Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators

Alexandru Plesco

N^o **d'ordre: 589**
N^o **attribué par la bibliothèque: __ENSL589**

THÈSE

en vue d'obtenir le grade de
Docteur de l'Université de Lyon
ÉCOLE NORMALE SUPÉRIEURE DE LYON

spécialité: Informatique
Laboratoire de l'Informatique du Parallélisme
École Doctorale de Mathématiques et Informatique Fondamentale

présentée et soutenue publiquement le 27 Septembre 2010
par Monsieur **Alexandru PLESCO**

Titre:

Transformations de programmes et optimisations de l'architecture mémoire pour la synthèse de haut niveau d'accélérateurs matériels

Co-Directeurs:

Monsieur	Christophe	Alias
Monsieur	Alain	Darte
Monsieur	Tanguy	Risset

Après avis de:

Monsieur Frédéric	Pétrot	(Rapporteur)
Monsieur Patrice	Quinton	(Rapporteur)

Devant la commission d'examen formée de:

Monsieur	Alain	Darte	(Examinateur/Co-directeur)
Monsieur	Steven	Derrien	(Examinateur)
Monsieur	Ronan	Keryell	(Examinateur)
Monsieur	Frédéric	Pétrot	(Président/Rapporteur)
Monsieur	Patrice	Quinton	(Examinateur/Rapporteur)
Monsieur	Tanguy	Risset	(Examinateur/Co-directeur)

N^o **order: 589**
N^o **assigned by the library: __ENSL589**

PhD THESIS

for the grade of

Doctor of Université de Lyon
ÉCOLE NORMALE SUPÉRIEURE DE LYON

field: Computer Science

Laboratoire de l'Informatique du Parallélisme
École Doctorale de Mathématiques et Informatique Fondamentale

presented and defended the 27 of September, 2010

by Mr. **Alexandru PLESCO**

Title:

Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators

Supervisors:

Mr.	Christophe	Alias
Mr.	Alain	Darte
Mr.	Tanguy	Risset

With the approval of:

Mr.	Frédéric	Pétrot	(Reviewer)
Mr.	Patrice	Quinton	(Reviewer)

With the dissertation committee composed of:

Mr.	Alain	Darte	(Member/Supervisor)
Mr.	Steven	Derrien	(Member)
Mr.	Ronan	Keryell	(Member)
Mr.	Frédéric	Pétrot	(President/Reviewer)
Mr.	Patrice	Quinton	(Member/Reviewer)
Mr.	Tanguy	Risset	(Member/Supervisor)

Alexandru Plesco: *Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators,* Doctorate, © September 27, 2010

Supervisors:
Christophe Alias
Alain Darte
Tanguy Risset

Faculty:
Computer Science

Department:
Laboratoire de l'Informatique du Parallélisme

University:
ÉCOLE NORMALE SUPÉRIEURE DE LYON

Location:
Lyon, France

Time Frame:
September 27, 2010

Version 1.0

Ohana means family.
Family means nobody gets left behind, or forgotten.

— Lilo & Stitch

Dedicated to the loving memory of Lidia Plesco.
1960 – 2009

Consider Columbus:
He didn't know where he was going.
When he got there he didn't know where he was.
When he got back he didn't know where he had been.
And he did it all on someone else's money.
(LINUX FORTUNE)

Acknowledgments

Many thanks to my supervisors for their moral support as well as for their help that made this thesis possible.

Thanks to Christophe ALIAS who joined forces during the last year of the thesis and who helped very much during the work and writing of the last chapter of this thesis.

Thanks to Alain DARTE for his guidance during the whole thesis, for his help and persistence that helped pass through seemingly unsolvable problems and many thanks for his major help during the writing of this thesis.

Thanks to Tanguy RISSET for proposing the internship on high level synthesis that I was searching for, for helping me find and apply for thesis scholarship, for helping me with the work and writing of Chapter 3 of this thesis, and for helping to find and apply for an ATER position that gave me time to finalize all the works on this thesis.

Many thanks to jury members for their time and pertinent remarks that helped me improve this writing.

Special thanks to my girlfriend Andreea CHIS for being with me on sunny days and especially during rainy ones.

Thanks to my family who made me who I am, and who supported me till the last moment of this thesis and to my precious mother that I miss a lot and who was with me till the last moments of her life.

Thanks to all the members of the LIP laboratory at ENS Lyon and TC laboratory at INSA Lyon that made the work here pleasant and fun.

Thanks to Bogdan PASCA and Christophe ALIAS for all the time passed in front of a white board during "tea breaks".

Thanks to my friends and everyone else that I didn't mention here for their help and support.

Abstract

A wide category of sold products including telecommunication and multimedia propose more and more advanced features and functionalities. These functionalities come at a cost of increased design complexity. For performance and power budget issues, these features can be accelerated using dedicated hardware accelerators. To meet the required time-to-market and development price, traditional hardware design methodologies are not sufficient and the use of high-level synthesis (HLS) tools is an appealing alternative. These tools are now getting more mature for generating hardware accelerators with an optimized internal structure, thanks to efficient scheduling techniques, resource sharing, and finite-state machines generation. However, interfacing them with the outside world, i.e., integrating the automatically-generated hardware accelerators within the complete design, with optimized communications, so that they achieve the best throughput, remains a very hard task, reserved to expert designers. The leitmotiv of this thesis was to study and to develop source-to-source strategies to improve the design of these interfaces, trying to consider the HLS tool as a back-end for more advanced front-end transformations.

In the first part of the thesis, as a case study, we designed by hand, in VHDL, an intelligent glue logic to interface an accelerator, for matrix-matrix multiplication, generated by the MMAlpha HLS tool. Using data dependence information, we implemented double-buffering and blocking techniques on a scratchpad-like local SRAM memory to exploit data reuse. This increased significantly the performance of the system but required also a significant engineering effort. We then showed, on several multi-media applications and with another HLS tool, Spark, that the same benefit could be obtained with a preliminary semi-automatic source-to-source (here C-to-C) transformations step. For that, we used an advanced state-of-the-art compiler front-end, based on the Open64 compiler and the WRaP-IT framework for polyhedral transformations. Significant improvements were shown in particular on the synthesis of part of the video color space conversion from MediaBench II benchmarks, for which data was fed through a processor cache memory. This study demonstrated the importance of loop transformations as a pre-processing step to HLS tools, but also the difficulty to use them depending on the HLS tool features to express external communications.

In the second part of the thesis, using the C2H HLS tool from Altera, which can synthesize hardware accelerators communicating to an external DDR-SDRAM memory, we showed that it is possible to automatically restructure the application code, to generate adequate communication processes in C, and to compile them all with C2H, so that the resulting application is highly-optimized, with full usage of the memory bandwidth. These transformations and optimizations, which combine techniques such as double buffering, array contraction, loop tiling, software pipelining, among others, were incorporated in an automatic source-to-source transformation tool, called Chuba, based on the polyhedral model representation. Our study shows that high-level synthesis (HLS) tools can indeed be used as back-end optimizers for front-end optimizations, as it is the case for standard compilation with high-level transformations developed on top of assembly-code optimizers. We believe this is the way to go for making HLS tools viable.

Keywords: High-level synthesis tools, hardware accelerators, DDR SDRAM, optimized communications, HPC, source-to-source program transformations, reconfigurable architectures, FPGA.

Résumé

Une grande variété de produits vendus, notamment de télécommunication et multimédia, proposent des fonctionnalités de plus en plus avancées. Celles-ci induisent une augmentation de la complexité de conception. Pour satisfaire un budget de performance et de consommation d'énergie, ces fonctionnalités peuvent tre accélérées par l'utilisation d'accélérateurs matériels dédiés. Pour respecter les délais nécessaires de mise sur le marché et le prix de développement, les méthodes traditionnelles de conception de matériel ne sont plus suffisantes et l'utilisation d'outils de synthèse de haut niveau (HLS) est une alternative intéressante. Ces outils sont maintenant plus aboutis et permettent de générer des accélérateurs matériels possédant une structure interne optimisée, grce à des techniques d'ordonnancement efficaces, de partage des ressources et de génération de machines d'états. Cependant, les interfacer avec le monde extérieur, c'est-à-dire intégrer des accélérateurs matériels générés automatiquement dans une conception complète, avec des communications optimisées pour atteindre le meilleur débit, reste une tche très ardue, réservée aux concepteurs experts. Le leitmotiv de cette thèse était d'étudier et d'élaborer des stratégies source-à-source pour améliorer la conception de ces interfaces, en essayant d'envisager l'outil HLS comme back-end pour des transformations front-end plus avancées.

Dans la première partie de la thèse, comme étude de cas, nous avons conu à la main, en VHDL, une logique intelligente permettant l'interfaage d'un accélérateur, calculant la multiplication de deux matrices, généré par l'outil de synthèse MMAlpha. En utilisant des informations sur les dépendances de données, nous avons implanté des techniques de double tampon et de calcul/-transfert par bloc (pavage), pour des mémoires locales SRAM de type scratchpad, pour améliorer la réutilisation des données. Ceci a permis d'augmenter de manière significative les performances du système, mais a également exigé un effort important de développement. Nous avons ensuite montré, sur plusieurs applications de type multimédia, avec un autre outil de HLS, Spark, que le mme avantage pouvait tre obtenu avec une étape préliminaire semi-automatique de transformations source-à-source (ici de C vers C). Pour cela, nous avons utilisé le front-end d'un compilateur avancé, basé sur le compilateur Open64 et l'outil WRaP-IT de transformations polyédriques. Des améliorations significatives ont été présentées, en particulier pour la synthèse de la conversion de l'espace couleur (extrait d'un benchmark de MediaBench II), dont les données étaient transmises via une mémoire cache. Cette étude a démontré l'importance des transformations des boucles comme étape de pré-traitement pour les outils HLS, mais aussi la difficulté de les utiliser en fonction des caractéristiques de l'outil HLS pour exprimer les communications externes.

Dans la deuxième partie de la thèse, en utilisant l'outil C2H HLS d'Altera qui peut synthétiser des accélérateurs matériels communiquant avec une mémoire externe DDR-SDRAM, nous avons montré qu'il était possible de restructurer automatiquement le code de l'application, de générer des processus de communication adéquats, écrits entièrement en C, et de les compiler avec C2H, afin que l'application résultante soit hautement optimisée, avec utilisation maximale de la bande passante mémoire. Ces transformations et optimisations, qui combinent des techniques telles que l'utilisation de double tampon, la contraction de tableaux, le pavage, le pipeline logiciel, entre autres, ont été intégrées dans un outil de transformation automatique source-à-source, appelé Chuba et basé sur la représentation du modèle polyédrique. Notre étude montre que ainsi qu'il est possible d'utiliser

certains outils HLS comme des optimiseurs de niveau back-end pour les optimisations effectuées au niveau front-end, comme c'est le cas pour la compilation standard o des transformations de haut niveau sont développées en amont des optimiseurs au niveau assembleur. Nous pensons que ceci est la voie à suivre pour que les outils HLS deviennent viables.

Mots-clés : synthèse de haut niveau, accélérateurs matériels, DDR SDRAM, optimisations des communications, HPC, transformations de programme source-à-source, architectures reconfigurables, FPGA.

Publications

Some ideas and figures have appeared previously in the following publications:

Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing DDR-SDRAM communications at C-level for automatically generated hardware accelerators. An experience with the Altera C2H HLS tool. In *21st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'10)*. IEEE Computer Society, July 2010

Alexandru Plesco and Tanguy Risset. Coupling loop transformations and high-level synthesis. In *SYMPosium en Architectures nouvelles de machines (SYMPA'08)*, Fribourg, Switzerland, February 2008

Contents

Acronyms

ALU arithmetic logic unit

ALUT Altera look-up table

ANSI American National Standards Institute

ANSI-C C standard published by American National Standards Institute

API Application Programming Interface

ASH Application-Specific Hardware

ASIC application-specific integrated circuit

Avalon-MM Avalon MM

Avalon-ST Avalon ST

B

BRAM block select RAM

C

CAD computer-aided design

C `C` programming language

CSP communicating sequential processes

C++ C++

CWB CyberWorkBench

COMU communication and interface unit

CABA cycle accurate and bit accurate

CLooGVHDL CLooGVHDL

C2H C-to-Hardware Acceleration

CPU central processing unit

CAS column address strobe

CL CAS latency

CPLI cycles per loop iteration

CLB configurable logic block

CDFG control/data flow graph

CODEC coder-decoder

D

DDP draft data-path

DSP digital signal processing

DFG data flow graph

DRAM dynamic RAM

DDR double data rate SDRAM

DDR SDRAM double data rate SDRAM

DMA direct memory access

DSOCM data-side OCM

DIMM dual in-line memory module

E

EDA electronic design automation

EDIF Electronic Design Interchange Format

ESL electronic system level

EDK embedded development kit

F

FSM finite state machine

FPGA field programmable gate array

FHM fast hardware model

FIFO First In, First Out

G

GLD gate-level description

GCC GNU Compiler Collection

GPP general purpose processor

GUI graphical user interface

GALS globally asynchronous locally synchronous

H

HLS high-level synthesis

HMMU hardware memory management unit

HA hardware accelerator

HDL hardware description language

HW hardware

HPC high-performance computing

HPF High Performance Fortran

HTG hierarchical task graph

I

IP intellectual property

IR intermediate representation

IC integrated circuit

ICD integrated circuit design

I/O Input/Output

IDE integrated development environement

ISA instruction set architecture

ISPS instruction set processor specification

II initiation interval

J

JEDEC joint electron device engineering council

JTAG joint test action group

L

LDP logic design phase

LIS latency insensitive systems

M

MEMU memory unit

MMU memory management unit

MM memory-mapped

N

NPA non-programmable accelerators

O

OOP object-oriented programming

Open64 Open64

OPB on-chip peripheral bus

OCM on-chip memory

ORC Open Research Compiler

P

PU processing unit

PLB processor local bus

R

RTL register transfer level

RAM random access memory

ROM read only memory

RISC reduced instruction set computing

R/W read/write

S

SW software

SOPC system on a programmable chip

SOC system on chip

S2S source-to-source

SSA static single assignment form

SCOP static control part

SRAM static RAM

SDRAM synchronous DRAM

ST streaming interface

SystemC SystemC

SIMD single instruction, multiple data

SSE streaming SIMD extensions

T

TLM transaction-level modeling

TLM-T TLM with timing

U

UML Unified Modeling Language

USP user specification phase

V

VHSIC very-high-speed IC

VHDL VHSIC hardware description language

Verilog Verilog

VLSI very-large-scale integration

VLIW very large instruction word

W

WHIRL Open64 intermediate representation

WRaP-IT WHIRL Represented as Polyhedra

WCET worst-case execution time

X

XNF Xilinx netlist format

XUP Xilinx university program

Chapter 1

Introduction

Recent trends in embedded system design have brought new concepts such as platform-based design, network on chip, and higher levels of specification formalisms. Using predefined hardware blocks, also known as intellectual property (IP) re-use, is now widely considered as the main way of improving design efficiency. Embedded software design, i.e., the design and optimization of the software, takes now a major part of the time of a system on chip (SOC) design (for example the complete design of a smartphone), which includes the design of the architecture and of the applications that will be running on it. However, for widely-sold products such as telecommunication and multimedia devices, the design of dedicated hardware accelerators (IP design) is still mandatory because it provides better trade-offs between performances (especially in terms of power consumption) and cost (chip area).

The increasing complexities of accelerated algorithms and of demands of processing power have led integrated circuit (IC) manufacturers to an ever increasing IC size and complexity. Using traditional languages such Verilog or VHDL [1] to specify a complex IC design becomes a very tedious and error-prone task. In order to minimize this design effort, the abstraction level of the specification languages was increased. The tools developed to synthesize a circuit from such a high-level abstraction have been called HLS tools.

For many years, HLS has been foreseen as the solution to accelerate dedicated hardware design. Ideally, HLS should enable the automatic generation of efficient hardware designs from functional specifications expressed in some high-level programming language. We think that HLS failed, up to now, to integrate the industrial design flow because it was not mature enough to solve important technical problems:

- A huge design space to explore: the potential parallelism and the variety of target architecture technologies imply the use of multi-criteria optimizations. Some choices must be made by the designer to reduce the design space.
- Interface design: in many HLS tools, the process of integration of the generated hardware accelerator into the system is either manual or based on template-based interfaces that do not take into account the data transfer pattern of the accelerator.
- The memory bottleneck: the memory size and the memory traffic have become major components in the chip power consumption. Optimizing data accesses is even more difficult than optimizing parallelism exploitation.
- Data reuse: while some high-performance compilers have very efficient code transformation

1. In VHSIC hardware description language (VHDL), VHSIC stands for very-high-speed IC.

techniques to increase the data reuse in central processing unit (CPU) with caches, HLS tools usually leave these transformations and optimization decisions to the user.

In recent years HLS has become a necessity, mainly because the exponential increase in the number of gates per chip far outstrips the productivity of human designers. Besides, applications that need hardware accelerators usually belong to domains, like telecommunications, multimedia and game platforms, where fast turn-around and time-to-market minimization are paramount.

Today, synthesis tools for field programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs) come in many shapes. At the lowest level, there are proprietary synthesis and place and route tools, whose input is a VHDL or Verilog specification at the structural level or register transfer level (RTL). The direct use of these tools will always be difficult and restricted to expert hardware designers. Indeed, a structural description is completely different from an usual algorithmic language description, as it is written in term of interconnected basic operators. Also, synthesis tools have trouble handling loops. This is particularly true for logic synthesis systems, where loops are systematically unrolled (or considered as sequential) before synthesis. More generally, a VHDL design is at a too low level to allow the designer to perform, easily, higher-level code optimizations. This is especially true on multi-dimensional loops and arrays, which are of paramount importance to exploit parallelism, pipelining, and perform memory optimizations.

In the last decade, important efforts have been made to make possible the generation of low-level VHDL specification from higher-level specifications. Two main approaches can be identified. The first one is the bottom-up approach. Different tools such as the "behavioral compiler" from Synopsys [18] tried to synthesize from a common language such as VHDL with added behavioral abstractions. However, this approach failed to establish itself as the solution since it did not diminish sufficiently the design effort and it did not increase the synthesis quality enough for pushing the designers to learn and use it. The other approach is a top-down approach. A lot of works considered a very commonly used and known language like such as `C` programming language (C) or C++ (C++). To increase the quality of the synthesis results, the languages support extensions for low-level hardware constructs and compiler directives that are used by the user to guide the synthesis process. At the beginning, these languages had very little success mostly due to poor synthesis results. However, later on, more advanced languages such as Handel-C [16] and tools appeared, both in academia, such as Spark [78], Gaut [90], Ugh [28], Nisc [9], and in industry such as C2H [2], CatapultC [8], Impulse-C [7], Pico Express [11], to quote but a few. These tools are now quite efficient for generating finite-state machines, for exploiting instruction-level parallelism, operator selection, resource sharing, and even for performing some form of software pipelining, for one given kernel. In other words, we believe that it now start to be acceptable to rely on them for optimizing the heart of accelerators, i.e., the compute part of it. In other words, HLS tool may be considered as the equivalent of back-end optimizers in standard (software) compilers.

However, this is only part of the complete design. In general, the designer seeks a pipelined solution with optimal throughput, where the mediums for data accesses (either to local memory or for outside communications) are saturated, in other words, a solution where bandwidth is the limiting factor. The HLS tool should then instantiate the necessary hardware and schedule computations inside the hardware accelerator so that data are consumed and produced at the highest possible rate. But, for most tools, the designer has still the responsibility to decompose the application into smaller communicating processes, to define the adequate memory organization or communicating buffers, and to integrate all processes in one complete design with suitable synchronization mechanisms. This task is extremely difficult, time-consuming, and error-prone. Some designers

even believe that relying on today's HLS tools to get the adequate design is just impossible and they prefer to program directly in VHDL. Indeed, some HLS tools do not consider the interface with the outside world at all: data are assumed to be given on input ports, available for each clock cycle, possibly with a timing diagram to be respected [77, 63]. Then, the user has to design all the necessary glue (explicit communications, scheduling of communications, synchronizations) in VHDL or with ad-hoc libraries and structures [70][47, Chap. 9]. Some other tools, for example Ugh and CatapultC [28, 8], can rely on First In, First Out (FIFO)-based communication. The designer still needs to define the FIFO sizes, the number of data packed together in a FIFO slot (to provide more parallelism), and to prefetch data to hide memory latencies. Finally, some tools, such as `C2H`, also allow direct accesses to an external memory and is (sometimes) able to pipeline them. But, again, the designer has to perform preliminary code transformations to change the computations order and the memory organization to hide the latency and exploit the maximal bandwidth.

The leitmotiv of this thesis consists in optimizing the design of the interface that connects the accelerator to the outside world. As we know, most of the time the performance of a hardware accelerator is limited by the data availability. Thus, optimizing the data availability can increase its performance. The first part of this work (Chapter 3) concerns a situation where the generated hardware accelerator can take advantage of an existing cache in the architecture, directly or indirectly. The second part (Chapter 4) explores the situation where the hardware accelerator can and has to access directly an external double data rate SDRAM (DDR) memory.

In the first part of Chapter 3, instead of using a template interface (limited by nature) to the outside world, we build a glue logic that contains the interface and a local memory architecture. Data access information from the application is used to find the possible data reuse. We manually generated a local memory architecture that conform to this data reuse information. The whole logic is connected to the processor and we analyze specific aspects of this type of connection. This study is done with the HLS tool MMAlpha for a particular application, the matrix-matrix multiplication of complex numbers, and is exposed in the first part of Chapter 3. It demonstrates, on one particular example, how difficult is the interfacing problem and what is the potential loss of performance of such an interface in a complete system.

In general, and not only for the tool we used (MMAlpha), designing such a glue logic by hand is a very time-consuming and error-prone process, and performances are not always as good as expected. Another solution to improve the data availability is to transform the input code of the HLS tool to increase the data reuse. But transforming the code by hand is not an easy task either. Such transformation could be implemented in the HLS tool itself. However, this would require an advanced knowledge of its intermediate representation (IR) and, anyway, most of the time, these tools are provided in binary format, so such a transformation stage cannot be integrated. Instead, in the second part of Chapter 3, we present a source-to-source automatic code transformation methodology based on the WRaP-IT loop transformation library and scripting language Uruk. Being source-to-source, this methodology can be used in front of any tool whose input is a specification in C, so as to perform, as a pre-processing step, some of the loop transformations that are mandatory to get the code in a suitable form to go through the HLS tool and get acceptable performances. Our study focuses on one particular HLS tool, Spark, and on optimizing the memory communications and data reuse. This requires a local memory where the reused data can be stored. As `Spark` is not able to generate local memories, we use the cache available on the platform to control the data accesses. But our study goes beyond the two particular tools we use (`Spark` and WRaP-IT, itself integrated in `Open64`): it demonstrates that HLS can be coupled with software

compilation to achieve even better results than HLS tools alone can do. This does not eliminate the need for loop transformations fully integrated in the HLS tool (some may be mandatory, depending on its design flow), but it lets the user control and apply some of them in a semi-automatic way at source level.

In Chapter 3, the performance improvements were obtained mostly thanks to data reuse, either by exploiting an available cache or by designing a local memory architecture. Designing such a memory architecture by hand is not only difficult but it is also not very flexible. Since, as mentioned earlier, HLS tools are getting more mature to generate compute-parts of applications, one should indeed be able to apply various code transformations at source level so as to optimize data transfers and reuse, and to rely on the HLS tool itself to synthesize the result of these transformations, including the required glue (if any). We believe that being able to automatically generate an efficient interface for an automatically-generated accelerator is a *sine qua non* condition for making HLS tools a usable thus viable solution to hardware design, in the same way a traditional front-end compiler can perform high-level optimizations on top of any assembly-code optimizer. It is no longer acceptable that the user spends so much time to interface the accelerator obtained by HLS as if, today, when compiling C to assembly code, the user had to spend a lot of time connecting the different pieces of automatically-generated assembly code. The challenge here is thus to be able to perform code optimizations at C level that are directly beneficial when used in front of an HLS tool, with no modification of the tool itself.

In Chapter 4, we show that such a memory architecture and interface can be generated automatically, using the HLS tool itself, in a form of "meta-compilation", at least with the HLS tool we consider. Our study is done with the `C2H` Altera tool, for accelerators with external accesses to a DDR memory, always keeping in mind that any code transformation we perform can be automated. We first analyze `C2H` and we identify the features that make DDR optimizations feasible or hard to perform. We then communicating processes, and of software pipelining that can lead to fully-optimized DDR accesses. The idea is to keep the bandwidth saturated as much as possible, orchestrating communications and computations in a pipelined fashioned, using local memories to store data before they are consumed.

In Chapter 5, we present a method to derive automatically from a naive kernel written in C, a set a C functions implementing the pipelined communicating accelerators optimized for `C2H`, as proposed in chapter 4. Original techniques are defined to minimize the communications with the DDR, manage the local memory and generate the final code. The user must delimit the kernel to be optimized and specify several other parameters with compiler directives. The method has been fully implemented and the first experiments gives promising results. We believe that our technique can also be used to optimize loop-based programs for GPUs (graphics processing units).

To summarize, this thesis manuscript is organized as follows. After this short introduction, Chapter 2 presents the related works in HLS and code transformations, in particular those related to HLS based on the C language. In Chapter 3, we present the manually-designed local memory architecture and interface for the example of MMAlpha, then the source-to-source methodology we experimented in front of the HLS tool Spark. In Chapter 4, we present our various attempts to optimize the DDR accesses with the Altera `C2H` tool and the solution we found with the constraint that it can be automated. In Chapter 5, we show that the process we designed can be make automatic, with adequate program analysis, program transformations, and code generation. Chapter 6 raises some conclusions and perspectives.

Chapter 2

Related works in high-level synthesis

The ever increasing complexity of the circuits imposed a transition from low-level hardware languages, used in traditional hardware design, to languages with a higher level of abstraction such as C used in high-level synthesis (HLS) tools. The use of a higher abstraction level eases the design process, making it more flexible. However, as observed in the first generation of HLS tools, the synthesis quality was far from manual-level designs [91]. As the circuit complexity increased, the use of HLS became mandatory to get a better time to market. To raise the quality of synthesis results, the synthesis tools were extended with some add-ons. One of them is the extension of the input language with low-level hardware constructs. Unfortunately, such constructs could be only understood by hardware designers eliminating the software developers from the potential users. Another approach is to use configuration files to drive the synthesizer at a coarse-grain level. A better approach is to use compiler directives that can guide the synthesizer at a very fine grain so that the generated design approaches the desired one. There are many other approaches like class libraries and template/draft architecture definition methods that can be used to guide the HLS tool. In this chapter, we describe these different approaches together with their advantages and disadvantages. The goal of this chapter is not to be exhaustive, this would be impossible, but to give an idea of the evolution of HLS tools and of their underlying ideas, focusing mainly on tools based on variants of the C language. We also describe, in the form of a catalog, some code transformations (in particular loop transformations) that can be useful for HLS.

2.1 Hardware description languages and early HLS

2.1.1 Conventional low-level hardware description languages

One of the most used abstraction of hardware circuits is RTL (register transfer level) [14]. It has the lowest abstraction level for defining a synchronous digital circuit in terms of logical operations. The circuit structure is defined by hardware registers and logical operators. They are connected by signals that perform transfers of data. RTL description is very close to the gate-level description (GLD) and is very well suited for the logic design phase of the integrated circuit design (ICD). One of the advantages of using such an abstraction is that the user have the full control of the system being implemented. However, as the designed systems became more complex, the amount of designer work used to describe complex and large digital systems became not economically viable.

To circumvent this problem, languages like structural VHDL [14, 17] were proposed. VHDL is used as a design language for FPGAs and ASICs. It was originally developed at the US Department

of Defense in order to document the behavior of the ASICs (described as RTL) that supplier companies were included in equipment. It was revolutionary when it was introduced. VHDL has a syntax mainly based on the ADA programming language, along with an added set of constructs to handle hardware parallelism. While it can be used for basic RTL description, the language incorporates higher-level constructs. One can describe multiple entities and their corresponding architecture implementations. Each architecture can use multiple components described elsewhere in entities. It is more flexible than RTL and provides IP. However, recent increase in hardware complexity has shown that manual structural VHDL coding is very time-consuming and error-prone. Behavioral VHDL has a higher level of abstraction than structural VHDL. The main difference is that a design described in behavioral VHDL is not always synthesizable as opposed to structural VHDL, i.e., it is not always possible to derive automatically a layout with a synthesis tool.

Another hardware description language (HDL) language used to model electronic systems is Verilog [14, 15]. It was invented by Phil Moorby at Automated Integrated Design Systems (later renamed to Gateway Design Automation) in 1985 as a hardware modeling language. Gateway Design Automation was later purchased by Cadence Design Systems in 1990. Cadence now has full proprietary rights to Gateway's Verilog and the Verilog-XL simulator logic simulators. As opposed to VHDL designers that opted for a RTL syntax (at least in the earlier versions of VHDL standard), the designers of Verilog wanted a language with a syntax similar to the C programming language so that it would be familiar to engineers and readily accepted. The language has a pre-processor like C, and the major control keywords such as `if`, `while`, etc., are similar. The formatting mechanism in the printing routines and the language operators (and their precedence) are also similar. Like VHDL, the language differs from a conventional programming language in that the execution of statements is not strictly linear. A Verilog design consists of a hierarchy of modules. Modules are defined with a set of input, output, and bidirectional ports. Internally, a module contains a list of wires and registers. Concurrent and sequential statements define the behavior of the module by defining the relationships between the ports, wires, and registers. Sequential statements are placed inside a begin/end block and executed in sequential order within the block. But all concurrent statements and all begin/end blocks in the design are executed in parallel. A module can also contain one or more instances of another module to define sub-behaviors.

Even though the VHDL language has higher level constructs and a higher behavioral level of abstraction, `Verilog` is still used worldwide much more than VHDL. Assuming that the user has zero knowledge in either languages, `Verilog` is the easiest to understand. VHDL is less intuitive for multiple reasons. One of the reasons is that it is strongly typed. This feature makes it robust and powerful for the advanced user. However, this is only after a longer learning phase. Another reason is that there are many ways to model the same circuit (especially those with large hierarchical structures). The VHDL roots are based on ADA. Verilog has more similarities with C because its constructs are based half on C and half on ADA. For this reason, an existing C programmer may prefer Verilog over VHDL.

Independently on the programming language (VHDL or Verilog), designing systems with an ever increasing complexity using these languages becomes more expensive, tedious, and error-prone. We must take into account the fact that the success of a language depends a lot on its understanding simplicity and similarities with well-known and greatly-used programming languages.

2.1.2 Early attempts in high-level synthesis

The concept of high-level synthesis started in early 1974 when Mario Barbe from Carnegie Mellon University noticed that one could compile (synthesize) an instruction set processor specification (ISPS) into hardware [47, Page 15]. Later, HLS was defined as a process of automatic (or semi-automatic) generation of hardware from an algorithmic description. Since then, there were many attempts to synthesize circuits from different types of input specifications. In this section, we first quickly recall different attempts and mention their advantages and disadvantages.

In order to increase the acceptance of the HLS tools, some industry electronic design automation (EDA) such as Synopsys focused on increasing the abstraction level of existing languages instead of trying to synthesize efficiently from higher-level specifications. As we mentioned earlier, the specification written in behavioral VHDL or Verilog is not always synthesizable. The "behavioral compiler" from Synopsys addressed this issue. This reduced the design effort of RTL synthesis designers. However, on the other hand, software and algorithm designers were still unable to use this tool because it still required a huge hardware design knowledge. Another disadvantage of behavioral synthesis was the rather poor and unpredictable quality of results. The design was very difficult to validate because its timing behavior was very dependent on the synthesis tool optimizations. This had a major impact on the HLS tool acceptance.

One of the first attempts in HLS to synthesize from a high-level specification was the work of Snow in late 80's during his thesis [47, Page 17]. He was one of the first to propose the use of control/data flow graph (CDFG) as an input specification to the synthesis system. In later year, Emil Gircyz in his thesis proposed the use of the ADA language to model hardware [47, Page 17], which is, as previously mentioned, the predecessor of the VHDL language.

In the domain-specific category, Cathedral and Cathedral-II [57] were the most well-known. The tools were used for digital signal processing (DSP)-oriented specifications. The specification language was called Silage and it was developed by IMEC. Silage is a pure behavioral language without any structural biases. It was oriented to describe DSP algorithms. However, the domain specialization was not appropriate for the majority of ASIC designers. Early ASIC designers indeed concentrated on control logic rather than on dataflow and signal processing. The designs were mostly written by hand in an rather poorly-structured random logic.

These high-level synthesis attempts failed to establish themselves in the hardware design cycle of many engineers. Most of them were only starting to move from schematic system design to the HDL methodology. Learning new and much different languages such as Silage, together with their different design methodologies, was not an easy task and was not considered as a good investment. Another important drawback of the first HLS tools were the very poor quality of the generated results compared to the manual approach. This was mainly due to the simplicity of the optimizing algorithms used.

2.2 Advances in high-level synthesis

In this section, we present some advances in high-level synthesis. We mostly focus on tools that synthesize from a C-like language, since C is very popular among hardware and software designers. We present different types of synthesis approaches used to increase the synthesis easiness and quality. We also try to classify the tools according to these different approaches, following their most important feature. But, of course, these tools usually incorporate multiple approaches and this classification may therefore seem a bit artificial, on some aspects. For a more detailed information,

one may consult the books of Diniz et al. [41] and of Coussy et al. [47], and the article "High-Level Synthesis: Past, Present, and Future" [91].

The first work that addressed the designers language familiarity problem was the work of Stroud et al [102]. They introduced a new C-like language called Cones. This was the very first time an input to the high-level synthesis system was a C-like language. The input specification was synthesized by the Cones HLS tool. It is an automated synthesis system that takes a behavioral circuit description written in C and generates a gate-level implementation. Its name is based on the logic cone that is formed by inputs and outputs connected by arrows representing data dependences.Each function is synthesized independently. It has input and output variable definitions. Cones supports a subset of C that includes `if`, `switch`, and `for` constructs. It also provides macros to handle array of inputs, outputs, memory elements, and temporary variables. Memory elements are defined explicitly with no memory controller or scheduler to fetch them. Cones can be used with single-clock, single-edge [1] synchronous designs, or purely combinatorial ones. Even though the language was very familiar to designers, being only able to synthesize combinatorial designs limited its usability.

2.2.1 Low-level hardware constructs

One of the problems with the synthesis from a high-level language like C was the unexpected synthesis results. To guide the HLS tool, hardware constructs were introduced in the language.

Ku and De Micheli from Stanford University addressed this issue of synthesizing sequential circuits and developed HardwareC [58, 84, 64] as an input to the Olympus synthesis system. It has a C-like syntax and it represents a behavioral HDL with support for declarative and procedural semantics. It supports the notion of concurrent processes, message passing, timing constraints, resource constraints, hardware structure, and hierarchy. It supports adjusting the degree of parallelism by grouping specific operations using different types of parentheses defined in the language. External interfacing is done thanks to port passing or message passing using channel mechanisms.

Another C-like language that has hardware constructs is `SpecC` [14, 64, 62]. It was developed by Gajski et al.'s at University of California, Irvine. As HardwareC, it is based on C and it offers special extensions (keywords) to cover the needs of embedded designs. In addition to low-level hardware constructs, it also provides constructs for finite state machines (FSMs), concurrency, pipelining, and structure. As such, the SpecC language provides a minimal set of orthogonal constructs that covers the concepts identified in embedded systems. Systems written in the complete language must be refined into the synthesizable subset. Channels are used for synchronization and communication between modules. Unlike previously-presented synthesis tools, `SpecC` was designed with system integration in mind. The obtained architecture contains a software processor and a custom hardware. A part of the initial algorithm will run on the software processor and the rest will run on custom hardware. The two parts communicate using a generated interface.

Another more advanced language with hardware constructs is `Handel C` [14, 16, 64]. Developed at Oxford University Computing Laboratory, it was adopted by Agility [1] until Mentor Graphics acquired its assets [14]. It is a programming language designed for compiling programs into hardware images of FPGAs or ASICs. It uses much of the syntax from C standard published by American National Standards Institute (ANSI-C) with addition of explicit parallelism. By default, the written instructions will be executed in a sequential order. To execute a part of the code in parallel, one must use the `par` keyword. In this case, the instructions that have to be executed

1. Digital designs can use the two edges of the clock, i.e., the rising and the falling one. When only one is used, the design is called single-edge.

in parallel will be executed at the same instant in time by separate pieces of hardware. The language also provides channel communications in order to synchronize parallel branches of a program. Handel-C semantics is inherited from the communicating sequential processes (CSP) formalism. It also imposes a restriction that a single variable can be written by only one parallel branch and may be read by several parallel branches. The variables can be mapped to memory elements such as random access memory (RAM) and read only memory (ROM) by using type extensions and cannot be accessed more than one element at a time, otherwise the compiler will generate an error. Before the start of the main function, the clock at which the main function will run needs to be defined. More than one main function can be defined if different parts of a program need to run with different clocks. Handel-C comprises all common expressions necessary to describe complex algorithms, but lacks processor-oriented features like pointers and floating-point arithmetic. The programs are mapped into hardware at the `netlist` level, currently in Xilinx netlist format (XNF) or Electronic Design Interchange Format (EDIF) format.

Another C-like language, designed at Sharp, `Bach C` [82] based on the C language had fewer low-level constructs. It has extensions to support explicit parallelism, communication between parallel processes, and a bit-width specification of data types and arithmetic. The semantics of the language is untimed so that users specify the hardware behavior without worrying about when operations will take place. It supports arrays but not pointers. The input specification was synthesized by Bach which was developed at Sharp Corporation in 2001. Sharp has also developed a simulation and debugging environment for Bach C, and a high-level synthesis tool for turning Bach C programs into RTL-VHDL. The algorithm described in Bach C is partitioned into the hardware (HW) (hardwired logic) and software (SW) parts. The SW part can be executed on an ARM7. The HW part is compiled into RTL circuits using the Bach synthesizer, meeting the given constraints. The compiler uses a logic-synthesis tool to estimate the delay and area of each functional unit in the design. It has some issues. HW/SW partitioning has to be done by hand. Performance, area, and power consumption cannot be evaluated at the level of Bach C. The HW/SW interface circuit is generated automatically and represents a `AMBA` bus. IP reuse is done manually by manual modifications in the VHDL code [82].

Finally, another language that restricted the number of low-level constructs was the language `Transmogrifier C` [71, 14]. It was developed at the University of Toronto and is a restricted subset of the ANSI-C programming language with a minimum number of extensions. The communication is performed by using Input/Output (I/O) ports defined in the language as low-level constructs and directives. It does not support multiply, divide, pointers, arrays, structure, and recursion. It is used by the Transmogrifier C HLS system. Now Transmogrifier C is called FPGAC and is an open-source project. Its usability is rather limited because of its very restricted subset of the C language and because it targets only the platforms based on the Xilinx XC4000 series FPGAs.

The HLS systems presented above were able to synthesize synchronous and much more complex circuits compared to systems like Cones. However, this does not come for free. To be able to optimize the synthesis results, the designers were forced to guide the HLS tool using very low-level constructs. Most of these constructs could be understood by only the hardware designers, eliminating the software designers from the possible users. One of the main disadvantages of low-level hardware constructs was that the designer was usually forced to completely rewrite its specification written in C in order to get it synthesized by the HLS tool. Poor quality, variable and unpredictable synthesis results, together with the difficulty to manage low-level hardware constructs were some of the main factors that perished the success of these tools.

2.2.2 Configuration files and compiler directives

The previous section mentioned some HLS tools that defined hardware constructs in the language in order to inform the HLS tool of some hardware specific attributes. As explained earlier, this however limited the code reuse between the hardware and software parts in a hardware/software co-design. There are multiple solutions to this problem.

Configuration files The first solution is to use a configuration file that guides the HLS tool. This solution is for example used by the HLS tool `Spark` [78]. It was designed as a continuation of the `MI-CRO` project 00-037 (2001) and was developed at the University of California, San Diego, funded by Semiconductor Research Corporation and Intel. It takes behavioral ANSI-C code as input and generates synthesizable register-transfer level VHDL. `Spark` does not accept code that contains pointers, function recursion, and irregular control-flow jumps. It uses hierarchical task graphs and dataflow graphs as the internal representation of the program. In a pre-synthesis phase, using parameters from the configuration file to tune heuristics, `Spark` applies several optimization techniques and coarse-level code restructuring as function inlining, loop unrolling, loop fusion, common sub-expression elimination (CSE), copy propagation, dead code elimination, loop-invariant code motion, induction variable analysis (IVA), and operation strength reduction. After this pre-synthesis phase follows a scheduler phase, which is organized in two parts. The first part contains heuristics that perform scheduling and the second part contains a toolbox of synthesis and compiler transformations such as percolation and trailblazing (code motions techniques), speculative code motions, chaining across conditions, and other. The resource allocation has to be done by the designer who has to specify in the configuration file the amount of available hardware elements and their timing information. All these transformations have a very important impact on maximizing the parallelism and performance of the final system. After the scheduling phase follows the control generation phase that minimizes interconnects between units and generates a FSM that implements the controller. Finally, a back-end code generation pass generates synthesizable a RTL VHDL description that can be synthesized by commercial tools. We will use this tool in Chapter 3.

A similar methodology was used for the HLS tool `C2Verilog`. It was introduced in 1998 by CompiLogic that was later bought by Synopsys in 2001 [101, 64]. The compiler accepts pointers, structure, arrays, loops, and function calls. Each function is synthesized independently and each function parameter is translated into an input and returned into an output port. The compiler can be informed, by the graphical user interface (GUI), of different global settings such as bit widths, multiplication and divider options, etc. In order to tune the performance of the system, the designer has to iterate through different compiler options, logic synthesis preferences and, if a finer grain change is required, using direct C code modifications.

Another example of tool that uses configuration files is CyberWorkBench (CWB), which was designed internally at `NEC`. It includes synthesis, verification, and hardware software co-simulation. The input language is an extension of ANSI-C and is called `BDL` or `Cyber-C` or SystemC. The basic concept of the framework is the all-modules-in-C and all-processes-in-C. Everything is expressed by means of the C language. The tool can perform such fusion, array expansion, loop pipelining (also called loop folding). The tool automatically instantiates a CPU, its corresponding bus interface, and the connection to the hardware modules.

Finally, it is worth mentioning AutoPilot, an electronic system level (ESL) synthesis system of AutoESL [3, 47], which uses configuration files containing user constraints and directives to guide the synthesis process. Based on the UCLA xPilot system [44], it can automatically synthesize an input code written in an untimed or partially timed C, C++, or SystemC (SystemC) code.

The tool is based on a commercial C/C++ compiler and can automatically perform optimizations such as constant propagation, dead code elimination, common sub-expression elimination, strength reduction, if-conversion, loop unrolling, loop flattening, loop fusion, intra and inter-procedural analysis, bit-width analysis and optimization, and pipelining. The synthesis process takes into account user constraints such as latency, throughput, and resources defined in the configuration file. Dynamic pointers, dynamic memory allocations, and function recursions are not supported.

To summarize, configuration files can express very general requirements of the designer. Compared to low-level hardware constructs, the use of configuration files does not change the code allowing a much more flexible hardware/software partitioning. However, even though some HLS tools have a very powerful internal optimization framework, the results are not always acceptable by the designer. Communicating with the tool with a rather limited tuning of the internally predefined heuristics is not always sufficient to obtain a design with the required performances and characteristics. Using a configuration file, the designer cannot express fine-grain specifications such as, for example, which loop to parallelize. The only solution is to change the code, in an iterative manner and by hand, and to check the obtained results, which is very tedious and error prone.

Compiler directives Instead of configuration files, used for coarse-grain configuration of the synthesis process, one can use compiler directives (pragmas). Compiler directives can describe code optimizations at a much finer grain. Also, as for configuration files, code compatibility is ensured with pragmas: a code written using pragmas can be synthesized to a dedicated hardware or compiled as a software for a standard processor. The compiler will then ignore pragmas that it does not understand without generating an error. At the same time, an advantage of using pragmas over low-level hardware constructs is the flexibility of the code. To change the behavior of a specific part of a code, one just has to write a pragma instead of changing the structure of the code, which would be error prone. We present below some of the tools that use compiler directives.

One tool that uses, in the input language specifications, compiler directives to guide the synthesizer is `Impulse C` [14]. It was developed by Impulse Accelerated Technologies for their `Impulse C` compiler. It represents a subset of the ANSI-C. The programming model is a variant of CSP. The synthesizer accepts multiple predefined pragmas such as `NONRECURSIVE` used to inform the optimizer that a given array variable cannot be scalarized, `PIPELINE` used to inform the synthesizer that the loop should be pipelined, and so on. From its subset of C, the `Impulse C` compiler generates an FPGA hardware accelerator using HDL. The architecture is generated from the CSP description described in the C language. It is more dataflow and streaming oriented, with processes accepting data, performing computations on it, and sending them after. The inter-process communications are performed using streams or shared memories.

Anther commercial tool that accepts pragmas to guide the synthesizer is `Nios II C2H` [22, 21]. It is a tool developed by Altera Corporation that synthesizes hardware accelerator co-processing modules to the `Nios II` soft-core processor. The design starts with an algorithmic description written in ANSI-C that is supposed to be executed on the processor. Some parts of the code (that are separated in functions) can be selected to be implemented in hardware. The `C2H` compiler will automatically redirect all calls to a hardware function to its hardware implementation. The calls to the hardware accelerator can be blocking or non-blocking. In the blocking call implementation, the processor waits in a while loop reading the state of the status register of the accelerator. In the non-blocking call, the designer must register an interruption routine. The compiler supports most C constructs such as pointers, arrays, structures, global and local variables, loops and subfunction calls. Each syntax element is mapped in a straightforward way into a hardware element

thus, in particular, with no resource sharing. Each loop or function is translated into a separate state machine that is synchronized to the upper hierarchy one. As opposed to `Impulse C`, the compiler will automatically pipeline the loops based on its data dependence analysis. There are also multiple pragmas defined in the input language specification that can be used to define, for the compiler, the connections of variables to memories, the blocking and non-blocking function executions, etc. The compiled hardware accelerator is integrated into the system on a programmable chip (SOPC) builder system and is connected to the rest of the system by means of the Altera's Avalon interconnect. The hardware accelerator supports parallel scheduling, which is performed automatically based on the data dependences, instead of using pragmas as in `Handel-C`. In a state machine, multiple constructs can be scheduled for parallel execution. The constructs can be simple statements but also other state machines. The hardware accelerator can access the (external) memory directly by using master port connections. The accesses are then pipelined in order to reduce the memory latency penalty. We will use this tool in Chapter 4.

One of the tools that does not really accept directives written in the code but, nevertheless, can specify code optimizations at a fine-grain level, using GUI or Tcl scripts, is Catapult C [14, 47]. Previously called PrecisionC, it is a ESL commercial HLS tool of Mentor Graphics. It takes as input ANSI-C/C++ and SystemC and generates an RTL description. The interface generation is done by parsing the C++ arguments and generating Catapult C supports multi-level clock gating, multi-block systems, loop unrolling, loop fusion, and loop pipelining. The Catapult C language supports pointers, classes, templates, and operator overloading. It uses some user memory directives to specify, for example, to split the mapping of an array into multiple memories. The input code should be statically determinable.

Instead of using compiler directives, one can also use code labels and a separate configuration file to inform the compiler about the required code transformations. This is used in the academic HLS tool CLooGVHDL (CLooGVHDL). It accepts a C description and generates a VHDL design. The tool is built on the side of the WHIRL Represented as Polyhedra (WRaP-IT) tool. The input C code is parsed by the Open64 (Open64) compiler. Its intermediate representation, whirl, is fed into the WRaP-IT tool that performs polyhedral loop transformations. CLooGVHDL accepts the so-called VIM-scripts and the `CLooG` outputs to generate a HW architecture, composed of functional computational blocks, loop control, internal FIFOs and memories, and a pre-fetch request module with synchronization points [61, 59]. However, CLooGVHDL is just a toy HLS tool, which does not include any complex optimizations or parallel execution. As explained in Section 2.3, our work in Chapter 3 uses WRaP-IT too, but in front of a more complete HLS tool, namely `Spark`.

As this brief survey shows, compiler directives are very useful in HLS. The designer does not have to change the C code that was written for a software part to get it synthesized into hardware: the user can just add, during the synthesis process, pragmas to the code. Unlike low-level hardware constructs, the code can still be compiled in software. For example, when using `C2H`, if a function is compiled into software, hardware pragmas are ignored by the software compiler. Actually, in practice, to get efficient designs, the process is a bit more complicated. In addition to pragmas, the C code may need to be rewritten in a particular way so that the hardware design generated by the HLS tool delivers adequate performance. But it can still be compiled into software, which is, in particular, very useful for verification and simulation. Also, compared to the configuration files, pragmas can specify at a much smaller grain the designer's requirements (when needed).

2.2.3 Class libraries, template architectures, etc.

Class libraries Another approach used to guide the synthesis tool to a suitable synthesis result is by using class libraries. These libraries allow the designer to use extension to an existing programming language, such as specialized data-types with corresponding methods or functionalities. We present below several HLS tools of this category.

`Gaut` [47] is an academic and open-source HLS tool dedicated to the synthesis of DSP applications. The synthesis starts with an algorithmic description written in C with Mentor Graphics class libraries that allow designers to specify bit-accurate data types such as `ac_int` and `ac_fixed` that represent bit-accurate integers and bit-accurate fixed-point data types. The compilation unit is based on GCC [2]. During the compilation phase, several code optimizations are performed such as dead code elimination, value range propagation, redundancy elimination, loop-invariant code motion, loop peeling, loop fusion, partial loop unrolling. The designer can specify the variable location (memory, register or both (dynamic)) using the so-called locality-threshold parameter so as to minimize the cost of storage elements. The user is guided by a lifetime variables histogram and the frequency of utilization. Using design constraints specified by the user, Gaut generates a hardware accelerator that is composed of a pipelined processing unit (PU), a memory unit (MEMU), and a communication and interface unit (COMU) based on latency insensitive systems (LIS). The synchronous components are encapsulated into combinational logic that drives them only when all the inputs are valid and all outputs can be stored. Gaut generates scripts used to simulate the result using cycle accurate and bit accurate (CABA) simulation and TLM with timing (TLM-T).

Ocapi [99, 64] is another tool that uses the class libraries mechanism. It supports simulation, verification, and synthesis. The object-oriented features of the C++ language allow the mix between high-level description of the components and a detailed cycle-accurate simulation. This approach also reduces considerably the simulation time compared to a RTL-level simulation.

A very popular framework that uses the class libraries methodology is `SystemC`. It has as origin the Scenic project from Synopsys [72] designed to create a language based on C++ that would allow RTL synthesis and would be able to generate executable modules of hardware system. SystemC is used mostly for hardware simulation. However, recent compilers such as Forte's `Cynthesizer` can perform hardware synthesis from `SystemC`. It provides `C++` high-level constructs as classes and macros, and also low-level ones such as structural hierarchy, delta cycles [3], 4-state logic, virtual platform modeling. The designer can simulate concurrent processes using `C++` syntax [14]. `Forte's Cynthesizer`, one of the HLS tools that use SystemC, is a high-level design product of Forte Design Systems [6]. It accepts SystemC-transaction-level modeling (TLM) design description as input and generates an RTL design. The user can use high-level C++ constructs like encapsulation, construction of custom data types, templates, etc. The tool also supports a few directives used to control the loop pipelining and unrolling.

One can also mention `C-to-Silicon`, the commercial HLS tool from Cadence, which uses input specifications based on class libraries. It accepts C, C++, and SystemC as input specification. The compiler synthesizes the design up to the physical level in order to generate accurate timings. The tool also incorporates a verification module. The design is translated into a so-called fast hardware model (FHM) that runs almost as fast as an untimed C-model [4].

Finally, the commercial tool `Synphony C Compiler` from Synopsys [12] is also based on class

2. GNU Compiler Collection (GCC)

3. Hardware simulation is performed using delta cycles. At a specific delta cycle are executed only hardware instances for which some signals have changed their state at the previous delta cycle.

libraries. It uses C/C++ constructs for synthesis. The tool has the key capability of automatically inserting, in the hardware design, clock gating in order to save power.

All HLS tools we just mentioned use class libraries in their input specifications. These class libraries are used to extend the original language specification. This has multiple advantages. One of the major advantage is the very fast simulation time when the required level of simulation does not need to be as accurate as VHDL simulation. Another advantage of these class libraries is the very easy hardware/software partitioning as well as a very rapid and painless co-simulation and co-verification. Unfortunately, all this comes at a cost. Because of the required use of these class libraries, the majority of the code is not directly usable. The user has to pass through painful transformation phases from its original specification into one that uses these class libraries.

Synthesis on a predefined template architecture Another methodology to guide the HLS tool in order to obtain a desired result is by imposing the HLS tool to generate a hardware acceleratorq based on a template architecture.

One tool that uses this approach, as opposed to classical synthesis approaches where the synthesizer decides the resource usage, placement, and schedule, is the User-Guided High-level synthesis framework (`UGH`) [28, 47]. `UGH` synthesizes the input C description on the draft data-path (DDP) specified by the user and it generates a co-processing unit. The DDP consists of a simplified structural hardware description of the target architecture. It is represented as a directed graph. The nodes of this graph represents operators and the arcs connecting them represent the allowed data flow between them. This allows a fine control by the user on the generated hardware. Along with this, the user should provide the timing constraints file, which specifies, among others, the desired target frequency. `UGH` supports several synthesis directives, for example to ask the tool to hardwire an `if` operator rather than adding it to the state machine, and the possibility to give only a partially-connected DDP. When an operation cannot be mapped into this DDP, the tool will automatically extend it. The tool does not accept pointers, recursive functions, and the use of functions from the standard C library such as `printf`. All the variables must be static or global. They should be mapped to memory elements (registers or RAMs). The I/Os are performed by means of FIFOs that are viewed, in the C code, as input and output streams, read and written using special functions.

`Pico` is a system that automatically synthesizes non-programmable accelerators (NPA)s to be used as co-processors for functions expressed as loop nests in C [100]. The `Pico` project was a research effort of the HP Labs aimed at automating the design of customized, optimized, general and special-purpose processors. `Pico` performs a series of operations to obtain a NPA architecture from a perfect loop nest written in C. The source language has some extensions, in the form of pragmas (to declare nonstandard data widths, to indicate that certain global variables are not live in or live-out). The generated architecture represents sequential processes in the form of very large instruction word (VLIW) processors that communicate using FIFOs in the parallel execution model of Kahn process networks [81]. Each VLIW processor executes a statically-parallelized block from a loop that is software pipelined. The parallelization and software pipelining are quite powerful thanks to the use of the Omega system that computes data dependences accurately and the use of the Trimaran and Suif compiler infrastructures. After its first developments at HP Labs, `Pico` was then developed as a commercial product (`Pico-Express`) by the company Synfora, recently bought by Synopsis. Note: `Pico` could also be classified in the category of tools based on compiler directives: we chose to mention it here because its generated accelerators, based on Kahn processes and VLIW architecture, impose some rigid design constraints.

One of the advantages of synthesizing using a template architecture is that the user can have an image of the result before synthesis. However, synthesis using such tools requires a good hardware knowledge from the designer. For `UGH` for example, the user has to define an adequate draft datapath in order to obtain good synthesis results. For `Pico`, the code has to be transformed into a form suitable for the HLS tool and its target architecture.

2.2.4 Other HLS tools for reconfigurable architectures

In this section, we present some other HLS tools that use a C-like language as a synthesis input and that generate an IP for reconfigurable fabrics [4]. They usually have some low-level constructs or compiler directives that are special to these systems.

One such tools is `Transmogrifier C` mentioned earlier. It has special low-level hardware constructs in the language to access the pins of the specific Xilinx XC4000 series FPGAs. Another tool that synthesizes for reconfigurable platforms is `PRISM-II` [106]. It is the successor of the `PRISM-I` system. It consists of a configuration compiler and a reconfigurable hardware platform. The compiler transforms the C language description into a hardware module that is mapped into Xilinx FPGAs. The software part is similar to the executable part produced by compilers but also includes support for accessing the hardware module during program execution. The input specification is a subset of the C language and includes `for`, `while`, and `switch` statements but does not include structures, floating point data, or unions. GCC is used as a pre-processor that can apply multiple code transformations such as dead-code elimination, sub-expression elimination, etc. The internal representation of the GCC compiler, called `RTL`, together with the machine description file, is used to generate hardware.

The `CASH` or Application-Specific Hardware (ASH) compiler [38] is an automated hardware synthesis tool that generates hardware from programs written in C. It is based on the SUIF research compiler [108]. Its approach consists of synthesizing at compile time a hardware accelerator and executing it on a reconfigurable fabric. The compiler exploits dynamic scheduling, speculation, and instruction level parallelism techniques. Each synthesized function gets computation structures, interconnection links, and local memories. It is generated into a separate circuit and independently optimized. Later on, these functions are synchronized together by asynchronous communication protocols, in a coarse-grain manner. As `Spark`, and other tools of the same category, `CASH` does not accept any user intervention by means of compiler directives or low-level hardware constructs.

Another HLS tool used in reconfigurable computing is the `GARP` compiler [40]. The compiler accepts ANSI-C programs, it automatically extracts computation-intensive loops, and it compiles them into a hardware architecture for dynamically-reconfigurable coprocessors inside a special reconfigurable chip called GARP.

There are many other HLS tools [5] used in reconfigurable computing that use a C-like language as an input specification [41], which demonstrates that this type of specification is very common and popular, mostly because of its similarities with the ANSI-C software programming language. Another advantages of using a C language for synthesis is the very easy hardware/software partitioning, co-simulation, and verification. This is why it is very often used in the HW/SW co-design

4. A reconfigurable fabric represents a digital circuit that can be reconfigured statically or dynamically with a different hardware design. One example of a reconfigurable system but not limited to is the FPGA.

5. For example, `SPC Compiler`, `A C to Fine-Grained Pipelining Compiler`, `DeepC`, `Stream-C`, `ROCCC`, `RaPiD-C`, `XPP-VC`, `DRESC`, `Chimaera-C`, `GARPCC`, `NAPA-C`, `DEFACTO`, `Cameron`, `DWARV`, `CoDe-X`, `COBRA-ABS` [41].

domain. Tools like AKKA [103], PRISM-II, and others, use this to partition, estimate, profile, co-simulate, and co-synthesize.

Other languages and tools We point out that there are other interesting languages used as input specification to a synthesis system. Here, we mostly focused on C-like languages. For example, languages such as Matlab have a higher level of abstraction than C. One could translate the specification in this language into a C one. For a language such as Alpha [39], the process is inverse. One can transform a C representation into Alpha by using array expansion techniques and dynamic static single assignment form (SSA) as described in [67].

The Alpha language was proposed by Christophe Mauras during his thesis in 1989 [92]. Alpha is a functional language (based on recurrence equations) for expressing regular algorithms, synthesizing regular architectures, or compiling to sequential or parallel machines from a high-level specification, thanks to its corresponding HLS tool MMAlpha[6]. An algorithm is described by equations involving variables defined on multi-dimensional domains. By successive transformations (uniformization, parallelization for instance), the description is refined until it may be interpreted as an architecture. Then, this description can be translated towards logic synthesis tools in order to generate a very-large-scale integration (VLSI) architecture. Alternatively, different analysis (scheduling, lifetime, etc.) may guide the transformations towards an imperative loop code for general-purpose (sequential or parallel) processors [107].

2.2.5 Conclusions

In the previous sections, we presented different HLS methodologies and languages. We started with a presentation of traditional hardware development methodologies and languages such as Verilog and VHDL. We showed that using such languages to implement complex hardware designs is not very efficient: such designs are difficult to write and maintain, and minor specification changes require a major redesign and rewriting. Another disadvantage of these languages is that they are not familiar to software developers hence limiting their use to only hardware designers. However, software developers are required to forward the algorithmic specification to hardware designers. Since the abstraction level of these hardware description languages is very low, the hardware designers have to pass through the tedious process of transformation from an algorithmic description into a hardware design.

In order to circumvent this effort, we showed how multiple HLS tools appeared that synthesize hardware from a higher level of abstraction. There are multiple high-level languages used in synthesis, but the most used is C and variants of it. Its advantages over other high-level languages is that it is very popular and well-known in the world of both software and hardware designers (just like, as previously described, Verilog has more users than VHDL). Designing hardware using C is much less bug-prone. The debugging and testing process, which takes a major part of a hardware design, is reduced considerably because one can find faster bugs in a higher abstraction level and also because the simulation times are much shorter than for the low-level ones.

When using a high-level language such as C, one can easily apply various high-level code transformations as opposed to low-level languages where it is difficult or even impossible to apply them. One can adapt many powerful transformations from high-performance computing (HPC) into HLS. However, inserting all these transformations into the HLS tool can be very difficult since one should

6. See also Chapter 3 where we define and design an interface for a hardware accelerator generated by MMAlpha.

adapt the IR of the tool to this transformation or vice versa. Another way of doing it is by passing the input code through a preprocessing tool before synthesis. This way, we separate the preprocessing tool from the HLS tool, thereby facilitating the reuse of implemented knowledge using an adaptation of the connection to other HLS tools. This way, the preprocessing tool is perpetuated and will not disappear if the HLS tool disappears.

Also, as mentioned earlier, HLS tools usually require a more or less important adaptation of the input specification so as to match the input specification restrictions of the tool. Performing correctly this adaptation is not trivial as one should perform it with performance metrics in mind, which are defined by the HLS tool documentation. Implementing them (semi-) automatically in the preprocessing tool will ease the task of the designer which can now be a software developer, with only a general idea of performance metrics of the code transformations, that wants to obtain quickly good synthesis results. This is the approach we explore in this thesis: performing high-level program transformations semi-automatically, in front of the HLS tool, trying to use the synthesis tool as a black box as much as possible, or at least as a back-end compiler that should take care of lower-level details that the designer does not want to take care of.

In the next section, we give a catalog of program transformations, in particular loop transformations, well-known in the HPC context, together with their interest in the context of HLS.

2.3 Code transformations for high level synthesis

Compiler technology is clearly penetrating deeper and deeper the HLS world in order to improve the synthesis results. Compilers integrate many complex optimizations, usually tuned to get efficient assembly code, but they can also be used for HLS. Examples of such optimizations are common subexpression elimination, dead code elimination, strength reduction, to quote but a few [45, 93]. In this section, we are interested in higher-level optimizations, in particular loop transformations [29], where parallelism extraction and memory optimizations can be better achieved. Some of these transformations are already implemented in recent state-of-the-art HLS tools such as `pico`, `C2H`, and others. However, the transformations are performed mostly at a lower-level of abstraction and on the complex internal data structures of these tools. Our goal is to facilitate the design cycle by adding a higher-level preprocessing step, which would be human readable and thus easily controlled by the designer. The preprocessed specification would then be synthesized by most HLS tools that take as input specification an C-like program.

General affine loop transformations were first implemented in parallelizing prototypes such as tiny [109], LooPo [75], Suif [108], or Pips [80] to try to exploit loop parallelism. Recently, dedicated loop optimization modules have been integrated into more popular open-source compilers [96], also because cache performances can be greatly improved by such transformations. Our goal is to use these tools or, at least, similar transformations, to provide a source-to-source (S2S) front-end to HLS tools so as to widen the space of possible hardware implementations given a particular initial sequential specification, thereby eliminating the need to reimplement all these transformations internally in the HLS tools.

For applying loop transformations, we will use the standard polyhedral model, a modeling technique for loop nests in programs. It abstracts a n-dimensional loop nest (i.e., n nested loops) by a polyhedron on a n-dimensional space enclosing all the integer vectors spanned by the vector of indices of the loop nest. It uses a classical internal representation for the instructions of the body of the loop nest. Performing loop transformations amounts to perform algebraic transformations

on these polyhedra. This polyhedral representation has also been successfully used to model other objects such as the memory layout of a program [42, 55], the communication volume between IPs of a SOC [104], cache misses [89], etc.

The polyhedral representation has the advantage that its size is independent of the number of iterations of the loop, which is particularly useful to manipulate loops unlike some HLS tools that need to unroll loops to exploit parallelism. It can also represent loops with a parameterized number of iterations (i.e., where the number of iterations is not known at compile time). It has been used in research prototypes such as MMAlpha, Compaan, or LooPo. However, its use implies a shift towards an IR quite different from the IR commonly used in compilers (abstract syntax trees or linear IR). Another restriction is that the polyhedral model is efficient to model *static control programs*, i.e., programs where the control flow is not dependent of the input data. It is also even often assumed that data dependences do not depend on input data either. As shown in [105], a major part of programs is composed of static control parts, especially for compute-intensive programs present in signal processing or multimedia applications. However, it is mandatory for a HLS tool to also handle the parts of the program that do not have static control.

Independently of our work, in particular the one presented in Chapter 3 and in [95], a study by Devos has been published [60] that uses, as we do, `WRaP-IT` as a loop transformation tool too. The HLS was done with an elementary (i.e., no complex back-end optimizations) homemade synthesis tool (`CloogVHDL`) and results compared to designs done with ImpulseC, starting from an initial unoptimized specification. The goal of this work was to show that loop transformations can indeed lead to performance improvements for HLS and how these transformations can be selected. Actually, many HLS tools already adopted compiler technology, including loop transformations, and they already proved the importance of them, even if most of them only implemented a small subset of compiler transformations. Therefore, if our first goal was similar to the work of [60], i.e., showing the importance of high-level loop transformations for HLS, we wanted to complete this first study with a different methodology so as a) to better analyze the interactions between high-level and low-level tools and optimizations, b) to show that such transformations can be made fully at source level, in front of a HLS tool that cannot be modified. Implementing a new compiler transformation indeed requires a very good knowledge of the internals of these tools. Even worse, it is impossible to add one to a HLS tool that is provided only in binary format such as `Spark`, `Pico`, and most of the proprietary HLS tools. One of the goals of our study is thus to show that one can split the HLS process into front-end optimizations (as a pre-processing step) and low-level optimizations (in the HLS tool).

The following two sections present, in the form of a catalog, classical loop transformations in code optimization and parallelization, that were first developed for HPC and that are potentially (or already proved) interesting for HLS. Our study in the next chapters will dive in more details, and with more experimental evidence, into their interest in the context of HLS. Some more details can be found in [41] too. These source-level transformations can be classified into two broad categories: those affecting the control flow, i.e., the order of computations, and those changing the organization in the memory and its accesses. The two types of transformations, one acting on the structure of loops, the other on the structure of arrays, are not independent. Usually one should apply a transformation of the first type for a transformation of the second type to have an interesting effect. We can find this duality in data-parallel programming languages such as High Performance Fortran (HPF) [83, 94], a language that focused research efforts for many years for the semi-automatic optimization of applications for distributed-memory machines. This language

has popularized the concepts of parallel loop, of scalar and array privatization, array alignment, block or cyclic distribution, communication vectorization, data and computation duplication to avoid transfers, etc. We find similar needs and solutions for the synthesis of hardware accelerators.

2.3.1 Transformations that change the computation order

Software pipelining Software pipelining consists in organizing the instructions of a loop by taking into account its cyclic execution, the latency of operations, and the computational/transfer resources available. To an iteration i of the loop and instruction S of the loop body corresponds a particular operation $S(i)$. These different operations are organized in a regular way, by the well-known modulo scheduling method [98], where $S(i)$ is initiated at time $\lambda i + \rho(i)$. In general, all operations $S(i)$ corresponding to the same instruction S are mapped to the same computational resource (cyclic allocation). The coefficient λ is called the initiation interval (II): a new iteration of the loop starts every λ cycles, and one occurrence of S is initiated every λ cycles. In the Altera HLS tool `C2H`, the initiation interval is called CPLI, for cycles per loop iteration, see Chapter 4.

In compilation, λ can be arbitrarily large. It depends on the resource constraints and on the dependences between operations. In synthesis, on the other hand, most designs target $\lambda = 1$, by allocating all necessary resources and by applying preliminary program transformations.

The tool `PicoExpress` from Synfora has a very good software pipeliner. The software pipeliner of Altera's C2H is interesting, mainly to hide latencies during the transfer from external memory. However, it is rather rudimentary and quickly becomes sub-optimal.

Loop unrolling Loop unrolling is a well-known transformation used to replace a set of operations, described by loop iterations, to the explicit sequence of these operations. Thus:

```
for (i = 0; i < 4; i++) {
   a [i] = ...
}
```

becomes:

```
a[0] = ...
a[1] = ...
a[2] = ...
a[3] = ...
```

This transformation is particularly useful for HLS tools that do not have a scheduler or optimizer that can exploit loops. It is possible only if the number of iterations is known (i.e., not a parameter). A partial unrolling by a factor r is also possible. Usually r is not a parameter. For example:

```
for (i = 0; i < N; i++) {
   a[i] = ...
}
```

unrolled by $r = 2$ becomes:

```
for (i=0; i<N; i+=2) {
   a[i] = ...
   if (i<N-1)
```

```
      a[i+1] = ...
}
```

It is possible to write the code differently, in particular in to avoid a conditional statement inside the loop. Some information on the value N can be also useful in order to simplify the code (here for example that N is even for example). Partial unrolling can be useful to improve the performance of software pipelining. In Chapter 4, we will use an unrolling by a factor of 2 to express a form of double buffering and, this way, overlap communications and computations.

Loop fusion Fusing (or merging) two loops consist of merging the control structure of the two loops to form only one. The inverse transformation of the loop fusion is called loop distribution. Thus:

```
for (i=0; i<n; i++) {
  a[i] = ...;
}
for (i=0; i<n; i++) {
  b[i] = a[i];
}
```

is transformed by loop fusion into:

```
for (i=0; i<n; i++) {
  a[i] = ...;
  b[i] = a[i];
}
```

The order of computations after loop fusion is changed. The resulting code performs an alternate execution of statements of the two initial loops rather than executing all statements of the first before those of the second. Loop fusion has several advantages:

- increasing the granularity of the loop body in order to provide more parallelism to the instruction scheduler;
- moving the computations within a single hardware accelerator if each loop corresponds to a unique hardware accelerator (some synthesis tools treats each loop separately);
- reducing the lifetimes of some variables allowing a better memory reuse (in the example above, we may contract the array into one scalar variable).

We will use loop fusion for all these reasons in Chapter 3 for our study with `Spark`. The negative effects of the fusion are the possible increased register pressure and the possible destruction of loop-level parallelism (in particular, the fusion of two parallel loops can lead to a sequential one, leading to NP-complete optimization problems [49]).

The legality of loop fusion is well known. The main difficulties of applying this transformation are: a) when the code generation does not accept loops whose bounds are different (we need to apply for example loop peeling before this transformation); b) the distance (in the program) between the loops; c) the absence of a clear cost model; d) the difficult interaction with other transformations (especially loop interchange and loop shifting to make the fusion legal).

Loop peeling Loop peeling [27] consists of removing a loop iteration (usually the first or the last one) and executing it out of the loop body without changing the order of computations. Thus, the peeling of the first iteration of:

```
for (i=0; i<n; i++) {
   a[i] = ...;
}
```

leads to:

```
a[0] = ...;
for (i=1; i<n; i++) {
  a[i] = ...;
}
```

In general, this transformation is used to extract from a loop an iteration having a particular behavior or to enable further transformations. It can be useful also to simplify the control of a program by cutting the iterations into subsets for which the control is similar.

The reverse operation called sinking is used to insert a statement, by adding conditional guards, in a loop defined at the same depth. This method is used to obtain perfectly-nested loops.

Loop shifting/retiming Sometimes, in order to change the distance and potentially the nature (loop-carried, loop-independent) of certain data dependences in the loops, it may be useful to shift (relative to the loop counter) some loop-body instructions. This technique is called loop shifting (even if this transformation should be called instructions shifting in a loop). For example, the two following loops cannot be fused as is:

```
for (i=0; i<N; i++) {
  a[i] = ...
}
for (i=0, i<N, i++) {
  b[i] = a[i+1]
}
```

but by shifting the second loop:

```
for (i=0; i<N; i++) {
  a[i] = ...
}
for (i=1; i<N+1; i++) {
  b[i-1] = a[i]
}
```

the two loops can be merged by peeling a few iterations:

```
a[0] = ...
for (i=1; i<N; i++) {
  a[i] = ...
  b[i-1] = a[i]
}
b[N-1] = a[N]
```

Loop shifting is similar, from an algorithmic point of view, to the retiming concept [86] that is well known in low-level hardware synthesis. It has also a strong link with software pipelining and loop compaction [50]. It is also sometimes used in vector form when several loops are nested. It is then called multi-dimensional shifting [51].

Loop interchange Loop interchange (not to mix up with loop inversion) is a classical code transformation [27]. It consists of exchanging the levels of two perfectly-nested successive loops. The two-nested loops:

```
for (i=0; i<N; i++) {
  for (j=i; j<M; j++) {
    a[i][j] = ...
  }
}
```

become:

```
for (j=0; j<M; i++) {
  for (i=0; i<min(j+1,N); j++) {
    a[i][j] = ...
  }
}
```

This transformation changes the order of computations. Therefore, it changes the data dependences as well as array accesses. For example, here, the first code accesses the array `a` following the direction of rows, and the second accesses it following the direction of columns. The main difficulty resides in the computation of loop bounds after the transformation. Techniques manipulating polyhedra allow to accomplish this computation by considering that a loop interchange represents an affine transformation, here $(i, j) \rightarrow (j, i)$, applied on a polyhedron (the iterations of both nested loops).

Loop skewing Loop skewing [27] is an intermediate transformation with little interest when applied by itself because it does not change the computation order. It consists of performing an affine transformation $(i, j) \rightarrow (i, j + i)$. Let us note however that, by definition, a skewing can be only the addition of an outer loop iterator to an inner loop iterator. In other words, skewing is always valid. The transformation $(i, j) \rightarrow (i + j, j)$ is not a skewing but a combination $(i, j) \rightarrow (j, i) \rightarrow (j, i + j) \rightarrow (i + j, j)$ of loop skewing with two loop interchange transformations.

Loop reversal The inversion of loops [27] (not to mix up with loop interchange) consists in reversing the loop counter, i.e., decrementing it if it was incremented and reciprocally. For example:

```
for (i=0; i<N; i++) {
  a[i] = ...
}
```

becomes:

```
for (i=N-1; i>-1; i--) {
  a[i] = ...
}
```

Strip-mining Strip-mining [27] is similar to partial unrolling, except the fact that the unrolled part is itself expressed by a loop. So:

```
for (i=0; i<N; i++) {
  a[i] = ...
}
```

becomes, after applying a strip-mining of s:

```
for (i=0; i<N; i+=s) {
  for (ii=i, ii<min(i+s,N), ii++) {
    a[ii] = ...
  }
}
```

It can be written in different, semantically-equivalent, forms. The interest of strip-mining is to identify a portion of iterations or a data block on which other transformations can be applied. When combined with loop interchange, loop tiling is obtained.

The opposite of strip-mining transforms a code that have two loops to one having only one and is called loop linearization. For example:

```
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {
    a[i][j] = ...
  }
}
```

is transformed into:

```
for (t=0; t<MN; t++) {
   a[t div N][t mod N] = ...
}
```

This transformation is sometimes useful when HLS tools synthesize better simple loops in comparison to nested loops. Of course, integer divisions and modulo can be avoided by re-computing variables i and j using increments by a small FSM. This is similar to the juggling method [54] used in the Pico project of HP Labs (and possibly in the tool PicoExpress from Synfora). An alternate code generation is to use the method of Boulet and Feautrier [37] as we do in Chapter 5.

Unroll-and-jam The unroll-and-jam transformation [27] acts on two perfectly-nested loops and consists in partially unrolling the outer loop and to merge the resulting loops. For example, the code:

```
for (i=0; i<2*N; i++) {
  for (j=0; j<M; i++) {
    a[i][j] = ...
  }
}
```

is transformed, by an unroll-and-jam of factor 2, into:

```
for (i=0; i<2*N; i+=2) {
  for (j=0; j<M; i++) {
    a[i][j] = ...
    a[i+1][j] = ...
  }
}
```

The interest of the unroll-and-jam is to be able to unroll loops which do not appear at the deepest level, while keeping the nesting order. This transformation changes the order of computations and also the order of array accesses. This can enable classic optimizations on memory. This transformation is in fact a particular form of "unrolled" tiling.

Tiling Tiling [110] is one of the most important transformations used to improve performances. It consists in transforming n perfectly-nested loops into $2n$ loops. It consists in performing a strip-mining on each of the n loops and to be move the n outer loops of strip-mining before the n inner loops of strip-mining. In this example:

```
for (i=0; i <N; i ++) {
  for (j=0; j <M; i++) {
    a[i][j] = ...
  }
}
```

Using tiles of size $s \times s$, we obtain:

```
for (i=0; i<N; i+=s) {
  for (j=0; j<N; j+=s) {
    for (ii=i; ii<min(i+s,N); ii++) {
      for (jj=j; jj<min(j+s,N); jj++) {
        a[ii][jj] = ...
      }
    }
  }
}
```

It is possible to write it in different forms than the one presented above. Tiling is particularly useful to perform computations by block, with a coarser grain of computation than at the instruction level. The transformed code usually has a better space and temporal data locality. It can also be used to vectorize communications, i.e., to perform transfers by blocks (burst communications). The legality conditions of the tiling transformation are well known but require dependence analysis and the search for permutable loops [56]. Code rewriting methods are not simple, because tiling is not an affine transformation, and many techniques have been proposed. But tiling has generated, above all, many papers on cost models.

Unimodular transformations A unimodular transformation [31] operates on n perfectly-nested loops. The counters of these n loops are represented as a vector (from the most external loop towards the most internal one). This vector is transformed, by a linear transformation invertible

in the integers, in a new vector that describes new counters and their corresponding loops (from the most external loop towards the most internal one). Such a transformation is characterized by a transformation matrix with integer coefficients and a determinant equal to ± 1. Loop reversal, loop interchange, and loop skewing are unimodular transformations. Also, any unimodular transformation can be expressed as a combination of these elementary transformations.

In general, the user does not think in terms of unimodular transformations. These are more a formalism allowing to think globally of elementary transformations and their composition, to prove their legality, and to develop algorithms that generate the loop bounds of the transformed loops.

Affine transformations Affine transformations are an additional stage in the formalization of elementary transformations compositions. By adding the possibility of having a different constant for each statement, they include into the model the possibility of shifting. By adding the possibility of different linear part for each statement, in the original space as in the transformed space, they allow to deal with loops that are not necessarily perfectly nested and to generate loops that have any type of nesting.

Algorithms manipulating such transformations exist, for example, the multidimensional affine scheduling algorithm of Feautrier [68], the approach of Lim and Lam [88], or the method of the group of Sadayappan and Ram [34], who developed the tool called `Pluto`. Code rewriting techniques also exist, as implemented in the tool `CLOOG`.

Loop parallelization Loop parallelization consists of annotating loops as sequential or parallel. A loop is parallel if the computation operations at iteration i do not depend on the computation operations at iteration j, where $i \neq j$. There may be dependences in the code, but they are between different instructions of the same iteration (loop-independent dependence) and not between different iterations (loop-carried dependence). The iterations of a parallel loop can be performed, not only in parallel, but in any order. For example, in HPF, such a loop is specified by the pragma "independent". It should not be mixed up with the HPF `FORALL` whose semantics is different.

Different algorithms used for loop parallelization exist. One of them is the algorithm of Allen and Kennedy [26] based on loop distribution. The most powerful to date, for the maximal extraction of parallel loops, is the algorithm of Feautrier [68], which builds general affine transformations. For a more complete list of parallelization techniques based on scheduling, see [53].

Analysis and transformations of `while` loops The previous transformations concern the `for` loops in C, when they can be proved to behave as `DO` loops in FORTRAN, i.e., with an obvious counter modified only in the loop header, with loop bounds not modified in the loop body, etc.

Such "static" `for` loops are better optimized by HLS tools, one of the reason being that they can anticipate their execution time. However, even in the HLS context, there are codes which contain `while`-type loops. Some of the HLS tools accept them and generate a corresponding control FSM. But, usually, this FSM represents a simple translation of this control structure without any particular optimization, neither pipeline, nor parallelism. Some other HLS tools can transform these loops into `for`-type loops with early exits and are able, in some cases, to pipeline them. Finally, some other tools simply reject them. Besides that, for the front stages of HLS tools (those that decide which transformations to apply or which scheduling to perform), even if the loop `while` can be treated, it is sometimes necessary to know an upper bound for its number of iterations (i.e.,

some kind of worst-case execution time (WCET)). For example, when several hardware accelerators need to be pipelined at coarse-grain level, it is important to have an idea of their latencies.

The analysis of `while` loops, the computation of upper bounds of their "iteration domain", and, possibly, the re-generation of these loops in the form of `DO` loops with early exits is an interesting add-on step for the HLS tools. Techniques such as those developed in [24] can be used.

2.3.2 Transformations that change the memory access order and its size

In the context of HLS, the interest of changing the memory accesses is triple:
- to change the data dependences of a program to extract more parallelism;
- to reuse memory so as to reduce its size;
- to reorganize the memory in order to change the total number of available parallel ports and connections to computation resources.

Scalar privatization/expansion The scalar expansion [27] consists in transforming a scalar variable, redefined in every loop iteration, into an array indexed by the loop counter. In this case, anti-dependences (writing after reading) and output dependences (writing after writing) are removed, increasing the potential for parallelism. For example:

```
for (i=0; i<N; i++) {
  t = a[i];
  a[i] = b[i];
  b[i] = t;
}
```

is transformed into:

```
for (i=0; i<N; i++) {
  t[i] = a[i];
  a[i] = b[i];
  b[i] = t[i];
}
t = t[N-1]; /* if t is used later */
```

Thanks to this memory expansion, the loop becomes parallel. Its iterations can be performed in any order. It is possible to not expand the scalar variable but only point out that, if the iterations are executed on p parallel processors, a separate scalar variable for each processor should be used. In this case, only p memory locations are needed instead of N. In this case, it is called scalar privatization. It is also possible to apply the same principle to an array by adding dimensions explicitly (array expansion) or implicitly (array privatization).

Single assignment The translation into single assignment form is a more general form of scalar and array expansion [65]. To eliminate all the dependences that are not due to the data flow (flow dependences) but to the memory reuse (anti-dependences and output dependences), thanks to an exact data flow analysis [67], the code is fully rewritten. Every computation operation now writes in a dedicated memory cell.

This strategy "explodes" the memory explicitly to express a maximum of parallelism. Memory folding techniques (i.e., memory reuse) are then necessary to avoid to obtain a code that uses too much memory.

Array unrolling In some HLS tools, the user has to specify the memory organization. Usually, it is based on the principle of equivalence between an array and an actual storage: two disjoint arrays having different names will be mapped to different memories and hence to different ports. Therefore, it will be possible to access them in parallel.

In order to increase the parallelism of some loops, it may be necessary to unroll, not only the code, but also the arrays in memory. For example, the following code:

```
for (i=0; i<3*N; i++) {
  a[i] = ...
}
```

cannot be efficiently pipelined if memory can sustain only one data per loop cycle, due to its limited ports. Even if the code is partly unrolled as follows:

```
for (i=0; i<3*N; i+=3) {
  a[i] = ...
  a[i+1] = ...
  a[i+2] = ...
}
```

the problem persists as the three statements access the same memory port. It is then necessary to write it as follows:

```
for (i=0; i<3*N; i+=3) {
  a_0[i] = ...
  a_1[i] = ...
  a_2[i] = ...
}
```

This requires to be able to analyze the constant terms 0, 1, and 2 of the array accesses in order to avoid an expensive `case` type structure. This transformation is therefore naturally adapted to loop unrolling as well as unroll-and-jam.

If the tool is able of treating in particular way rows or columns of an array when parallel memory banks are available, it would be possible to write it this way:

```
for (i=0; i<3*N; i+=3) {
   a[i][0] = ... /* or a[0][i] depending what tool does */
   a[i][1] = ... /* or a[1][i] */
   a[i][2] = ... /* or a[2][i] */
}
```

This would avoid a sometimes difficult array index analysis or the use of `case` statements.

Array padding Array padding consists in adding "ghost" dimensions or to increase artificially the sizes of arrays. Depending on the allocation strategy of the code generator, it is used to reposition differently an array in the memory. For classical compilation, this transformation is used to better exploit the cache and also to resolve the alignment problems in some instructions (single instruction, multiple data (SIMD) notably). In HLS, depending on the tool, it can have an effect on possible parallel accesses of some data.

Array linearization and strip-mining The (canonical) linearization of an array consists in accessing it explicitly in memory using successive addresses. In C, if a is a bi-dimensional array of size $N \times M$, we can replace the access `a[i][j]` by `b[Mi+j]` where b is a one-dimensional array of size $N \times M$. This does not change the access sequences. On the other hand, it is also possible to re-organize the memory with any other linearization function, for example as $(i, j) \rightarrow N * j + i$.

As well as for the geometric loop transformations presented earlier, it is possible to apply array reallocation transformations in order to change the way they are accessed. On the other hand, unlike the transformations of loops, space transformations are always valid as long as they do not assign two memory locations of the initial array into one in the final array. The simplest and undoubtedly the most useful transformations are the linearization, its inverse the strip-mining, and any form of dimension exchange so as to define blocks (tiling). Alignment transformations (equivalent to loop shifting) are also possible. These have similarities with the alignment and distribution directives (block, cyclic, block-cyclic) of the HPF language.

Array contraction The last important transformation is the inverse transformation of scalar and array expansion. The array contraction allows to transform an array into a scalar variable (or to remove one dimension of the array) if each of the location to be transformed is read before another one (which will be finally mapped at the same place) is written. It is often in association with loop shifting and loop fusion that this contraction is possible [52]. For example, the following code has one explicit temporary array:

```
for (i=0; i<N; i++) {
  t[i] = a[i];
}
for (i=0; i<N; i++) {
  a[i] = b[i];
}
for (i=0; i<N; i++) {
  b[i] = t[i];
}
```

If the array t is not used later and the objective is to minimize the memory utilization, the code can be optimized into:

```
for (i=0; i<N; i++) {
  tmp = a[i];
  a[i] = b[i];
  b[i] = tmp;
}
```

Nevertheless, it is not always possible to fold the array into a scalar or to delete a dimension entirely. Sometimes it is necessary to reuse memory, notably if the code was previously transformed in the single assignment form [85]. One solution is to fold the memory not just with projections, but also using the modulo operation, combined with linearization or unimodular transformations. This corresponds to the memory reduction technique based on admissible lattices developed in [55]. For HLS, interesting folding are those obtained by simple dimensions exchange and/or linearization, with modulos restricted to powers of 2. This technique will be used in Chapter 5 to reduce the memory size needed to store communicated data.

Note: many other transformations, not described here, are useful to clean up the code, either before the previously-mentioned transformations so that they can now be applied, or after them to generate code that optimizes hardware resource utilization.

2.4 Conclusions

In this chapter, we presented different languages, tools, and code transformations used in HLS. We showed that traditional hardware development languages are not efficient in the world of big and complex circuit designs. We therefore presented multiple HLS tools that ease the task of hardware design by raising the abstraction level of the input languages. As can be deduced, most of the tools use a C-like language with some extensions. The use of C reduces the bug-prone tasks of the synthesis process, reduces the debugging and simulation time, and was intended to allow a direct use of the tools by software developers.

Because of the increasing capacity of the FPGAs, the computer community is becoming interested in the possibility of FPGA-based computing machines to accelerate the execution time of different algorithms. Using the C language as the input to the synthesis system greatly facilitates the task of hardware/software partitioning, co-simulation, and verification that takes a significant amount of time of the hardware design process, minimizing significantly the time to market. Also, with a C-like input language, one can use directly the already available utilities for the C language such as compilers and debuggers, and to implement easily simulators or emulators.

Unfortunately, the HLS tools still failed to attract the community of software developers or are still too inefficient or not satisfactory for hardware developers. Most of the HLS tools accept only a subset of the ANSI-C standard. Because of this, the user is still required to transform its specification into one that the tool accepts. Also to obtain good synthesis results, the user is required to insert different low-level constructs, compiler directives, or even to define different hardware configurations.

We propose to automate this design flow by inserting a preprocessing step before synthesis, which would be semi-automatic and at source level. Our study focuses mainly on interface and communication optimizations. Working with the high level of abstraction offered by C, compared to VHDL or Verilog (Verilog), allows us to adapt multiple standard but efficient compiler transformations from HPC that were developed in the 90's and to include powerful polyhedral-based transformations. However, this adaptation also requires the development of new analysis and code optimization, as shown in Chapter 5.

Chapter 3

Tightly-coupled architectures with cache: experiments with MMAlpha & Spark

3.1 Introduction and motivation

Most of the HLS tools described in Chapter 2 synthesize hardware accelerator IPs with a very primitive interface. This implies a difficult integration of the hardware accelerator in a system requiring usually at least some glue logic. This glue logic implements the interface of the accelerator to the rest of the system. It can be based on primitive templates with minor syntactic adjustments to fit the synthesized hardware accelerator to a specific interface in the system. In general, designing such an interface is difficult.

The first disadvantage of such a design flow, based on an additional primitive glue logic, is that it does not take into account the specific communication requirements to the global system in order to obtain a maximum of performance. For example, in a bus system, better performances are obtained when data are transferred in a burst mode, i.e., by packets. Instead, most HLS tools define the performance metrics of the circuit by using only the synthesis results such as timings and resource utilization. However, in most cases, the synthesized hardware accelerator is integrated into a system that includes multiple accelerators, general purpose processors (GPPs), and their corresponding interconnections. Thus, the appropriate performance metric should take into account all the system components and their interactions.

A second disadvantage is that the communication optimization (if any) does not take into account the access pattern of the IP and thus it is not able to apply optimizations based for example on data reuse locality. In other words, when compiling the function that describes the hardware accelerator, it may already be too late to transform the code to optimize communications. It may be necessary to optimize the application at a higher level, i.e., by analyzing and transforming the code that describes both the function to be accelerated and the context in which it is called.

In this chapter, we explore these two issues through the analysis and the use of two academic HLS tools, MMAlpha (Section 3.2) and Spark (Section 3.3). In Section 3.4, we conclude these two studies with an analysis of the encountered problems and a summary of the possible solutions.

Experiments with MMAlpha We first present (Section 3.2) a hand-made solution, for one particular design: a hardware accelerator, generated by the MMAlpha HLS tool, performing a complex matrix-matrix multiplication. The full design of this application, including the interface, was the purpose of the Memocode'07 HW/SW co-design contest. This challenge was a good exercise to explore the interface problems we were interested in. Our manually-designed solution uses double buffering techniques, blocking, and temporal local storage to the MMAlpha IP. We show some experimental results, which prove that designing an interface while understanding the hardware accelerator communication information is very beneficial to the overall real performance of the entire system. Finally, we discuss the problems we encountered, which are related to the "tightly-coupled cache architecture" (see below) that we use.

Experiments with Spark Designing an interface by hand, as we did for MMAlpha and matrix-matrix multiplication, can be error prone and very time consuming. Generating such an interface automatically is difficult too, since the generating tool has to incorporate all the specific details of the hardware accelerator, which requires mostly to work using reverse-engineering. In Section 3.3, we try to avoid this. We propose to move all the "intelligence" part from the interface into the actual hardware accelerator itself. We treat the HLS tool as a black box and hence we cannot access any internals or perform modifications to them. One solution to this problem is to change the HLS result by the only possible way: by changing the input specification, in particular the order in which data arrives. But, of course, the final result of the synthesis must be a hardware accelerator producing the same result for the same input data. We present a source-to-source loop transformation solution used as a preprocessing step to the HLS tool `Spark`. This way, we can change the access pattern of the data to the outside world by respecting the dependence-driven execution of the code.

In order to obtain data reuse, some data has to be kept in a local storage memory. The usual solution is to use a local memory managed as a scratchpad memory. However, we decided to use a better alternative, considering that we design hardware accelerators that are located very close to the processor. In order to save a high-priced static RAM (SRAM) memory and to use it as a scratchpad, we propose to use the processor's cache memory as a local storage. One advantage of using such a memory as the local storage for a hardware accelerator versus a standard SRAM memory is that the cache memory integrates into itself a transparent fetching and storing mechanism of data from the local storage to the rest of the system. We do not consider its price on silicon versus the price of a SRAM since it is already present in a system having a CPU. This advantage can be also viewed as a minor disadvantage when we need very fine control of the data transfers and storage, but usually it is not the case. The coarse-grain control is implemented using the data access pattern. We analyze the impact of each transformation on the memory bandwidth and on the low-level hardware aspects of the generated hardware accelerator.

Tightly-coupled architectures Most of the systems usually contain at least a basic GPP/CPU, which is responsible for executing the parts of the program that are not adequate to be synthesized in hardware. They correspond to standard library calls and to pieces of code that are not cost and space efficient when synthesized. They can be selected using code profiling techniques. Both studies in this chapter, with MMAlpha and Spark, consider an architecture where the hardware accelerator is closely connected to such a CPU. We say such an architecture is "tightly coupled".

Depending on how the hardware accelerator is connected to the rest of the system, different

architectures are possible. The most common one is depicted in Figure 3.1. It is not tightly coupled, but distributed. The hardware accelerator is connected to the rest of the system by a bus or some other interconnect. The hardware accelerator usually has a dedicated local memory with a fast access time. In this memory, intermediate data is stored. The second type of architecture, depicted in Figure 3.2, is (strongly) tightly coupled: the processor has a dedicated connection link to specialized hardware implementing special instructions that are not present in the processor (found for example in Virtex II Pro PowerPC processors). Usually, the instructions are not very complex and there is almost no dedicated memory except for some registers.

In some cases, when the code size is very small, these CPUs could be tightly connected to local fast SRAMs. Most of the time however, this is not the case and the CPUs are connected to an external dynamic RAM (DRAM). To mask the high latency access to this memory, it is usually accessed through a dedicated cache memory, as in Figures 3.1 and 3.2. In a multiprocessing system, the lowest level cache memory (L2 or L3) is shared between several GPPs. Here, since this cache memory is already present in the design, it can also be reused as a local cache memory for a hardware accelerator, as in the architecture of Figure 3.3. No additional SRAM memory is then necessary. Also, as the CPU shares the cache with the hardware accelerator, data coherency between the CPU and the hardware accelerator is easy to guarantee. On the contrary, in the architecture of Figure 3.1, to preserve the data coherency, the CPU has to perform a cache flush if it is working with the same data as the hardware accelerator.

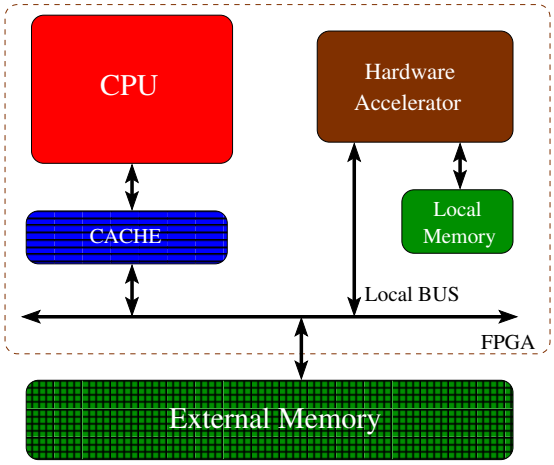


Figure 3.1: Distributed architecture

For the design with MMAlpha, We use the architecture type of Figure 3.2. The CPU is then responsible for bringing data from external memory through the cache, then a memory architecture interface is designed between the CPU and the hardware accelerator. For our study with Spark, we use the architecture of Figure 3.3. The hardware accelerator can then managed the data transfers directly, without occupying the CPU. The CPU is still closely connected to the hardware accelerator, but only through the cache. We say that such an architecture is “weakly tightly coupled”.

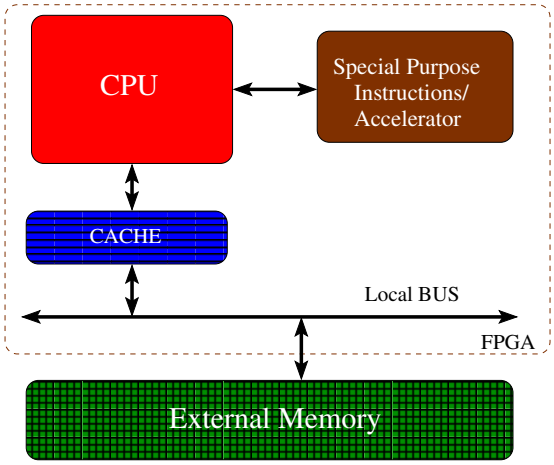


Figure 3.2: Strongly tightly-coupled

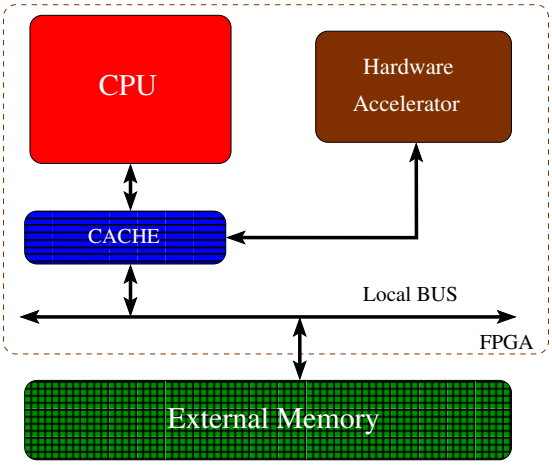


Figure 3.3: Weakly tightly-coupled

3.2 A case study for a matrix-matrix multiplication design generated by MMAlpha

In our general study and search for ways of interfacing automatically-generated hardware accelerators, we found interesting to consider the Memocode'07 HW/SW co-design contest. The challenge was to implement a hardware and software design on a FPGA platform for a matrix-matrix multiplication algorithm. The reference design, implemented in software only, was provided for the Xilinx university program (XUP) development board. The complete source code and copyright is given in the appendix (see Section 7.1.1).

3.2.1 Algorithm description

The algorithm to be implemented, summarized in Figure 3.4, represents the multiplication of two matrices with complex numbers. The elements represent complex numbers and the arithmetic operations performed on them belong to the complex domain. A complex data is stored in a packed format as shown in Figure 3.4(d). Each element consists of a 32 bit integer divided in two 16 bits parts. The lower-most bits store the imaginary part and the uppermost bits the real part. The two parts are represented as a fixed point format with 1 bit for the sign, 1 bit for the integer part, and 14 bits for the fractional part.

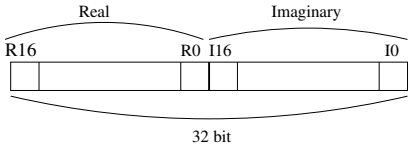
Each complex arithmetic operations on packed elements requires an unpacking, computation, and packing (Figure 3.4(a)). The packing and unpacking (Figure 3.4(b)) require a significant amount of CPU cycles because it is implemented using slow shift instructions available in the instruction set architecture (ISA). Since most of the CPUs (except for the DSPs) do not have fused increment and compare instructions, one complex arithmetic multiplication will take at least N_{multc} cycles. We evaluate this number by taking into account that a CPU, with a reduced instruction set computing (RISC) architecture, does not have a barrel shifter [1] and a shift operation by `k` bits will take:

$$N^k_{shift} = k * (N_{increment} + N_{control} + N_{shift}) = 3 * k \text{ cycles} \tag{3.1}$$

1. a digital circuit that can shift a data word by a specified number of bits in one clock cycle.

```
static inline
Number complexMult(Number A, Number B) {
//  A, B, and return use 2's complement
//  fixed-point format with 1-bit sign,
//  1-bit integer, 14-bit fraction
  signed long Ar=UNPACKR(A);
  signed long Ai=UNPACKI(A);
  signed long Br=UNPACKR(B);
  signed long Bi=UNPACKI(B);
  Number Cr, Ci;
  Cr=Ar*Br-Ai*Bi; Ci=Ar*Bi+Ai*Br;
  Cr=Cr>>(WIDTH-2); Ci=Ci>>(WIDTH-2);
  assert(!OVERFLOW(Cr));
  assert(!OVERFLOW(Ci));
  return PACK(Cr,Ci);
}
```

(a) Multiplication of two packed complex numbers



(d) Complex data pack

```
typedef signed long Number;
#define WIDTH (16)
#define MASK (0xffff)
#define PACK(r,i) (((r)<<WIDTH)|((i)&MASK))
#define UNPACKR(r) ((r)>>WIDTH)
#define UNPACKI(i) (((i)<<WIDTH)>>WIDTH)
#define OVERFLOW(f) (!((((f)&0xffff8000)==0) \
           ||(((f)&0xffff8000)==0xffff8000)))
```

(b) Packing and unpacking

```
typedef signed long Number;
void mmm(Number* A, Number* B, Number* C, int N) {
  int i, j, k;
  for (j = 0; j < N; j++) {
    for (i = 0; i < N; i++) {
      for (k = 0; k < N; k++) {
        C[i*N+j] = complexAdd(C[i*N+j],
              complexMult(A[i*N+k],B[k*N+j]));
      }
    }
  }
}
```

(c) Matrix-matrix complex multiplication, main C code

Figure 3.4: Matrix-matrix complex multiplication

where $N_{increment}$ is the number of cycles required to increment the counter in the shifting loop, $N_{control}$ to verify the termination condition of the loop, and N_{shift} to shift by one.

From Figure 3.4(b) and Equation (3.1), we get the number of clock cycles required to unpack the real part, see Equation (3.2), and the imaginary part, see Equation (3.3).

$$N_{unpackr} = N_{shift}^{WIDTH} = 3 * 16 = 48 \text{ cycles} \tag{3.2}$$

$$N_{unpacki} = N_{shift}^{WIDTH} * 2 = 96 \text{ cycles} \tag{3.3}$$

The total number of CPU cycles required for a complex multiplication is:

$$\begin{aligned} N_{multc} &= 2 * (N_{unpackr} + N_{unpacki}) + 4 * N_{mult} + 2 * N_{add} + 2 * N_{readjust} + N_{packri} \\ &= 2 * (48 + 96) + 4 * 1 + 2 * 1 + 2 * (3 * (16 - 2)) + 17 = 395 \text{ cycles} \end{aligned}$$

As can be observed from Equation (3.2), the complex multiplication of two packed numbers is a very time-consuming operation on a processor. Mostly this is due to the shift operations. In hardware, this constant shift operations are provided at zero cost since they can be implemented using connection shifts.

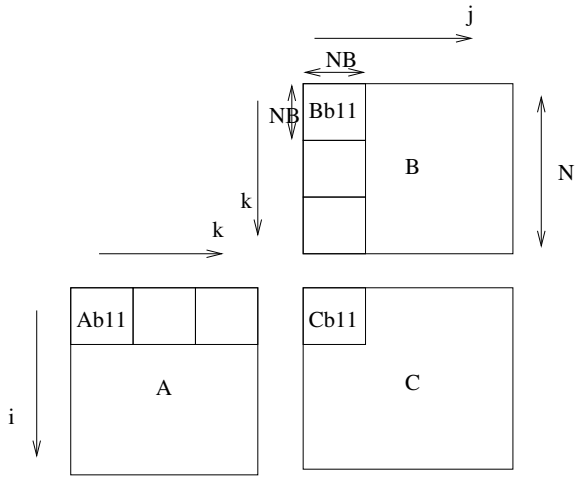
In hardware, a straightforward implementation of the matrix-matrix multiplication has a very bad data reuse locality. A classical loop transformation like blocking can be applied to improve the locality, see Figure 3.5. For simplicity of the design we assume that N is an integer multiple of NB and the elements of the array `C` were already initialized. After a blocking transformation, the code is separated into two parts. The first part is written in the mmmBlocked function, which traverses

```
// mmmBlocked - traversal of blocks
typedef signed long Number;
void mmmBlocked(Number* A, Number* B,
           Number *C, int N, int NB) {
  int j, i, k;

  for (j = 0; j < N; j += NB)
    for (i = 0; i < N; i += NB)
      for (k = 0; k < N; k += NB)
        mmmKernel(&A[i*N+k], &B[k*N+j],
                  &C[i*N+j], NB);
}

// mmmKernel - computation inside one block
void mmmKernel(Number* A, Number* B,
               Number* C, int N) {
  int i, j, k;
    for (j = 0; j < N; j++)
      for (i = 0; i < N; i++)
        for (k = 0; k < N; k++)
          C[i*N+j] = complexAdd(C[i*N+j],
            complexMult(A[i*N+k],B[k*N+j]));
}
```

(a) C code



(b) Graphical representation

Figure 3.5: Blocked matrix-matrix complex multiplication

the matrix blocks used for computation. It calls the second part, the mmmKernel function, which implements the multiplication of one block of a matrix. The blocking computation does not change the total number of computations, which remains N^3. We assume that `NB` is sufficiently small so that the blocks of `A`, `B`, and `C` fit together in the cache memory of the processor. As can be seen from Figure 3.5(b), the block `Cb11` can be fully stored in the cache while iterating over the loop `k` in the mmmBlocked function. After this iteration, the values of the block `Cb11` are computed and it can be stored in the memory. This technique reduces considerably the memory bandwidth requirements. For the original example `mmm` (Figure 3.4), if N is large enough, the accesses to `A` and `B` always correspond to cache misses, while the accesses to `C` correspond to cache hits, so the total number of memory accesses is:

$$\begin{aligned} N_{origmmmaccesses} &= N^3 * (N_{Aelem} + N_{Belem}) + N^2 * N_{Celem} = \\ &= (1+1)N^3 + 2 * N^2 \simeq 2 * N^3 \end{aligned} \tag{3.4}$$

where N_{Aelem}, N_{Belem}, and N_{Celem} represent the number of elements related to arrays `A`, `B`, and `C`, respectively, in the inner loop body. For the blocked version, we get:

$$\begin{aligned} N_{blockedmmmaccesses} &= \frac{N^3}{NB^3} * (N_{Ablock} + N_{Bblock}) + \frac{N^2}{NB^2} * 2 * N_{Cblock} = \\ &= \frac{N^3}{NB^3} * (NB^2 + NB^2) + \frac{N^2}{NB^2} * 2 * NB^2 = \\ &= 2 * \frac{N^3}{NB} + 2 * N^2 \end{aligned} \tag{3.5}$$

where N_{Ablock}, N_{Bblock}, and N_{Cblock} represent the number of elements of the arrays `A`, `B` and `C` accessed in a block of size $NB \times NB$. This formula takes into account the fact that the block elements of `C` will only be read and written once, from and to the external memory, and reused from the local memory.

The total gain of the blocking version is the following:

$$Gain_{blocking} = \frac{N_{origmmaccesses}}{N_{blockedmmmaccesses}} = \frac{2 * N^3 + 2 * N^2}{2 * \frac{N^3}{NB} + 2 * N^2} \simeq NB \tag{3.6}$$

3.2.2 Implementation

In this section, we present the details of the implementation we made for the matrix-matrix complex multiplication on an FPGA platform, including the design of an interface controller to optimize data transfers.

Xilinx university program (XUP) development board The design was implemented using the XUP FPGA development platform (Figure 3.6). This platform has a Xilinx Virtex II Pro (Figure 3.7) FPGA connected to peripherals such as a DDR1 dual in-line memory module (DIMM) memory module. The Virtex II Pro FPGA contains two PowerPC405 hard IP cores RISC processors implemented in the silicon being capable of running at 300 MHz. Each of the processor has 32 32-bit general-purpose registers, a memory management unit (MMU), and two 16 kB two-way set-associative caches for instruction and data. The processors are surrounded by configurable logic blocks (CLBs), corresponding to a total of 13696 slices and dedicated lines of 136 18x18 bit hardware multipliers and 2448 kb block select RAMs (BRAMs). Dedicated multipliers usually are faster and consume much less power than multipliers implemented using CLBs. Their other advantage is their placement with BRAMs, in lines that have a very fast access time.

The Xilinx embedded development kit (EDK) was used to build the system, using some IP cores that are provided (Figure 3.8) such as the joint test action group (JTAG) controller, BRAM controllers, DDR memory controllers, bridges such as the `PLB2OPB` bridge, and buses such as on-chip memory (OCM), the on-chip peripheral bus (OPB), and the processor local bus (PLB).

Architecture selection One design choice in the implementation is to decide where the hardware accelerator is located in the provided architecture. As discussed in Section 3.1, it could be located very close to the processor or directly connected to the system bus. In the provided reference design provided by the Memocode'07 contest, the accelerator was connected to the data-side OCM (DSOCM) (Figure 3.9). This interface is normally used to connect to BRAM modules. In this case, the accelerator has to be located in the `HARD IP` block. The data connection is bidirectional with 32 bits in each direction. The maximum theoretical attainable speed is 400MB/s.

MMAlpha IP Writing the matrix-matrix multiplication by hand, in the VHDL language, is a very tedious and error-prone task. The easier solution is to use one of the many available HLS tools. As the code is fairly regular, we decided to use the MMAlpha tool, which generates a systolic-like architecture.

To be able to use MMAlpha, the specification has first to be translated into the Alpha language (Figure 3.10). The language is based on the polyhedral model. The system is represented as a set of recurrence equations rather than loops. In this example, the first part of the system `matmult`

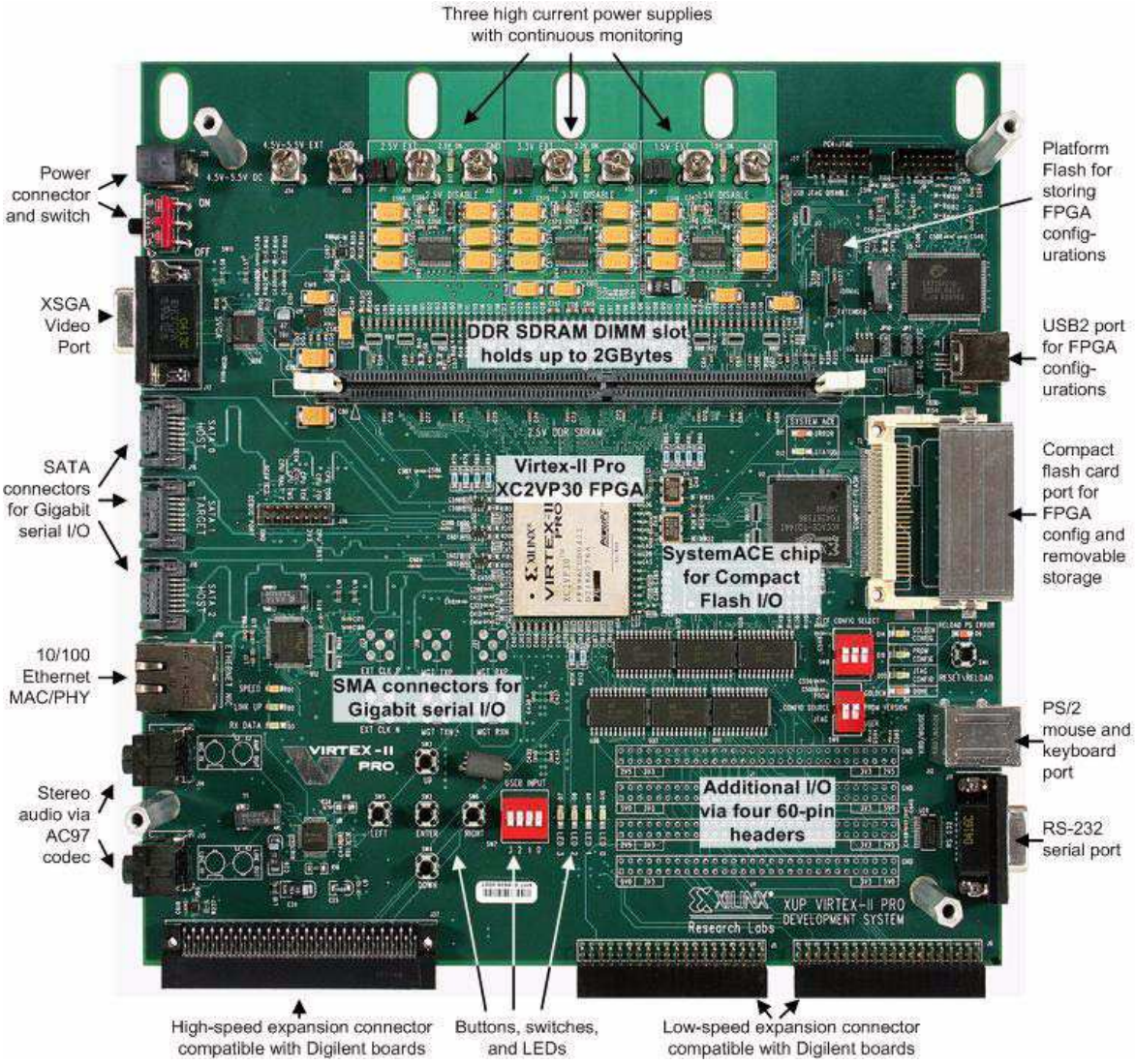


Figure 3.6: XUP FPGA development platform

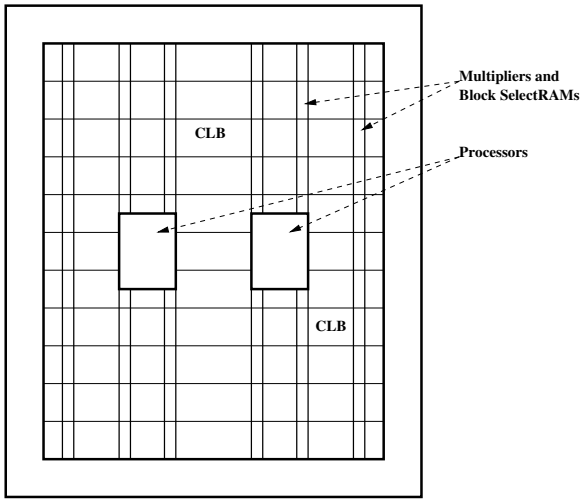


Figure 3.7: Virtex II Pro internal architecture using EDK

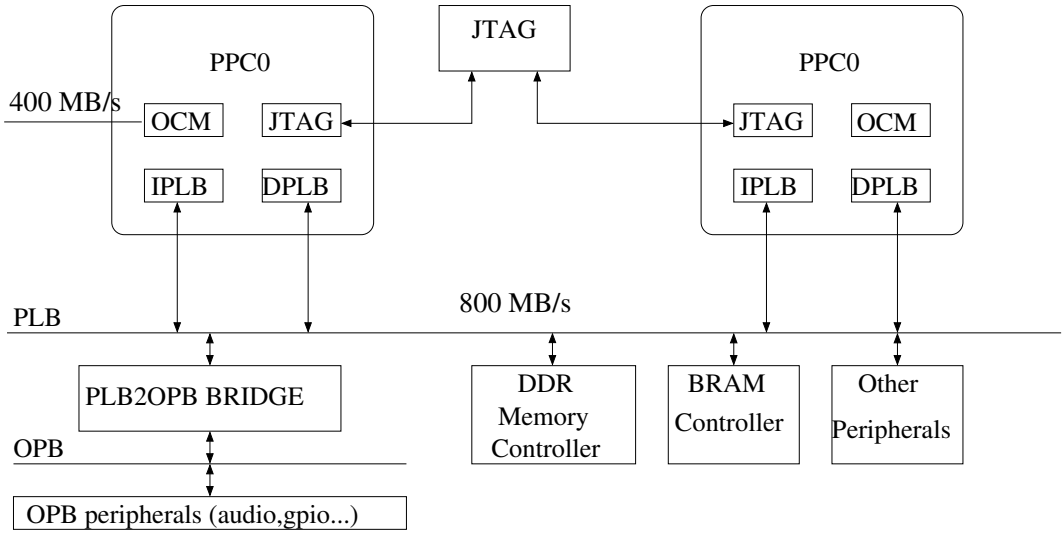


Figure 3.8: Virtex II Pro internal architecture using EDK

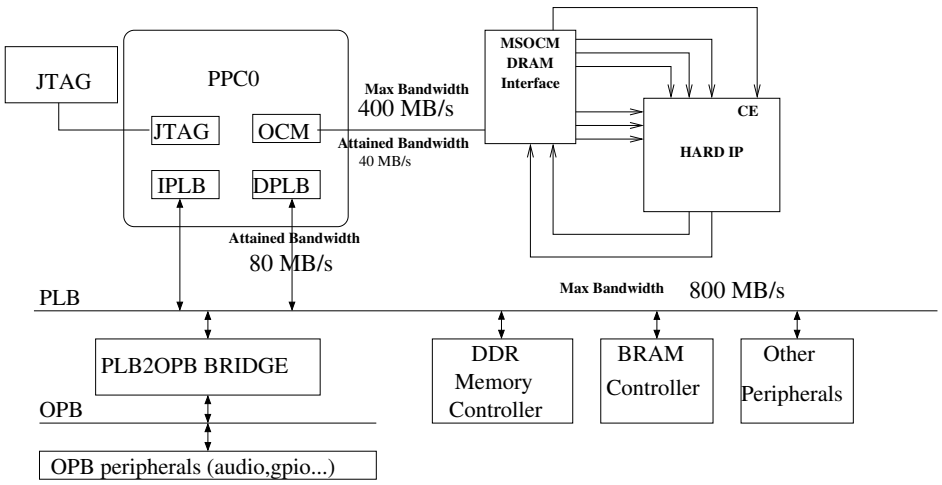


Figure 3.9: MSOCM connection of the accelerator and PowerPC405 processor

describes the points of the polyhedron. The lines such as `resRe1 = Mult(opARe, OpBIm);` describe the method of computation of, in this case, `resRe1`. The last part of Figure 3.10, containing case equations, defines that the element `CRe` on the point $\begin{pmatrix} i & j & k \end{pmatrix}$ is computed using its value on the point $\begin{pmatrix} i & j & k-1 \end{pmatrix}$ and the `resRe` value on the point $\begin{pmatrix} i & j & k \end{pmatrix}$ when $1 \leq k \leq N$ and 0 when $k = 0$.

The ALPHA representation is translated successively into ALPHA0, ALPHAHARD (see Appendix, Section 7.1.2), and finally VHDL. After each translation, the code resembles more to the hardware structure. For example, the ALPHAHARD system is divided into multiple modules: the control module `ControlmatmultModule`, the modules `cellmatmultModule1` till `cellmatmultModule4` describing the computation cells (different cells for special, including boundary, conditions), and the `matmult` module encapsulating all the others and providing the external interface.

The result of the synthesis is an IP that contains a systolic array (Figure 3.11) performing a 4x4 matrix-matrix multiplication. One complex multiplication requires 4 multipliers and therefore a 4x4 cells systolic array uses 64 multipliers. An 8x8 version would require 256 multipliers and would not fit in the selected FPGA that has 136 multipliers. [2] The inputs of the IP are the clock signal (`Clk`), control signals such as clock enable (`CE`) and (`Reset`), and data-path signals. `aIm` and `aRe` represent the buses used to transfer the imaginary and real values of the matrix `A`. Each of them is composed of 4 buses, directly connected to the 4 `A` inputs of the systolic array. The same is valid for the matrix `B` and `C`. The IP is supposed to get the required values on its inputs at a specific time and in a specific sequence. For example, the input `aIm0/aRe0` is supposed to read the `a11` element at time $t = 0$, `a12` at time $t = 1$, and so on. After an interval of time T ($T = 8$ for this example), at the output `cIm0/cRe0`, the first computed element `c11` of the matrix appears. After 6 more cycles, the last element `c44` is available at the output `cIm3/cRe3`.

Interface controller As previously explained, the IP generated by MMAlpha does have specific timing and order requirements. In order to connect it to the system, a dedicated interface controller

2. Note that we did not try to exploit partitioning techniques in MMAlpha, for which the interface would be even more complicated. Also, another possibility would have been to use hierarchical tiling so as to exploit the cache memory. The goal was here to develop an interface with some data reuse in the interface itself.

```
system matmult : {N,NB |N>NB>1}
  (aRe,aIm: {i,j | 1<=i<=NB; 1<=j<=N} of integer[U,16];
   bRe,bIm: {i,j | 1<=i<=N; 1<=j<=NB} of integer[U,16])
returns
  (cRe,cIm : {i,j | 1<=i,j<=NB } of integer[U,16]);
var

  CRe,CIm : {i,j,k | 1<=i,j<=NB; 0<=k<=N} of integer[U,16];
  opARe,opAIm,opBRe,opBIm : {i,j,k | 1<=i,j<=NB; 1<=k<=N} of integer[U,16];
  resRe1,resIm1,resRe2,resIm2,resRe,resIm : {i,j,k | 1<=i,j<=NB; 1<=k<=N} of integer[U,16];
let
  cRe[i,j] = CRe[i,j,N];
  cIm[i,j] = CIm[i,j,N];
  opARe=aRe.(i,j,k->i,k);
  opAIm=aIm.(i,j,k->i,k);
  opBRe=bRe.(i,j,k->k,j);
  opBIm=bIm.(i,j,k->k,j);

  resRe1 = Mult(opARe,opBIm);
  resRe2 = Mult(opAIm,opBRe);
  resRe = resRe1+resRe2;
  resIm1 = Mult(opARe,opBRe);
  resIm2 = Mult(opAIm,opBIm);
  resIm = resIm1-resIm2;
  CRe[i,j,k] = case
        {|k=0} : 0[];
        {|1<=k<=N} : CRe[i,j,k-1] + resRe[i,j,k];
esac;
  CIm[i,j,k] = case
        {|k=0} : 0[];
        {|1<=k<=N} : CIm[i,j,k-1] + resIm[i,j,k];
   esac;
tel;
```

Figure 3.10: Matrix-matrix multiplication written in Alpha

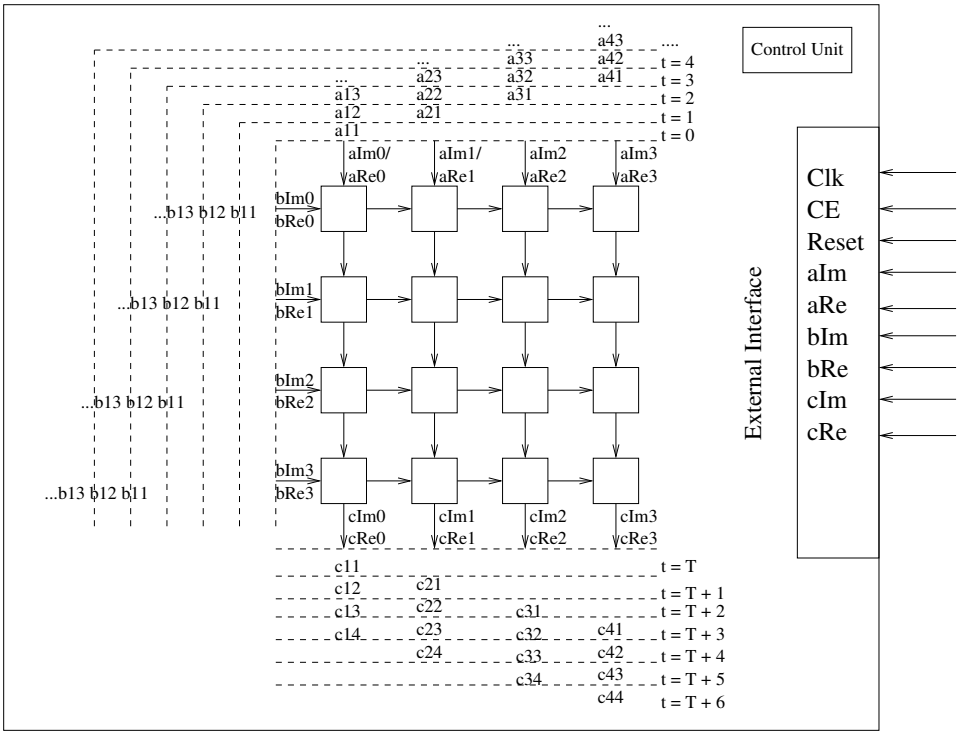


Figure 3.11: 4x4 cell hardware accelerator generated by MMAlpha

was designed (Figure 3.12). The controller connects to the DSOCM interface and acts as a slave to the CPU. It has multiple configuration registers. The control register is used to control the behavior of the interface controller. The master CPU can start or stop the controller, change the working matrix width and height. The CPU program will use the status register to verify the state of the accelerator. This register is accessed in polling mode.

As observed earlier, one of the major performance bottleneck of the matrix-matrix multiplication algorithm is the data availability. In this case, the data is received from the processor. Even though the maximum theoretical bandwidth of the DSOCM bus is 400MB/s, the experiments show that the real data transfer rate from the processor to a feedback FIFO connected to the DSOCM bus is about 40MB/s (in the best case, i.e., if the data is in a register of the CPU). This is mainly due to the loop control and other instructions overhead of the processor, and also due to the hardware communication protocol. The MMAlpha module requires 256 bits of data to be available at every clock cycle (3200MB/s) and therefore 80 times more.

In order to address this problem, we tried to exploit data reuse by adding to the design three memories composed of BRAM located on the FPGA. According to Equation (3.6), we can hope to improve the design by a factor 4. Each memory has 5 ports that can be accessed simultaneously. One port is used to receive the data from the CPU and the other 4 to feed the MMAlpha accelerator. This type of memory can be obtained by replicating by 3 a dual-port memory available in the FPGA. The downside of this type of memory is the replication of data, and thus inefficient use of the storage resource. In order to optimize the data reuse we chose to store a whole $4 * A_{width}$ of the matrix `A` and $4 * B_{height}$ of the matrix `B`. After the multiplication of these two blocks of matrices,

a block of size 4x4 of the `C` matrix containing the final result is obtained. The memory size for the arrays `A` and `B` is 8 kB. The address zone is divided in two and implements a double buffering technique. Half of the space is used to receive data from the CPU and half is used to feed the matrix-matrix multiplication accelerator. The first zone of the address space represents the registers used to store the intermediate matrix values. These values are accumulated as required using the accumulating blocks located in the `IP Dispatcher Accumulator for C` block (Figure 3.12). Normally, this accumulation could be performed inside the MMAlpha accelerator and thus eliminating the accumulation of 4x4 matrices. However, in our case, this was not the option. The design was required to be able to compute variable size matrices and MMAlpha IP cannot be parameterized during execution. The size of the computed matrix is only 4x4 and, in order to be able to obtain a better performance burst transfer, the result is stored in a 4 kB memory, and is sent to the processor when filled.

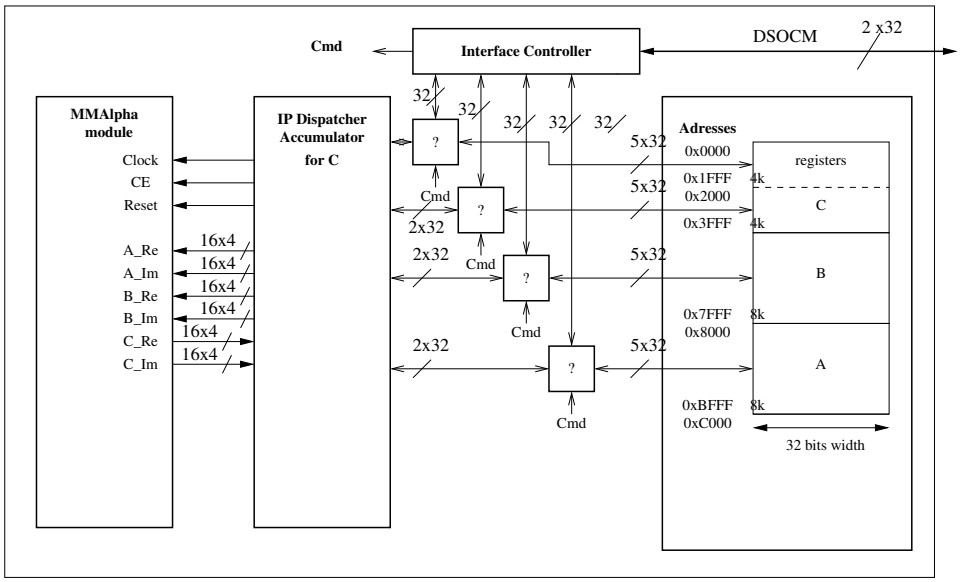


Figure 3.12: MMAlpha external communication interface and memory module controller

3.2.3 Final results and discussion

The accelerator and the interface were synthesized, obtaining a frequency of 103.4 MHz [3], which was sufficient because the interface of the processor was running at 100 MHz. The processor acted as a dispatcher of the matrices located in the DDR memory.

The theoretical maximum transfer connection of the processor to the DDR is limited by the transfer rate of the PLB bus and is 800MB/s. However, the real measured transfer rate is only about 80MB/s. This is mainly due to the instruction, cache memory, and bus overhead. The cache is blocking the processor execution from the beginning of a cache miss until the end of the cache miss is serviced and the required data is present in the cache. Therefore, each such cache miss is paid with an important access penalty. Each PLB transfer requires an arbitration, thus increasing

3. mostly limited by the BRAM and dedicated multipliers-to-multipliers interconnections.

the latency. The DDR controller itself has a very important access latency. Connecting accelerators as close as possible to the source of the data can remove this communication overhead. The final transfer rate we obtained was about 32MB/s [4], compared to the required 400MB/s! The design was thus heavily limited by the bandwidth. The real minimum execution time was, according to Equation (3.5):

$$t = \frac{N_{blockedmmmaccesses} * 4 \text{ (in Bytes)}}{\text{Bandwidth (in Bytes/s)}} = \frac{(2 * \frac{1024^3}{4} + 2 * 1024^2) * 4}{32 * 10^6} \simeq 67s \tag{3.7}$$

However, even with this data bandwidth limitation, the accelerator was 5 times faster compared to the 338 seconds performed by the PowerPC on the blocking matrix-matrix multiplication, which justifies the interest of using a dedicated hardware.

In these experiments, we used Xilinx EDK version 9.1i that provided the DDR interface controller with only one port connected to the PLB bus. In a later version (version 10), the DDR controller has multiple ports that can be connected to peripherals. Connecting the accelerators to these ports would theoretically offer a bandwidth of 3200MB/s from all 4 ports, which could be more than sufficient for the current design. In other words, instead of considering a strongly tightly-coupled architecture with cache as in Figure 3.2, we could have used a distributed architecture, as in Figure 3.1. Unfortunately, we were not able to test this as the new controller IP did not support our platform. We explore this type of distributed architecture in the next chapter, with the C2H Altera HLS tool.

3.3 Design flow with Spark and WRaP-IT

Today, existing commercial HLS tools require the original functional code (usually in a language with a C-like syntax, as we previously explained) to be written in a very specific manner in order to get good synthesis results. Hence, a source-to-source (S2S) preprocessing step is mandatory to get the code from the designer specification to a specification suitable to a particular HLS tool. Another important remark concerns the internal representation of loop nests. After many years of research in automatic parallelization, a modeling technique was developed for loop nests: the so-called *polyhedral model* [69, 87], also exploited in MMAlpha as mentioned in Section 3.2. This model provides an intermediate representation suitable for loop transformations.

This section presents the experiments we conducted to demonstrate the interest of adding advanced preprocessing code transformations to HLS tools. Our approach is to use an advanced state-of-the-art compiler front-end as an independent C-to-C preprocessing step before synthesis. By using this approach, recent state-of-the-art compiler advances could be used directly in HLS. We eliminate the re-engineering of them into modern HLS tools and the preprocessing effort can be reused by multiple HLS tools. We focus on efficient synthesis of loop nests and we use the WRaP-IT loop transformation framework integrated in the `Open64` compiler (Figure 3.13). As HLS back-end, we chose to rely on the `Spark` framework. Our study shows that important improvements are obtained in the resulting RTL design thanks to the fact that WRaP-IT uses a polyhedral representation for nested loops and provides a flexible framework for loop transformations. Improvements are shown, in particular, on the synthesis of part of the H263 decoder from `MediaBench II` benchmarks. Our study goes beyond these two particular tools (WRaP-IT and

4. 40MB/s processor to accelerator + processor to DDR overhead.

Spark). It demonstrates that HLS can be coupled with software compilation to achieve even better results than HLS tools alone can do (even those that have already integrated some loop transformations such as Spark, PICO-NPA, and others) and, even possibly, to eliminate the need for compiler transformations integration internally in the HLS tool.

Section 3.3.1 gives a brief review of the software compiler and HLS frameworks we used. Section 3.3.1 introduces our synthesis flow, which uses Spark and WRaP-IT. Section 3.3.2 summarizes the interest of loop transformations in the context of Spark. In Sections 3.3.3 and 3.3.4, we present two synthesis examples and analyze the performance improvements obtained using various loop transformations before synthesis.

3.3.1 Choice of tools: Spark and WRaP-IT

As a code transformation framework for our flow, we selected the WRaP-IT tool [33]. This tool explicitly implements a polyhedral internal representation and it is fully integrated in the Open64 compiler. This allows us to rely on Open64 for the non-static control parts of programs. We point out that WRaP-IT is not a parallelization tool, it manipulates sequential code. But it can be used to prepare the code for an HLS tool that does parallelization by itself or to prepare parallelization to be exploited by an HLS tool, for example if the code is annotated with parallelization pragmas. As we explained later, this was one of our initial motivation. Another nice property of WRaP-IT is its user interface to manipulate loops. The user can very easily specify loop transformations and can provide new ones thanks to the URUK script language.

As a back-end, we preferred to rely on a HLS tool such as Spark for the following reasons. First, it is important to take into account the possible interactions of high-level transformations with back-end optimizations. Possible performance loss cannot be clearly seen with a basic HLS tool, so Spark was a better choice. Second, we selected Spark because its main strength is in control-intensive programs and thus we can rely on the optimized FSM output since loop transformations will alter the complexity of the generated FSM. Third, relying on independent tools for both parts (front-end and back-end) shows the feasibility of our two-phases approach. Indeed, the fact that we use Spark as a black box, with no possible source modification, shows that we could do the same with a commercial tool. This is what we do in the next chapter with Altera C2H. Finally, using sophisticated tools for the front-end and back-end brings the best of the two worlds. For example, we can consider a full application, not just static-control programs, even if we perform

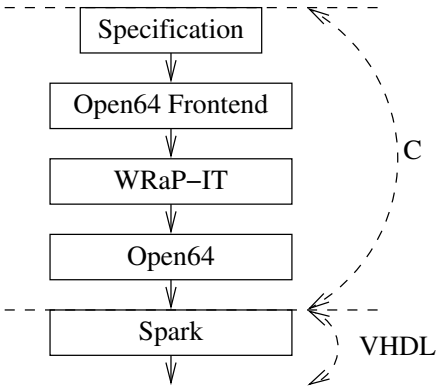


Figure 3.13: Our methodology and design flow

loop transformations only on static-control parts.

We point out that, unlike [60], the loop transformations used in this chapter are selected manually. However, we can analyze the impact of every single transformation (or a series of transformations) on the resulting hardware (size, latency, etc.).

Code representation in the polyhedral model To illustrate the functionality of WRaP-IT, we first present a short example of polyhedral code representation.

Polyhedra are used to represent the set of elements that a system of affine recurrence equations defines. Domains of nested loops with affine lower and upper bounds can be defined in terms of polyhedra too. Figure 3.14 gives an illustrating code example with its corresponding representation in the polyhedral model.

The polyhedral representation has a weakness: it is impossible to represent in this abstraction a loop bound that cannot be expressed as an affine function of surrounding loop counters. However, the analysis from [73] of various benchmark codes showed that most of the loops, with exceptions for some special ones (e.g., FFT), have affine loop bounds. The parts of a program that are enclosed in loops with affine functions on their iteration domains are called static control parts (SCOPs). Most of the computational intensive parts of programs are SCOPs. The polyhedral model often assumes also that the computations themselves are affine, i.e., it consists in array operations with affine access functions. This is a much stronger restriction in practice.

In [73], several examples are given that prove that syntactic representation limits the detection and the applicability of loop transformations. The polyhedral representation of the code has proved to be better than syntactic ones. The advantage of this representation was observed by many compiler designers. In particular, it allows to manipulate nested loops as a whole, with a multi-dimensional view, without unrolling, and even in a parametric fashion. Loop transformations based on the polyhedral model framework named `Graphite` were included in GCC [96]. This is another advantage of the WRaP-IT framework that guided us to choose it for our study.

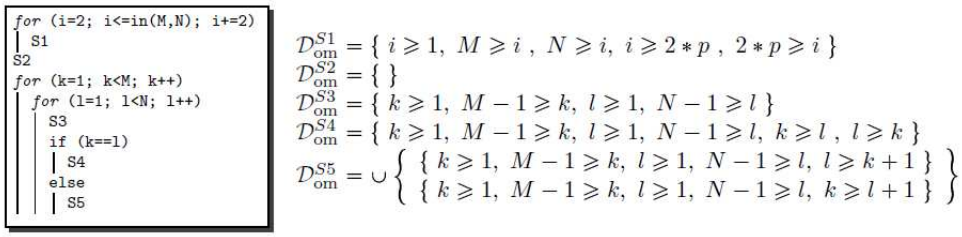


Figure 3.14: Code representation in polyhedral model

The loop transformer WRaP-IT The WRaP-IT framework was developed by the Alchemy team. It is described in details in Sylvain Girbal's PhD thesis [73]. Its flow diagram is presented in Figure 3.15. It was designed as a complement to the `Open64` compiler from Silicon Graphics. It replaces its loop nests optimizer with a more powerful one, as proved by benchmarks implemented in the `Uruk` tool.

The input C code passes through the first phase of the `Open64` compiler, the pre-optimizer, where various code transformations like constant propagation, dead code elimination, loop normalization into do-loops, and other transformations are applied. The `Open64` compiler was modified to stop

its execution and to dump its internal intermediate representation, called `whirl`, into a file. This file is read by the `whirl to polyhedral` module, which transforms all static control parts of the code into a `polyhedral` representation.

`URUK` applies loop transformations on this `polyhedral` representation based on a specification in the `URUK` script file. The final step is the re-insertion of the transformed code back into the original `whirl` representation. First, the polyhedron of each statement is scanned using `CLOOG` [32] by means of projections and separations. As a result, a set of iteration domains is obtained for each statement. The statements with same iteration domain or intersecting iteration domains are grouped and form the body of a loop. Finally, this loop information is reinserted back into the `whirl` representation. The `Open64` compiler starts again from this representation and generates an executable.

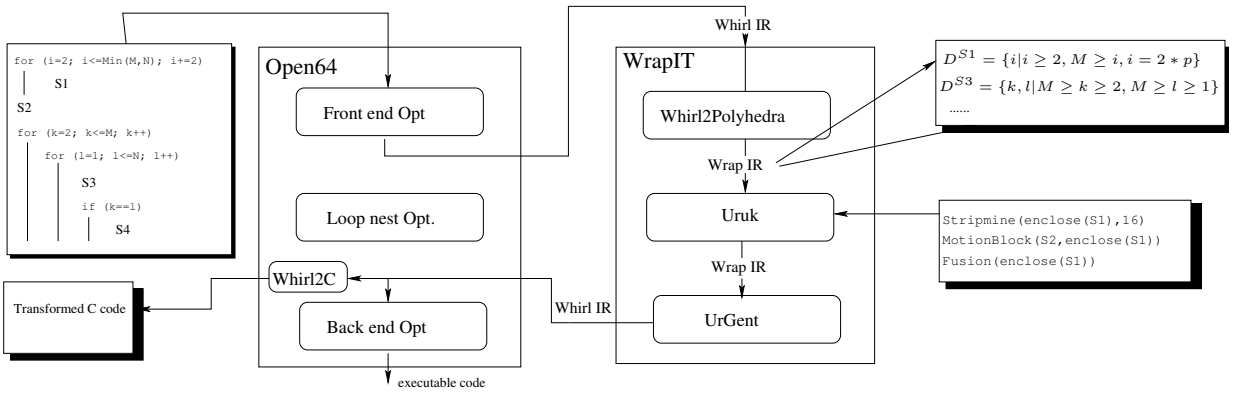


Figure 3.15: WRaP-IT framework

The high-level synthesis tool Spark In Chapter 2, some of the most known HLS tools were presented. An analysis of these tools was performed in order to identify the most appropriate HLS tool for our goals. Most of the tools accept, as input, hardware specific languages. The loop transformation framework we use (WRaP-IT) generates a transformed code that is still software-like and more precisely written in `C` (i.e., more specific for a use on general-purpose processors). Working with such HLS tools would mean writing a new module to transform a design from a sequential software-like language to a hardware specific language. But our goal is to study the impact of high-level loop transformations on the resulting hardware accelerator, not on the whole software-to-hardware design methodology. Since the input language of the HLS framework was chosen to be *C*, a framework that accepts as input a pure ANSI-C language was more suitable. The most appropriate framework that was freely available at the moment was the `Spark` framework (Altera C2H, used in the next chapter, was not yet available).

`Spark` is an academic HLS tool (see Chapter 2), provided in binary format, that takes as input a code written in a subset of ANSI-C and that incorporates many code transformation techniques to improve the quality of the synthesized circuit (see Figure 3.16). They are applied on its internal representation called hierarchical task graph (HTG). The user has to specify the resource constraints using a configuration file. `Spark` generates a synthesizable VHDL state machine, produces performance statistics, and is able to generate C code that emulates the parallel execution of the hardware generated. `Spark`'s scheduler computes execution cycles statistics of the scheduled

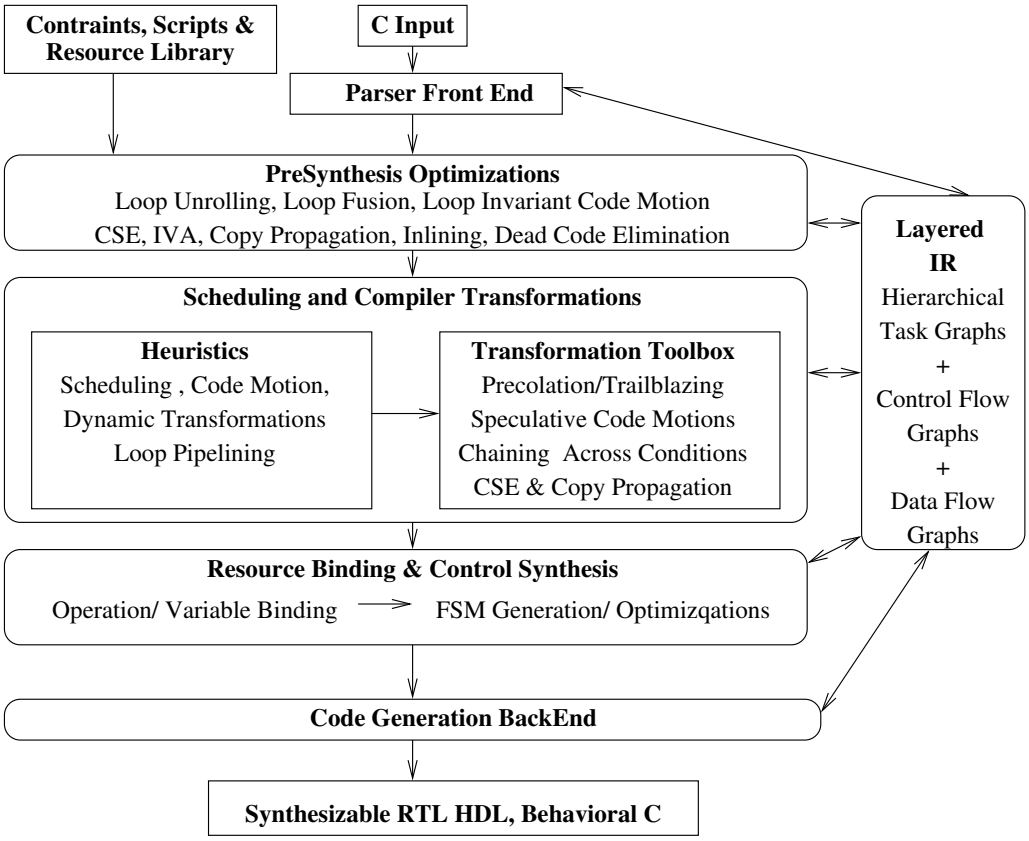


Figure 3.16: Spark framework internal transformations and optimizations

design. These are the minimum, maximum, and the average number of cycles that the generated hardware accelerator will run. This is very useful for comparison of the performance of the code, without any simulation. The other important statistic is the number of states of the generated FSM. The number of states will be used in order to estimate the area utilization. `Spark` can also generate a `C` code that represents the actual finite state machine with, in each of the state, the operations scheduled for parallel execution during the current clock cycle. This file will be used later to perform cache access simulations: we will compare the actual run-time of the design based on the cache miss delay and memory delay of different designs. The cache miss statistics file can be used to compute the memory power consumption of different designs.

In addition, we used `dineroIV`, a trace-driven uni-processor cache simulator [5], developed at the University of Wisconsin. It uses the cache access trace of an execution of a program and simulates the cache behavior, reporting significant performance results.

Design flow We now have all the components in our design flow (Figure 3.17): the WRaP-IT framework [33], the `Open64` compiler, the `Spark` HLS framework [78], and the `dineroIV` cache simulator [5].

We start from an initial sequential C specification. This code is fed to `Open64`. In the front-end of `Open64`, multiple code transformations can be performed such as procedure inlining, dead function/variable elimination, constant propagation, etc. The use of procedure inlining can be used

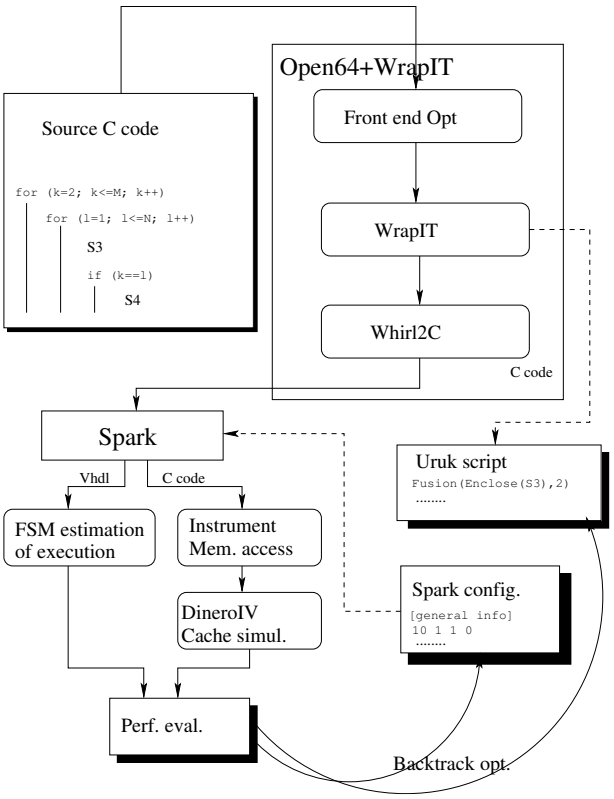


Figure 3.17: Our VHDL design flow combining WRaP-IT and Spark.

to transform a code having multiple function calls into a form that can be synthesized by `Spark` (i.e., without function calls). In the next step, loop transformations are performed by `WRaP-IT`, guided by the user (the user indicates which transformation to perform thanks to the `URUK` script language). The `Whirl2C` program, provided with `Open64`, is used to generate C code from the `whirl` representation. At this step, because of some bugs found in `Whirl2C` at the generation, we needed to modify this generated code. Then, this modified C program is fed to `Spark`, which is configured with resource constraints provided in the configuration file mentioned in Section 3.3.1. `Spark` outputs a VHDL file of the synthesized hardware accelerator and a C file. The C file contains the equivalent of the hardware accelerator state machine written in C. The performances of the resulting circuit are evaluated from these files. We instrumented the C file produced by `Spark`. This C file, when compiled and executed, generates a trace of memory accesses, which is finally used by the cache simulator to analyze memory access performances.

We point out that the VHDL designs generated by `Spark` cannot be synthesized directly. As previously mentioned, `Spark` does not synthesize a memory controller for its inputs and outputs, instead the circuit contains many I/O pins. The communication is done by means of bidirectional ports, which does not have any flow control signals. Each I/O data bit generates a new I/O pin. Since it was conceived for control-intensive programs, it does not generate a real memory controller (we recall that we chose `Spark` because of its state-of-the-art internal optimizations). As opposed to the MMAlpha design, it is impossible to generate an interface or memory controller for the design generated by Spark, unless it uses only very few inputs. Indeed, when MMAlpha produces

predictable I/O, specified by the user, Spark applies many non-controllable HTG transformations before synthesis. Without knowing the exact timing of the I/O, we cannot feed the accelerator with the required data when needed. A solution would be to connect its I/O ports to dedicated memory cells in BRAM memories. However, these memories have a maximum of two ports and `Spark` requires that all I/O ports can be accessed in parallel. Thus, one would need a memory location for each I/O port. Since there is no way to know when these data are consumed and thus when the memory can be reused, such a solution would require a huge local memory, one cell per value read, which is not practical. Another solution would be to generate such a dedicated port memory using the logical cells in the FPGA. However, this will reduce dramatically the running frequency of the circuit and usually the circuit will exceed the routing capacity of the FPGA. Nevertheless, the schedule of the I/O operations made by `Spark` and the hardware generated for the computational kernel are correct, i.e., they respect dependence and resource constraints. Thus, we think that our evaluation with the cache simulator is realistic and that the impact of loop transformations we analyze is relevant.

As we just explained, we thus did not synthesize any memory controller but we simulated the effect, on the cache, of the IP generated by `Spark`, thanks to `dineroIV`. This tool was configured to simulate a cache with a size of `8 kB`, block size of `32 B`, and associativity `4` for the first example. For the second example, it was configured to simulate a cache with the size of `32 kB`, block size of `64 B` and associativity `8`. For both examples the replacement policy is `LRU`, fetch policy as `demand`, write allocation `always`, and write back `always`. The write allocation policy was chosen because of the spatial locality of writes found in multimedia applications as well as in our example. This improves dramatically the burst write modes to the external memory. Synthesis results are obtained by providing to `Spark` the following timing constraints: 20 ns clock cycle, 10 ns for each of the arithmetic operators available, 20 ns for multiplications, etc.

A note on pragmas for parallel execution One of our initial goals in this study was to include, in the WRaP-IT framework, the possibility to generate a C code with `OpenMP` pragmas that would allow the parallel execution of different statements. This way, we would obtain a code with an explicit description of parallelism. The HLS tool would then have to exploit these pragmas. Unfortunately, `Spark` does not recognize any of these pragmas and modifying it was not an option (`Spark` is freely available as binary but not as source code). We were thus not able to analyze, with `Spark`, the benefit of expressing parallelism, in particular loop parallelism, at source level.

Another problem we faced was in the limitation of the C code generator from the `whirl` IR. When a C code containing a `pragma` is passed through the first stage of `Open64`, then transformed into the `whirl IR`, then translated back to C using the `whirl2C` tool, the final code does not contain any `pragmas`, labels, and other preprocessor constructs anymore. The implementation of a new `whirl2C` tool, which would capture then regenerate the pragmas from whirl if present, was then considered. But we decided to not put effort in such developments since designing an extended and stable version of `whirl` within `Open64` was not in the heart of our research objectives. However, the framework flow we used showed that, at least for the hardware accelerators we wanted to generate with `Spark`, i.e., with bounded parallelism in a single hardware accelerator, there is no need for such constructs at all. It is enough to give to `Spark` the parallel version of the code (without marking it) and, after data dependence analysis, it will find out the blocks that can run in parallel, possibly helped with some loop unrolling. This is because data dependence analysis in `Spark` is powerful enough. This is not be the case for some other tools, for example C2H, used in the next chapter.

3.3.2 Loop transformations in HLS, in the context of Spark

In section 2.3, we gave a brief overview of the most known code transformations that could be useful in HLS (including some that we identified). We now analyze more precisely, in the context of the `Spark` HLS tool, these various transformations and their impact on the performance and quality of the synthesis. The circuit architecture and platform model we use are those presented earlier.

Loop fusion Loop fusion combines two loops with the same iteration domains into one containing the bodies of the two fused loops. It has many advantages in HLS.

In `Spark`, each synthesized loop scans its own subspace of states. A part of the loop states implements the actual loop body. The other part implements the loop control part that increments the loop counter and verifies the termination condition. When two loops are merged (we suppose that the loop iteration spaces are identical), the two loop bodies are managed by only one control part instead of two. This decreases the number of states and the FSM complexity, which decreases the utilized area of the circuit.

When the original loop bodies have dependences between them, the two loops cannot be scheduled to run in parallel, unless the dependences are only from the body of the first loop to the body of the second one, for the same iteration `i` as in the code below on the left. After the merge, the bodies of the loops will still be scheduled in a sequential order, but the overall runtime will be reduced, since there is a single loop control for both bodies, see the code below on the right.

```
for (i=0; i<n; i++)                  for (i=0; i<n; i++){
  a[i] = ....                          a[i] = ....
for (i=0; i<n; i++)                    ... = a[i]
  ... = a[i]                         }
```

Another advantage of the loop fusion transformation is that it can promote array contraction, thus replacing an entire array (here the array `a`) by a scalar. Spark synthesizes scalar variables into internal registers and it keeps them in the chip itself. Thus, this leads not only to a reduced memory utilization, but also to faster accesses to the data because the access time of the internal registers of the architecture is roughly ten times faster than the access time to the external memory. It also leads to a decreased power consumption of the entire system, since the power consumption of a memory access is higher than the one of a register access.

Fusion can also be used to indirectly instruct the hardware scheduler of the HLS tool to reuse the accessed data of one instruction for another one. For example, suppose two parallel loops access, with the same pattern, the same set of data that is larger than the cache size. These two loops will be scheduled by `Spark` in sequence and hence the accelerator will have to access twice the same set of data, with the same cache misses. However, if we fuse the loops (and if the resulting loop is still parallel), we indirectly warn the scheduler that we want the bodies of the two loops to be executed in parallel in a synchronized manner improving the cache hit ratio and the data reuse. This will decrease the power consumption of memory accesses, decrease the data access latency, and increase the execution speed by removing this access latency.

Loop distribution (or fission) This is the inverse of loop fusion. The body of the original loop is distributed between two new loops, with identical bounds.

This transformation can be used to improve the cache locality, in some cases. If the loop body has multiple parts with different data access sets, loop distribution can be used to separate the

original loop body into two bodies executed in separate sequential loops. This will reduce the local data working set of the loop, thereby increasing the potential cache hit. Of course, unlike loop fusion, loop distribution duplicates the control part of the loop and thus increases the resource need.

Code motion Code motion is particularly useful when loop fusion or loop distribution are not valid. The loop fusion is limited to loops with the same dimensions and iteration domains, and they also need to be consecutive (i.e., no code between them). Code motion is more flexible since it can move arbitrary pieces of codes. But, of course, it is more difficult to use, as more general. It can be used to move a data producer closer to the consumer thus, as in case of fusion, for decreasing the cost of memory accesses by increasing the cache hits. This reduces the memory access power consumption and increases the speed of the circuit.

Loop interchange This transformation permutes two nested loops. To get an equivalent code, this may require to redefine loop bounds.

As an example, consider two nested loops such that the outer loop (loop over `i`) is parallel and the inner loop (loop over `j`) is sequential. To increase the performance and exploit potential parallelism, one can unroll by 2 the loop over `i`:

```
for (i....) {
  for (j...) {
     S0
  }
}
```

⇒

```
for (i....){
  for (j...)
    S0
  for (j...)
    S0
}
```

The two loops over `j` can be executed in parallel, which decreases the execution time. However, this unrolled code has replicated not only the statements performing the computations but also the FSM that control them. To avoid replicating the loop control overhead, one can apply loop interchange and, only after, apply the unroll of the obtained inner parallel loop:

```
for (j....)
  for (i...){
    S0
    S0
}
```

Loop reversal Loop reversal changes the direction of its traversal, i.e., the traversal will be backward to the original one: if the loop counter is incremented, it is now decremented. This transformation is mostly used to allow other loop parallelization transformations to be performed, such as loop fusion. It is valid only for loops that have no loop-carried dependences.

Loop skewing Loop skewing is a transformation that is primarily used, when combined with loop interchange, to make a loop parallel when it was sequential in the original form. It was invented to handle the so-called `wavefront` computations, because the updates to the array propagate like a wave across the iteration space. This type of computations is often found in image processing algorithms. On these types of codes, `Spark` usually synthesizes a sequential hardware accelerator. After applying this transformation, the code can run in parallel. However, in order for `Spark`

to really make it run in parallel, one needs to instruct `Spark` to unroll the parallel loops by a specified factor. This way, the successive parallel bodies will be detected as parallel and parallelism will be exploited. Loop skewing adds significant computational overhead to the loop iterators computations and bounds verifications. But `Spark` schedules and maps the iterators and loop bounds computations to the same arithmetic elements used to perform the statements inside the loop. Thus, synthesizing a skewed loop with `Spark` only increases the FSM size and the loop control part compared to the original code. On the contrary, if syntax-driven HLS tools are used, the loop control computations will be mapped to dedicated arithmetic elements, which will increase the arithmetic elements need.

Loop peeling Loop peeling removes the first (or last) iteration of a loop. This transformation can be used to make possible the application of other transformations. The first use can be the removal of a few iterations (the first or last ones) when they have data dependences that prevent the parallelization of the loop. Another use, more frequent, is to remove some iterations from the iteration domain so as to enable loop fusion. For a code represented in the polyhedral model, it is not necessary to adjust the loop bounds in order to fuse them because it is done transparently by the `CLOOG` tool. After the fusion of two loops with different bounds, `CLOOG` will perform an intersection of the domains and will generate the loop bounds for the three loops, i.e., the loop bounds for the code corresponding to the intersection of the iteration domains, and two other bounds for the remaining domains.

Loop unrolling Loop unrolling changes the structure of the loop, but leaves unchanged the computations performed by an iteration of the loop body and their order. Loop unrolling by a factor r replicates the body of the loop r. Unrolling can reduce the loop overhead and improve data locality. The iteration space of the unrolled loop is r times smaller, hence the loop control computations are executed r times less often. The basic block of the unrolled loop is r times larger, which increases the possible data reuse and data distance minimization inside the loop body.

Another advantage of this transformation is that it increases the possible parallel execution within the loop basic block. `Spark` already contains an implementation of loop unrolling. It is used to increase the execution parallelism of the loops that are supposed to run in parallel. The synthesis of an unrolled loop using `Spark` was tested for several examples. If the loop is parallel, the execution speed increases by the exact unroll factor (when sufficient resources exist on which the computations can be scheduled). All the array accesses and functions of the iterators are pre-computed in parallel at the beginning of the loop iteration.

Strip-mining Strip-mining is a widely-used method for adjusting the granularity of a parallelizable operation on vector and multicore architectures. The original loop is transformed into two nested loops that cover the same iteration space of the original loop. The iteration space is divided into strides of a specific length.

For synthesis using `Spark`, we can use strip-mining to obtain a better data reuse. It can also be used to obtain a suitable granularity of loop parallelism, after performing strip-mining by a factor (strip length) of the desired parallelism and asking `Spark` to then fully unroll the innermost loop. The result is equivalent to a partial unroll of the same factor. However, we point out that, after strip-mining, it is not always possible to unroll the innermost loop if it has unknown (parameterized) bounds or bounds hard for `Spark` to evaluate, for example with `min` and `max` functions.

Strip-mining has also one major disadvantage when used for HLS. The computation of the indices of the inner loop contains a multiplication, unless the strip factor is a power of 2. After full unrolling of the inner loop, the access function to arrays have the form *strip_factor*iterator_outer + constant*. Since all the access functions will be scheduled to be computed in parallel by `Spark`, the amount of area utilization will also increase because of the high area utilization of the multiplier unit. However, `Spark` can detect the case of several identical multiplications scheduled at the same state, and can reuse the result of one multiplication to avoid the others, thus decreasing the area utilization to one multiplier and as many adders as expressions, i.e., as many as the strip factor. The speed degradation can also be significant, since at each parallel iteration of the loop, there is also a need for computing the next iterator and this implies a multiplication and an addition, unless `Spark` avoids it thanks to strength reduction.

In the next sections, we first highlight the potential gains that loop transformations can bring to HLS designs on a simple edge detection example. Then, we consider a H263/H264 decoder application. For both cases, remember that the underlying architecture is weakly tightly coupled, with a cache, as in Figure 3.3.

3.3.3 First example: edge detection

The first example we detail here consists of a C code (Figure 3.18) that performs edge detection on a 100-by-100 pixels 16-bit-depth image. The code performs several function calls that first apply the Laplacian filter on an image, then apply horizontal and vertical Sobel filters on the image. At the end, the results are merged. This code contains many temporary arrays, it is typically written by an application designer who does not take into account memory access optimizations and would rather have a modular and readable code.

In order to be synthesized by `Spark`, this code has to be pre-processed. In the first phase, the code is inlined using the `Open64` inline stage (Figure 3.19). The `Open64` is instructed to pass only through the front-end and optimization phases and to dump the `whirl` representation into a file. The `Whirl2C` tool is used to transform this representation back into a C code. The obtained code is instrumented by hand with special `URUK` labels (Figure 3.19). In the `whirl` representation, these labels are identified by `WRaP-IT`. They are used to apply a succession of loop transformations. The loop transformations are specified by using the `URUK` script from Figure 3.20.

Loop transformations The loop transformations applied to the example, thanks to the `WRaP-IT` tool, are classical ones. First of all, code motion is applied in order to move the initialization of array `C` close to its use. This operation is described as motion_block in the `URUK` script. Between the parenthesis two arguments are specified: the first one represents the label of the block to be moved and the second one the block before which to put the moved block. In this example, motion_block performs a motion of a block specified by the label `LBL2` before the block specified by the label `LBL6`. The iteration domains of the block to be moved remains the same, only the sequential ordering in the scattering function changes. Because of the polyhedral representation, this transformation implicitly fuses the iteration spaces of both blocks (Figure 3.21). If they were distinct, `CLOOG` would perform an intersection of the domains and would generate a code with a prolog and an epilog.

The second transformation (`fusion`) specifies a loop fusion: the fusion of the current loop and the one immediately after. The `enclose` is a modifier that removes the innermost loops from the iteration domain. In this example, the `enclose(LBL4,2)` (Figure 3.20) specifies the `i` loop of the block with the label `LBL4` (Figure 3.19). Thus, the whole `fusion(enclose(LBL4,2))` means the

```
#define N 100
int A[N * N], A1[N * N], B[N * N], B1[N * N], B2[N * N], C[N * N];

// Init array C
    init(C);

// Laplacian filter
    laplacian_filter(A);

// Horizontal Sobel filter stores the result in array B1
    horizontal_sobel_filter(B1, A);

// Vertical Sobel filter stores the result in array B2
    vertical_sobel_filter(B2, A);

// Merge the results
    merge_results(C, A1, B1, B2);
```

Figure 3.18: Initial source code of the edge detection example

fusion of the `i` loop of the statement `LBL4` with the ı loop of the statement `LBL5`. After the fusion, the `j` loop of the block `LBL4` can be fused with the `j` loop of the block `LBL5`: it is performed using `fusion(enclose(LBL4))`. The remaining transformations are similar and perform a sequence of doubly-loop fusions and array scalarizations. The scalarization transformation was applied by hand. The unroll transformations were performed using `Spark`. The final code slightly simplified by hand for better readability is presented in Figure 3.22.

Experimental results Each cache miss implies an execution stall during which the cache is fetching the required line from memory. We used in our examples a memory fetch latency of 40. We take this into consideration and the real number of execution clock cycles is computed using the formula $\text{Cycles}_{real} = \text{Cycles}_{ideal} + 40 * \text{Cache}_{miss}$. We point out that, at the end of each experiment, we did not performed any cache flush, which means that the data are not really written back to memory. If required, the real amount of data transfers can be obtained by adding the cache size and the size of the write-back buffer. This however does not change much the general conclusions. Figure 3.23 gives the improvements of the resulting hardware design, obtained after these transformations, in terms of number of cache misses, total number of communications between the accelerator and the memory, total number of execution clock cycles, and the number of states in the FSM of the accelerator generated by `Spark`.

The first transformation (motion_block) minimizes the reuse distance of the vector `C`. As a result, the total number of memory accesses and cache misses is minimized. Due to this transformation, the number of cycles also decreased by 39%. Each loop control has an overhead. For the doubly-nested loop, this overhead is $(N + outer\_loop\_control) * (N + inner\_loop\_control)$. Since the statements `LBL6` and `LBL2` share the same state machine, the loop overhead is minimized and so is the number of clock cycles. A part of cycles are saved thanks to the sharing of array access address computations.

```
#define N 100
int A[N * N], int A1[N * N], int B[N * N], int B1[N * N];
int B2[N * N], int C[N * N];

// Init temporal A1, B1, C1 with 0 (inlined)
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
{
__URUK_LBL1:  A1[i * N + j] = 0;
              B1[i * N + j] = 0;
              B2[i * N + j] = 0;
}

// Init C with 0 (inlined)
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
__URUK__LBL2: C[i * N + j] = 0;

// Laplacian filter stores the results in array A1 (inlined)
for (i = 1; i < N - 1; i++)
  for (j = 1; j < N - 1; j++)
__URUK__LBL3:  A1[i * N + j] = (-1) * A[(i - 1) * N + j - 1] +
      (-1) * A[(i - 1) * N + j] + (-1) * A[(i - 1) * N + j + 1] +
      (-1) * A[i * N + j - 1] + (8) * A[i * N + j] +
      (-1) * A[i * N + j + 1] + (-1) * A[(i + 1) * N + j - 1] +
      (-1) * A[(i + 1) * N + j] + (-1) * A[(i + 1) * N + j + 1];

// Horizontal Sobel filter stores the result in array B1 (inlined)
for (i = 1; i < N - 1; i++)
  for (j = 1; j < N - 1; j++)
__URUK__LBL4:  B1[i * N + j] = (-1) * A[(i - 1) * N + j - 1] +
      (-2) * A[(i - 1) * N + j] + (-1) * A[(i - 1) * N + j + 1] +
      (1) * A[(i + 1) * N + j - 1] + (2) * A[(i + 1) * N + j] +
      (1) * A[(i + 1) * N + j + 1];

// Vertical Sobel filter stores the result in array B2 (inlined)
for (i = 1; i < N - 1; i++)
  for (j = 1; j < N - 1; j++)
__URUK__LBL5: B2[i * N + j] = (1) * A[(i - 1) * N + j - 1] +
      (-1) * A[(i - 1) * N + j + 1] + (2) * A[i * N + j - 1] +
      (-2) * A[i * N + j + 1] + (1) * A[(i + 1) * N + j - 1] +
      (-1) * A[(i + 1) * N + j + 1];

// Merge the results into C (inlined)
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
__URUK__LBL6: C[i * N + j] = A1[i * N + j] + B1[i * N + j] + B2[i * N + j];
```

Figure 3.19: Source code of Figure 3.18 after Open64 inline and insertion of Uruk labels

```
0: motion_block(LBL2,LBL6)
1: fusion(enclose(LBL4,2))
2: fusion(enclose(LBL4))
3: fusion(enclose(LBL3,2))
4: fusion(enclose(LBL3))
5: fusion(enclose(LBL5,2))
6: fusion(enclose(LBL5))
7: fusion(enclose(LBL1,2))
8: fusion(enclose(LBL1))
```

Figure 3.20: Uruk script specification of loop transformations

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
__URUK__LBL2: C[i * N + j] = 0;
__URUK__LBL6: C[i * N + j] = A1[i * N + j] +
               B1[i * N + j] + B2[i * N + j];
```

Figure 3.21: LBL2 and LBL6 after motion block

```
#define N 100
int A[N * N], C[N * N];

int main(void)
{
  int i,j;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      if (i == 0) || (j == 0) || (i == N-1) || (j == N-1)
        C[i * N + j] = 0;
      else
        C[i * N + j] = - A[(i - 1) * N + j - 1] - 3 * A[(i - 1) * N + j]
                       - 3 * A[(i - 1) * N + j + 1] + A[i * N + j - 1]
                       + 8 * A[i * N + j] - 3 * A[i * N + j + 1] + A[(i + 1) * N + j - 1]
                       + A[(i + 1) * N + j] - A[(i + 1) * N + j + 1];
    }
  }
}
```

Figure 3.22: Resulting code after all transformations

Another part of the cycles is saved due to the scheduling of the initialization assignment of C in parallel with the computation of the sum of arrays A1, B1, and B2.

The remaining transformations (Figure 3.20, lines 1-8) are performing a double fusion of the nested loops. The first fusion (Figure 3.20, lines 1 and 2) is performed on the horizontal and vertical Sobel filters (Figure 3.19 for `LBL4` and `LBL5`). The cache misses and total memory reads decreased because we can reuse elements of $A[u, v]$ where $u \in [i-1, i+1]$ and $v \in [j-1, j+1]$ from the `LBL4` and `LBL5` statements. The number of cycles decreased due to the elimination of one nested loop control overhead and to the sharing of address calculations for the read addresses of A and write address of B. Spark preserves the `WAW` (write after write) dependences. Hence, the writes of the elements of B1 and B2 are scheduled sequentially. All the reads are scheduled in parallel. Similar improvements are obtained with the remaining transformations (Figure 3.20, lines 3-8).

With unrolling, `Spark` can perform parallelization and loop pipelining more efficiently. This explains the improvement in the number of clock cycles. However, the transformations performed with `WRaP-IT` (which does not include any unroll) already leads to better performance on each metrics. This experiment presents dramatic improvements: 5.57 times speedup in the number of total cycles, 6.97 times better in the number of cache misses. An analysis of the final code, for the reuse distances of the elements of the array `A`, can show that, after the optimizations, the synthesized architecture has only a very small working set that fits in a few lines of cache.

The synthesis with `Spark` was performed with the following hardware constraints: 2000 arithmetic logic unit (ALU), 1000 multipliers, 2000 comparators, 2000 shifters, 5000 logic gates. As can be seen from the table in Figure 3.23, all the transformations decrease the number of clock cycles. There is one exception after the `LBL5` fusion. The number of states increased because the iteration space of the symmetric difference of the iteration domain of the `LBL5` and `LBL6` is not empty ($D_{iter}^{lbl5} \triangle D_{iter}^{lbl6} \neq \emptyset$), and thus there are an epilog and an prolog that increase the number of states. Since the state machine generated by `Spark` is monolithic, the bigger the number of states, the lower the possible running frequencies. For this reason, the number of states could be used as a measure of performance of the generated hardware accelerator. More precisely, the number of ideal cycles must be divided by the frequency to get a good view of the performances of the system. Then, to take into account memory transfers, one should again add the number of cache misses multiplied by the memory latency (40).

Design Flow	`WRaP-IT` transformation	cache miss	memory (#bytes)	#cycles ideal	#cycles real	FSM states
`Spark` alone	-	4 364	219 200	273 077	447 637	53
`Spark` + `WRaP-IT`	code motion	4 051	199 168	195 548	357 588	41
`Spark` + `WRaP-IT`	+ double fuse LBL4	3 738	189 152	147 233	296 753	32
`Spark` + `WRaP-IT`	+ double fuse LBL3	3 425	179 136	98 918	235 918	23
`Spark` + `WRaP-IT`	+ double fuse LBL5	2 504	149 664	79514	179 674	28
`Spark` + `WRaP-IT`	+ double fuse LBL1	1 565	90 144	59 213	120 813	22
`Spark` + `WRaP-IT`	+ scalarization	626	30 048	55 302	80 342	33
`Spark` + `WRaP-IT`	+ unroll j by 20	633	30272	15 302	40 622	115
`Spark` alone	unroll j 20	4 364	219 200	74 969	249 256	173

Figure 3.23: Performance improvements on edge detection (cache size of 8 kB, line of 32 B)

3.3.4 Second example: h264/h263 YUV to RGB

We now present the same performance improvements on an example taken from MediaBench II Benchmark [5]: a H263 decoder. Profiling of the decoder (see Appendix, Section 7.2.1) shows that an important part of the execution time (62.33%) is taken by the YUV-to-RGB conversion. This color space conversion is present on all the video codecs like `mpeg4` and image ones as `jpeg`. On a desktop computer, this conversion is performed by the video card when the frame is displayed. However, when image processing filters have to be applied, the conversion is performed on the processor. Most of the embedded systems do not have a video card and this conversion is done either by the `CPU`, a `VLIW` processor, or a hardware accelerator. The code consists of multiple function calls (Figure 3.24). Inlining was used as a preprocessing step to get all computations in a single function, see the code in Appendix, Section 7.2.2. The first two function calls are CONV420TO422 for the U space and V space. The function represents a vertical interpolation using 6 adjacent vertical elements. An unrolled version of the function that converts the V color space is presented in Figure 3.25. The code consists of two nested loops iterating over a square domain. The computational loop body consists of a weighted vertical interpolation that uses a small array `clp` for look-up. The loop body has also six `if` statements that are the boarder guards.

The CONV420TO422 calls are followed by two CONV422TO444 calls that use the V422 and U422 just computed. The functions perform a horizontal interpolation and the code is similar to the vertical interpolation one. The resulting V444 and U444 representations are used to generate a RGB color space representation.

```
void yuvrgb(int advance)
{
   /* ..........*/
   /* Conversion from YUV420 to YUV422 color space */
   conv420to422(srcU, u422);
   conv420to422(srcV, v422);
   /* Conversion from YUV422 to YUV444 color space */
   conv422to444(u422, u444);
   conv422to444(v422, v444);
   /* Conversion from YUV444 to RGB color space */
   convYUV444toRGB(u444, v444, rgb);
}
```

Figure 3.24: Conversion from YUV to RGB color space

The analysis of the data computation and flow is presented in Figure 3.26. As can be observed, a 7×7 block of U420 is used to generate a single cell of RGB (7 vertical elements of U420 are used to compute 2 vertical elements in U422, and 7 horizontal elements of U422 then define U444). Between the transformations, there are 4 temporary data storage arrays U422, U444, V422, V444.

The synthesis and simulation were done on a fixed 1000x1000 pixels image. The synthesis results of Figure 3.27 are obtained by applying various loop transformations as summarized below:

- `Spark_Only`: no use of `WRaP-IT`.
- `WRaP-IT0`: Passing through `WRaP-IT` without loop transformations.
- `WRaP-IT1`: loop interchange on vertical interpolation (`U`).
- `WRaP-IT2`: `WRaP-IT1` plus loop interchange on vertical interpolation (`V`).

5. `http://euler.slu.edu/~fritts/mediabench/mb2/`

```
 /* conv420to422(srcV,v422); */

  /* intra frame */
  for (i=0; i < w/2; i++) {
__URUK_BEG2:  for (j=0; j < h/2; j++) {
__URUK_IF11:    j21 = j*2;
__URUK_IF12:    if(j<3)
__URUK_IF121:     jm31 = 0;
__URUK_IF122:   else jm31 = j-3;
__URUK_IF13:    if(j<2)
__URUK_IF131:     jm21 = 0;
__URUK_IF132:   else jm21 = j-2;
__URUK_IF14:    if(j<1)
__URUK_IF141:     jm11 = 0;
__URUK_IF142:   else jm11 = j-1;
__URUK_IF15:    if(j<h-1)
__URUK_IF151:     jp11 = j+1;
__URUK_IF152:   else jp11 = h-1;
__URUK_IF16:    if(j<h-2)
__URUK_IF161:     jp21 = j+2;
__URUK_IF162:   else jp21 = h-1;
__URUK_IF17:    if(j<h-3)
__URUK_IF171:     jp31 = j+3;
__URUK_IF172:   else jp31 = h-1;

__URUK_IF18:    v422[dst_index01 + w*j21] =
                  clp[(int)(3*srcV[src_index01 + w*jm31]
                  -16*srcV[src_index01+ w*jm21]
                  +67*srcV[src_index01 + w*jm11]
                  +227*srcV[src_index01 + w*j]
                  -32*srcV[src_index01 + w*jp11]
                  +7*srcV[src_index01 + w*jp21]+128)/256];

__URUK_IF19:    v422[dst_index01 + w*(j21+1)] =
                  clp[(int)(3*srcV[src_index01 + w*jp31]
                  -16*srcV[src_index01 + w*jp21]
                  +67*srcV[src_index01 + w*jp11]
                  +227*srcV[src_index01 + w*j]
                  -32*srcV[src_index01 + w*jm11]
                  +7*srcV[src_index01 + w*jm21]+128)/256];
              }
    src_index01 = src_index01 + 1;
    dst_index01 = dst_index01 + 1;
  }
```

Figure 3.25: Conversion of V space from 420 to 422

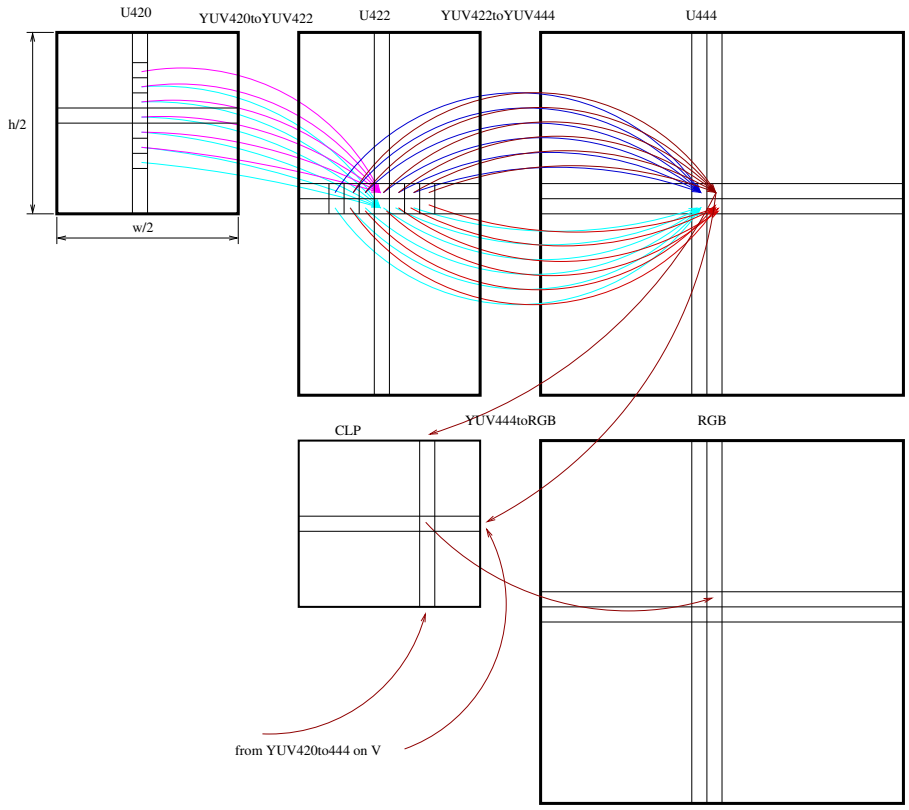


Figure 3.26: Data reuse distance of YUV420toRGB conversion

– `WRaP-IT3`: `WRaP-IT2` plus loop transformations (fusion, strip-mining, and shifting) on horizontal and vertical interpolations on `U` and `V`.
– `All_transf`: `WRaP-IT3` plus loop transformations (code motion, strip-mining, loop interchange) on `U` and `V` and `RGB` loops and multiple loop fusions.

Results and discussions We now give the performance results obtained from the `Spark` synthesis report (number of clock cycles) and from the `dineroIV` cache simulator using the C code generated by `Spark`. Figure 3.27 gives the results for cache behavior (cache misses and effective read/write (R/W) from memory), total number of clock cycles, and the FSM size of the generated hardware.

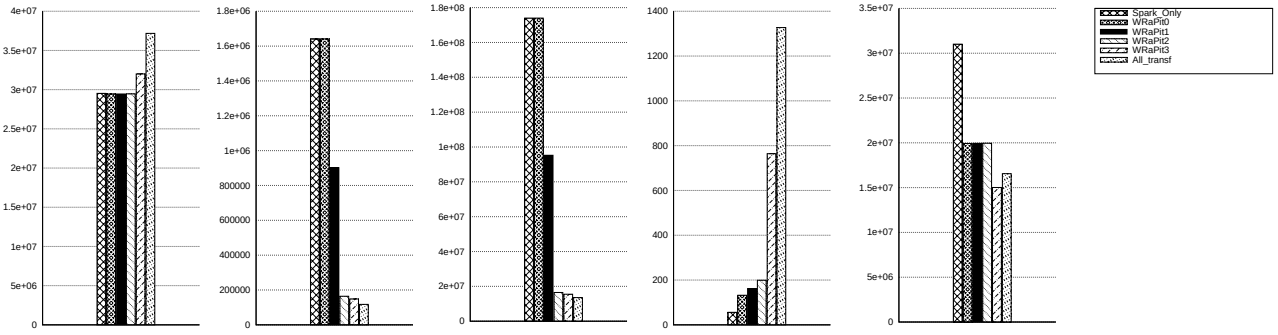


Figure 3.27: Synthesis results after applying transformations on YUV-to-RGB conversion

There is an improvement in the number of cycles between `Spark_only` and `WRaP-IT0` transformations because, in the original code, a lot of `if` statements are used to check border conditions. Each `if` statement is dividing the iteration space in two parts. `CLooG`, the loop generator of `WRaP-IT`, will generate a code for each part. The resulting code does not have any `if` statements and their computational control overhead disappeared.

The `WRaP-IT` 1 transformation performs a loop interchange of the two loops of the vertical interpolation on `U`. Because the original interpolation is done vertically and the array elements are stored horizontally, the interchange improves cache miss ratio thanks to spatial locality. The access distance to the same line of cache is reduced from $\frac{h}{2}$ to 7 which does not depend on the height of the frame and is smaller than the number of lines of the used cache. The `WRaP-IT` 2 transformation brings similar improvements.

The `WRaP-IT` 3 transformation improves the temporal locality between the vertical interpolation and the horizontal one thanks to loop fusion. Because of the data dependences between them, a shift by 2 iterations was performed before, on the loop of the vertical interpolation, to enable this fusion. An important improvement of the number of execution cycles can be observed after this transformation because most of the loop control hardware is shared between the two fused loops. There is also a sharing of the array access index calculations that are in the critical path. This transformation is also used to increase the level of the parallelism inside the loops that `Spark` can then easily explore. A loop fusion will not possibly increase the cache miss ratio if the two fused loops have the same working set and there is a locality in their time-sliced accesses. In other cases, loop fusion can perturb smooth cache operations by increasing the working set of data and thus increasing the cache miss ratio. The best performance are obtained if a loop fusion enables the scalarization transformation. The scalarization enables the storage of the data in the local

registers, and thus reduces the number of data access ports. The strip-mining and interchange [6] (see Appendix, Section 7.2.3) were used to increase the locality and to align the accesses of the data produced in the function `conv420to422` and the data consumed in the function `conv422to444`.

The last transformations (`All_transf`) improves the temporal locality of the writes for the previously-obtained nested loops. The code motion is used to move a block of the code located between the functions `conv420to444` and `conv444toRGB` so as to enable the fusion of these two functions. This block produces data used in the `conv444toRGB` function and it can be placed at the very beginning of the process. Tiling (through strip-mine and interchange) is applied on the two loops of the function `conv444toRGB` and all loops are then fused, in order to align the iteration spaces of the data producer and data consumer.

With these last transformations (`All_transf`), the cache misses are reduced even more. However, as it can be observed from Figure 3.27, the number of execution cycles increases as compared to `WRaP-IT` 3. In this case, the performance degradation of the blocking transformation can be very clearly observed. Each blocking transformation is generating a new loop nest inside the original loop nest thus increasing the overhead. The cache locality improvement brought by the blocking transformation has to be balanced with the degradation of the number of cycles.

As previously mentioned, the hardware generated by `Spark` cannot be synthesized directly, however one can have a rough idea of its area complexity: the hardware consists of an execution unit, the same for each design, and a finite state machine controlling the execution unit. Figure 3.27 gives the number of states of the FSM. It increases after each new transformation, especially for `WRaP-IT` 3 and `All_transf`. Here again, a trade-off must be done between cache performance and hardware complexity.

The goal of these experiments was to show that important improvements can be obtained by using loop transformations as a front-end to HLS, especially if these loop transformations are guided by the user (automatic loop optimization is still not applicable to HLS). Similar improvements were obtained in [60]. The main benefit of our approach is that it provides the flexibility of adding a powerful loop transformation front-end to existing HLS frameworks. This is useful especially when the HLS framework is provided in binary format, which is usually the case. Of course, this applies only if the synthesis framework accepts ANSI-C as input. `WRaP-IT` being now integrated into the GCC compiler, GCC could also be used instead of `Open64`.

The original code as well as the `Uruk` script is given in the appendix. The transformations were found by hand and the results measured by experiments as explained previously. It is easy to be convinced that loop transformations can be useful to improve HLS designs, as they affect the way the memory is accessed. For example, some tools do incorporate such transformations. However, to our knowledge, the belief that they can be useful at source level, before using an HLS tool, was not really supported by experimental results in a complete setting. Our experiments provide such an evidence. It also shows that trade-offs must be considered during the design: for example, tiling can optimize memory transfers but it can also make the circuit more complex, which means a larger surface but also a lower frequency. We believe that such trade-offs should be left to the designer and not to heuristics within the HLS tool. However, it is not reasonable to expect the designer to do such transformations by hand. Our approach, which uses a `Uruk` script, makes the code generation automatic.

6. also known as blocking or tiling

3.4 Conclusions

In this chapter, we presented two studies that improve the performance of automatically-generated (by HLS tools) hardware accelerators, one for `MMAlpha`, one for `Spark`. Both tools generate circuits without local memories, only registers. To store larger set of data, we exploited the fact that the target FPGA platform has a processor, with a cache. In the next chapter, we consider a more standard situation where the HLS tool can generate local memories, which are more power-efficient than a cache and will directly access the external memory, i.e., with no intermediate cache.

The first study is the memory architecture and interface generation for a matrix-matrix complex multiplication IP, synthesized using the MMAlpha HLS tool. As we have seen, an efficient synthesis is not enough to obtain good overall performances when integrating the hardware accelerator into a complex system. The data availability problem causes a significant performance bottleneck. The solution is to have a local memory architecture exploiting the knowledge of the IP data access patterns and enabling data reuse, and a dedicated fast link to the location where data is stored. Building and verifying the memory architecture and external interface by hand for the generated `MMAlpha` IP is difficult. Most of the time, the design should be changed after each minor modification of the IP proving a very inflexible design method. An automatic generation of this memory architecture and interface is thus becoming mandatory for a HLS system.

The second study we performed shows the interest of source-to-source code transformations, in front of a HLS tool. This step has an advantage over the previous solution for MMAlpha design, which acts as a slave. Here, loop transformations push all the reuse "intelligence" from the interface into the hardware accelerator itself, which acts as a master: the order of its accesses brings lines to the cache, which then produces spatial and temporal reuse. The choice of transformations depends slightly on the HLS tool but they do not have to be integrated into the complex IR representation of the tool (if, in the best case, the sources of the HLS tool are available). We showed that, with such transformations, it is possible to greatly improve the data reuse and optimize the internal structure of the hardware accelerator. Data reuse minimizes the external data requirements thus improving the performances when the IP performances are limited by an external data link. (Such transformations can also optimize accesses to local memories, but `Spark` does not have such memories.)

This method however relies greatly on the cache dedicated data management system. Unfortunately, in most of today's systems, the cache cannot be used for this purpose. When the processor resides in a different chip, its cache is not accessible. It may also be inaccessible even when it resides in the same chip. Indeed, usually, the soft-core and hard-core processors are provided as IPs, which do not allow access to their caches from the outside world. Maybe in the future, a multi-core processor or system will have a reconfigurable fabric having access to the cache, much like the streaming SIMD extensions (SSE) instructions found in today processors.

The two previous studies show that to improve high-level synthesis, i.e., to improve the synthesis results and to ease the synthesis process, we need a tool that can generate automatically both the interface and the local memory architecture. Furthermore, at the same time, we would like this tool to be a source-to-source tool. Customized memory architectures increase the possible optimization space allowing transformations that are not possible to apply when using a typical static memory architecture. We address all these concerns in the following chapter.

Chapter 4

Local SRAM with external DDR: Altera C2H flow

4.1 Introduction and motivation

As illustrated in the previous chapter, the data availability is one of the major performance bottlenecks when synthesizing hardware accelerators. In general, one of the objectives of the designer is to be able to generate a hardware accelerator with optimal throughput, i.e., a circuit where the data transfer rate (either to local memory or to external storage) is at its maximum, in other words, a solution where bandwidth is the limiting factor. The circuit itself should then have all the necessary hardware to be able to consume and produce data in a fast-enough manner. No need to have a super-parallel implementation or a very-high-frequency solution if the hardware accelerator spends its time to stall, waiting for data to arrive [1].

When using HLS tool to get such designs, the problem is the same: this is still the responsibility of the user to write the code in such a way the bandwidth is saturated. Indeed, in general, the tool is not able to transform the code to reach such a solution. For example, if the interface of the hardware accelerator with the external world is specified, in the C code, through FIFOs, it is too late to do anything. The tool would need to know both the caller function and the callee function to be able to re-organize the code and produce different access patterns. Thus, even if a HLS tool is used, the designer has still to decompose the application into smaller communicating processes, to define the adequate memory organization or communicating buffers, and to integrate all processes in one complete design with suitable synchronization mechanisms. This task is extremely difficult, time-consuming, and error-prone. As explained in the introductory chapter, this is one of the reasons why designers give up using HLS tools. This is also what we illustrated in the previous chapter. For MMAlpha, interfacing was feasible though difficult, but it was not as efficient as we could expect. For Spark, this was in a way worse as we did not even succeed to interface it completely: we needed to rely on simulation to evaluate the benefit that loop transformations could have if Spark was designed so that I/O pins are more exploitable.

Completing the study we did in the previous chapter for high-level transformations with Spark

1. As shown in Section 3.2, this was one of the problems with MMAlpha. If a tile larger than $4 \times 4 \times 4$ is mapped to the FPGA, the circuit needs too many resources (multipliers) and, anyway, it is much too powerful compared to the available bandwidth. Techniques to slow it down, while sharing resources, are required. This is what the research on LSGP partitioning tries to address. It is a general situation: the circuit should be adapted to the available bandwidth.

and WRaP-IT, the goal of this chapter is to demonstrate that program transformations, in particular loop transformations, are needed in front of HLS tools, if one wants to better exploit the available bandwidth, and that these transformations can be automated. We believe that this is a *sine qua non* condition for making HLS tools a usable thus viable solution to hardware design, in the same way a traditional front-end compiler can perform high-level optimizations on top of any assembly-code optimizer. We want to contribute to make this possible for hardware accelerator generation too.

The challenge is thus to be able to perform code optimizations at C level that are directly beneficial when used in front of an HLS tool, with no modification of the tool itself. For that however, the HLS tool should behave in a predictable way, in particular with predictable behavior and predictable performances. This may require to be able to pass higher-level information to the (back-end) HLS tool, such as non-aliasing information, or to use the tool in a very specific and controllable way. Furthermore, the tool should provide an integrated interface with more flexible accesses to the outside world than just FIFOs. Our study is done with the `C2H` Altera tool, which has such specificities. Our goal is to try to optimize accelerators with external accesses to a DDR memory, always keeping in mind that we should be able to automate any code transformation that we first apply by hand, until we identify the right way to do it.

Our contributions, summarized in [25], are the following:

1. We analyze `C2H` and we identify the features that make DDR optimizations feasible or hard to perform, in general, and with `C2H` in particular.
2. We propose a technique based on tiling, the generation of communicating processes, and of software pipelining that can lead to fully-optimized DDR accesses.
3. We show how our scheme can be automated. The complete automation will be detailed in the next chapter, using analysis and transformation techniques, primarily developed in the context of high-performance compilation.

Target architecture In Chapter 3, we used a tightly-coupled architecture, with a cache, for the hardware accelerator placement. This architecture selection has multiple advantages, however it cannot be used when the cache memory of the processor is not accessible. This can be either because the processor is located in a separate chip or because the processor is provided as an IP module. In this chapter, we use a distributed-type architecture as depicted in Figure 3.1, in the previous chapter. This type of architecture is used by the Altera hardware design flow (Figure 4.1).

The system is composed of a processor, called `Nios II`. It is a soft-core processor that can be mapped onto Altera's FPGA configurable logic. The rest of the system is composed of IP cores. They are connected by means of a switch fabric `Avalon` interconnect. Each of the IP cores can have multiple ports that are used for data exchange. The ports can communicate using different types of interfaces such as Avalon MM (Avalon-MM), Avalon ST (Avalon-ST), Avalon-MM tri-state etc. Avalon-MM are address-based R/W interfaces based on master-slave connections. Avalon-ST is an interface used for unidirectional stream of data. Avalon-MM tri-state interface is used for communicating with off-chip peripherals.

In this chapter, we are interested in Avalon-MM only. This interface is used to connect peripherals like the `Nios II`, local FIFOs, local DRAM and DDR controllers, and hardware accelerator generated using the `C2H` tool. The advantage of this interface over a bus interface is the point-to-point connection. Multiple components can communicate in parallel without any usual bus contention. The multiplexers are used to perform the required connection. If multiple master

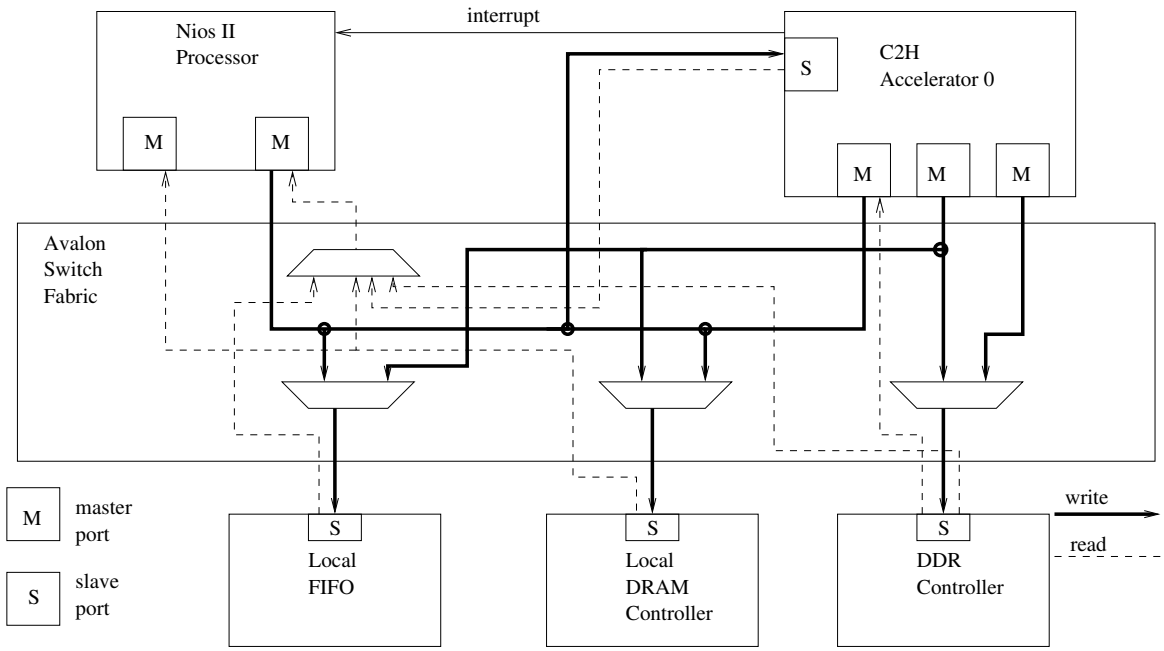


Figure 4.1: Altera's Nios II architecture

ports require an access to a single slave port, an arbitration is performed. Each slave port has a maximum number of arbitration shares. They specify the number of operations that a master can perform without the need for re-arbitration. The user can assign different arbitration shares (with the pragma `arbitration_share` described hereafter) per master port that accesses the same share thus prioritizing some master ports or allowing an atomic burst transfer of a specific length if the corresponding requests are present in the pool of requests. These shares do not enforce a minimal transfer length: the master can cease the transfer at any time.

Each port of the interface can have dedicated independent write and read connections. Read connections can perform a pipelined read transfer: a master port can send multiple data requests without waiting for the result to be sent by the slave. The slave peripheral should send the request results in the same order it received them. The wait penalty is paid only once at the beginning of the transfer sequence, and is called pipeline latency. Write transfers are not pipelined by nature, data is sent together with the write request.

Finally, there are also multiple signals such as `dataavailable`, `waitrequest`, and `readyfordata` to control the flow of data. Thanks to suitable pragmas, these signals are automatically generated by C-to-Hardware Acceleration (C2H). This is very useful for example to implement a blocking read from a Avalon-MM FIFO memory.

4.2 Altera C2H important features

Some HLS tools rely on a somehow straightforward mechanism to map the C syntax elements to their corresponding hardware parts, for example a loop is encoded into a simple FSM and not unrolled, a scalar variable is mapped to a register and an array to a local memory with no memory reorganization, etc. This can be a disadvantage from the point of view of automatic code synthesis.

Indeed, to get good performance results, the user should know very well the underlying synthesis concepts and methods together with the hardware structures that the tool generates from the C-level algorithmic description. It implies that the user is required to have a good expertise and knowledge of the low-level hardware design. This is a major issue of a majority of the HLS tools.

However, a direct correspondence between software and hardware can also be an advantage as it gives a mean to control what the HLS tool will produce. This is the approach of Ugh (user-guided HLS) [13], where each scalar variable is register-allocated at C level by the user to guide the hardware generation. This is also an advantage if the HLS tool is used with a source-to-source preprocessing tool, as we do. The more information the preprocessing could give to the HLS tool, the more precise version of the required hardware will be obtained. This is one of the reasons why we chose `C2H`, the HLS tool designed by Altera Corporation, as a target for our source-level optimizations.

We now summarize the main features of `C2H` that will impact our technique. Further details are provided in the Nios II C-to-Hardware Acceleration Compiler (`C2H`) User Guide [22] and in the Altera's Embedded Design Handbook [21].

4.2.1 Input language and interface

`C2H` supports most of the C constructs such as pointers, structures, loops, and subfunction calls. It is integrated in the development flow of the Altera FPGAs and works with `Quartus II`, `SOPC Builder`, and `Nios II` integrated development environement (IDE). This makes the integration in the complete software/hardware design much easier. Being connected with other modules of the system thanks to the `Avalon` system interconnect fabric, the hardware accelerator can communicate, not only by means of FIFOs, but also using the mapped address space connection. In fact, this is the default communication method. [2] It eliminates the need for the hardware interface designer to accommodate and integrate the accelerator to the complete system as required in many popular HLS tools described in [47]. This communication interface also supports pipelined memory accesses, which is mandatory to achieve good performance of the final system. Indeed, as we explained before, the performances are most of the time limited by the external data transfer bandwidth and rate, and not by the lack of parallelism within the function to be accelerated.

`C2H` is aimed to create a custom hardware accelerator that offloads the `Nios II` processor (Figure 4.1). The description of the hardware accelerator is passed as a function written in the ANSI-C language from which a specialized hardware accelerator implementing it will be generated. Multiple hardware accelerators can be generated in the same system. The accelerators are controlled by using a slave port. In order to start an accelerator, the function input parameters are sent together with a start token. During the execution, the processor can be blocked in a polling read to the slave, waiting for it to finish the execution. This is the default method of execution, ensuring the correct sequential execution of the code run in the processor and in hardware. The call to the accelerated function can be also implemented using interrupts, specified by a pragma. After a call to such a function, the processor will continue its execution without blocking. Just like interrupt servicing on systems without operating systems, the user just has to write an interrupt routine, which will implement the synchronization if needed. This way, the processor can run multiple accelerators in parallel.

The user should take care of the data cache coherency in the system generated by `C2H`. By

2. To communicate using a FIFO, there should be a FIFO instantiated in the system by means of the `SOPC Builder` Builder and connected using a memory mapped interface to the specified port of the accelerator.

default, the processor is forced to flush its full cache before starting a hardware accelerator. If the working sets of the processor and the hardware accelerator are disjoint, then the user can choose not to flush the cache.

4.2.2 Mapping to hardware

`C2H` uses a somehow direct syntactic translation to hardware. For example, it generates a dedicated master port connected to the program memory (we use only an external DDR) each time it encounters a pointer or array access to data located outside of the function. Local scalar variables are synthesized as internal registers. Uninitialized local arrays are mapped to on-chip memories and initialized arrays are mapped to the DDR.

As explained in C2H documentation, the arithmetic and logical operations are mapped to dedicated modules without any resource sharing, in other words, there is a hardware resource for each textual operator, even if the code is purely sequential. In particular, in `C2H`, unrolling a code multiply the resource need. Unrolling a fully sequential loop uses more resources without increasing the performances. The unrolled basic block instructions are executed sequentially on each of the dedicated parallel hardware units. The arithmetic operations such as shifts by a constant, multiplications and divisions by a power of two, bitwise operations by a constant, are implemented as wires in hardware without consuming any resources. Multipliers and adders consume arithmetic resources. Implementing a very long expression is performed by chaining these resources. This type of expression should be avoided as long timing paths degrade the frequency of the resulted circuit. Again, C2H does not cut such paths into smaller paths, with intermediate storage: such control is left (on purpose) to the user. Finally, multiple assignments to the same variable also degrade the circuit frequency. This is because each of the assigned values should be passed to an input of a multiplexer that selects which value to pass.

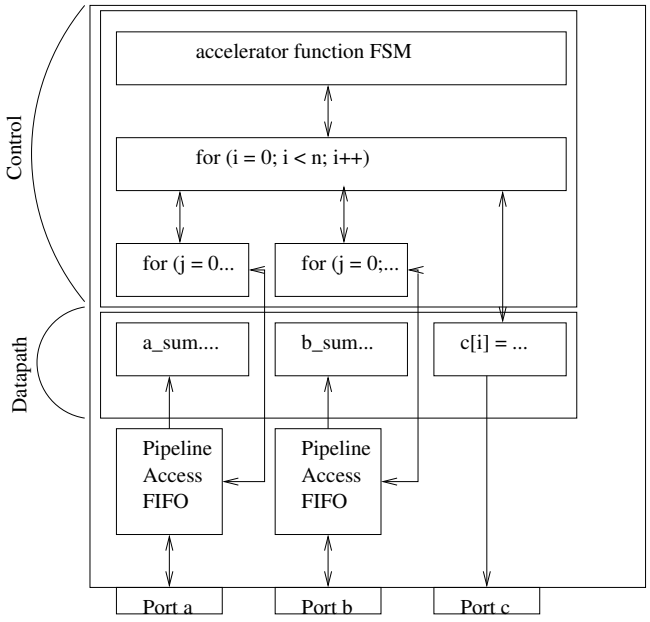
There are several type of loops: `for`, `do`, and `while` loops. Each of these loops has a termination condition. `C2H` will use a `if` structure statement to verify this condition. It will always try to pipeline the loops in order to increase the throughput of the system. In most cases, the loop control statement will not add an overhead to the loop pipeline, being executed in parallel to the loop basic block. However, when the loop control expression has a data dependence from the loop basic block, a performance penalty is of course paid due to the loop overhead. But more important is the way loops are scheduled and, in particular nested loops, as we explain in Section 4.2.3.

`Switch` statements are mapped to an equivalent structure composed of `if` statements. However, there is some subtle differences between a `switch` and an `if` because the compiler knows that the `then` part and the `else` part cannot both occur while it seems to analyze different `case` statements as if they can all occur, even when each one has an ending `break`, i.e., when only one case can occur.

The function calls are mapped by the `C2H` compiler to separate accelerators. When a call is encountered, the current state machine (the caller) passes the token to the called function (the callee) and stalls its execution until the state machine of the callee returns. In some cases, when the execution time of the callee can be determined at compilation time (no loops, no branches, etc.), the call to the function is pipelined. The function calls can be used to implement resource sharing techniques. Indeed, as explained earlier, `C2H` maps each operation to a dedicated hardware instance. If operations are encapsulated in a function call, `C2H` creates only one hardware instance of the function. The calls to this function are then sequentialized, thus providing an indirect way of scheduling and resource sharing fully implemented and controlled by the user.

4.2.3 Scheduling: inner control with state machines

In Figure 4.2, the generated architecture for an ad-hoc example, called *accelerator*, is depicted. Each `for` loop has its own state machine. The `i loop` is controlled by the *accelerator* function FSM and it controls the two `j` loops. In this example, there is no data dependence between the statements of the loop computing `a_sum` and the statements of the loop computing `b_sum`, therefore these loops are scheduled in parallel. The data is fetched from the outside of the accelerator using dedicated ports. The data-path is connected to the ports via FIFOs that are used to store the access requests for the pipelined access execution. These FIFOs are implicit for the user, i.e., they are automatically generated by the compiler.




```
int accelerator  (int __restrict__ *a,
int __restrict__ *b, int __restrict__ *c)
{
    int i, j, k;
    int a_sum, b_sum;
  for (i = 0; i < n; i++)
  {
      for (j = 0; j < m0; j++)
          a_sum += a[j];
      for (k = 0; k < m1; k++)
          b_sum+= b[k];
       c[i] = a_sum + b_sum;
   }
}
```

Figure 4.2: Architecture of the generated hardware accelerator and code for the *accelerator* function

Figure 4.3 presents the decomposition and the schedule of the read of array element a[0] as well as the corresponding accumulation from the `j` loop. The read is decomposed into several sub-instructions, in particular a request and a receive instructions. Because of the time required to fetch the data, they are separated by a waiting time, which is evaluated by the compiler. This waiting time is encoded into the FSM using a state for each wait cycles. In this example, the state machine has 38 states encoding the waiting time. If this latency is higher at run-time, the FSM stalls, i.e., the state that contains the receive instruction stops until the data is received.

In order to hide the access latency, `C2H` will software-pipeline the loop as shown in Figure 4.4, using a "modulo schedule". Since the loop `j` does not have loop-carried dependences, the *cycles per loop iteration (CPLI)* is equal to 1. The CPLI represents the number of states of the current FSM after which it is possible to schedule a new iteration of the loop. In other words, for each instruction, there is an occurrence every CPLI cycles. Here, `a[i+1]_req` is scheduled at the next state, immediately after `a[i]_req`. When the memory access latency is smaller than $m_0 * CPLI$ (remember that m_0 is the number of iterations in the loop, see Figure 4.3), after some initial iterations, the state machine sends a read request and receives a data at the same state. Note that

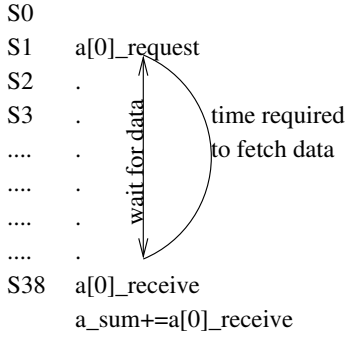


Figure 4.3: DDR access latency incorporated in the schedule

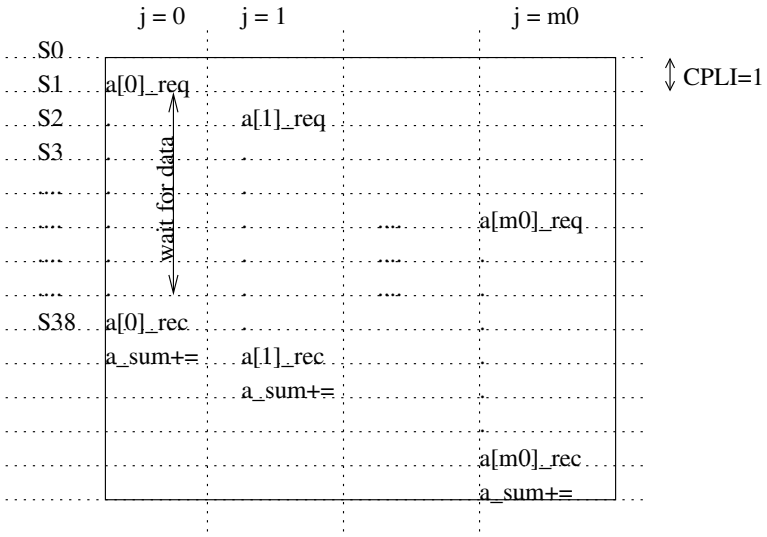


Figure 4.4: Pipelined execution with DDR latency

the notion of CPLI is similar to the notion of *initiation interval* used in compilers for programmable processors. However, there is a small difference: as will be explained hereafter, software pipelining is applied to all loops, when possible, and not only to innermost loops. The consequence is that the "cycle" used in CPLI is not a real-time cycle, it is a virtual cycle, i.e., a stage of the FSM: stages can indeed have different durations and even unpredictable durations. In other words, even if Altera calls it CPLI, it would be more accurate to call it "stages per loop iteration".

To reduce the CPLI, `C2H` often needs to perform variable replications, as illustrated by the small example of Figure 4.5. The value of the loop iterator variable `i` is used in the `i++` statement and in the `i<N` statement. Because of this, the loop cannot be pipelined with CPLI equal to 1 since `i++` and `i<N` cannot be executed at the same time. Indeed, the former operates on the old value and they access the same variable in R/W at the same time. Actually, there is a cycle of length 2, with the flow dependence from `i++` to `i<N` and the anti-dependence from `i<N` to `i++`. The loop is pipelined with an inefficient $CPLI = 2$, just because of the loop counter, not because of the computations. In order to be able to pipeline the loop with CPLI equal to 1, the variable `i` that does not allow a good pipelining is replicated. One way to do it is given in Figure 4.6.

If a loop contains another loop, the inner loop is considered as an atomic instruction for defining the software pipelining of the outer one, leading to a hierarchical software-pipelined execution of loops. Figure 4.7 presents the software pipeline of the outer loop (loop iterating over `i`). The loop statements are scheduled using 6 states. At state `S0`, the initialization of the inner loop iterators are scheduled, the first loop execution condition test is computed, and the increment of the outer loop counter is performed. At the state `S1`, the increment of the replicated `i` is performed, together with verification of the loop termination condition using the original `i`. At state `S2`, the loop termination condition is verified using the replicated `i` and the loop FSMs of `j` and `k` are started. The synchronization of the outer-loop FSM with the inner-loop FSM is performed using three states scheduled one after each other (here `S2`, `S3` and `S4`). In the first state, the outer-loop FSM starts the execution of the inner-loop FSM and passes to the next state. During the second state, the

outer-loop FSM waits for the inner-loop FSM to finish, after which the outer-loop FSM passes to the next state. In our example, the state waiting for the loops `j` and `k` to finish is `S3`. Finally, at the third state (here `S4`), the outer-loop FSM can use the data on which the inner-loop FSM was performing computations. In our example, at the state `S4`, both of the loops `j` and `k` finished their execution, the `a_sum` and `b_sum` values are valid. The sum of the accumulated values `a_sum` and `b_sum` can then be performed. At the state `S5`, the sum of the accumulated values using replicated values is performed.

As can be observed, the outer loop is pipelined with aCPLI equal to 1. The "latency" of the loop, as provided by the Altera compiler, is equal to 6 cycles. As mentioned before, this latency and this CPLI of the outer-loop do not depend on stalls and on latencies of the inner-loop FSMs. These values can only be used as a performance metric for the outer-loop FSM: they define the "virtual" cycles of the FSM and not the "real" hardware clock cycles of the accelerator.

In Figure 4.8, a detailed pipelined execution of the accelerator is presented. It includes the pipeline of the two inner-loops `j` and `k`. The two loops are started in parallel. In the figure, we assume $m1 > m0$ and the execution pipeline of the loop `k` is longer. Let the execution time of the loops `j` and `k` be $t_j(i)$ and $t_k(i)$ respectively. The FSM of the outer loop `i` will pass to the state 4 only when both the `i` and `j` loops finish their execution, i.e., after $\max(t_j(i), t_k(i))$. Although the CPLI of the loop `i` is 1, its real execution time depends on the execution time of the inner loops `j` and `k`. Both of the loops `j` and `k` have the DDR access latency incorporated in their pipeline. However, when this statically computed latency is smaller than the actual run-time latency of the access, the state machine will stall. The execution time of the loops `j` and `k` can thus vary. If a stall is encountered in the `j` loop, and $t_j(i) < t_k(i)$ still holds, then the stall will be (possibly partially) hidden by the execution of the loop `k`. Otherwise, the stall of the loop `j` will increase the quantity $\max(t_j(i), t_k(i))$, and the execution of the state 4 of the outer-loop will be delayed.

This hierarchical FSM semantics is an important feature to consider for optimizing external DDR accesses, as illustrated in Figure 4.9. Suppose the `j` loop reads data from an external DDR. Even if its CPLI is 1, which means that one data item is scheduled to be consumed at each cycle, its total latency can be quite long, due to a transfer latency, because the inner loop needs to empty its pipeline before the outer loop can proceed to its next stage. This latency is thus paid as a penalty for each iteration of the `i` loop.

Let us see this more precisely. A pipelined loop at depth p, with n iterations, with a loop counter i from 0 to $n-1$ (without loss of generality), is characterized by its CPLI and its latency ℓ, which is the total number of states in the FSM: for $i = 0$, the first operation is started at state 0

```
for (i=0; i<N; i++)
    c+=i;
```

Figure 4.5: Variable replication in software pipelining example

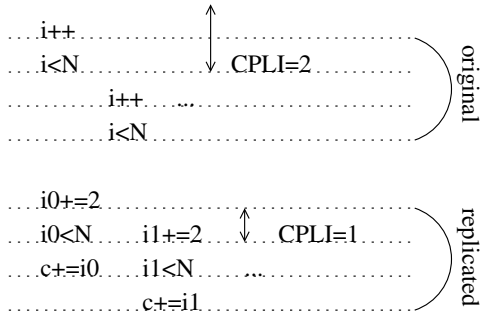


Figure 4.6: Schedule with variable replication

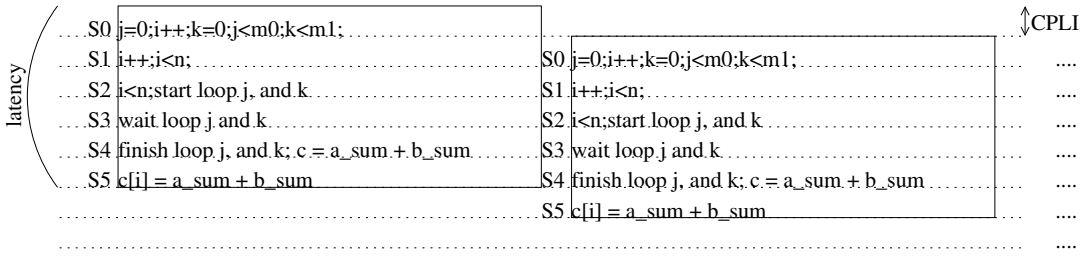


Figure 4.7: Outer-loop software pipelining

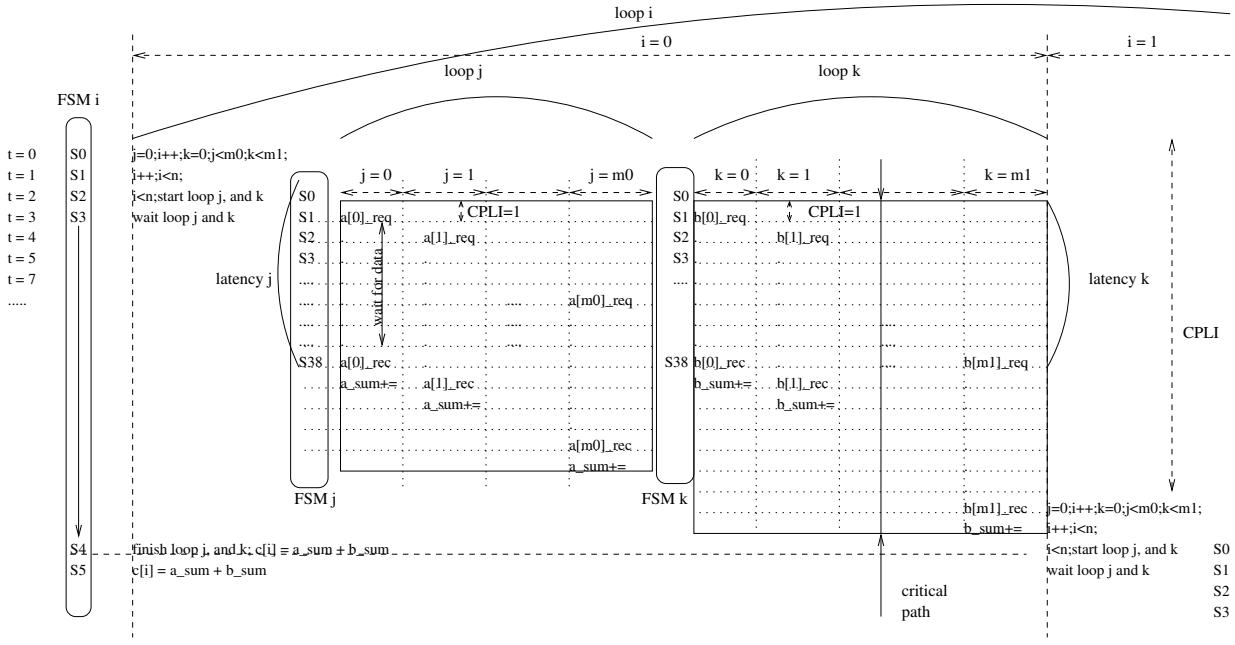


Figure 4.8: Detailed pipelined execution of the example

and the last operation at state $\ell - 1$. Each operation S at iteration i is scheduled with a standard modulo schedule, i.e., at state $\sigma(S, i) = \text{CPLI} \times i + \rho_S$ where $0 \leq \rho_S < \ell$. In other words, the first occurrence of instruction S starts at state ρ_S and one occurrence of S is run every CPLI states. The last operation of the whole loop is thus scheduled at state $\text{CPLI} \times (n-1) + \ell - 1$ and the total number of executed stages is:

$$N_{state}(p) = \text{CPLI} \times (n-1) + \ell \tag{4.1}$$

Now, each state has a duration, which depends on arithmetic operations and macro-operations, such as function calls and nested loops, at a deeper level. The duration of a state is the maximal duration of all operations scheduled at this state. Even if each operation does not induce a stall, giving a general formula is quite complicated. We do it below. Before, let us assume that the duration of each state at depth p is determined by one macro-operation at depth $p+1$ and that all these macro-operations have the same duration $N_{clk}(p+1)$. Then, we have:

$$N_{clk}(p) = N_{state}(p) \times N_{clk}(p+1)$$

If the macro-operation at depth $p+1$ is a loop with n_{p+1} iterations, with CPLI equal to 1, containing only elementary operations of duration 1, then its execution time is $(n_{p+1} - 1) + \ell_{p+1}$ according to Equation 4.1. Then:

$$N_{clk}(p) = N_{state}(p) \times n_{p+1} + N_{state}(p) \times (\ell_{p+1} - 1) \tag{4.2}$$

The first term is what would be expected with a latency of 1, the second term can be considered as an overhead due to a long latency. Of course, in practice, even in the simplest case, the situation is a bit more complicated. For example, if the outer loop has CPLI equal to 1 and contains a single macro-operation, the overhead does not appear for each state, but only n_p times. Nevertheless, Equation (4.2) gives the general idea. We will refine it when needed.

In the general case, it is more difficult to guess what will be the duration of a state. Nevertheless, we can give a quite general formula with some reasonable assumptions. We start from the general expression of $N_{clk}(p)$:

$$N_{clk}(p) = \sum_{c \in N_{state}(p)} d(c) \qquad \text{with } d(c) = \max\{d(S, i) \mid \sigma(S, i) = c\}$$

and $d(S, i)$ is the duration of the operation S at iteration i. We then identify blocks of CPLI states where all operations are running. The first such block is after state $\ell_p - 1$ (included), where all

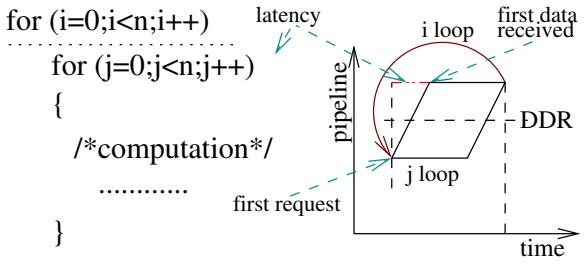


Figure 4.9: Loop latency penalty for an outer loop

operations of the first iteration at depth p have started. We round it to the next multiple of CPLI, thus we start the first block at state $s_p = \lceil (\ell_p - 1)/\text{CPLI} \rceil \times \text{CPLI}$. The last such block starts when the first operation of the last iteration at depth p is initiated, i.e., at state $(n_p - 1) \times CPLI$, thus the first non-complete block starts at state $e_p = n_p \times \text{CPLI}$. We get:

$$N_{clk}(p) = \sum_{c=0}^{s_p-1} d(c) + \sum_{c=e_p}^{N_{state}(p)} d(c) + \sum_{j=\lceil \frac{\ell_p-1}{\text{CPLI}} \rceil}^{n_p-1} \sum_{c=0}^{\text{CPLI}-1} d(c + j \times \text{CPLI})$$

Now, let us assume that the duration of each inner macro-operation is constant, i.e., it does not depend on the loop counter at depth p and no stall occurs. We write $\sigma(S, i) = \text{CPLI} \times (i + q_S) + r_S$ with $0 \leq r_S < \text{CPLI}$, as it is traditionally done in software pipelining based on modulo scheduling. This leads to:

$$N_{clk}(p) = \sum_{c=0}^{s_p-1} d(c) + \sum_{c=e_p}^{N_{state}(p)} d(c) + \sum_{j=\lceil \frac{\ell_p-1}{\text{CPLI}} \rceil}^{n_p-1} \sum_{c=0}^{\text{CPLI}-1} \max\{d(S) \mid r_S = c\}$$

The last equality is due to the fact that the set of operations scheduled at state c modulo CPLI is always the same in a "complete" block. Simplifying the expression, we get:

$$N_{clk}(p) = \sum_{c=0}^{s_p-1} d(c) + \sum_{c=e_p}^{N_{state}(p)} d(c) + (n_p - \lceil \frac{\ell_p - 1}{\text{CPLI}} \rceil) \sum_{c=0}^{\text{CPLI}-1} \max\{d(S) \mid r_S = c\}$$

Here, we assumed that $n_p \geq \lceil \frac{\ell_p-1}{\text{CPLI}} \rceil$. When n_p is large, one can see immediately the overhead due to inner macro-operations in the third term involving $\max\{d(S) \mid r_S = c\}$. This overhead is not paid for each macro-operation, it is "shared" by macro-operations scheduled at the same state modulo CPLI.

It may happen that the maximum (critical) path has multiple successive loops that cannot be run in parallel either because of some data dependences to be respected or because of some resource sharing. Figure 4.10 shows the pipeline of the same example but without the `restrict` keyword (see explanations on this pragma in Section 4.2.4) on the array pointers `a` and `b`. Since `C2H` does not know if the pointers alias or not, the two loops are scheduled sequentially. The critical path in this example contains the two loops `j` and `k`. In other words, the execution time of nested loops can be increased significantly by the latency of the inner loops. If the inner loops are scheduled in parallel, only the latency of one loop will be paid at each iteration of the outer loop. If the loops are scheduled sequentially, the sum of the latencies will be paid for each outer-loop iteration as depicted in Figure 4.11.

4.2.4 Pragmas for aliasing and connections

As many tools, the dependence analysis of `C2H` is limited to an analysis of "names" instead of memory locations. In particular, it has no dependence analysis of array elements, unlike Pico Express, which relies on the Omega Library (see [47, Chap. 4]). For example, `C2H` is unable to pipeline the following loop due to a false dependence:

```
for (i = 0; i < n; i++)
    a[i] = a[i] + 1;
```

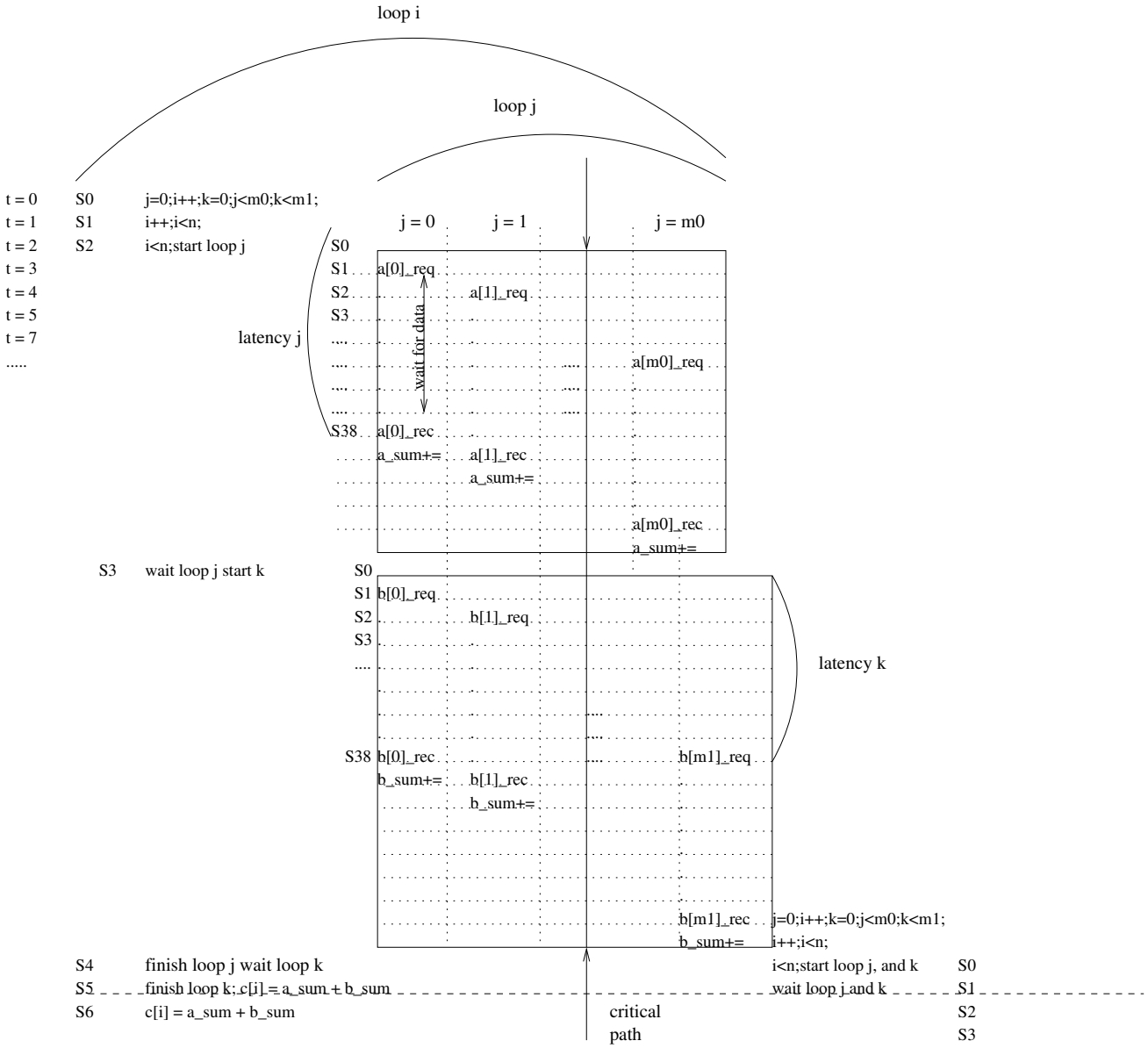


Figure 4.10: Detailed pipelined execution of the example when loops j and k cannot run in parallel (because the `restrict` keyword was not used for example)

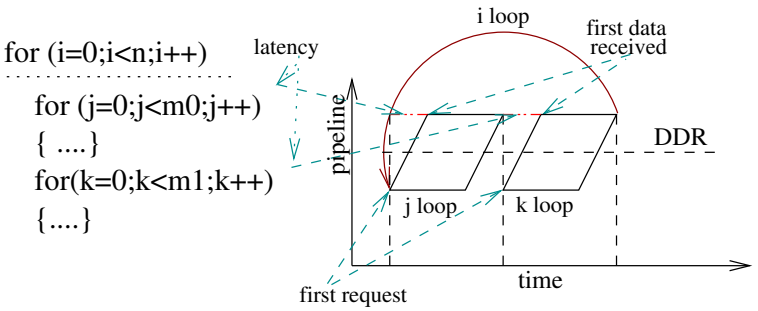


Figure 4.11: Latency penalty when there are two sequentially-executed inner loops

The value of the element `a[i]` is read, incremented, and written back at the same loop iteration. There is no loop-carried data dependence, and therefore the loop could be pipelined. The Altera documentation proposes a manual solution to this problem. One could use a shadow memory as done in the following example.

```
int *ptr;
for (i = 0; i < n; i++)
    a_out[i] = a_in[i] + 1;
ptr = a_out;
a_out = a_in;
a_in = ptr;
```

The inputs and outputs in this case reside in different memories facilitating the pipelining. If this code is in the body of an outer loop, the next iteration of the outer loop will use the computed values as input and will store the results in the original input array. However, this cannot be applied when not all the elements of `a_out[i]` are written in the loop i: one should analyze the written elements and copy them to the output array.

However, some potential memory aliasing can be removed by the user, thanks to the pragma `restrict`, defined in the ANSI-C standard as a pointer type qualifier. The `restrict` keyword is used to inform the compiler that `*a`, the value of the pointer, does not alias with any other pointer, as follows: `int* __restrict__ a;` The analysis of pointer aliasing is usually too conservative and it is not available in the Altera compiler. So the use of the `restrict` keyword is very important since the user may know better the complex aspects of the system to be implemented. However, as we will see, the fact that the pragma `restrict` is global, which means that a pointer does not alias with any other one, is often too restrictive. A useful feature that should be added to `C2H` (and possibly to other HLS tools as well) would be a pragma specifying that a subset of pointers do not alias with each other (a subset of size 2 may be already good enough for many situations).

Specific pragmas can also be used to specify the connection of the generated communication ports to other modules in the system. If there is no connection port specified, `C2H` connects each port to every other port in the system, since it does not know what memory address space the port associated with a pointer variable will address. In the example below, the pragma keyword `connect_variable` is used to inform the `C2H` compiler that the variable `a` from the accelerated function `accelerator` is connected to the slave port of `altmemddr_0`. In this case, the slave has

only one port, and when there are multiple ports, the connection should specify the port name (altmemddr_0/s1, where s1 is the slave port).

```
#pragma altera_accelerate connect_variable accelerator/a to
altmemddr_0 arbitration_share 8

int accelerator(int* a)
{
....
}
```

This connection pragma reduces considerably the arbitration logic and thus maximizes the operating frequency of the Avalon interconnect.

Finally, an `arbitration share` pragma, defined to each master/slave port pair, can be used to specify how many transfers the master can perform with the slave, without requiring to re-arbitrate. In the example above the arbitration share is set to 8. Altera Avalon allows a maximum of 128 arbitration shares per slave port. This pragma is useful to force consecutive accesses to a resource and obtain an improved throughput and latency. However, of course, it cannot re-order computations that belong to different cycles of a FSM and works only if the accesses are scheduled consecutively by the FSM.

4.3 Motivating examples with DDR accesses

Our goal in this chapter is to optimize a particular class of hardware accelerators: those working on a large data set that cannot be completely stored in local memory, but need to be transferred from a DDR memory at the highest possible rate, and possibly stored temporarily locally. This section motivates, with multiple design examples, why a naive use of `C2H` does not achieve the adequate performances.

4.3.1 DDR-SDRAM principles

We first recall the general principles of a double data rate SDRAM (DDR SDRAM) memory (DDR for short). For more details, refer to the JEDEC (joint electron device engineering council) specification of the DDR standard [19]. Figure 4.12 presents a simplified view of a typical DRAM architecture. Each DRAM unit is organized in multiple banks, from 4 (DDR1 standard) to 8 (latest DDR3 memories [20]) and the address is divided in two parts (row and column) and multiplexed from the input `address` ports to save pins on the chip. Each bank has its own state machine as depicted in Figure 4.13, which can work in parallel to hide latencies. The host DDR interface cannot command multiple banks in parallel, the command is multiplexed on the control I/O pins.

Selecting a bank automatically activates a row: a read or write command can then be performed to that row, after a specific amount of time. A transition of the controller state machine takes some time, defined in the technical memory specification. For example, going from the `PRECHARGE` state to the `IDLE` state takes a minimum amount of time t_{RP}. Also, when, after performing a read, the controller is in the `BANK ACTIVE` state and a read request to another row arrives, the controller has to meet some timing constraints [3] before it can pass to the `READ` state on the other row. Consider

3. These timing constraints are implied by the physical design limitations of the memory circuit.

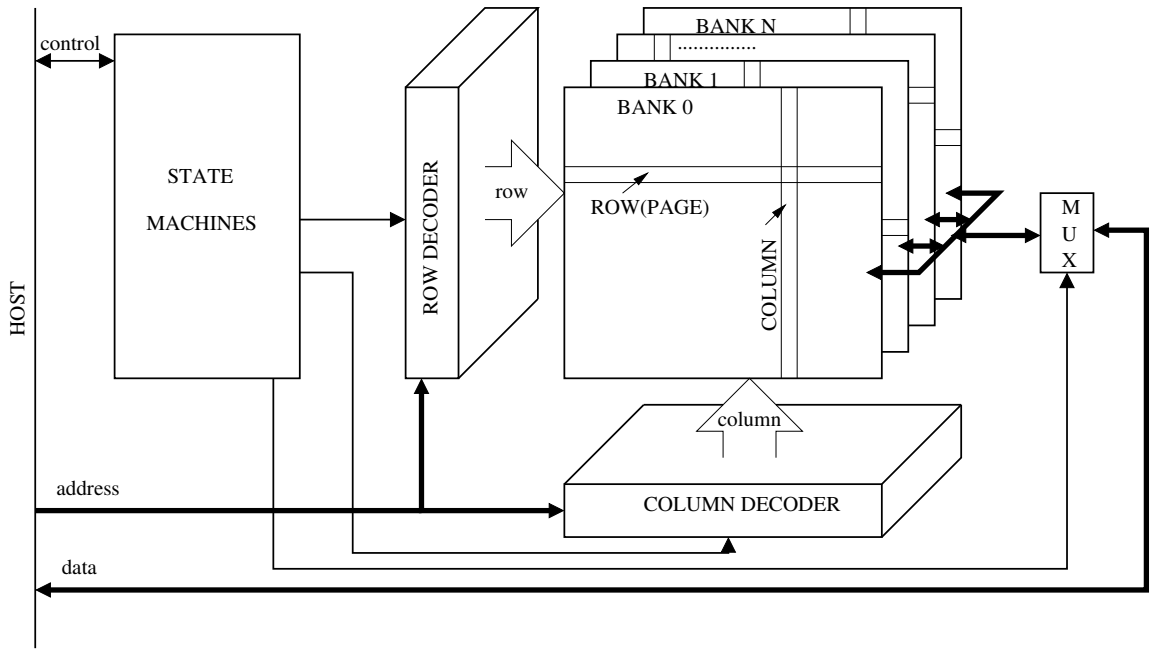


Figure 4.12: Typical DRAM architecture

for simplicity the most significant timing constraints when the controller has to pass to another row. As depicted in Figure 4.14, going from `BANK ACTIVE`, through `PRECHARGE`, `IDLE`, `BANK ACTIVE`, to `READ` takes at least as many cycles as given as Equation (4.3). There is a maximum between different timing constraints since the DDR controller requires all the timing constraints to be met.

$$t_{PASSr} = \max(t_{RAS}, 2 * t_{ck} + t_{RCD}) + t_{RP} + t_{RCD} \tag{4.3}$$

where

- t_{PASSr} represents the minimum time required to pass from an active `READ` command to another `READ` command on a different line of the DDR bank.
- t_{ck} represents the clock time.
- t_{RAS} represents the minimum time between `BANK ACTIVE` and `PRECHARGE`.
- t_{RP} represents the minimum time between `PRECHARGE` and `BANK ACTIVE`.
- t_{RCD} represents the minimum time between `BANK ACTIVE` and `READ`.

When two sequences of reads performed on different lines are interlaced by the port arbitrator of the DDR, the memory will receive the access requests in an interlaced manner, i.e., one access of the first read sequence followed by one access of the second read sequence, and so on. In this case, Equation (4.3) gives the latency between these reads. The write-to-read or read-to-write latencies can be computed in the same manner and depend on numerous scenarios. We point out that these latencies are of course not the total transfer latencies but the latencies between two accesses, i.e., they are more related to the throughput of the DDR.

In our experiments, we use a DDR memory with the following specifications: DDR-400 128 Mb x8, size of 16 MB, and CL = 3 at 200 MHz memory clock.[4] The memory parameters are $t_{ck} = 5$ ns, $t_{RAS} = 40$ ns, $t_{RCD} = 15$ ns, and $t_{RP} = 15$ ns. Using the memory parameters specification,

4. The CAS latency (CL) represents the minimum delay between the time when the memory receives a read

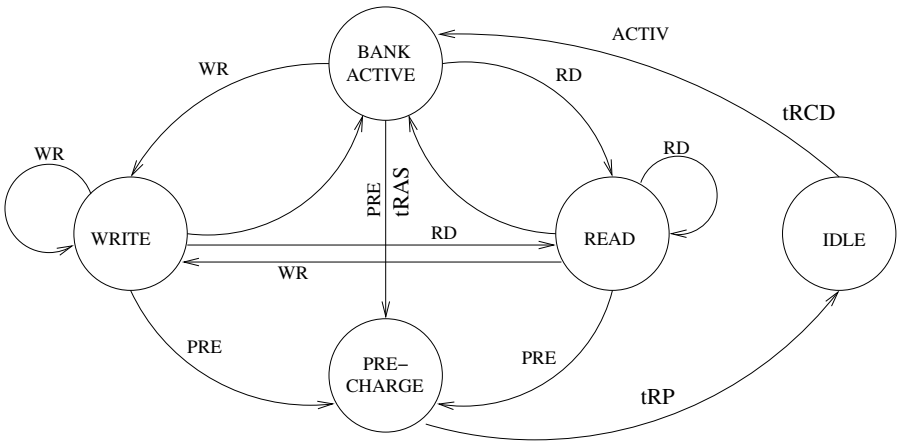


Figure 4.13: Simplified DDR state diagram

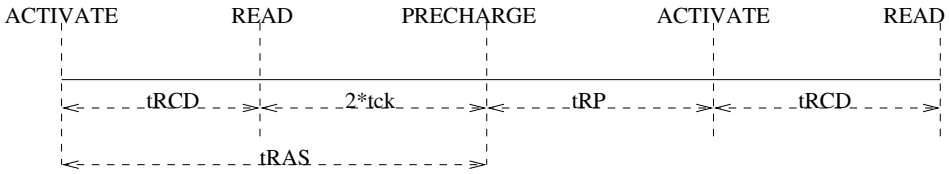


Figure 4.14: DDR read to read from different lines latencies

$t_{PASSr} = \min(40\ ns, 10\ ns) + 15\ ns + 15\ ns = 80\ ns$. A successive read to the same row takes $t_{NORM} = 2 * t_{CK} = 10$ ns. In other words, a read from the same row is $t_{samerow} = \frac{t_{PASS}}{t_{NORM}} = 8$ times faster. In practice, for a non-optimal implementation of the DDR controller JEDEC specification, this ratio can be larger. On the Altera platform, we did measure this order of magnitude.

As can be deduced from above, the memory approaches its maximum throughput when the state changes in the DDR controller are reduced. If the hardware accelerator is optimized, as it is often the case, to have a CPLI equal to 1, its performance directly depends on the throughput of the data accesses. If successive data items are accessed within the same row, the accelerator will work at full speed, otherwise it will stall, waiting for data to arrive. Thus, we seek transfers with as many successive reads or writes to the same row as possible, without changing the direction (read/write). The expected gain can be important, an adequate code re-organization could speed up the accelerator execution by a factor of 8!

4.3.2 Examples

We will illustrate the different problems we need to solve on some simple examples. Despite their simplicity, we believe that these examples are representative of a general situation that can occur in practice on more complex applications. In any case, being able to optimize them is a *sine qua non* condition to be able to treat more applications.

command on a particular column on an already opened row and the time when the data is ready at the memory data ports. CAS stands for column address strobe.

DMA transfer

The first example is the synthesis of a direct memory access (DMA) transfer from and to the external DDR memory (Figure 4.15(a)). Such a transfer occurs in software when invoking the function call `memcpy`. The scheduling information of the `C2H` synthesis results are presented in Figure 4.15(b). There are no inter-iteration data dependences and `C2H` succeeds to pipeline the loop with CPLI equal to 1. The loop latency is 40 and is due to the (evaluation of the) access latency to the DDR memory. The state machine has 40 states in order to implement the pipelined accesses. In the FSM, 40 cycles are paid from the first data request until the first data is received (Figure 4.15c). In the steady state, from the ideal point of view of the schedule in the FSM, the accelerator will be able to receive one data and send one data at each clock cycle.

The presence of cache memory in modern processors imposes a data burst transfer of the cache line size from the main memory. The data is not fetched one by one, even though data is accessed one by one in the code. We used this cache memory feature in the previous chapter to increase the data reuse. However, as explained earlier, most current IPs providing a processor with a cache do not offer an access to this cache from the outside. In this case, adding a dedicated cache memory is not practical because of its high price compared to SRAM memory. Also, because usually accelerated algorithms are very regular, SRAM memory data transfer management is not very complicated.

In this DMA transfer example, we are interested in the case where the size of the arrays `a` and `b` are much larger than the size of a row in the memory (in our example, the system row size is 1 kB). In this case, for the majority of accesses, the elements `a[i]` and `b[i]` are placed in different rows. To pass from the read of `a[i]` to the write of `b[i]`, the DDR state machine has to pass through the states of `PRECHARGE`, `BANK ACTIVE`, and `READ`, with an important performance penalty. Passing from the write of `b[i]` to the read of `a[i]` implies a state change through `PRECHARGE`, `BANK ACTIVE` and `WRITE`. The resulting memory accesses (see Figure 4.15(d)) are very inefficient from the point of view of the DDR memory architecture. The read access of one element of array `a` using the Altera DDR controller takes 80 ns and a write access takes 70 ns. Although the hardware accelerator is optimal from the point of view of resource usage and schedule, it operates $\frac{80/10+70/10}{2} = 7.5$ times slower than the version when all the elements of the array `a` and `b` are fetched in an non-interleaved way.

Convolution 1D

Another interesting example is the 1D convolution. It is often used in image processing to apply a 1D mask on a image (e.g., edge detection, filtering, etc.). The code represents two nested loops that apply a mask with coefficients c_1, c_2, and c_3 on an image array a and store the result into an image array b. The 2D image array is linearized before the function call since the ANSI-C restricts the use of a bi-dimensional array with variable line width in a calling function. The synthesized hardware accelerator contains two loop control FSMs. The first loop over i has CPLI 1. The loop does not have any loop-carried dependences and can be fully pipelined. The latency of 5 cycles is due to the 5 states of the pipeline. The loop over j has a CPLI 3. This is due to the resource access contention of 3 accesses on array `a` on one DDR memory port.

For each loop iteration, the hardware accelerator accesses sequentially the elements $a[i*n+j-1]$, $a[i*n+j]$, and $a[i*n+j+1]$. As in the previous example, the write of $b[i]$ is not considered as a contention, which means it can be scheduled at the same time as the reads. The memory access latency is 46, hence there are at least 46 states in the FSM. The pipeline and synchronizations of

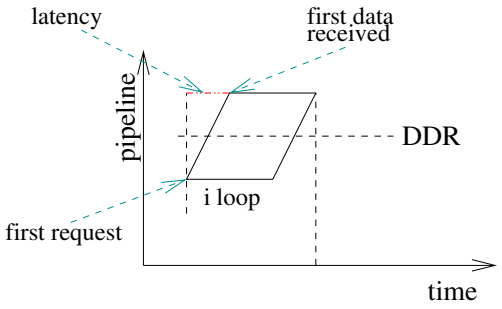
```
#pragma altera_accelerate connect_variable /
   dma_transfer/a to altmemddr_0
#pragma altera_accelerate connect_variable /
   dma_transfer/c to altmemddr_0

int dma_transfer (int* __restrict__ a,
  int* __restrict__ b, int n) {
    int i;
    for (i=0; i<n; i++) b[i] = a[i];
    return 0;
}
```

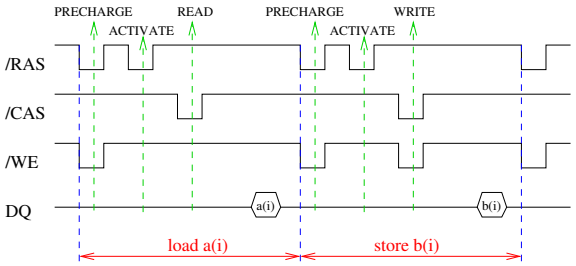
(a) Source code

```
LOOP i (b[i] = a[i]):
   CPLI = 1, Loop latency = 40
   Schedule:
     c[i] = a[i]; : State 1-->39
     i++; : 0-->1
```

(b) State machine parameters



(c) Pipeline



(d) DDR accesses

Figure 4.15: DMA transfer example

the two nested loops are presented in Figure 4.16(c). The loop j has a start-up latency of 46 cycles, which is paid for every iteration of the loop i, more precisely from state 3 (for iteration $i = 1$) to state $3 + n - 3 = n$ (for iteration $i = n - 2$). With the same reasoning used for Equation (4.2), the nesting overhead in this example is thus $(n - 3 + 1)(46 - 1) = 45(n - 2)$. For large values of n, the overhead is small (compared to the execution time of order n^2) but, for small values of n, it can take a significant portion of the execution time.

An iteration of the convolution 1D (see Figure 4.16(b)) requires three reads of elements of the array a and one write to the array b. The addresses of the reads are consecutive, therefore it is very likely that the elements will be located in the same row of the memory (except for the border cases). In this case, the three elements of the array `a` can be accessed without any latency access penalty (see Figure 4.16(d) on the right). However, there is still a need for a write of the element of the array `b`. As in the example of the DMA transfer, this will induce a significant performance penalty after each burst of three elements of `a`. This example shows that part of the memory accesses in some algorithms can benefit from the DDR burst data transfer performances. Unfortunately, usually, there are multiple write transfers in between that can significantly minimize the data transfer performances from the DDR. There are only a few exceptions, such as "cycle redundancy check" algorithms. These types of algorithms are characterized by a very big data read-to-write ratio, i.e., they consume a lot of input data for a single output data.

Sum of two vectors

This example computes the sum of two vectors a and b and writes it into the vector c (see Figure 4.17(a)). The loop over i has a latency of 2 cycles (see Figure 4.17(b)) because a and b are

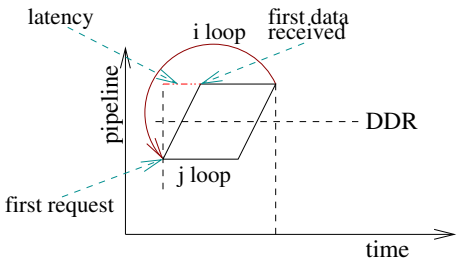
```
#pragma altera_accelerate connect_variable /
conv/a to altmemddr_0
#pragma altera_accelerate connect_variable /
conv/c to altmemddr_0

int conv (int* __restrict__ a,
    int* __restrict__ b, int n) {
  int i;
  for (i = 1; i < n-1; i++)
    for (j = 1; j < n-1; j++)
      b[i] = c1*a[i*n+j-1] + c2*a[i*n+j]
           + c3*a[i*n+j+1];
  return 0;
}
```

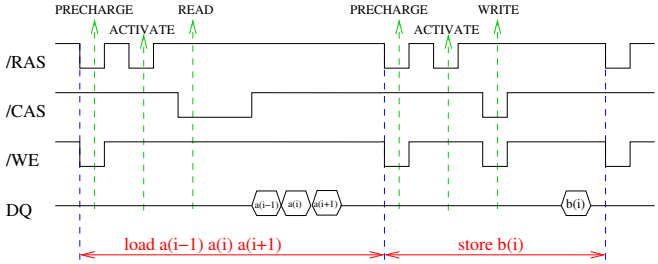
(a) Source code

```
LOOP i (b[i] = c1 ... ):
   CPLI = 1, Loop latency = 5
   j = 0; i++; : State 0-->1
   LOOP j: State 2-->4
LOOP j (b[i] = c1 ...):
   CPLI = 3, Loop latency = 46
   Schedule:
     j++; 0-->1
     b[i] = c1* ...; 0-->45
```

(b) State machine parameters



(c) Pipeline



(d) DDR accesses

Figure 4.16: Convolution 1D example

connected to the same shared resource `altmemddr_0` which represents the DDR memory port while the write is not considered as a contention (dedicated separate memory ports for writes and reads). The loop latency is 42 cycles and, as in previous examples, it represents the latency of accessing data from memory. The state machine has 42 states that implement the pipelined memory accesses (see Figure 4.17(c)).

The DDR memory accesses are depicted in Figure 4.17(d). The hardware accelerator performs a read of an element of `a`, followed by a read of an element of `b`, followed by a write of an element of `c`. Each of these operations requires a change of line in the DDR since `a`, `b`, and `c` are located in distinct lines. The resulting hardware accelerator is using the DDR in a very inefficient way. Even worse, in the DDR, there is an automatic burst of the whole memory bitwidth even if the required data is shorter than the words managed by the DDR. For example, DDR memories are usually packed in DIMMs (dual in-line memory module) that are 72-bits wide. A single memory transfer that is by default in burst mode by 4 will fetch from the memory $72 * 4 = 288$ bits. From this 288 bits, only 32 bits of data will be used (an integer element has 32 bits). The rest of the transferred data is immediately discarded when a new request to a different line arrives. In other words, not only the transfer is slow but, in addition, useful data are transferred to be discarded, before being requested again in the next iterations. In our case, the situation is not as bad because we instantiated a 8-bits wide DDR, with burst of 4, so no data is discarded if 32-bits words are requested.

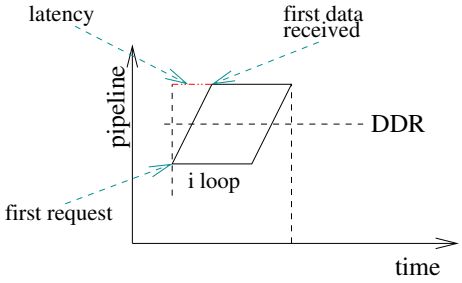
```
#pragma altera_accelerate connect_variable /
 hw_acc_orig_ddr_restrict/a to altmemddr_0
#pragma altera_accelerate connect_variable /
hw_acc_orig_ddr_restrict/b to altmemddr_0
#pragma altera_accelerate connect_variable /
 hw_acc_orig_ddr_restrict/c to altmemddr_0

int vector_sum (int* __restrict__ a,
  int* __restrict__ b,
  int* __restrict__ c, int n) {
  int i;
  for (i=0; i<n; i++)
     c[i] = a[i] + b[i];
    return 0;
}
```

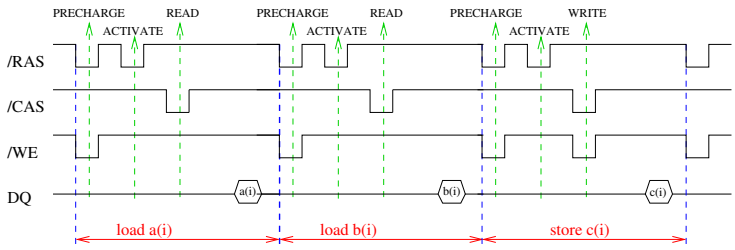
(a) Source code

```
LOOP i (c[i] = a[i] + b[i]):
   CPLI = 2, Loop latency = 42
   Schedule:
     c[i] = a[i] + b[i]; : State 1-->41
     i++; : 0-->1
```

(b) State machine parameters



(c) Pipeline



(d) DDR accesses

Figure 4.17: Vector sum example

Matrix-matrix product

The inputs of the matrix-matrix product code (Figure 4.18) are two linearized matrices `a` and `b`. The access to the matrix `c` was replaced by a local accumulator `tmp` and is copied to `c` at the end of the iteration loop k. The accumulator is used to remove the unnecessary R/W to the DDR, where the matrix `c` is stored, otherwise the design would not be pipelined. The `restrict` and `connect_variable` keywords are used to inform `C2H` about the connection and aliasing of variables `a`, `b`, and `c`.

The source code is composed of three nested loops iterating over `i`, `j`, and `k`. The loop over `i` has a CPLI of 1 and a latency of 5 (i.e., it has 5 states). Its critical path corresponds to the loop over `j`. This loop has a CPLI of 1 and its critical path contains the loop over `k`. The latency of the `j` loop is bigger than the latency of the `i` loop since, besides the `tmp` initialization, there is a write instruction to the element `c`. However, unlike reads, the writes do not increase too much the latency of a loop since a write operation does not have any waiting period to be encoded in the FSM. The third loop, iterating over `k`, has a latency of 43 since it has to encode the latency of the DDR reads of `a` and `b`. Its CPLI is 2 as parallel accesses to `a` and `b` arrays using a single DDR memory port will possibly cause a congestion. It thus runs in $(m_0 - 1) \times \text{CPLI} + 43$ states, according to Equation 4.1. Compared to its number of iterations m_0, the difference is paid for each iteration of the `j` loop, and then for each iteration of the `i` loop, (see Figure 4.20), thus $n0 * m1$ times, reducing considerably the execution time of the hardware accelerator.

The inner loop has two interleaved read requests to the DDR, with the same access penalty seen for `a` and `b` in previous examples (Figure 4.21). Besides that, every `m0` cycles, the accelerator performs a write to store the value of `c`. As in previous examples, the interleaving of accesses diminishes considerably the possible burst performances of the DDR to the same row in a bank.

```
#pragma altera_accelerate connect_variable /
 matrix_multiply_hw_orig/a to altmemddr_0
#pragma altera_accelerate connect_variable /
 matrix_multiply_hw_orig/b to altmemddr_0
#pragma altera_accelerate connect_variable /
 matrix_multiply_hw_orig/c to altmemddr_0
int matrix_multiply (int* __restrict__ a,
    int* __restrict__ b, int* __restrict__ c,
    int n0, int m0, int m1) {

  int i, j, k, tmp;
  for (i=0; i<n0; i++)
    for (j=0; j<m1; j++) {
      tmp = 0;
      for (k=0; k<m0; k++)
        tmp += *(a+(i*m0+k))**(b+(k*m1+j));
      *(c+(i*m1+j)) = tmp;
    }
  return 0;
}
```

Figure 4.18: Matrix-matrix multiplication

```
LOOP i
   CPLI = 1, Loop latency = 5
   Schedule:
     j = 0; : State 0-->0
     assignment control: State 0-->1
     Loop j: State 2-->4
     i++; : State 0-->1
LOOP j
   CPLI = 1, Loop latency = 7
   Schedule:
     k = 0; : State 0-->0
     tmp = 0; : State 3-->3
     loop k :  State 2-->4
     *(c+(i*m1)+j) = tmp: State 2-->6
LOOP k
   CPLI = 2, Loop latency = 43
   Schedule:
     tmp += *(a+....); : State 0-->42
     assignment control: State 1-->2
     k++; : State 0-->1
```

Figure 4.19: State machine parameters

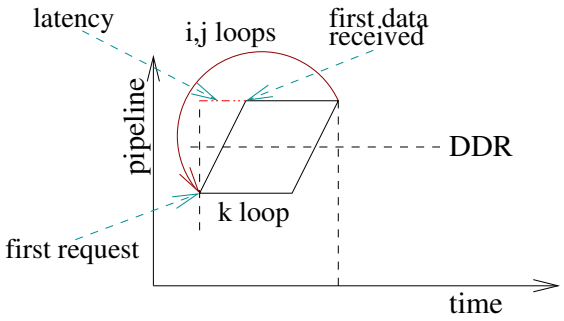


Figure 4.20: Pipeline latencies of matrix-matrix multiply

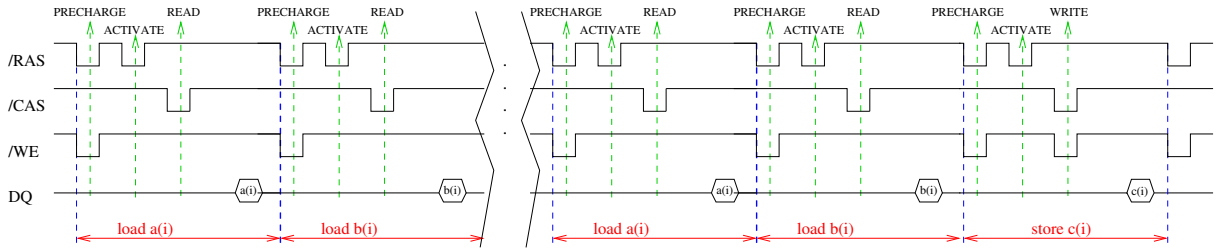


Figure 4.21: DDR accesses of matrix-matrix multiply

4.3.3 Optimizing DDR accesses

The examples detailed in the previous section illustrate two important reasons for performance loss. The first one is due to the non-consecutive DDR accesses, we call it (*row change penalty*). The second due is due to to nested loops containing DDR accesses, we call it (*data fetch penalty*). To summarize, for our platform and designs:

- at best, the accelerator can receive 32 bits every 10 ns (8 bits per rate, double data rate, 200 MHz), thus 3.2 Gb/s.
- if successive accessed data are not in the same row, the accelerator is able to receive 32 bits only every 80 ns, roughly.
- if, before sending a new request, the accelerator needs to wait the complete communication delay from the request to the arrival of a data, it will have to wait an order of magnitude longer. For simple `C2H` designs, this delay is roughly 400 ns (40 cycles at 100 MHz).

The row change penalty can occur in inner loops and thus directly impacts the throughput of the accelerator. The data fetch penalty due to access latency occurs less often (not for inner loops, unless they are not pipelined), but with a higher penalty. To get better performances, the code must be restructured so that:

- arrays are accessed by blocks of elements belonging to the same row of the DDR.
- the re-organization of accesses should not increase the CPLI of the computations.
- nested loops containing data accesses should be avoided in order to not pay long data fetch latencies.
- all necessary glue and house-keeping should be written at C-level and compiled into hardware by the same HLS tool, i.e., `C2H`.

To make all this possible, we will have to use the local memory to store some data that cannot be consumed immediately. Section 4.5 will explain the generic solution we designed to achieve these goals, based on multiple accelerators that orchestrate communications. Figure 4.22 gives the type of communication patterns that we will obtained, here for the vector sum, with an optimal DDR usage. The next chapter will then show how this process can be automated, in particular how the required transformations and code generations, that we designed to be as systematic as possible, can be performed. Before, in Section 4.4, we demonstrate why direct and maybe more natural approaches do not work. The context of HLS and the use of an HLS tool as a black box make things much more complicated than one can think.

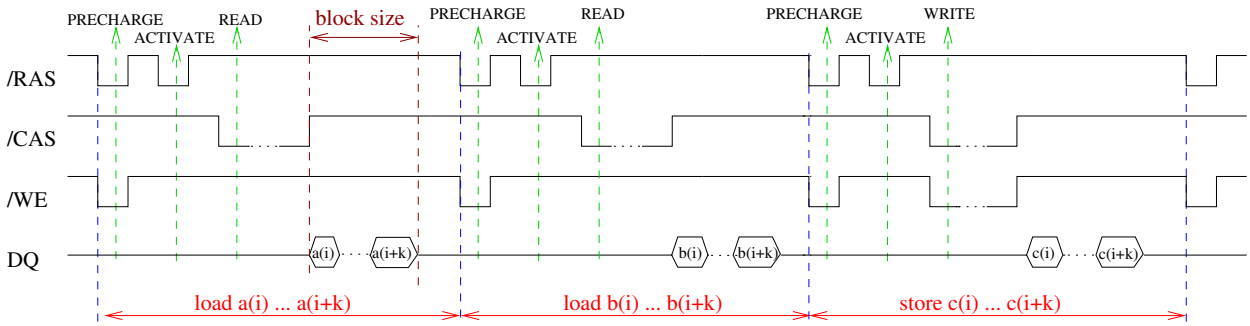


Figure 4.22: Accesses for optimized vector sum

4.4 First attempts toward a solution

In this section, we try to apply different compiler transformations in order to optimize the performances of the communications. The attempts are illustrated with the vector sum code (Figure 4.23), synthesized using `C2H`. The other examples studied in Section 4.3.2 can be treated similarly.

```
for (i = 0; i < MAX; i++)
    c[i] = a[i] + b[i];
```

Figure 4.23: Vector sum

4.4.1 Blocking

The vector sum code does not have any nested loops. Thus, the DDR access latency are paid only once at the loop start-up. The rest of the access latencies is hidden by the loop pipeline. However, as explained earlier, the vector sum code accesses the DDR memory in a very inefficient way. The code performs successively two reads and a write to different rows in the DDR memory (Figure 4.17). A direct technique to optimize the performance is to apply loop distribution and to introduce a local storage to store the received data as depicted in Figure 4.24.

The resulting code represents 4 loops iterating over the original iteration domain. The first two loops fetch arrays `a` and `b` from the DDR memory and store then into arrays `a_tmp` and `b_tmp`. These two arrays are stored into local memory, which is usually implemented using an SRAM.

```
for (i = 0; i < MAX; i++)
    a_tmp[i] = a[i];
for (i = 0; i < MAX; i++)
    b_tmp[i] = b[i];
for (i = 0; i < MAX; i++)
    c_tmp[i] = a_tmp[i] + b_tmp[i];
for (i = 0; i < MAX; i++)
    c[i] = c_tmp[i];
```

Figure 4.24: Vector sum array privatization

Such memories have much more faster access times and, as opposed to the DRAM, the access time does not depend on the address that is accessed. The third loop performs the actual computation and stores the result into a local array (stored into local memory) `c_tmp`. The last loop stores the computations into the result array `c`. With respect to the DDR, requests are done as successive accesses (at least in the code), as desired. Unfortunately, such a code cannot be used in most cases. Indeed, the size of the arrays can be very large (usually in MB or even GB) while the size of the local memories implemented as SRAM are very small (usually in kB) because of the expensive technologies they are implemented with. Therefore, in most cases, the temporary arrays `a_tmp`, `b_tmp`, and `c_tmp` will not fit into local memory.

To be able to fit arrays in local memory, their sizes can be reduced by using blocking on each of the `i` loops and, after that, a fusion of the outer loops iterating over the obtained blocks (Figure 4.25). This can also be obtained by a first strip-mining by a factor `BLOCK`, then a loop distribution to put each statement in a different loop. Note that all this is of course legal since the original loop is parallel.

```
for (i=0; i<MAX; i=i+BLOCK) {
  for(j=0; j<BLOCK; j++) a_tmp[j] = a[i+j];//prefetch
  for(j=0; j<BLOCK; j++) b_tmp[j] = b[i+j];//prefetch
  for(j=0; j<BLOCK; j++) c_tmp[i+j] = a_tmp[j] + b_tmp[j];/compute
  for(j=0; j<BLOCK; j++) c[i+j] = c_tmp[i+j];//store
}
```

Figure 4.25: Vector sum with blocking: strip-mining + loop distribution

The code obtained this way is composed of an outer loop `i` iterating over blocks of size `BLOCK`. The inner loops perform the prefetches into local arrays for a block, the computations, and store the results. The `BLOCK` parameter can be used to increase the size of the local arrays with respect to the available local memories. The larger is `BLOCK`, the smaller the row change penalty, but the larger the local memory. The scheduling and the other FSM parameters of the blocked code are given in Figure 4.26).

The loop over `i` has a CPLI of 2 because of the pointer overlap of `a_tmp` in write and `a_tmp` in read. Since `C2H` cannot perform a fine grain dependence analysis, it cannot schedule at the same time the loop fetching elements of `a` into `a_tmp` and the loop performing the computations on them. The loops that fetch elements of the array `a` and `b` do not have data dependences between them, thus they are scheduled in parallel from state 2 to state 4. Then, they are followed, sequentially,

```
LOOP i
   CPLI = 2, Loop latency = 7
   Schedule:
     j = 0; : State 0-->0
     assignment control: State 0-->1
     Loop j (a_tmp): State 2-->4
     Loop j (b_tmp): State 2-->4
     Loop j (c_tmp): State 3-->5
     Loop j (c):     State 4-->6
     ...
LOOP j (a_tmp)
   CPLI = 1, Loop latency = 40
   Schedule:
     a_tmp[j] = a[i+j];: State 1-->39
     ...
LOOP j (b_tmp)
   CPLI = 1, Loop latency = 40
   Schedule:
     b_tmp[j] = b[i+j];: State 1-->39
     ...
LOOP j (c_tmp)
   CPLI = 1, Loop latency = 10
   Schedule:
     c_tmp[j] = a_tmp[j] + b_tmp[j];: State 1-->9
     ...
LOOP j (c)
   CPLI = 1, Loop latency = 9
   Schedule:
     c[i+j] = c_tmp[i+j];: State 1-->8
     ...
```

Figure 4.26: Blocked vector sum: schedule

by the computation and store loops. The pipelining of the code is presented in Figure 4.27. Since the loops that fetch `a` and `b` from the DDR are scheduled in parallel, for each iteration of the `i` loop, only one start-up latency penalty of `a` or `b` is paid. The initial source code (Figure 4.23) has only one latency penalty, while the blocked code has this penalty $\frac{MAX}{BLOCK}$ times.

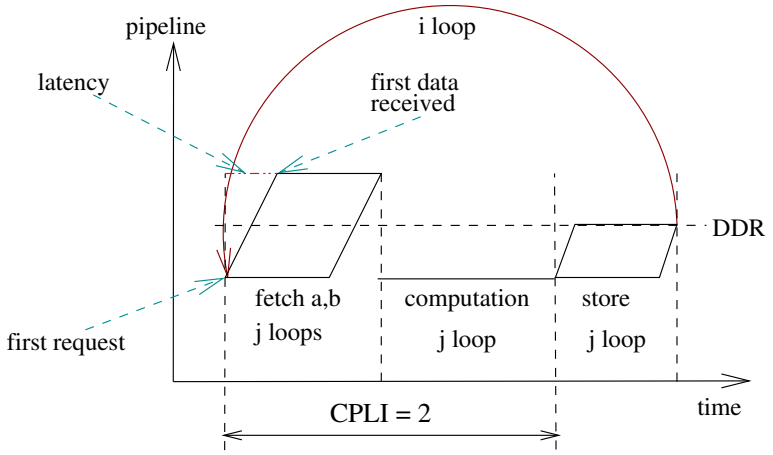


Figure 4.27: Block vector sum: pipeline

Thus, from the point of view of the pipeline efficiency, the performed transformations decrease the performance of the hardware accelerator compared to the original one. But the hope is to win from the point of view of DDR accesses. Unfortunately, the transformation does not optimized any memory accesses (Figure 4.28). The elements of the arrays `a` and `b` are still accessed in an interleaved manner since the two loops that fetch them are scheduled in parallel. Also, due to the pipelined execution of the outer loop, at its second iteration, the fetching of `a` and `b` of the second iteration are also interleaved with the writes of `c` from the previous iteration. The use of the `arbitration share` pragma could (partially) avoid this interleaving. However, its size is limited to only 128 per slave port. The DDR has multiple connections including at least a data and instruction cache of the CPU, each of them having 8 shares and arrays `a`, `b`, and `c`. Thus, the `arbitration share` can only be used for a limited block size of 37, assuming that no other accelerator is connected to the DDR. Also, besides the performance loss due to these non-optimal accesses to the DDR, there is a performance loss due to the startup loop latency that can be seen at the beginning of the DDR access diagram.

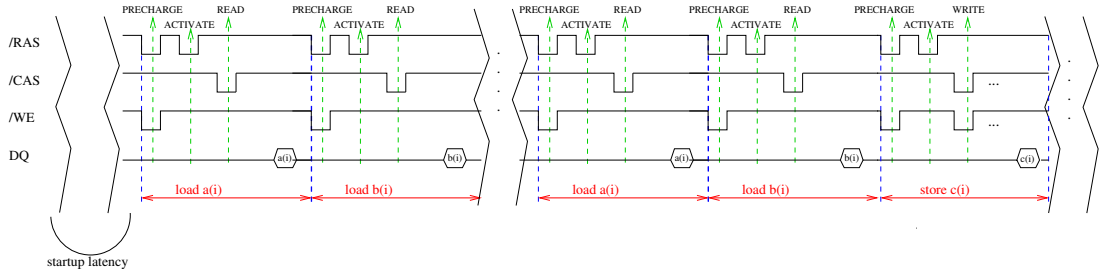


Figure 4.28: Blocked vector sum: DDR access

To eliminate the interleaved accesses of at least `a` and `b`, a simple solution is to introduce a false

data dependence between the two fetching loops (see Figure 4.29) to impose a sequential execution order. Here, the false dependence is done with the scalar `tmp`, used to pass the loop iteration count from the loop prefetching `a` to the loop prefetching `b`.

```
for (i=0; i<MAX; i=i+BLOCK) {
  for(j=0; j<BLOCK; j++) {tmp = BLOCK; a_tmp[j] = a[i+j];//prefetch
  for(j=0; j<tmp; j++) b_tmp[j] = b[i+j];//prefetch
  for(j=0; j<BLOCK; j++) c_tmp[i+j] = a_tmp[j] + b_tmp[j];/compute
  for(j=0; j<BLOCK; j++) c[i+j] = c_tmp[i+j];//store
}
```

Figure 4.29: Blocked vector sum with sequential accesses obtained with false dependence

However, this transformation increases the "virtual schedule time" of the loops (Figure 4.30), i.e., the CPLI increases to 3, as the two `j` loops that prefetch `a` and `b` are in the critical circuit of the outer loop. Also, for each iteration of the loop, the latency access penalty is now paid twice (see Figure 4.30). As in the previous example, the outer loop pipelining will cause an interleaved access between the reads of the elements of `a` (or possibly `b`, depending on the schedule), and the writes of elements of the array `c`.

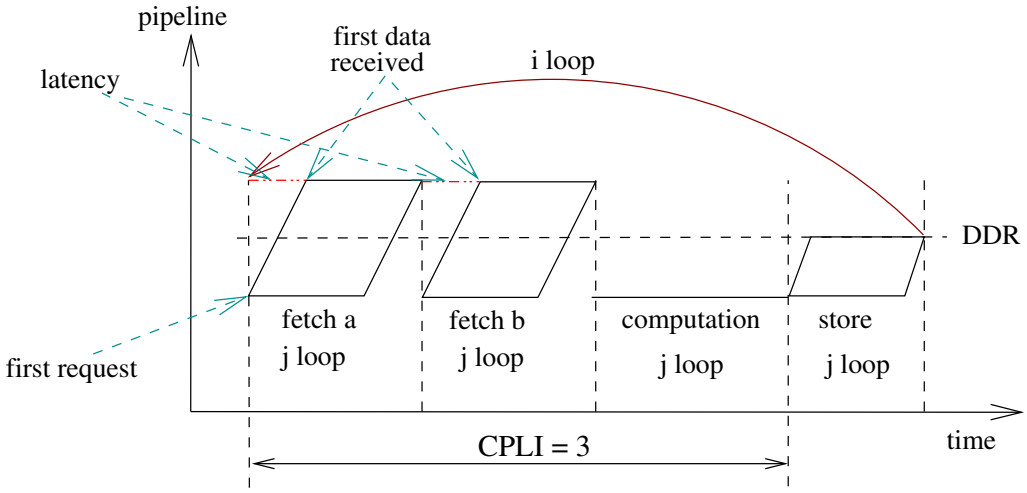


Figure 4.30: Blocked vector sum: pipeline with false dependence

To avoid these data fetch penalties, a possibility is to unroll the inner loops that prefetch `a` and `b` and store each data read in a different scalar variable, as in Figure 4.31. For a `BLOCK` of size 8, the `i` loop will have a CPLI of 16 (Figure 4.32) and performs 16 read accesses, which is optimal in terms of CPLI. This strategy does not remove some interleaving with the write accesses of array `c`. But with some luck, most of the time (this can be checked by simulation), data are fetched in the textual order of the requests, thus reducing the row change penalty. In general however, there is no guarantee that instructions are scheduled by C2H in the textual order. For example, as shown in Figure 4.32, the stores of `c` are in the reverse order of the text, except the operation `c[i+7] = c_tmp7`, which is scheduled after `c[i+6] = c_tmp6`. Nevertheless, in this particular example, from the point of view of DDR memory accesses, the code read accesses are optimal since the arrays `a`

and b are accessed by block. However, the store of an element of array c starts immediately after its computation, thus polluting the memory accesses starting from the 38th execution cycle of the accelerator.

The code obtained this way contains only one loop, thus the startup latency of 62 is paid only once. By including all statements in a single basic block, we thus enabled the possibility of fine-grain scheduling performed by C2H. The downside of this approach however is the code explosion and hence the hardware resource explosion since C2H will map directly every sum operator to a separate resource, using also many register storage for scalar variables. Also, this approach requires a non-parametric unrolling factor BLOCK, which should be a divisor of MAX. For all these reasons, this solution is again not suitable.

4.4.2 Juggling

Blocking is necessary to re-organize the data accesses. However, as we have seen, relying on successive loops to prefetch blocks of data leads to a sub-optimal code due to the pipeline latency of successive loops. A more involved solution is to linearize the 3 inner loops of the blocking code obtained earlier into a single loop k. Conceptually, the iterations of the loops are emulated with an automaton, written in C. This can be done with if statements or with case statements. We tried both, obtaining different results that are detailed here. This technique is a particular form of the juggling technique introduced for the HLS tool Pico [54]. As we will show, again, this will not be enough to get the right performances.

Juggling using if statements The code for vector sum with linearized loops using if statements is given in Figure 4.33. The desired loop scanning is emulated thanks to an automaton that retrieves the original loop counters. In this example, bi is used to iterate on blocks, i to iterate inside the block. The variable j is used to schedule the operations on a specific block. For j = 0, the loop prefetches the values of a and stores them in a_tmp. For j = 1, the loop prefetches the values of b and stores them in b_tmp. For j = 2, the loop computes the sum of the two blocks of vectors. For j = 3, the loop stores the computed results to the DDR memory into the array c. The loop over k has the number of iterations of the original loop multiplied by the number of states (in this case 4) so that the linearized code behaves the same as the blocking code.

The scheduling information is presented in Figure 4.34 and its graphical representation in Figure 4.35. The fetch of elements of array a and b, the computations, and the store all belong to different branches of the nested if statements. Here, the C2H scheduler finds a very inefficient solution with CPLI equal to 21, which is not at all expected. This is because the software pipeliner is not powerful enough. Considering the schedule it finds, we can guess how it proceeds. Since all the statements of the same iteration can be scheduled in parallel, the maximum number of required states for the FSM seems to be fixed to its minimum, given by the DDR access latency, here 40, plus one state for initial counter computations, leading to a latency of 41. Now, within this window of 41 states, let us follow one chain of dependent computations: the instruction c_tmp[i] = a_tmp[i] + b_tmp[i] can potentially depend on a DDR access from the previous iteration (see Figure 4.35). This access ends at state 40 but, analyzing different schedules, it seems that the computation c_tmp[i] = a_tmp[i] + b_tmp[i] can be initiated a bit earlier, at state 39 (see later examples also). As the situation is different if a_tmp[i+1] was read, this is certainly because the data is written in local memory only at state 40 but it can be forwarded one state earlier from a local register storage: $18 + 1 = 19$ and $19 + 21 = 40$. Now, if the four branches of the if tree were

```
for (i = 0; i < MAX; i=i+BLOCK) {
  a_tmp0 = a[i+0];
  a_tmp1 = a[i+1];
  a_tmp2 = a[i+2];
  a_tmp3 = a[i+3];
  a_tmp4 = a[i+4];
  a_tmp5 = a[i+5];
  a_tmp6 = a[i+6];
  a_tmp7 = a[i+7];

  b_tmp0 = b[i+0];
  b_tmp1 = b[i+1];
  b_tmp2 = b[i+2];
  b_tmp3 = b[i+3];
  b_tmp4 = b[i+4];
  b_tmp5 = b[i+5];
  b_tmp6 = b[i+6];
  b_tmp7 = b[i+7];

  c_tmp0 = a_tmp0 + b_tmp0;
  c_tmp1 = a_tmp1 + b_tmp1;
  c_tmp2 = a_tmp2 + b_tmp2;
  c_tmp3 = a_tmp3 + b_tmp3;
  c_tmp4 = a_tmp4 + b_tmp4;
  c_tmp5 = a_tmp5 + b_tmp5;
  c_tmp6 = a_tmp6 + b_tmp6;
  c_tmp7 = a_tmp7 + b_tmp7;

  c[i+0] = c_tmp0;
  c[i+1] = c_tmp1;
  c[i+2] = c_tmp2;
  c[i+3] = c_tmp3;
  c[i+4] = c_tmp4;
  c[i+5] = c_tmp5;
  c[i+6] = c_tmp6;
  c[i+7] = c_tmp7;
}
return 0;
```

Figure 4.31: Unrolled blocked vector sum

```
LOOP i
   CPLI = 16, Loop latency = 62
   Schedule:
     i = i+80; : State 0-->1
     a_tmp0 = a[i+0];:State 1-->37
     a_tmp1 = a[i+1];:State 1-->38
     a_tmp2 = a[i+2];:State 1-->39
     a_tmp3 = a[i+3];:State 1-->40
     a_tmp4 = a[i+4];:State 1-->41
     a_tmp5 = a[i+5];:State 1-->42
     a_tmp6 = a[i+6];:State 1-->43
     a_tmp7 = a[i+7];:State 1-->44
     b_tmp0 = a[i+0];:State 1-->45
     c_tmp0 = (a_tmp0+b_tmp0);: State 45-->46
     c[i+0] = c_tmp0;:State 38-->61
     b_tmp1 = a[i+1];:State 1-->46
     c_tmp1 = (a_tmp1+b_tmp1);: State 46-->47
     c[i+1] = c_tmp1;:State 38-->60
     b_tmp2 = a[i+2];:State 1-->47
     c_tmp2 = (a_tmp2+b_tmp2);: State 47-->48
     c[i+2] = c_tmp2;:State 38-->59
     b_tmp3 = a[i+3];:State 1-->48
     c_tmp3 = (a_tmp3+b_tmp3);: State 48-->49
     c[i+3] = c_tmp3;:State 38-->58
     b_tmp4 = a[i+4];:State 1-->49
     c_tmp4 = (a_tmp4+b_tmp4);: State 49-->50
     c[i+4] = c_tmp4;:State 38-->57
     b_tmp5 = a[i+5];:State 1-->50
     c_tmp5 = (a_tmp5+b_tmp5);: State 50-->51
     c[i+5] = c_tmp5;:State 38-->56
     b_tmp6 = a[i+6];:State 1-->51
     c_tmp6 = (a_tmp6+b_tmp6);: State 51-->52
     c[i+6] = c_tmp6;:State 38-->54
     b_tmp7 = a[i+7];:State 1-->52
     c_tmp7 = (a_tmp7+b_tmp7);: State 52-->53
     c[i+7] = c_tmp7;:State 38-->55
```

Figure 4.32: Unrolled blocked version: scheduling

```
i = 0;
j = 0;
bi = 0;
for (k = 0; k < 4*MAX; k++) {
  if (j == 0)
    a_tmp[i] = a[bi + i];
  else if (j == 1)
    b_tmp[i] = b[bi + i];
  else if (j == 2)
    c_tmp[i] = a_tmp[i] + b_tmp[i];
  else
    c[bi + i] = c_tmp[i];

/* loop iterators, control unit */
  if (i < BLOCK-1) i++;
  else {
    i = 0;
    if (j < 3) j++;
    else {
      j = 0;
      bi = bi + BLOCK;
    }
  }
}
```

Figure 4.33: Linearized blocked vector sum

```
LOOP k
   CPLI = 21, Loop latency = 41
   Schedule:
     k++; : State 0-->1
     j++; : State 0-->1
     i++; : State 0-->1
     i=0; : State 1-->1
     i=0; : State 1-->1
     a_tmp[i] = a[bi+i];:State 1-->40
     b_tmp[i] = b[bi+i];:State 1-->40
     c_tmp[i] = a_tmp[i] + b_tmp[i]; State 18-->37
     c[bi+i] = c_tmp[i]; State 28-->35
     bi = bi + 80; : State 0-->1
     if (i< BLOCK-1): State 0-->1
     if (j < 3): State 0-->1
     if (j == 2): State 35-->36
     if (j == 1): State 34-->35
     if (j == 0): State 33-->34
```

Figure 4.34: Linearized version: scheduling

scheduled at state 1 (which is possible as they never occur at the same iteration), then the CPLI would be at least 38 (as $39 = 18 + 21$ and also $39 = 1 + 38$). To avoid such a high CPLI, it seems that the software pipeliner tries to delay the computation `c_tmp[i] = a_tmp[i] + b_tmp[i]` to obtain the right distance (in time) between the reads and the write at the next iteration. But, it proceeds in a very sub-optimal way as the following study shows: actually, a CPLI equal to 3 is possible.

When the number of states are not restricted, it is possible to obtain a more efficient schedule (see Figure 4.36). Many flow and anti data dependences need to be met by the scheduler. Taking into account the flow dependence from the write of `a_tmp[i]` to the read of `a_tmp[i]`, we get:

$$CPLI + d + 1 > \ell - 1 \tag{4.4}$$

where ℓ represents the number of states fixed for a read access to the DDR (in the previous discussion, $\ell = 40$) and d the state at which the statement `c_tmp[i] = a_tmp[i] + b_tmp[i]` is scheduled (in the previous discussion, $d = 18$). Taking into account the anti-dependence from the read of `a_tmp[i]` to the next write of `a_tmp[i]`, we get:

$$CPLI + \ell - 1 > d + 1 \tag{4.5}$$

By combining Equations (4.4) and (4.5), we get:

$$\ell - 1 - CPLI < d + 1 < \ell - 1 + CPLI \tag{4.6}$$

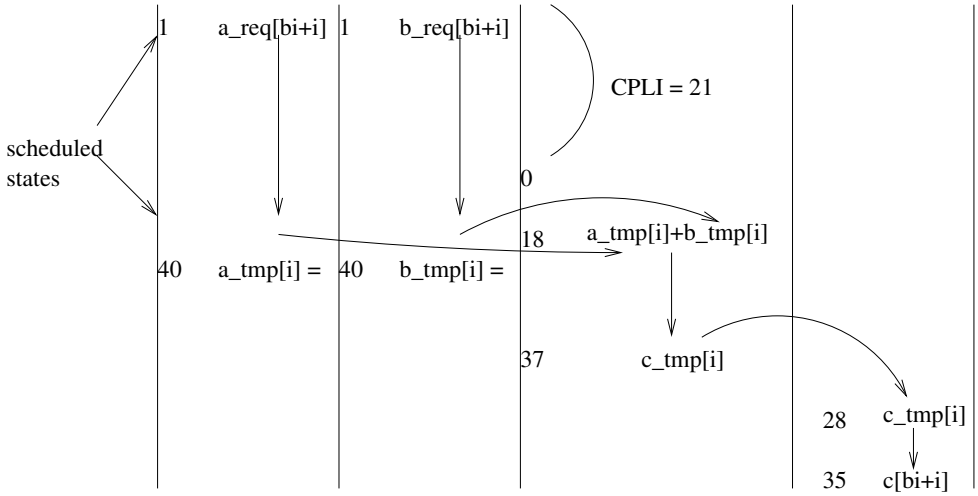


Figure 4.35: Vector sum juggling: graphical representation of the schedule

To obtain a CPLI equal to 1, $d+1$ must be strictly between $\ell-2$ and ℓ, thus for $d=\ell-2$. In the example, the computation `c_tmp[i] = a_tmp[i] + b_tmp[i]` should be scheduled at state $d=38$ and not at state $d=18$ as the software pipeliner of Altera wrongly chooses. But this would imply a latency ℓ larger than the DDR access latency.

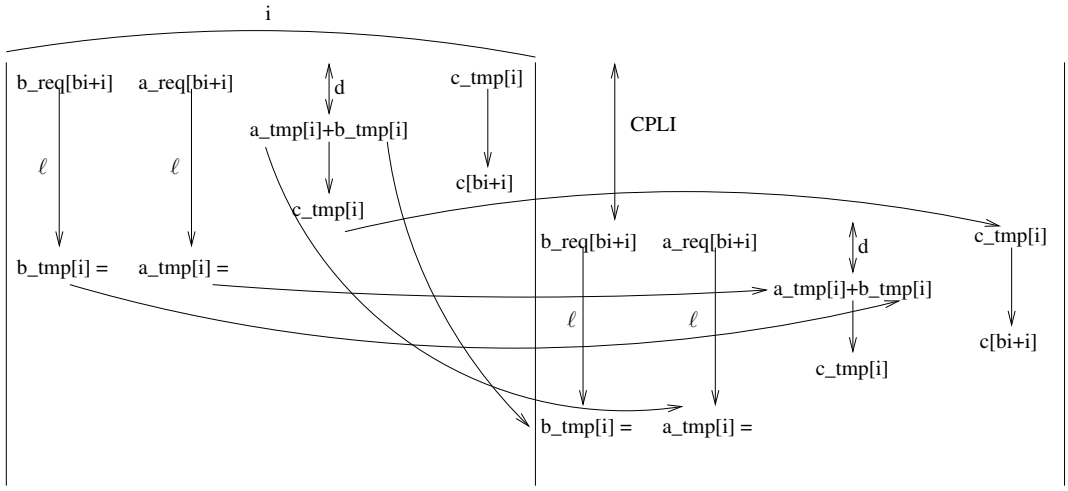


Figure 4.36: Vector sum juggling: graphical representation of possible optimal schedule

Juggling using `case` statements The juggling/linearization technique can also be implemented using a `case` statement. The Altera documentation specifies that a `case` statement is implemented using multiple `if` statements. However, as will be shown below, the `C2H` scheduler behaves differently if `if` statements are directly used and if `case` statements are used.

The source code of the juggling technique using a `case` statement is presented in Figure 4.37. It is similar to the previous code, just with one difference: the nested `if` statements are substituted by a `case` statement.

```
i = 0;
j = 0;
bi = 0;
for (k = 0; k < 4*MAX; k++) {
  switch(j) {
    case 0: a_tmp[i] = a_in[bi + i];
            break;
    case 1: b_tmp[i] = b_in[bi + i];
            break;
    case 2: c_tmp[i] = a_tmp[i] + b_tmp[i];
            break;
    default: c_out[bi + i] = c_tmp[i];
  }

  /* loop iterators, control unit */
  if (i < BLOCK-1) i++;
  else {
    i = 0;
    if (j < 3) j++;
    else {
      j = 0;
      bi = bi + BLOCK;
    }
  }
}
```

Figure 4.37: Linearized version with `case`

```
LOOP k
   CPLI = 3, Loop latency = 41
   Schedule:
     k++; : State 0-->1
     j++; : State 0-->1
     i++; : State 0-->1
     i=0; : State 1-->1
     i=0; : State 1-->1
     a_tmp[i] = a[bi+i];:State 1-->40
     b_tmp[i] = b[bi+i];:State 1-->40
     c_tmp[i] = a_tmp[i] + b_tmp[i]; State 39-->48
     c[bi+i] = c_tmp[i]; State 47-->55
     bi = bi + 80; : State 0-->1
     if (i< BLOCK-1): State 0-->1
     if (j < 3): State 0-->1
     break; : State 0-->3
     break; : State 0-->6
     break; : State 0-->9
     switch_expression0 = 1: State 3-->3
     switch_expression0 = 1: State 6-->6
```

Figure 4.38: Corresponding schedule

The scheduling information of the code is presented in Figure 4.38. As can be observed, the obtained code has a CPLI of 3. The behavior of the `case` statement can be different from the `if` statement as it depends on the presence of `break` statements. Here, the compiler does not even try to detect if the cases can occur at the same iteration or not, they are assumed they can. Therefore, it cannot detect the possible parallel execution of the different `case` statements. When the scheduler is not able to detect the possible parallel execution of the statements, even though they will never execute at the same iteration, the schedule looks as depicted in Figure 4.39. All statements are scheduled sequentially one after another. In order for the scheduler to pipeline the loop and to preserve the data dependences, the following constraints need to be respected:

$$d + 1 > \ell - 1 \text{ and } d + 1 < CPLI + \ell - 1 \tag{4.7}$$

In other words, the closer the reads of `a_tmp[i]` and `b_tmp[i]` are from their writes, the shorter the CPLI. The minimal value is 2. because the hardware accelerator is not allowed to perform, simultaneously, a write and a read of the same array (vector variables cannot be replicated [22]). However, we are not quite sure to understand why C2H finds a CPLI equal to 3 and not to 2.

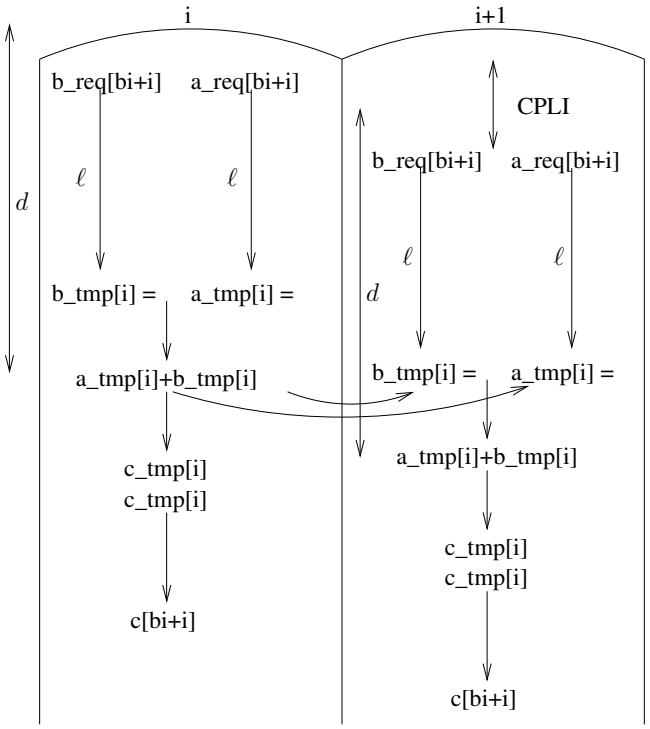


Figure 4.39: Graphical representation of a schedule with dependences in the same loop iteration

Still an interleaving problem So far, we succeeded to remove the data fetch penalties due to successive inner loops within an outer loop. For that, we enforced the order of requests to the DDR by an automaton, encoded in C, in a single loop iterating over variables and blocks. The first problem we faced was that the software pipeliner of C2H is quite weak, leading to hard-to-predict CPLIs. Depending on how the code is written, we can get a very bad CPLI (21) or a quite good CPLI (3).

Nevertheless, this is not enough, we would like to get the highest possible communication rate, i.e., a CPLI equal to 1. A possibility would be to unroll by a factor equal to CPLI and to try to avoid false dependences by renaming. This may be possible, after some brainstorming. Unfortunately, even if we succeed, this will not be sufficient. Indeed, as observed from simulations, a new problem arises. To be able to pipeline DDR requests, `C2H` has to speculatively send read requests to the memory when the access statements are inside a `if` statement. Only when the condition of the `if` is determined, the data is read from the FIFO or not. This is also due to the fact that `C2H` generates hardware: each statement corresponds to hardware, even if it is within a `if` and is not activated for a given iteration. The consequence is even worse than before: not only DDR requests are interleaved again, resulting in important row change penalties, but also dummy requests are repeated multiple times, even if they are not used.

To eliminate these dummy DDR accesses, we can try to use pointer dereferencing, as depicted in Figure 4.40, again with `if` statements. When $j = 2$, the sum of elements is computed and stored directly into `c` in the DDR. When $j = 0$ (resp. $j = 1$), the pointer `ptr_ddr` points to the required location of `a` (resp. `b`). The data transfer is performed immediately after with a single transfer instruction in the `else` part of the first `if`. Thus there are no dummy DDR accesses, except for $j = 2$. Furthermore, the two read requests are scheduled at the same state (but at different iterations), so there is no interleaving between them and accesses to `a` and `b` should be nicely scheduled. However, for the write accesses to the DDR, two phenomena can occur. First, as the write is scheduled at a much later state (state 39), the last write requests of a block actually occur *after* the first read requests of the next block. Also, dummy write requests still pollute the DDR access medium. The pointer address computations do not increase the CPLI since pointers are treated as scalar variables thus `C2H` can replicate them. Here, the CPLI is again equal to 3, for the same reason as for the previous code.

Let us try to avoid this undesired interleaving between reads and writes. For this, we can use a code similar to the code of Figure 4.33, with local arrays `a_tmp`, `b_tmp`, and `c_tmp`. Then, we introduce two new pointers `ptr_read` and `ptr_write` and define a unique data transfer statement `*ptr_write = *ptr_read`, outside the `if` statements. Depending on the branch $j = 0$, $j = 1$, or $j = 4$, `ptr_read` is initialized to an address of `a`, `b`, or `c_tmp`, and `ptr_write` to an address of `a_tmp`, `b_tmp`, or `c`, respectively. Now, there will be no dummy transfers at all. Note that `ptr_write` and `ptr_read` are both connected to the DDR. Thus, for the loop to be pipelined, they must be declared as `restrict`. It remains to write the computation statement `c_tmp[i] = a_tmp[i] + b_tmp[i]` while preserving data dependences and loop pipelining. To access `a_tmp` in this statement, the only way to guarantee a correct schedule is to read it through the pointer that defines it, thus `ptr_write`. We can then write the computation for $j = 3$ as a sequence of operations. What we would like is something like:

```
ptr_write = &(a_tmp[i]); tmp_a = *ptr_write;
ptr_write = &(b_tmp[i]); tmp_b = *ptr_write;
tmp_c = tmp_a + tmp_b;
ptr_read = &(c_tmp[i]); *ptr_read = tmp_c;
```

However, again, if these transfer statements are written in the branch $j = 3$, they will create dummy speculative transfers. We thus need to perform these 3 transfers thanks to the unique transfer statement `*ptr_write = *ptr_read` previously defined. For that, we can add three more states for j, one for each access, to read or write the right data. However, we cannot use the unique transfer statement since `ptr_write` is on the left-hand side of the unique transfer statement and

not on the right-hand side. Anyway, we cannot use `ptr_write` both on the left and on the right of an assignment, otherwise the loop would not be pipelined as `ptr_write` is a pointer that accesses the DDR. The only remaining solution seems thus to use `ptr_read` to read in `a_tmp` and `b_tmp` thanks to the statement `*ptr_write = *ptr_read`. Then, the loop is pipelined, with CPLI equal to 1. But, in some cases, depending on the scheduler and of runtime latencies, the code is incorrect: the computation should start after the last data of array `b` has arrived, not just after the request itself. But because `ptr_read` and `ptr_write` are `restrict` and point to the same location, there is no way to guarantee that the schedule is correct.

Actually, the problem we faced here several times is that the `restrict` pragma is too global, it cannot express the restriction between only two pointers. In this example, we would need a pointer `ptr_local_a`, connected only to the local memory `a_tmp` and not to the DDR, and a pragma that could specify that `ptr_local_a` can only alias with `ptr_write`, which writes into the local array `a_tmp`. The accesses to `c_tmp` and `b_tmp` could be handled similarly. This way, the loop will be pipelined and the data dependences will be preserved.

```
int* __restrict__ ptr_ddr;
bi = 0; j = 0;
for (k=0; k<3*MAX; k++) {
  if (j==2)
    c[i+bi] = a_tmp[i] + b_tmp[i];
  else {
    if (j==0) ptr_ddr = a+(i+bi);
    else ptr_ddr = b+(i+bi);

    tmp = *ptr_ddr; /* data transfer */

    if (j==0) a_tmp[i] = tmp;
    else b_tmp[i] = tmp;
  }

  if (i++==BLOCK) {
    i=0;
    if (j++==3) {
      j=0;
      bi += BLOCK
    }
  }
}
```

Figure 4.40: Vector sum source code using juggling and pointer dereferencing

```
LOOP k
   CPLI = 3, Loop latency = 48
   Schedule:
     if (j == 2): State 0-->1
     if (j == 0): State 0-->1
     ptr_ddr = (b + (i + bi));:State 0-->1
     ptr_ddr = (a + (i + bi));:State 0-->1
     tmp = *ptr_ddr;: State 1-->37
     a_tmp[i] = tmp;: State 37-->39
     b_tmp[i] = tmp;: State 37-->39
     c[i+bi] = a_tmp[i] + b_tmp[i];: State 39-->47
     k++; : State 0-->1
     j++; : State 0-->1
     i++; : State 0-->1
     i=0; : State 1-->1
     i=0; : State 1-->1
     bi = bi + 80; : State 0-->1
     if (i< BLOCK-1): State 0-->1
     if (j < 3): State 0-->1
```

Figure 4.41: Scheduling of vector sum using juggling and pointer dereferencing

4.4.3 Conclusions

In this section, we detailed our numerous attempts to try to express block communications at source code without loosing performance. The transformed versions of code always failed to supply the required performance.

One of the biggest concern is that, with `C2H`, data requests in `if` instructions are still initiated,

speculatively, so as to enable their pipelining. This leads to correct non-interleaving read accesses to `a` and `b`, but possibly a pollution with writes to the DDR. We tried many variants, some that we explained here, with different pointers, different writing, trying to enforce dependences when needed and to remove false dependences. We did not find any satisfactory solution. Either the code is potentially incorrect, depending on the schedule and because we needed to use the pragma `restrict`, or its CPLI increases, or it is not pipelined at all.

The best solutions we obtain led to small but sub-optimal CPLIs, for example CPLI equal to 3. Unrolling may be a possible solution to retrieve an execution equivalent to one with CPLI equal to 1. But, because of false dependences and an uncontrollable software pipeliner, this strategy is more likely to not work, especially for more complex applications. Indeed, the techniques tried in this chapter already fail to optimize a very simple code such as the sum of two vectors. Also loop unrolling is bad for resource usage, as `C2H` does not perform resource sharing. Finally, more importantly, in all solutions we tried, we did not succeed to reach our initial goal, which was to completely remove interleaved accesses to the DDR. Dummy accesses due to speculative requests were an unexpected problem that could be discovered only through simulation. The only practical solution we found is to separate the fetches and stores to the local memory locations from the computations, defining *communicating hardware accelerators*. This solution is discussed in the following section. It is extremely painful to write by hand a correct version of this strategy but we show later how it can be automated.

4.5 A solution with multiple communicating accelerators

The previous section showed that it is inefficient, if not impossible, to write the code managing the communication in the code managing the computation. If it is placed in a previous loop, the accelerator has to wait for the data to arrive before starting the computation (data fetch penalty). If it is embedded within the computation code, controlled by an automaton as for the juggling technique, it is very difficult to ensure that this extra housekeeping code does not alter the optimal data rate and does not create artificial interleaved DDR accesses. This is even more true when trying to mix a communication code, whose CPLI should be 1, with computations at $CPLI > 1$ but, possibly, at higher frequency. A natural solution would be to implement the data transfers in a single-loop accelerator, synchronized with the computation accelerator. However, since all instructions from a loop are guaranteed to be executed only when the loop state machine finishes its execution, it is again not possible to enforce data coherence with a simple correct synchronization between the two accelerators. All these considerations pushed us toward a more involved solution that we now expose.

4.5.1 Architecture description

The template architecture is presented in Figure 4.42. The generated accelerators are represented as bold rounded rectangles, local memories as normal bold rectangles, and the rest are FIFOs. The data required for a given block of computations are transferred, in the order desired for optimizing DDR accesses, using a "double buffering"-like approach implemented by two accelerators `BUFF0_LD` and `BUFF1_LD`. They prefetch data in local memories `BUFF0` and `BUFF1`. Using the same type of double buffering, two other hardware accelerators, `STORE0` and `STORE1`, are used to store the computation results from local buffers `ST0` and `ST1` to the DDR memory in a specific optimized order.

A dual buffering approach allows the use of single-port local memories, which consume fewer resources and are preferred over dual port ones in ASIC design, even if they are now usually available on FPGA chips. More importantly, with two accelerators, we will be able to use the data transfer of one to hide the data fetch penalty of the other one.

In this design, FIFOs are used only for synchronization and control, which means that they do not contain data used for computation. Multiple tokens are passed between the hardware accelerators using these FIFOs. The FIFO `BUFF0_BUFF1` is used to pass the DDR resource access token from `BUFF0_LD` to `BUFF1_LD`. This resource token is passed through all the hardware accelerator accessing the DDR resource in this order: `BUFF0_LD`, `BUFF1_LD`, `STORE0`, `STORE1`, then back to `BUFF0_LD`, passing through the FIFOs `BUFF0_BUFF1`, `BUFF1_ST0`, `ST0_ST1`, `ST1_BUFF0`.[5] As can be observed, the token is passed using a single path circular link thus enforcing an ordered and deadlock-free execution. The FIFOs `C01_ST0`, `C01_ST1`, and `BUFF01_C01` are used for passing a data dependence token. The `C01_ST0` is used to pass the token from computation hardware accelerator `COMP0/1` to the accelerator `STORE0`. This token is transmitted when the computations using the `BUFF0` memory elements are finished: `STORE0` can then store the results from `ST0` to the DDR. The same is true for `STORE1`. The `BUFF01_C01` FIFO has two input sources: `BUFF0_LD` and `BUFF1_LD`. To distinguish which prefetch accelerator sends the token, we use a token equal to 0 for `BUFF0_LD` and 1 for `BUFF1_LD`. Note that there will be no non-determinism: because of the whole synchronization diagram, the values are alternatively 0 and 1. We could also decompose the computation accelerator into two parts, but it is not needed and it would waste hardware resource.

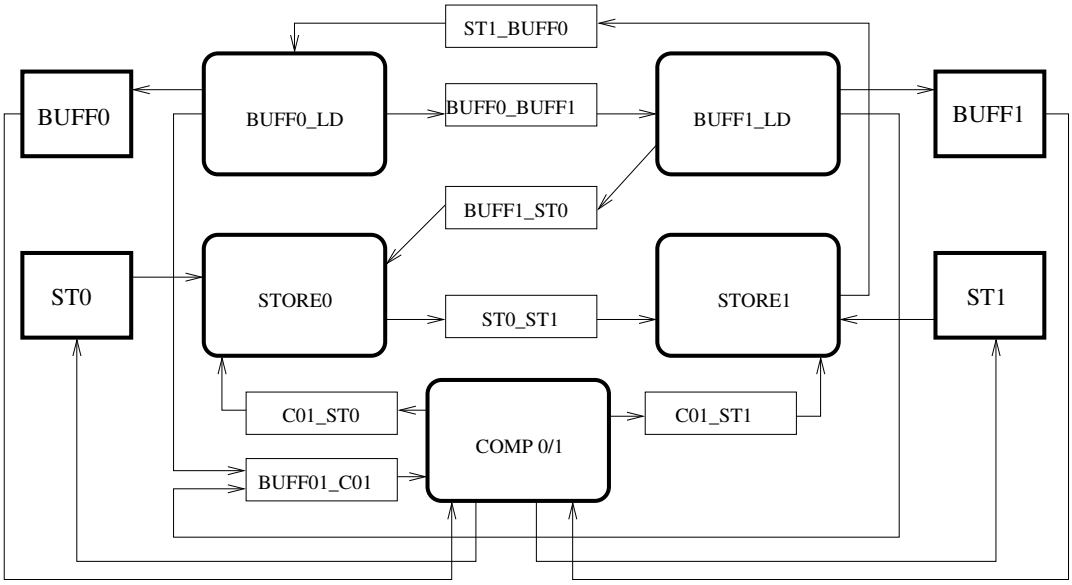


Figure 4.42: Accelerators module architecture

As can be seen, the design is partitioned in multiple accelerators: for data transfer management and for computation. As explained earlier, we are interested only in source-to-source transformations. Designing manually such hardware modules is not a convenient solution (even if we first

5. This is the order we chose for the solution presented here. As will be explained later, other orders are possible, with different performances.

had to do it!) since it requires hardware expertise, and a considerable verification and simulation time. To avoid all these difficulties, every hardware accelerator should be generated using HLS methodology. FIFOs and memories are not hardware accelerator, and they are instantiated in the system using vendor's IP cores.

4.5.2 Hardware accelerator implementations

We now explain how the hardware accelerators of Figure 4.42 can be generated and synchronized with `C2H` and the Altera design environment.

Synchronization diagram In the following, each accelerator operates on a block of data identified by an iterator t, so we write for example `BUFF0_LD(t)` to represent the execution of this accelerator at iteration t.

Figure 4.43 shows a possible synchronization of the whole system, with two kinds of synchronization, due to data dependences (for example from `BUFF0_LD(t)` to `COMP0(t)`) and due to resource utilization (for example from `BUFF0_LD(t)` to `BUFF1_LD(t)`). The communication accelerators are represented as parallelipipeds to express the fact that the transfers are pipelined with a long latency. The left diagonal edge represents the transfer of the first data read in a block: the lower-left corner corresponds to the time when the request is sent, the upper-left corner represents the time when the corresponding data arrives. Then, other successive requests are pipelined the same way, until the last data transfer in a block which is represented by the right edge of the parallelipiped. Computations are represented as rectangles as the latency of iterations is in general much smaller. With this parallelipiped representation, it is easier to see when the synchronizations take place. A dataflow synchronization (in blue normal line) starts after the last data is received. A synchronization due to force a sequential access to the DDR (in red dotted line) starts after the last data request has been sent.

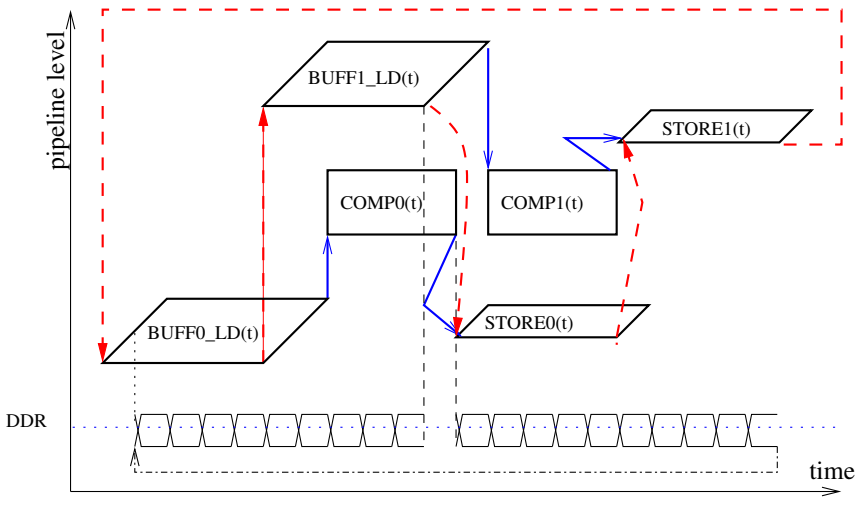


Figure 4.43: Synchronization diagram: first solution, possibly with a time gap

In Figure 4.43, the DDR transfers may still be not optimal: indeed, there may be a small gap between the loads and the stores, if the computation in `COMP0(t)` takes longer than the time required

to send all read requests of `BUFF1_LD(t)`. This gap can be eliminated by reducing the computation time with parallelization techniques. Another solution is to shift all stores to the right (i.e., delay them by one iteration) as depicted in Figure 4.44. This time, we synchronize `BUFF1_LD(t)`, at its last data request, to `STORE0(t-1)` instead of `STORE0(t)`. `STORE0(t-1)` stores the results obtained at the previous iteration, hence it can start before `COMP0(t)`. However, this requires duplicating the local memory of the computed data, as `COMP0(t)` now overlaps with `STORE0(t-1)`. Another solution, with no duplication, is given in Section 4.5.4, once the problem is formulated as a software pipelining problem (see Figure 4.66).

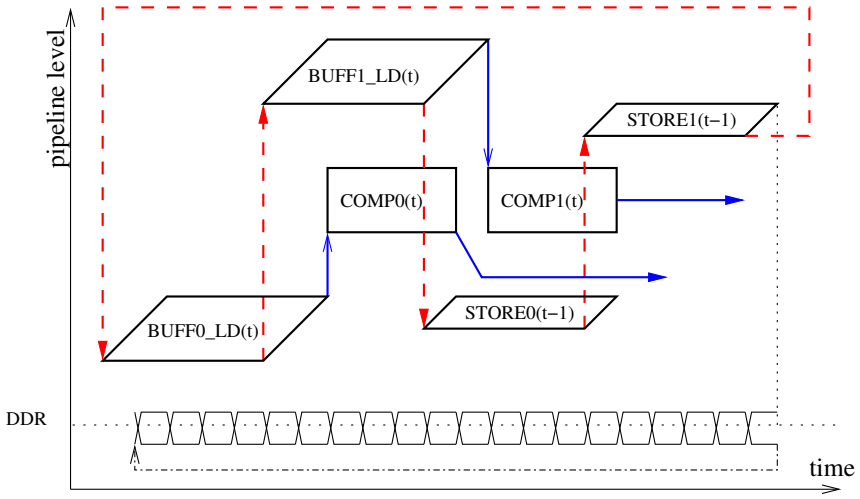


Figure 4.44: Synchronization diagram: second solution with memory duplication

Template codes for communication and computation accelerators When compiling, with `C2H`, a system of communicating accelerators written in C, there are only two mechanisms to guarantee a given sequential order of operations: the use of FIFOs between accelerators which offers a mechanism for blocking reads (if the FIFO is empty) and blocking writes (if the FIFO if full), and the fact that data dependences inside a given accelerator are preserved by any schedule. However, data dependences between different accelerators must be guaranteed by the designer. If two accelerators access to a given shared memory, for example in a producer/consumer relation, i.e., one writing a value that should be read by the other, how can we make sure that the read occurs when the data is here? If the HLS tool does not provide a way to express that a data has arrived, there is no way to define a consistent semantics of operations.

In `C2H`, when an access to a local memory is performed at a given state of a FSM, it is guaranteed that, at the next state of the same FSM, the data has been written or read. In other words, it is an atomic operation. However, this is no longer true for distant accesses such as DDR accesses, otherwise transfers to the DDR could not be pipelined: after a data request is sent at a given state of a FSM, other operations can be performed (if data dependences allow to do so) before the data has arrived. Therefore, to warn another accelerator that a read from the DDR has been accomplished, it is not enough to send a token in a FIFO just after the C instruction that initiates the read. As `C2H` performs pipelining with a fixed pre-computed latency (and a stall mechanism if

the transfer is longer), one could exploit this latency to wait sufficiently many states of the FSM to be sure that the data arrived. However, this latency (40 in the previous examples) depends on the platform and, moreover, making sure that this latency is respected after software pipelining, although we cannot control how operations are precisely scheduled, would be quite hard, if not impossible. Fortunately, `C2H` provides a way to do it, at source level, although it is very likely that this feature was not designed for that. As we previously explained, each loop has its own FSM and, for the schedule of the code surrounding it, it is considered as atomic. In other words, when a loop is scheduled at a given state of an outer FSM then, at the next state, we are sure that all its operations are finished, including reads and writes to the DDR. This hierarchical scheduling principle, which was a problem before because of what we called the data fetch penalty, is now an advantage. After a loop performing transfers from or to the DDR, we can send a token to specify that the transfers are done.

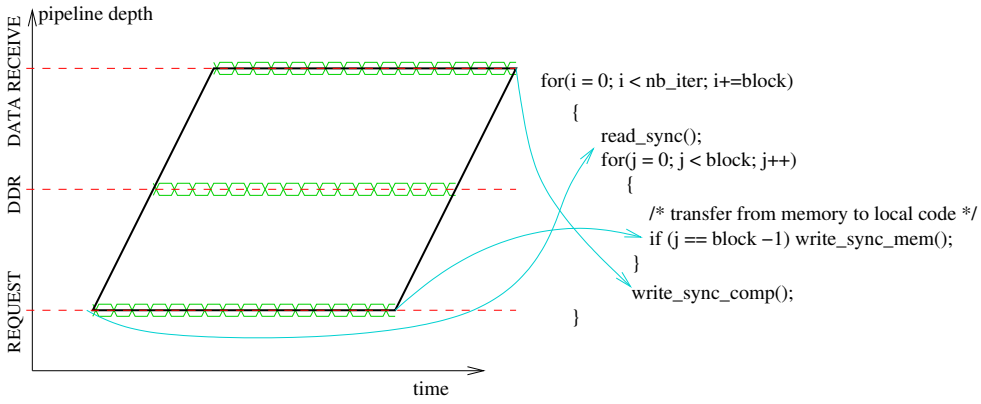


Figure 4.45: Synchronization and C semantics

These principles are illustrated in Figure 4.45. The code has two nested loops. The outer loop iterates over the blocks (tiles). Before executing a tile, a blocking read from a FIFO is performed, then the inner loop can start the transfers. At the last iteration of the inner loop, the last request has been performed, a token can be written to give access to the DDR to another accelerator. After the loop, a token can be written to specify that the transfers are done and some computation can start.

Figure 4.46 shows the template code of the `BUFFO_LD` accelerator for a situation similar to the vector sum example where two arrays `a` and `b` must be prefetched from the memory to local storage. The general mechanism is as shown in Figure 4.45 except that, in addition, we need to make sure that the inner loop is not scheduled before the blocking read (indeed, they are parallel). To impose such a sequential execution, a dummy variable `dummy_read` is used to translate the execution order dependence into a false data dependence. Its value is read inside the inner loop to define another dummy variable `tmp`. The inner loop iterates over a tile. In this example, the tile represents a block of the iteration space of array `a`, followed by a block of the iteration space of array `b`. The number of iterations of the inner loop is controlled by a parameter `r_sup`. If it is too small, the DDR accesses are not optimized enough and the data fetch penalty is paid too often. If it is too large, the required local storage can become too large.

The inner loop uses the same mechanism as the juggling code of Section 4.4 to emulate the traversal of the read requests, in the right order. After the desired local and external addresses are computed, a data request is initiated to perform the transfer from external to local memory.

Here, as there are only reads, the code can be fully pipelined with CPLI equal to 1. The solution developed in Section 4.4 is well adapted as the problem of interleaving writes and reads does not occur (there are only reads).

In order to optimize the double buffering transfer, the token-passing statement to `BUFF1_LD` is located at the end of the iteration space of the inner loop scanning a tile. This ensures that the token is passed from `BUFF0_LD` to `BUFF1_LD` at the same time as the last pipelined request to the memory is sent. Only the resource token is sent at this stage. To send a data dependence token, we have to ensure that the data was prefetched and stored in the local memory. Thus the data-dependence token is sent to the computation unit immediately after the inner loop. Again, the dummy variable `tmp` is used to ensure that the inner loop is scheduled before the statement `*buff0_c01_write = tmp`, which sends the data-dependence token.

The template code for `BUFF1_LD` is similar, see Figure 4.47. The difference with `BUFF0_LD` is its initial guard (`if` statement) to handle correctly the cases when the number of tiles is not a multiple of 2. When the guard is false, the inner loop is not executed but the resource token must still be forwarded (last line of the code). The synchronization statements are similar but involve different FIFOs, according to the synchronization diagram of Figure 4.42. Also, the initial tile is different than for `BUFF0_LD`.

The template code for the accelerator `STORE0` is presented in Figure 4.48. The accelerator has to wait for two tokens before starting executing a tile. The first blocking read represents a read of the resource token and the second one represents a data-dependence token. Dummy variables `dummy_read` and `dummy_read1` carrying false data dependences ensure again the execution of the inner loop only after both tokens are read. The inner loop writes the computed results of a block of array `c` to memory. The resource token is finally sent at the same time as the last write request to the DDR to the next user, which is `STORE1`. There is no token to send to any computation accelerator.

The template code of the hardware accelerator `STORE1` is similar, it is presented in Figure 4.49. As the `BUFF1_LD` accelerator, it contains an `if` statement for the cases where the number of tiles is not a multiple of 2. The inner loop transfers data from the local memory to the DDR. At it last iteration, it sends a resource token to the hardware accelerator `BUFF0_LD`. When the execution guard is false, the inner-loop is not executed but the resource token is still sent to `BUFF0_LD` so that the whole process can start again for the next iteration.

The template code for the `COMP0/1` computation accelerator combines the computations that correspond to data brought by the two accelerators `BUFF0_LD` and `BUFF1_LD`, see Figure 4.50. The

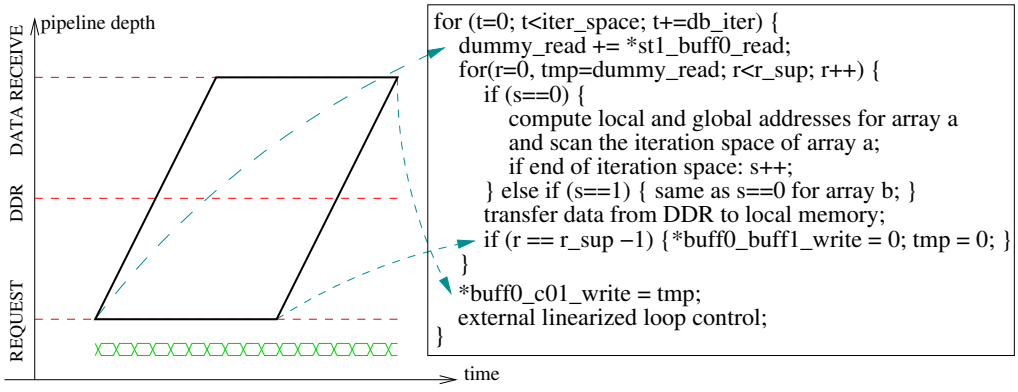


Figure 4.46: Simplified template C code of the BUFF0_LD using Altera's `C2H` semantics

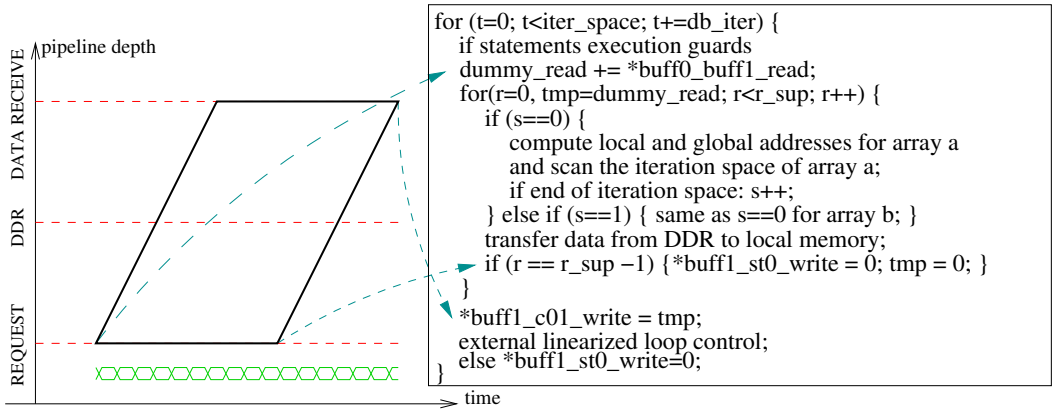


Figure 4.47: Simplified template C code of the BUFF1_LD using Altera's `C2H` semantics

outer loop is iterating over the tiles. The iteration space is multiplied by 2 since at each loop iteration only one tile is computed. When the data-dependence token is received over the `BUFF01_C01` FIFO, the received value is analyzed. When 0 is received, the data stored in `BUFF0` is used for computation and when 1 is received, the data stored `BUFF1` is used. At the end of each computation tile, the data-dependence token is forwarded to the corresponding store accelerator, `STORE0` or `STORE1`.

With this generic technique, it is possible to fetch, in an optimized blocked manner, many blocks of different sizes, each with its individual access addresses, without increasing too much the hardware resources (the only increase is the state machine size of the inner loop). Another advantage is that we can dispatch one or multiple arrays to multiple memories. This should be used jointly with optimizations of the computation accelerator so that parallel computations can be performed on data from different local memories. Therefore, we reached our goal: communication to the DDR is done in a burst manner, controllable by a parameter (the tile size), and the data fetch penalty due to nested loops is paid at the end of each tile, but hidden thanks to the double buffering mechanism.

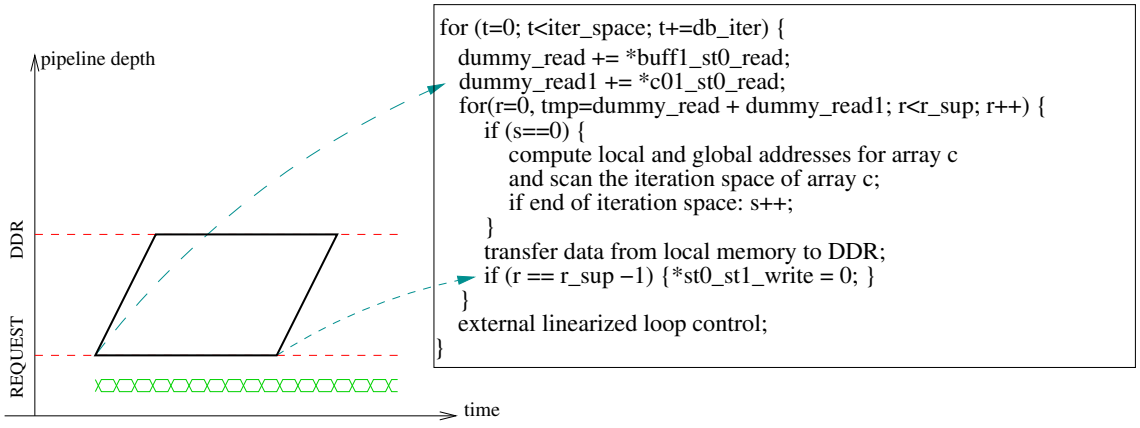


Figure 4.48: Simplified template C code of the STORE0_LD using Altera's `C2H` semantics

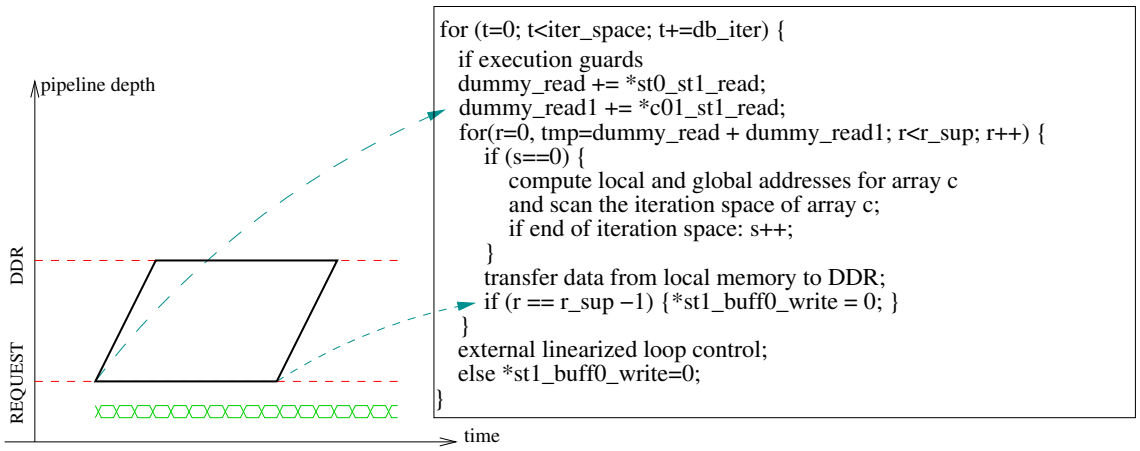


Figure 4.49: Simplified template C code of the STORE1 using Altera's `C2H` semantics

```
for (t = 0; t < iter_space * 2; t+= db_iter)
{
  dummy_read+ = *buff01_c01_read;
  if(dummy_read == 0)
  {
    for (r = 0; r < r_sup; r++)
    { perform computations in a tile; tmp = ...}
    *c01_st0_write = tmp;
  }
  if(dummy_read == 1)
  {
    for (r = 0; r < r_sup; r++)
    { perform computations in a tile; tmp = ...}
    *c01_st1_write = tmp;
  }
  external linearized loop control
}
```

Figure 4.50: Simplified template C code of the COMP01 using Altera's `C2H` semantics

4.5.3 Experimental results

We described earlier the methods and templates that can be used to implement the double-buffering approach using C2H. We now give implementation details and experimental results for three of our previous examples: the DMA transfer, the sum of two vectors, and the matrix-matrix multiplication with parametric sizes.

DMA In Section 4.3.2, we presented the performance bottleneck of the DMA transfer example. To optimize memory accesses, we transformed the code to the double-buffering architecture using the templates presented earlier. The code of the BUFF0_LD accelerator is given in Figure 4.51. The complete code is available in the appendix, Section 7.3.1.

```
#define MAX 16384

for (i = 0; i < MAX; i = i + 2 * BLOCK)
{
     dummy_read += *st1_buff0_read;
     for(j=0,tmp=dummy_read;j<BLOCK;j++)
     {
         if (j == BLOCK-37) //35+1+1
         {
           tmp = 0;
           *buff0_buff1_write=0xdeadbee0;
         }
         buff0[j] = mem_pointer[i + j];
     }
     /* buffer 0 is ready */
     *buff01_c01_write = tmp;
}
```

Figure 4.51: DMA BUFF0_LD source code

```
LOOP i
   CPLI = 8, Loop latency = 13
   Schedule:
     j=0; tmp=dummy_read;:State 0-->9
     i=i+2*100;:State 0-->1
     i< MAX; : State 1-->2
     dummy_read+=*st1_buff0_read;:State 3-->9
     Loop j: State 8-->10
     *buff01_c01_write = tmp;:State 10-->12
     j < BLOCK;: State 6-->7
LOOP j
   CPLI = 1, Loop latency = 40
   Schedule:
     if (j == BLOCK - 37): State 0-->1
     buff0[j]=mem_pointer[i+j];:State 1-->39
     j++: State 0-->1
     j< BLOCK: State 1-->2
     *buff0_buff1_write=0xdeadbee0:State 35-->37
     tmp = 0; State 1-->1
```

Figure 4.52: Scheduling of DMA BUFF0_LD

Since there is only one array to fetch, the code complexity is reduced. As can be observed from the schedule in Figure 4.52, the resource token is scheduled only at state 35. However, the memory transfer is scheduled at the state 1. In order to align the timings of the two operations, we can shift the token-sending statement with the distance between the two states divided by the CPLI. In this example, the difference is 35 and the CPLI is one. We take into consideration that reading and writing to the FIFO takes one cycle each and therefore we add this to the shifting value. The statement `if(j == BLOCK - 37)` implements the shift by 37. Note that this optimization is just to avoid yet another small gap in the DDR accesses: for a block of size 1000, this additional optimization gains 3.7% (37 cycles every 1000 cycles). Also, such a shift is always valid: the token is never sent before the last read request occurs. Anyway, even if the shift was too important, we would only create some interleaving accesses to the DDR (writes and reads), but this would not affect the correctness of the code. For a flow dependence, it would not be safe to play this game however.

The source code of the hardware accelerator BUFF1_LD is presented in Figure 4.53. It is very similar to the BUFF0_LD except for the shifted start by BLOCK of the outer loop and the FIFOs.

As can be observed from its schedule in Figure 4.54, the read from the `*buff0_buff1` FIFO is performed at state 3 of the outer-loop FSM. The read fetch from the memory statement (`buff1[j] = mem_pointer[i+j]`) however starts at state 1 of the inner loop. The inner loop being scheduled at cycle 8 of the outer-loop FSM, the latency of the fetch statement from the FIFO is $8-3+1=6$ cycles. A simple solution to hide this latency is to send the resource token earlier by 6 cycles, however this will imply that the `BUFF1_LD` state machine will start before the last DDR request was sent from the `BUFF0_LD`. If some memory stall occurs during this time, the FSM of the `BUFF0_LD` will stall, but not the FSM of the `BUFF1_LD` (only `BUFF0_LD` is waiting for DDR requests). In this case, after the stall, `BUFF0_LD` will continue sending requests to the DDR at the same time as `BUFF1_LD`. The accesses to the DDR will become interleaved and thus less efficient from the performance point of view. The problem here is thus slightly different: playing this shifting game may lead to send the token before the last read request.

```
#define MAX 16384

for (i = BLOCK; i < MAX; i=i+2*BLOCK)
{
     dummy_read += *buff0_buff1_read;
     for(j=0,tmp=dummy_read;j<BLOCK;j++)
     {
          if (j == BLOCK-37) //35+1+1
          {
              tmp = 1;
              *buff1_st0_write=0xdeadbee2;
          }
          buff1[j] = mem_pointer[i + j];
     }
     /* buffer 1 is ready */
     *buff01_c01_write = tmp;
}
```

Figure 4.53: DMA `BUFF1_LD` source code

```
LOOP i
   CPLI = 8, Loop latency = 13
   Schedule:
     j=0;tmp=dummy_read;:State 0-->9
     i = i + 2 * 100;: State 0-->1
     i< MAX; : State 1-->2
     dummy_read+=*buff0_buff1_read;:State 3-->9
     Loop j: State 8-->10
     *buff01_c01_write = tmp;:State 10-->12
     j < BLOCK;: State 6-->7
LOOP j
   CPLI = 1, Loop latency = 40
   Schedule:
     if (j == BLOCK - 37): State 0-->1
     buff1[j]=mem_pointer[i+j];:State 1-->39
     j++: State 0-->1
     j< BLOCK: State 1-->2
     *buff1_st0_write=0xdeadbee2:State 35-->37
     tmp = 1; State 1-->1
```

Figure 4.54: Scheduling of DMA `BUFF1_LD`

The same latencies are encountered when sending tokens from any hardware accelerators. These latencies are significant only for very small values of the block. As can be seen from Figures 4.55 and 4.56, our double buffering approach with a block size of 2 is actually slower than the original non-blocked version because of these latencies. However, by increasing the block size, the speed-up starts to rise fast, stabilizing at a speed-up of about 6×. The maximum ideal speed-up can be computed as follows:

$$\sigma_{theoretical} = \frac{t_{read}^{interleaved} + t_{write}^{interleaved}}{t_{read}^{burst} + t_{write}^{burst}} = \frac{70ns + 80ns}{10ns + 10ns} = 7.5 \tag{4.8}$$

The theoretical maximum speed-up is thus close to the speed-up we obtained: the original version takes 2.496 ms for 32768 transfers of 32 bits, thus 420 Mb/s, the optimized version provides 2,5 Gb/s. This proves that our double-buffering approach is useful even for algorithms without any data reuse, here a simple move of data, back and forth.

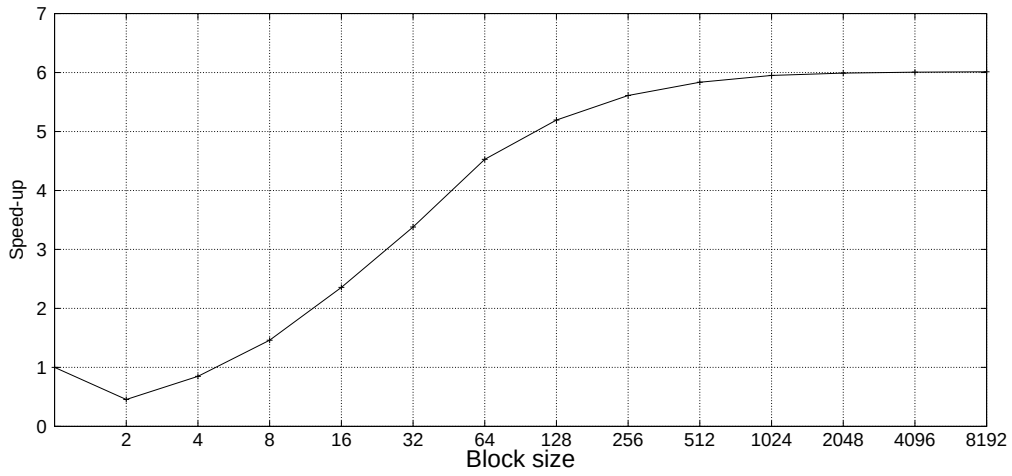


Figure 4.55: Double buffering vs. original DMA experimental results figure

Block size	2	4	8	16	32	64	128
Blocked	5481760	2943940	1709070	1059120	738340	551320	480610
Speed-up	0.45	0.85	1.46	2.36	3.38	4.53	5.19
Block size	256	512	1024	2048	4096	8192	
Blocked	444930	427740	419410	416730	415570	415110	
Speed-up	5.61	5.84	5.95	5.99	6.00	6.01	

Figure 4.56: Double buffering vs. original DMA experimental results table (in ns)

Vector sum Another example we optimized by hand with the double-buffering approach is the sum of two vectors. The performance bottleneck of the vector sum was presented in Section 4.3.2. The optimized code for the `BUFF0_LD` accelerator is given in Figure 4.57. The complete code is available in the appendix, Section 7.3.2.

This source code is more complex that the DMA one, since the inner loop has to fetch two arrays `a` and `b`. As in previous example, the statement sending the resource token `*buff0_buff1_write = ...` was shifted because it was scheduled quite late compared to the DDR request statement `*lmemp = *ememp`. The `j` loop is pipelined with CPLI equal to 1. Increasing the number of arrays to prefetch does not increase neither the CPLI, nor the latency. This points out that the template code can scale without any overhead, even when many different arrays need to be prefetched.

The experimental results comparison between the original version and the double-buffering one is presented in Figures 4.59 and 4.60 (the original version takes 3.828 ms). Again, the double-buffering approach with the block size of 2 is slower because of the inter accelerators communication latencies. However, by increasing the block size, the speed-up starts to rise fast, stabilizing at a speed-up of about 6.5×. The maximum ideal speed-up can be computed as follows:

$$\sigma_{theoretical} = \frac{2 * t_{read}^{interleaved} + t_{write}^{interleaved}}{2 * t_{read}^{burst} + t_{write}^{burst}} = \frac{2 * 80ns + 70ns}{2 * 10ns + 10ns} = 7.66 \tag{4.9}$$

Again, the speed-ups we obtain are quite close to the theoretical speed-ups. The transfers are done at 2.7 Gb/s, i.e., 28 M additions of 32-bits words per second.

Matrix-matrix multiplication Another example we implemented using the same double-buffering approach is the matrix-matrix multiplication (Section 4.3.2, Figure 4.18). We now briefly describe the series of code transformations that we performed by hand to get a code implementing the double buffering and temporal storage.

For the sake of space, the code is presented only in the appendix, Section 7.3.3. We started with the initial code (Appendix 7.3.3) and performed a series of classical loop transformations. The first transformation was to apply loop tiling, which consists in two transformations: strip-mining on `i`, `j`, and `k` loops – with strip sizes as parameters `block_ii`, `block_jj`, and `block_kk` – then loop interchange of the loops obtained after strip-mining. The resulting code iterates over tiles with the loops `ii`, `jj`, and `kk`, and inside the tiles with the loops `i`, `j`, and `k`. To be able to reuse data, we inserted local arrays of the size of the original ones. However, since the local storage is usually smaller than the external DDR memory, the arrays are downsized using a (here straightforward) packing technique to the size of a tile. Loop distribution is then applied to the code on `i`, `j`, and `k` to separate the loads of the arrays `a` and `b` from the computation and from the stores of the array `c`. An unroll by 2 is applied on the loop `ii` and code motion is applied to get a code that looks like an execution using double buffering. To be able to fully separate the two parts of the double buffering approach, we apply array privatization on local arrays `tmp_a`, `tmp_b`, and `tmp_c` obtaining arrays `tmp_a_0` and `tmp_a_1`, same for `b` and `c`. From here, to obtain a code that corresponds to the desired template code, we applied a loop linearization technique to transform the nested loops iterating over the tiles and the ones iterating inside the tiles.

The obtained code was partitioned into multiple parts (functions). Each part was later synthesized by `C2H` as one of the accelerators of Figure 4.42. To validate the correct behavior of the code, we executed each part as a Linux process. Each process was instructed with read-from and send-to FIFOs using Linux FIFO constructs. FIFOs and shared-memory elements emulating local

```
/* n = 16384 */
for(ii = 0; ii < n; ii = ii + 2 * block)
{
    dummy_read += *st1_buff0_read;
    j = 0; i = 0; ij = 0;bool_nsent = 1;
   //*2 for a and b
    for(k=0,tmp=dummy_read;k<block*2;k++)
    {
        offset_bufflocal = i;
        offset_ext_memp = ii + i;

        if (i < block - 1) {i++; }
        else  i = 0;

        if (ij == block) {j++; ij = 0;}
        else ij++;

        if (j == 0) {
            lmemp=buff0_0+offset_bufflocal;
            ememp = a + offset_ext_memp;
        }else if (j == 1) {
            lmemp=buff0_1+offset_bufflocal;
            ememp = b + offset_ext_memp;
        }
        /* transfer code */
        *lmemp = *ememp;
        /* sync code */

        if(((k==block*2-35)|(k==block*2-1))
             & bool_nsent) //-34 -1
        {
            bool_nsent = 0;
            tmp = 0;
            *buff0_buff1_write = 0xdeadbee2;
        }
    }
    /* buffer 0 is ready */
    *buff01_c01_write = tmp;
}
/* void the fifo */
dummy_read += *st1_buff0_read;
```

Figure 4.57: Vector sum BUFF0_LD source code

```
LOOP i
   CPLI = 8, Loop latency = 13
   Schedule:
     j = 0; tmp = dummy_read; : State 0-->9
     i = i + 2 * block;: State 0-->1
     ii< n; : State 1-->2
     dummy_read+=*st1_buff0_read;:State 3-->9
     Loop k: State 8-->10
     *buff01_c01_write = tmp;:State 10-->12
     k < block;: State 6-->7
     bool_nsent = 1;: State 9-->9
     ij = 0; State 9-->9
     i = 0; State 9-->9
     i = 0; State 9-->9
LOOP j
   CPLI = 1, Loop latency = 41
   Schedule:
     offset_ext_memp=(ii+i);: State 0-->1
     if (i < block -1) : State 0-->1
     i++; State 0-->1
     if (ij == block) : State 0-->1
     ij++; State 0-->1
     j++; State 0-->1
     if (j == 0): State 1-->2
     if (j == 1): State 1-->2
     ememp=(b+offset_ext_memp);:State 1-->2
     ememp=(a+offset_ext_memp);:State 1-->2
     *lmemp = *ememp; : State 2-->40
     if(((k==block*2-35)|... : State 0-->1
     k++ : State 0-->1
     k < block * 2; : State 0-->1
     *buff0_buff1_write=0xdeadbee2:\
                              State 36-->38
     tmp = 0; State 1-->1
     bool_nsent = 0; State 1-->1
     lmemp=(buff0_0+offset_bufflocal);:\
                              State 37-->38
     ij = 0; State 1-->1
     i = 0; state 1-->1
     lmemp=(buff0_1+offset_bufflocal);:\
                              State 37-->38
     offset_bufflocal=i; State 37-->37;
```

Figure 4.58: Scheduling of vector sum BUFF0_LD

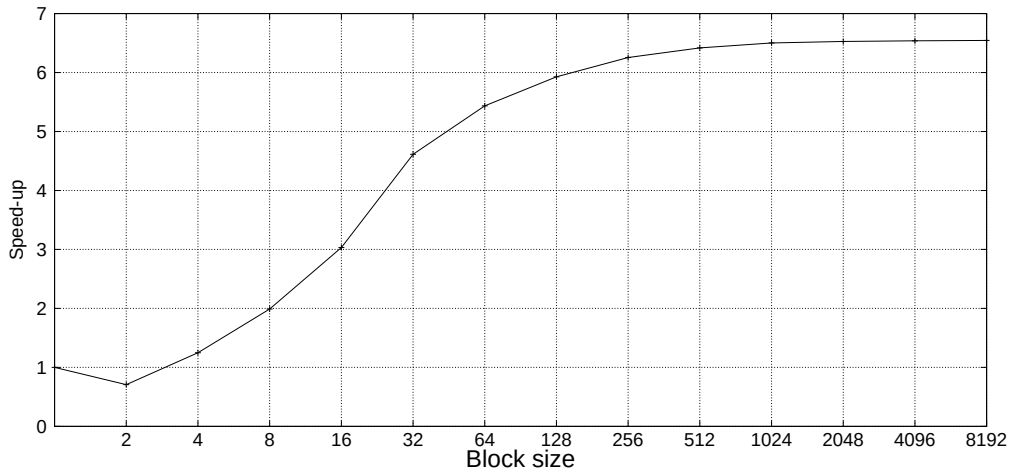


Figure 4.59: Double buffering vs. original vector sum experimental results figure

Block size	2	4	8	16	32	64	128
Blocked	5412340	3067580	1924200	1262690	829860	704490	645900
Speed-up	0.71	1.25	1.99	3.03	4.61	5.43	5.93
Block size	256	512	1024	2048	4096	8192	
Blocked	612160	596620	588960	586580	585680	584990	
Speed-up	6.25	6.48	6.50	6.53	6.54	6.54	

Figure 4.60: Double buffering vs. original vector sum experimental results table (in ns)

memories were instantiated using Linux constructs. After the verification, the code was translated in a straightforward way to the final code (see the appendix, Section 7.3.3), transforming only local memory address connections and FIFO connections.

To obtain a better data reuse, the tiles of array `c` were stored in the local memory (see Figure 4.61). The computations were performed by two parts `double_buffering0` and `double_buffering1`. The front plane of the cube representing the iteration space corresponds to the part where a tile of elements are loaded from the memory. In our case, we inserted the initialization with zero in the accelerators, so we did not prefetch elements of the array `c` from the memory (this would not have been possible if the double buffering was not done along the axis `k`). After each tile, the elements of the array `c` are not stored to the DDR but reused at the next tile. Only for the back plane, i.e., the last iterations of `kk` and `k`, the elements of `c` are stored to the DDR memory.

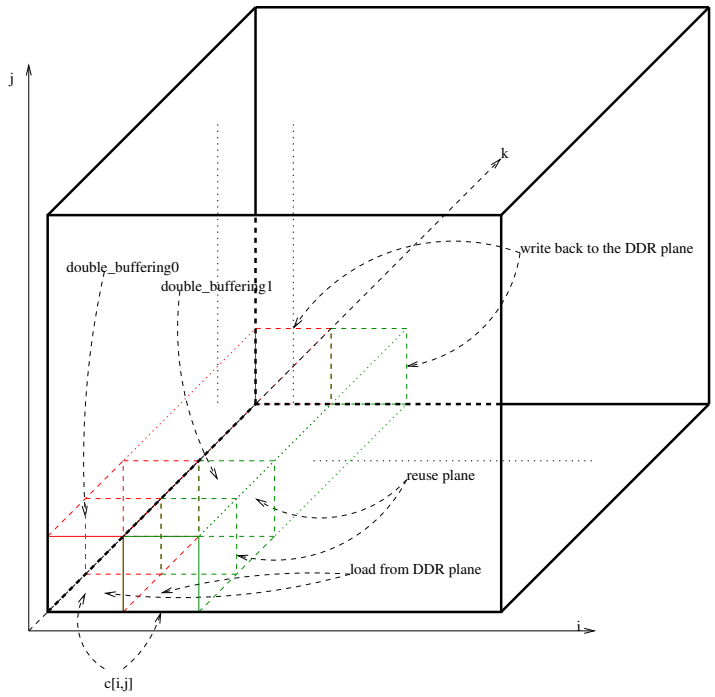


Figure 4.61: Matrix-matrix multiplication blocking reuse

The experimental results for the matrix-matrix multiply example, using our double buffering technique, are presented in Figures 4.62 and 4.63. In these experiments, the values of `n0`, `m1`, and `m0` were fixed to 20. The values of `block_ii`, `block_jj`, and `block_kk` were doubled for every benchmark iteration, obtaining the block sizes from Figure 4.63. As in previous examples, for a small block size, the double-buffering code runs slower because of the inter-accelerator communication latency overheads. The speed-up peaks at a block size of 216 is equal to about 7.37 (the original version takes 1.412 ms). The speed-up for a block of size 1296 is smaller than the previous one because the iteration space of outer loops is not an exact multiple of the tile sizes. As there are only a few writes compared to read operations, the theoretical speed-up of the double-buffering approach, for matrix-matrix multiply, is then roughly:

$$\sigma_{theoretical} = \frac{t_{read}^{interleaved}}{t_{read}^{burst}} = \frac{80ns}{10ns} = 8 \tag{4.10}$$

The maximum theoretical speed-up (8) is very close to the speed-up (7.37) we obtained. But this is without taking into account the fact that the data reuse increases with the block size. According to Equation (3.5), $2 \times \frac{20^3}{b} + 20^2$ data are accessed (here, the matrix c is initialized in the hardware accelerator), where b is the block size. For the best version, for $b = 64$, the data transfer rate we obtained is 109 Mb/s. The reason why the average transfer rate is so low is because now the accelerator is limited by the computation rate: the computation kernel is not parallelized and needs, for each tile, order b^3 states for order b^2 transfers. Since the hardware accelerator runs at 100 MHz and performs (at least) 20^3 sequentially-executed operations, the best execution time we could expect is 80000 ns, and communications should not slow the accelerator down. However, here, the implementation uses the synchronization diagram of Figure 4.43 where no computation is performed during `BUFF0_LD` and `STORE1`. Therefore, communications are not completely hidden by computations. We would need to use a more suitable coarse-grain software pipeline, for example those of Figures 4.44 or 4.66.

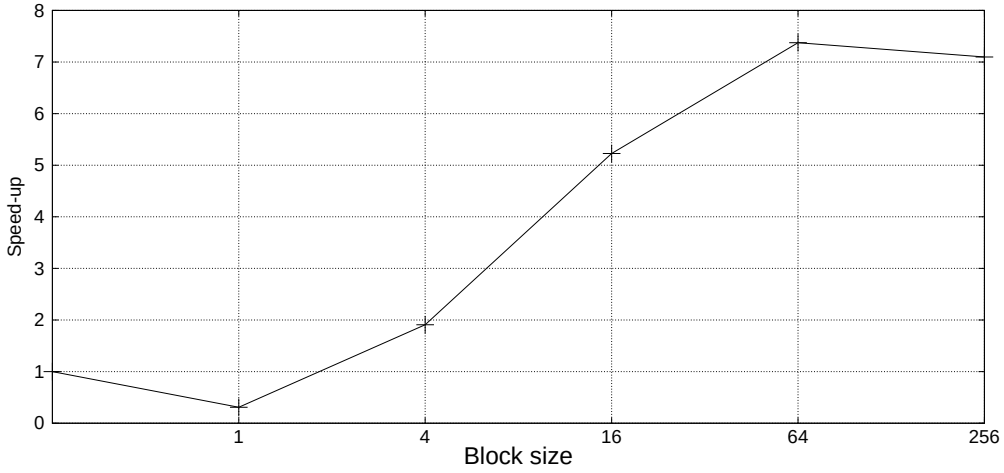


Figure 4.62: Double buffering vs. original matrix-matrix multiplication experimental results figure

Block size	1	4	16	64	256
Blocked	4578800	740430	270300	191580	199060
Speed-up	0.30857	1.90822	5.22719	7.37503	7.09791

Figure 4.63: Double buffering vs. original matrix-matrix multiplication experimental results table

Synthesis results Earlier, we presented the functional simulation time of three examples. However, in order to validate the complete hardware design flow, we have to analyze other hardware-related parameters. In Figure 4.64, we present the synthesis results of the direct and the optimized versions of the previously-explained examples.

The designs were synthesized on the Altera Stratix II `EP2S180F1508C3` FPGA. The FPGA system is running at 100 MHz. It is connected to the outside DDR having the following specification parameters: JEDEC DDR-400 128 Mb x8, column address strobe (CAS) = 3.0. ALUT represents the number of Altera look-up tables used. "Dedicated registers" represents the number of the registers the design uses. "Total number of registers" includes dedicated registers and additional registers used by the synthesis tool to optimize some specific parameters such as the circuit frequency. The "total block memory bits" represents the utilization of BRAM memories from the FPGA. DSP block 9-bit elements represents the utilization of hard 9-bit multiplication IP cores that are present on the FPGA. Finally, we present the maximum frequency at which the whole system can run.

The comparisons of the original and optimized simulation time show that the optimized version can run 6 times or more faster than the direct implementation (Figure 4.64). There is a small price to pay from the point of view of hardware resources to achieve this. The optimized designs use two twice more Altera look-up tables (ALUTs) and almost three times more registers and memory blocks than the original design. However, we have to take into account that the optimized design uses only about 6% of the FPGA resources while saturating the DDR memory bandwidth.

The optimized version also has a slightly smaller maximum running frequency than the original design. This is mostly due to the Avalon interconnect routing. However, if the design already saturates the memory bandwidth at 100 MHz, running the system at higher frequencies will not speed up the design. Note that, as we mentioned previously, the situation is different for matrix-matrix multiplication: either computations can be accelerated so that the algorithm is limited by bandwidth or it is important to always overlap communication with computation.

Kernel	Speed-up	ALUT	Dedicated registers	Total registers
System alone	-	4406	3474	3606
DMA direct implementation	1	4598	3612	3744
DMA double buffering	6.01	9665	10244	10376
Vector sum direct implementation	1	5333	4607	4739
Vector sum double buffering	6.54	10345	10346	11478
Matrix-matrix multiplication direct impl.	1	6452	4557	4709
Matrix-matrix multiplication double buffering	7.37	15255	15630	15762

Kernel	Total block memory bits	DSP block 9-bit elements	Max Freq. (MHz)
System alone	66908	8	205.85
DMA direct implementation	66908	8	200.52
DMA double buffering	203100	8	167.25
Vector sum direct implementation	68956	8	189.04
Vector sum double buffering	269148	8	175.93
Matrix-matrix multiplication direct impl.	68956	40	191.09
Matrix-matrix multiplication double buffering	335196	188	162.02

Figure 4.64: Synthesis results of DMA, vector sum and matrix-matrix multiply examples using direct and double buffering techniques

4.5.4 Coarse-grain software pipelining

In Section 4.5, the optimization scheme we propose, with communicating accelerators, was illustrated with one particular synchronization pattern. We now show how such a synchronization structure can be found automatically.

Each (reading, computing, or writing) accelerator works at the block granularity, and iterates thanks to an outer loop over blocks. As depicted in Figure 4.42, the different accelerators synchronize each other, at the block boundary, using blocking FIFOs of size 1, each acting as a token. Depending on the desired synchronization semantics, the token is sent either at the last iteration of the inner loop that scans a block, or just after this loop (see Figure 4.45). In the first case, the token is used to enforce a *resource constraint*, i.e., to sequentialize the accesses to a given resource (here the communication medium on which DDR requests are sent). In the second case, the token is used to enforce a *dependence constraint*, e.g., to make sure that a data read in the local memory has indeed arrived. These synchronizations, all together, finally enforce a particular pipelined execution of the blocks computed by the accelerators, as depicted in Figure 4.43.

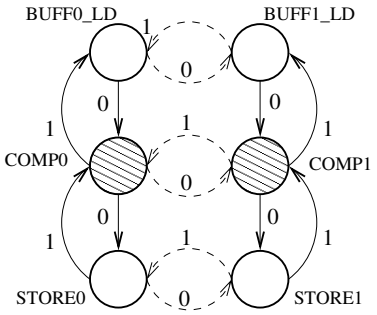


Figure 4.65: Dependence graph

What we did here is nothing but a coarse-grain *software pipelining* at block level, considering each individual accelerator has a macro-instruction that reads or writes (local and external) memories, in a common outer block loop to be pipelined. The standard approach is to define a directed multi-graph, where each vertex is a macro-instruction and each edge describes a dependence, as depicted in Figure 4.65 for a simple read/write case with double buffering as in Section 4.5. Each vertex requires a particular resource, either a computing accelerator or the communication medium, if possible with corresponding durations. Edges are labeled with dependence distances, usually 0 (loop independent) or 1 (loop carried) and, if possible, with corresponding latencies. Additional dependencies can be added, to constrain the problem, as for example to ensure that the block orders are preserved (dashed lines in Figure 4.65). Then, standard software pipelining techniques [98] can be applied to define a periodic schedule for the particular instance to solve. This periodic schedule defines a sequential order for each resource, in particular for the communication medium. When this order is not already structurally ensured, a synchronization is added. For example, if the selected schedule placed `BUFF0_LD(t)` just before `BUFF1_LD(t)`, a synchronization is added, thanks to a FIFO `BUFF0_BUFF1`, as in Figure 4.43. This model makes the search for a good schedule more systematic. For example, to avoid the gap mentioned in Section 4.5 when `COMP0(t)` delays `STORE0(t)`, it is possible to find a solution without an overlap between `STORE0(t-1)` and `COMP0(t)`, thus with no extra memory duplication, as shown by the (non intuitive) software pipeline of Figure 4.66.

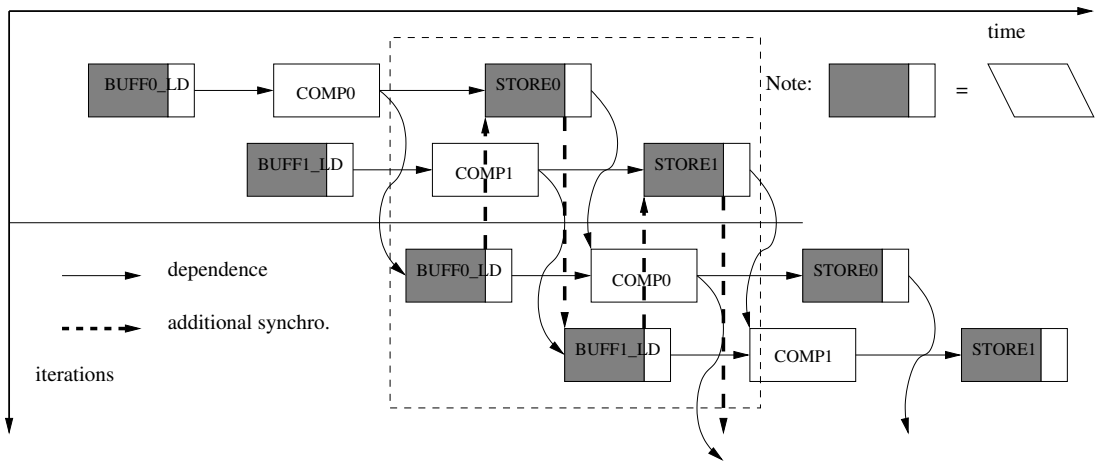


Figure 4.66: Software pipeline

4.5.5 Automation of code transformations

Sections 4.5 and 4.5.4 defined the synchronization mechanisms that enable to pipeline communications by blocks, avoiding both the row change penalty, due to the DDR specification, and the data fetch penalty, due to the way `C2H` schedules nested loops. The latter can be considered as a limitation of `C2H` but, actually, this is also what makes the synchronization at C level possible. These synchronized accelerators can now be considered as a kind of run-time system, described at C level and compiled by `C2H` itself, on top of which high-level transformations are made to re-organize the code. They identify blocks of computations, prefetch the required data from the DDR, store them locally in the accelerator, and transfer results to the DDR, in other words, i.e., they fill the communication and computation templates defined in Section 4.5 with the adequate codes.

These tasks require both program analysis and code transformations, in particular optimizations based on polyhedral techniques. We list here the main steps that are necessary, as well as related references.

Loop tiling and partial unrolling First, loop tiling [110] (or, for a single loop, strip-mining) divides the kernel into elementary blocks of computations to be executed with a double buffering scheme. For the matrix-matrix product, this corresponds to a block-based version. The double buffering scheme is then performed along the last loop describing tiles (loop unrolling by 2). A good tiling should reduce the volume of necessary data transfers, as well as the sizes of the local memories needed (memory footprint optimization), while trying to leave some data in local memory from one tile to another (temporal reuse). Many approaches exists in the HPC community to compute such a tiling (see references in [110]), even for non perfectly-nested loops [35].

Communication coalescing The second step is to identify, for a given tile, the data to read from and to write to the DDR, excluding those already stored locally or that will be overwritten before their final transfer to the DDR (as the array `c` in the matrix-matrix product of Section 4.3.2). This is a particular form of communication coalescing as described in [43], for HPF, to host communications outside loops (here the loops describing one tile). Then, the set of transferred data is scanned [97,

37], preferably row by row. Even if an array is accessed by column in a tile, as for the matrix-matrix product, the corresponding data can be transferred by row.

Contraction of local arrays Then, a mapping function must be defined that convert indexes of the global array (in the DDR) to local indexes of a smaller array in which the transferred data are stored. Standard lifetime analysis and array contraction techniques [74, 85, 23] can be used for that.

Code specialization The final transformation is to replace nested loops that scan data sets or that define the computations in a tile by a linearization, as in the juggling code of Section 4.4. This, again, is to avoid any data fetch penalty. Also, once the different accelerators are defined, a coarse-grain software pipeline is determined, as explained in Section 4.5.4, and synchronizations (`fifo_read` or `fifo_write`) are placed as explained in Section 4.5.

All these steps will be detailed in the next chapter.

Chapter 5

Automation of code transformations

5.1 Introduction

High-level synthesis tools provide a convenient level of abstraction to implement complex designs. However, the program optimizations proposed by these tools are not as sophisticated as those of high-performance compilers, and the input program must be written in the proper way to get performances. This often leads the designers to give up the abstraction level and to guess how the code must be written to get performances and even, sometimes, to make it correct. For HLS tools to be viable, this issue needs to be addressed and HLS-specific optimizing program restructuring must be designed to allow the user to fully take advantage of the input abstraction level. This is particularly true for the `C2H` C-to-VHDL compiler from Altera, for which we showed in Chapter 4 how the input program needs to be restructured to get performances.

In this chapter, we show how to automate this restructuring. Given a naive input implementation in C, we present a method to automatically derive a `C2H`-compliant version following the optimization scheme discussed in Chapter 4. We first recall some preliminary concepts and notations, and outlines the three main steps of our method, which are detailed in Sections 5.2, 5.3, and 5.4. Our method has been fully implemented and Section 5.5 provides experimental results comparing the performances of the hardware accelerators generated from the program optimized by hand and from the program optimized automatically, thanks to our method. Finally, Section 5.6 concludes this chapter, and gives future directions.

5.1.1 Preliminaries

The polyhedral model [69] is a general framework of program analysis and transformations applied to static control programs. A *static control program* verifies the following properties:

- **Data types.** Only array and scalar variables are allowed. Pointers are forbidden.
- **Control.** The control is restricted to `for` loops with clearly-identified loop counters, conditionals (`if`), and sequence. Irregular control structures as while loops are forbidden.
- **Predictability.** Loop bounds, conditions, and array access functions must be *affine expressions* of the surrounding loop counters and structure parameters (*e.g.*, array size). This way, the execution of the program does not depend on the input values. It is always the same and can be predicted at compile-time.

Operations and iteration domains Given an assignment S, the vector built with surrounding loop counters is called an *iteration vector* of S. The set of iteration vectors of S reached during program execution is called the *iteration domain* of S. Because of static control restrictions, the iteration domains are **invariants**. They are still the same whatever the input is. The execution of S at the iteration $\vec{i}$ is exactly characterized by the couple $(S, \vec{i})$, called *operation*. The ability to produce program analysis and transformations at the *operation level* (*instance-wise*) rather than at *assignment level* is the key feature of the polyhedral model, and makes the polyhedral analysis powerful and far beyond classical approaches. As loop bounds and conditions are affine, an iteration domain is exactly the set of integral points lying in a polyhedron, a $\mathbb{Z}$-polyhedron. This fundamental property of the polyhedral model allows to design program analysis in terms of $\mathbb{Z}$-polyhedra manipulation. Figure 5.1 shows the iteration domain for the example of the polynomial product. The operation-level data-dependence are depicted with red arrows.

```
    for (i=0; i<= 2*N; i++)
S1:    c[i] = 0;

    for (i=0; i<=N; i++)
      for (j=0; j<=N; j++)
S2:      c[i+j] = c[i+j] + a[i]*b[j]
```

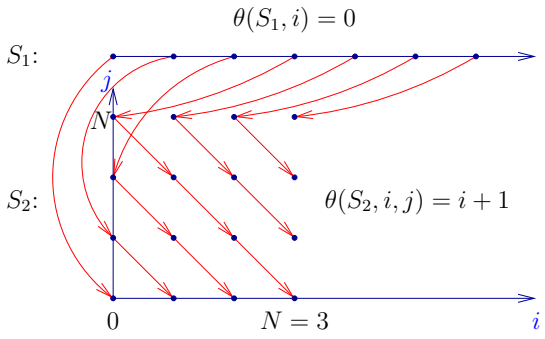


Figure 5.1: Polynomial product: iteration domains, data dependences, and possible schedule

Affine schedules Many program transformations change the execution order of the operations. This can be specified thanks to a *schedule*, an application mapping each operation ω to an execution date $\theta(\omega)$. A schedule is *valid* if the data dependences are respected. If there is a data dependence from an operation α to an operation β: $\alpha \to \beta$, then α must be executed before β: $\theta(\alpha) < \theta(\beta)$. The number of different execution dates is called the *latency* of the schedule.

In the polyhedral model, we define a schedule with an affine function: $\theta(S, \vec{i}) = \vec{\tau_1} \cdot \vec{i} + \vec{\tau_2} \cdot \vec{p} + c$ where $\vec{\tau_1}$ and $\vec{\tau_2}$ are integral vectors, $\vec{p}$ is a vector of parameters, and c is an integer constant. For example, in Figure 5.1, the affine schedule given by $\theta(S_1, i) = 0$ and $\theta(S_2, i, j) = i + 1$ is valid for the polynomial product (Figure 5.1). This means that the program will be executed in $N + 2$ steps. The step 1 (at date 0) executes in parallel all the instances of S_1. Then, each following step (dates from 1 to $N + 1$) executes in parallel the operations in a column of the iteration domain of S_2.

This works only for schedules with a linear latency. When the latency is more than linear, we specify a *multi-dimensional affine schedule*. The execution date $\theta(S, \vec{i})$ is defined by a vector whose components are also affine expressions of loop counters and parameters. The execution dates are then totally ordered with the *lexicographic order* $\ll$. For example, the multi-dimensional affine schedule given by $\theta(S_1, i) = (0, i)$ and $\theta(S_2, i, j) = (1, i, j)$ specifies the sequential order for the product of polynomials.

In the remainder, we will deal with mono- and multi-dimensional affine schedules, that we will refer as *schedules* for the sake of simplicity.

Systems of clauses Affine schedules are a key notion of the polyhedral model, which subsumes most of loop transformations and compositions thereof. Also, when thinking with polyhedra, the design of a program optimization almost boils down to compute an affine schedule. Another point is that a program can be specified completely with a system of clauses:

$$\left[\vec{i} \in D_1 : S_1(\vec{i})\right] \vee \ldots \vee \left[\vec{i} \in D_n : S_n(\vec{i})\right]$$

together with a schedule θ, where D_k is a $\mathbb{Z}$-polyhedron specifying the iteration domain of the assignment S_k, $S_k(\vec{i})$ stands for the instance of the assignment S_k with the iteration vector $\vec{i}$ (actually, the operation $(S_k, \vec{i})$), and θ gives the execution date for each valid operation $(S_k, \vec{i})$. A system of recurrence equations (SRE) and a system of affine recurrence equations (SARE) can be viewed as specific systems of clauses representing a program in dynamic single assignment form. There exist algorithms [97, 37] to generate a compilable C program from a system of clauses, thus giving iteration domains, assignments, and schedules is sufficient to generate C code.

Loop tiling Loop tiling [79, 110] is a key loop transformation in program optimization, which has proven to be effective for automatic parallelization and data locality improvement [110, 88, 35]. The iteration domain of a loop nest is partitioned into rectangular tiles, which are executed atomically. A first tile is executed, then another tile, and so on. Loop tiling is often defined as a composition of several loop strip-mines and several loop interchanges. The loop strip-mine introduces two kinds of loops: the *tile loops*, which iterate over the tiles, and the *intra-tile loops* which iterate into a tile. Then, the loop interchange pushes the intra-tile loops in the inner dimensions of the loop nest. It is not always possible to tile a loop nest. Sometimes, extra loop transformations are necessary to make a loop nest tilable. In some cases, we need a skewing (torsion) of the iteration dimension, meaning that a tile in the original iteration domain is a parallelogram rather than a rectangle.

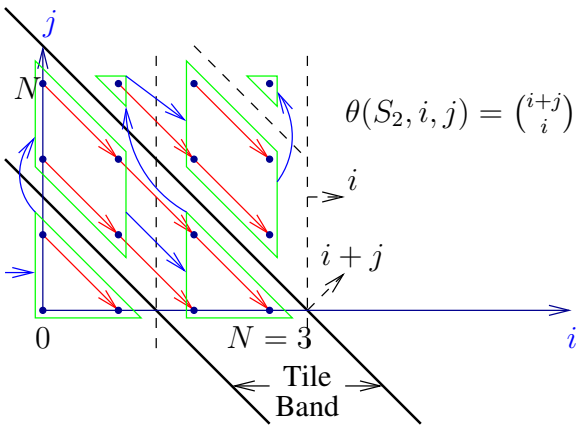


Figure 5.2: Polynomial product: loop tiling

In the polyhedral model, we can specify a loop tiling for a perfect loop nest of depth n with a set of affine hyperplanes $\mathcal{H} = (H_1, \ldots, H_n)$, where $H_k = \{\vec{i} \mid \vec{\tau_k} \cdot \vec{i} = 0\}$. The vector $\vec{\tau_k}$ is the normal to the hyperplane H_k and the vectors $\vec{\tau_1}, \ldots, \vec{\tau_n}$ are supposed to be linearly independent. Then, the iteration domain of the loop nest can be tiled with regular translations of the hyperplanes of $\mathcal{H}$, keeping the same distance b between two hyperplanes. For H_1, this gives H_1, then $H_1 + b\vec{\tau_1}/||\tau_1||^2 = \{\vec{i} \mid \vec{\tau_k} \cdot \vec{i} = b\}$, then $H_1 + (2b)\vec{\tau_1}/||\tau_1||^2 = \{\vec{i} \mid \vec{\tau_k} \cdot \vec{i} = 2b\}$, and so on. Similarly, for H_2, this gives H_2,

then $H_2 + b\vec{\tau_2}/||\tau_2||^2$, then $H_2 + (2b)\vec{\tau_2}/||\tau_2||^2$, ..., and so on, and the same for H_3, ..., H_n. (This is if all tile sizes have the same size b.) A valid iteration vector $\vec{i}$ belongs to a tile if, for each hyperplane H_k, there exists an integer I_k such that:

$$I_k b \leq \vec{\tau_k} \cdot \vec{i} \leq I_k b + b - 1, \text{ for each } 1 \leq k \leq n$$

The integers $I_1, \ldots, I_n$ actually define a tile, and can be viewed as tile loop counters. Fixing an integer value for b, and adding these constraints to those of the original iteration domain gives an iteration domain D' of dimension $2n$, which is tiled according to the hyperplanes of $\mathcal{H}$. If the body of the original perfect loop nest has a n-dimensional schedule θ, compatible with loop tiling (i.e., the n dimensions of the schedule are permutable), then a valid sequential schedule of the tiled loop nest, for statement S, is:

$$\theta_{\text{tiled}}(S, I_1 \ldots I_n, \vec{i}) = (I_1, \ldots, I_n, \theta(S, \vec{i}))$$

Finally, giving S, D', and θ_{tiled} to a polyhedral code generator [97, 37, 32] will generate the tiled code. Figure 5.2 gives the tiled version for the product of polynomials discussed above. The tiling hyperplanes are defined by the normal vectors $\vec{\tau_1} = (1, 1)$ and $\vec{\tau_2} = (1, 0)$, and $b = 2$. The tiling is correct, as $\vec{\tau_1} \cdot (1, -1) = 0 \geq 0$ and $\vec{\tau_2} \cdot (1, -1) = 1 \geq 0$. The tiled iteration domain is defined by:

$$\{(I, J, i, j) \mid 0 \leq i, j \leq N \wedge 2I \leq i + j \leq 2I + 1 \wedge 2J \leq i \leq 2J + 1\}$$

Also, the schedule for the tiled code for statement S_2 is $\theta_{\text{tiled}}(S_2, I, J, i, j) = (I, J, i + j, i)$.

A *tile band* is the set of tiles described by the last tile loop, on a given iteration of the outer tile loops (see the tile band for $I = 1$ in Figure 5.2). This notion will be widely used in our approach, as most optimizations will be done within a tile band, parameterized by outer tile loops counters.

5.1.2 Overview of the method

Given a kernel written in C, we present a method to *derive automatically the* C2H*-compliant C functions for the pipelined accelerators* load, store, and compute. For convenience, the kernel is supposed to fit into the static control model, so polyhedral analysis can be applied. In this program model, the blocks of computation to be executed with the double-buffering scheme [1] can be specified with a *loop tiling*. Then, *inside each tile band*, the double buffering is applied on the tiles two-by-two, following the sequential execution order. Figure 5.3 shows how the double buffering is applied on the matrix multiply example. The loop nest is tiled along the canonical directions, *i.e.*, the tiling hyperplanes are defined by the normals $\vec{\tau_1} = (1, 0, 0)$, $\vec{\tau_2} = (0, 1, 0)$ and $\vec{\tau_3} = (0, 0, 1)$. In the remainder, the tile loop counters will be written I, J, and K.

We leave the choice of tiling to the user, which must be specified by means of affine hyperplanes for each statement, thanks to a schedule. Then, our method executes the following steps:

Step 1. **Communication coalescing** identifies for each tile the data to be loaded from and stored to the DDR. This is a critical step, as the traffic with the DDR must be reduced and performed by blocks. Also, this impacts the size of the local memory. The goal is also to exploit as much as possible the data reuse between tiles. This step is described in Section 5.2.

1. Actually, "double-buffering scheme" is a language simplification: we do not really use two buffers, but one larger buffer. However, two blocks of computation are indeed pipelined with two blocks of communications, so as to overlap communications and computations.

```
for(i=0; i<=N; i++)
  for(j=0; j<=N; j++)
    for(k=0; k<=N; k++)
      c[i][j] += a[i][k]*b[k][j];
```

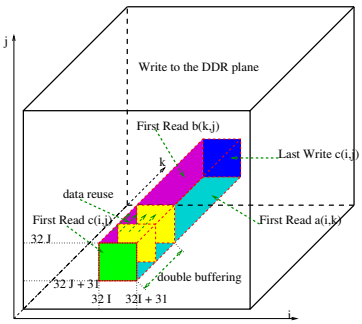


Figure 5.3: Applying the double-buffering scheme on matrix multiply

Step 2. **Local memory management** computes the size of the local buffers, together with an adequate access function. Given a communication coalescing, the local memory size must be reduced as much as possible. This step is described in Section 5.3.

Step 3. **Code Generation** generates the final C code for the accelerators, given the pieces of information computed above. The code must meet several requirements to get effective performances: a) the loops must be linearized and b) the execution order must respect the schedule defined by the tiling. This step is described in Section 5.4.

5.2 Communication coalescing

This section presents a method to select the array regions to be loaded from and stored to the DDR for each tile. This step will impact two important criteria: a) the amount of communications with the DDR and b) the size of the local memory. At first glance, it may seem that these criteria are antagonistic. Actually, we prove that, with our scheme, this is not the case. Both can be minimized at the same time.

A naive solution. Given the loop tiling, a naive solution would be, for each tile, to load all the data read in the tile and to store all the data written in the tile. This solution, although correct, would maximize the communications with the DDR and would lead to poor performances. Also, as the next load is executed before the current store, this would forbid dependences between two consecutive tiles within a tile band.

Our solution. We send load and store requests to the DDR only when it is needed, with *communication coalescing*. For each tile, we load from the DDR the data read for the *first time* in the current tile band. And we store to the DDR the data written for the *last time* in the current tile band. Meanwhile, the data is kept and used (read and written) in the local memory. This means that the data reused between the tiles of a tile band are kept in local memory. This way, the transfers with the DDR are *minimized*. On the matrix multiply example, the array `c` is loaded from the DDR only by the first tile of the band, and finally stored to the DDR by the last tile. With the naive approach, `c` would have been loaded and stored for every tile of the band.

As a bonus, the method no longer requires two consecutive tiles of a tile band to be dependence-free. Indeed, the data concerned by the inter-tile data dependences are kept in local memory and the compute accelerators are executed sequentially, guaranteeing the correctness of the program. Consequently, any tiling would produce a correct result.

Nonetheless, the tiling hyperplanes need to be chosen carefully to reduce the communication. With our approach, the loads from the DDR correspond to dependences (including input depen-

dences) entering the tile band. Similarly, the stores to the DDR correspond to dependences getting out of the tile band. These dependences are responsible for the communications with the DDR, and must be reduced as much as possible. A way to proceed is to find tiling hyperplanes such that the last hyperplane traverse as many dependences as possible. This is equivalent to push the dependences in the innermost loop. Several approach exist to find such hyperplanes, for perfect loop nests [110] as well as imperfect loop nests [35]. Figure 5.4 show what happens for the polynomial product example, with a naive tiling and with the skewed tiling previously defined, focusing on the communications related to array `c`. With the tiling defined from the schedule $(-j, i)$ (which corresponds to a loop interchange and a loop reversal of the j loop), many data dependences get in and out of the tile band causing as many loads and stores for the array `c` as shown in Figure 5.4(a). With the skewed tiling depicted in Figure 5.4(b), the data dependences are kept in the tile band. This way, the loads and stores for `c` only arise on the first tile and on the last tile of the tile band. Notice that the loads and stores for the array `a` are the same in both cases. However, communications for array `b` are reduced with the first tiling, with full reuse within a tile band.

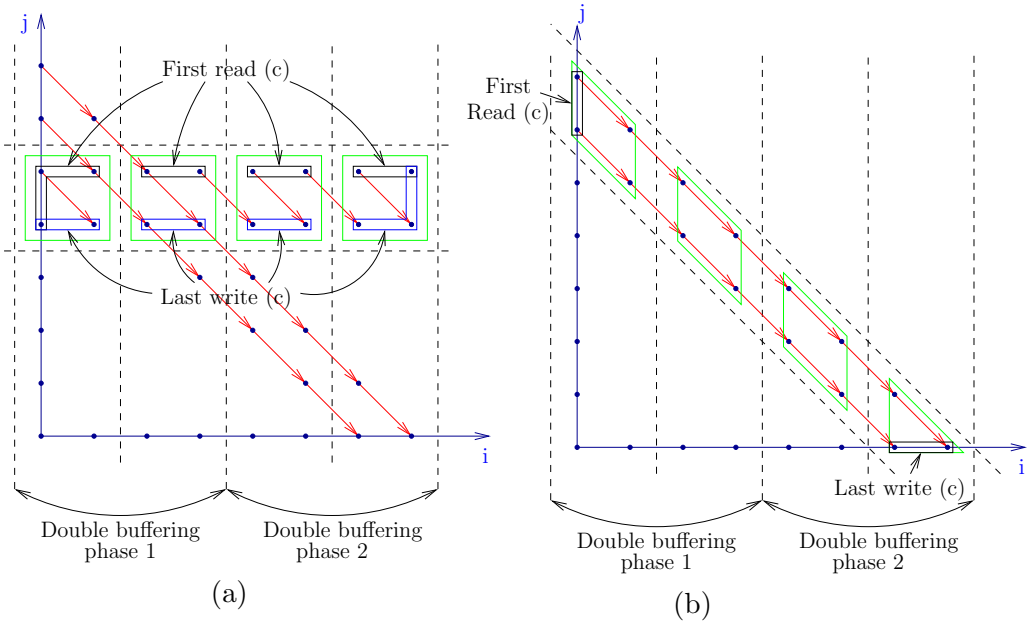


Figure 5.4: Impact of the loop tiling on the communication volume

5.2.1 Specification

For every tile t, we want to specify Load(t), the data to be loaded from the DDR just before executing the tile, and Store(t), the data to be stored to the DDR just after executing the tile. Let In(t) be the data read in the tile t (before being possibly rewritten) and Out(t) be the data written in t. Here, we assume the sets In(t) and Out(t) to be exact. The approximations are studied later.

Definition 1 (Valid Load**)** *The function $t \mapsto \text{Load}(t)$ is valid if and only if (iff) the following conditions hold for any tile T.*

(i) $\cup_{t \leq T} \{\text{In}(t) \setminus \text{Out}(t' < t)\} \subseteq \text{Load}(t \leq T)$.

(ii) $\text{Out}(t < T) \cap \text{Load}(T) = \emptyset$.

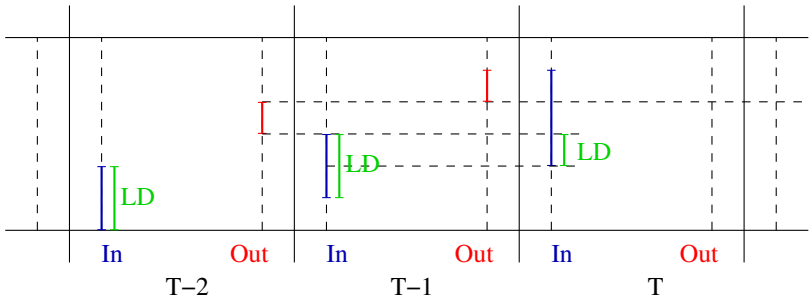


Figure 5.5: Valid load formalization for a two dimensional rectangular tiling example.

For a two dimensional rectangular tiling the formalization can be expressed as shown in Figure 5.5. The notation $\text{Load}(t < T)$ stands for $\cup_{t<T}\text{Load}(t)$ (same for the other expressions). Condition (i) means that all the data needed by the tile T, i.e., those input to the tile T but not produced by a previous tile, are loaded just before this tile or earlier. Condition (ii) means that there is no overwriting of a data already alive and modified in the buffer. This arises when a data is written in a previous tile before being read in the current tile. Without this condition, some data would possibly be loaded from the DDR and overwrite the existing value, which would be incorrect.

Definition 2 (Valid Store**)** *The function* $t \mapsto \text{Store}(t)$ *is valid iff the following conditions hold:*

(i) $\text{Out}(t \leq T_{\max}) = \text{Store}(t \leq T_{\max})$ *for* $T_{\max}$ *the last tile of the tile band.*

(ii) $\text{Store}(T) \cap \text{Out}(t > T) = \emptyset$ *for any tile* T*.*

For a two dimensional rectangular tiling the formalization can be expressed as shown in Figure 5.6. Conditions (i) and (ii) simply mean that we expect to store exactly the data locally modified. Condition (ii) means that a data is stored after its last write. This is actually stronger than what is really needed for defining a validity condition, as a value could be stored several times. But this assumption will simplify the proofs, without hurting the correctness of the whole construction. Similarly, we could also define more complex schemes, allowing for example to load from the DDR a value modified in the tile band. But this would imply to be able to guarantee that this value was already stored in the DDR and not modified again before the load. This would also imply a combined definition of the functions Load and Store.

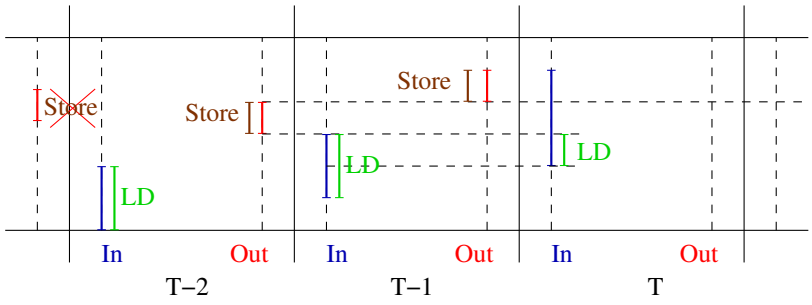


Figure 5.6: Valid store formalization for a two dimensional rectangular tiling example.

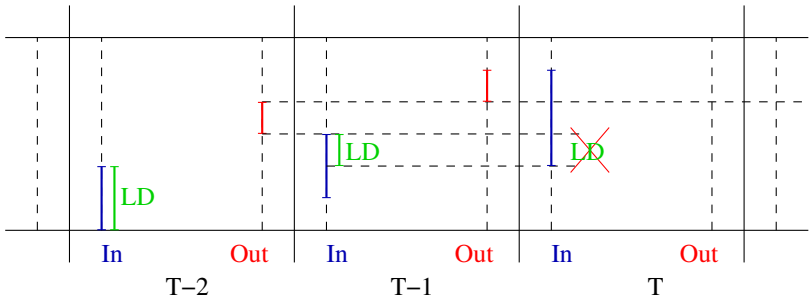


Figure 5.7: Exact load formalization for a two dimensional rectangular tiling example.

Definition 3 (Exact Load**)** Load(t) *is exact iff the following conditions hold:*

(i) $\cup_{t \leq T} \{\text{In}(t) \setminus \text{Out}(t' < t)\} = \text{Load}(t \leq T)$ *for any tile* T*.*

(ii) $\text{Load}(T) \cap \text{Load}(T') = \emptyset$ *for any tiles* $T \neq T'$*.*

For a two dimensional rectangular tiling the formalization can be expressed as shown in Figure 5.7. In Definition 1, Condition (i) was a simple inclusion in the validity definition, which allows to define valid solutions that load more data than needed. Now, the equality means that we load exactly the data needed and only them. The difference with Out$(t' < t)$ avoids to load the data already modified before executing the tile T. Condition (ii) means that all Load(T) must be disjoint, thus forbids redundant loads, i.e., data loaded several times. Note however that this may increase the size of the local memory. But, again, this assumption simplifies our general scheme.

Definition 4 (Exact Store**)** Store(t) *is exact iff the following condition hold.*

(i) The function $t \mapsto \text{Store}(t)$ *is valid.*

(ii) $\text{Store}(T) \cap \text{Store}(T') = \emptyset$ *for any tiles* $T \neq T'$*.*

For a two dimensional rectangular tiling the formalization can be expressed as shown in Figure 5.8. Similarly, the equality in Condition (i) means that we expect to store exactly the data modified: here, it cannot be an over-approximation otherwise the execution of the tile band would store an undefined value to the DDR, possibly leading to an incorrect code if one of the extra stores overwrites a meaningful value. Condition (ii) means that all Store(T) are disjoint, thus forbids redundant stores, i.e., a value defined by the tile band is stored only once.

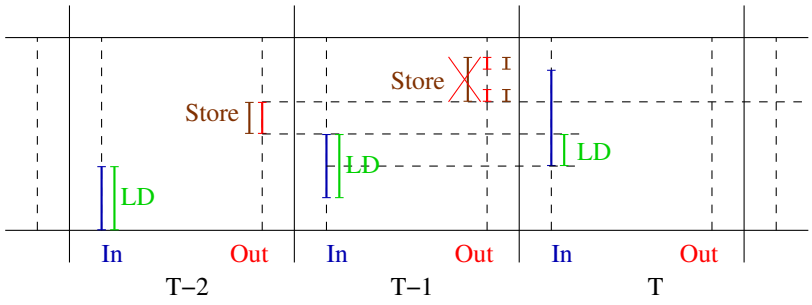


Figure 5.8: Exact store formalization for a two dimensional rectangular tiling example.

An exact solution

The previous definitions do not explicit the Load and Store operators for a given tile T. The following theorem expresses a solution, which corresponds to the case where loads are performed as late as possible and stores as soon as possible. Note that, unlike for the Store operator, an exact Load operator is completely determined by $\mathrm{Load}(T_{\min})$, the data loaded for the very first tile $T_{\min}$ of the tile band, as $\mathrm{Load}(T) = \mathrm{Load}(t \leq T) \setminus \mathrm{Load}(t < T)$, and these two terms are fully defined by the functions In and Out.

Theorem 1 *Let us define the following load and store operators.*

$$\begin{aligned} \mathrm{Load}(T) &= \mathrm{In}(T) \setminus \{\mathrm{In}(t < T) \cup \mathrm{Out}(t < T)\}. \\ \mathrm{Store}(T) &= \mathrm{Out}(T) \setminus \mathrm{Out}(t > T). \end{aligned}$$

Then, the functions $T \mapsto \mathrm{Load}(T)$ and $T \mapsto \mathrm{Store}(T)$ are valid and exact.

Intuitively, $\mathrm{Load}(T)$ gets all the data read in the tile T and removes data already read ($\mathrm{In}(t < T)$) and data already alive ($\mathrm{Out}(t < T)$). The latter contains the data read earlier, and data written earlier without a previous read, that we actually want to remove. As for $\mathrm{Store}(T)$, it is exactly the data written for the last time in T. We select the data written in the current tile ($\mathrm{Out}(T)$), which are not written later ($\mathrm{Out}(t > T)$).

Proof. First, let us show that Load is valid and exact. By definition, $\mathrm{Load}(t \leq T)$, which is equal to $\cup_{t \leq T} \{(\mathrm{In}(t) \setminus \mathrm{In}(t' < t)) \setminus \mathrm{Out}(t' < t)\}$ is a subset of $\cup_{t \leq T} \{\mathrm{In}(t) \setminus \mathrm{Out}(t' < t)\}$. Conversely, let x be in this latter union and let t_0 be the smallest tile index such that $x \in \mathrm{In}(t_0) \setminus \mathrm{Out}(t < t_0)$. By construction, for all $t < t_0$, $x \notin \mathrm{Out}(t)$. This implies $x \notin \mathrm{In}(t)$ for all $t < t_0$, otherwise this would contradict the minimality of t_0. Thus x belongs to $\mathrm{Load}(t_0)$ and, finally, to $\mathrm{Load}(t \leq T)$. This proves $\mathrm{Load}(t \leq T) = \cup_{t \leq T} \{\mathrm{In}(t) \setminus \mathrm{Out}(t' < t)\}$, which is Condition (i) of Definition 3 (exact Load). As for Condition (ii) of Definition 3, it holds from the fact that, for $T' < T$, $\mathrm{Load}(T) \cap \mathrm{Load}(T')$ is a subset of $(\mathrm{In}(T) \setminus \mathrm{In}(t < T)) \cap \mathrm{Load}(T')$, thus a subset of $(\mathrm{In}(T) \setminus \mathrm{In}(T')) \cap \mathrm{Load}(T')$ and finally of $(\mathrm{In}(T) \setminus \mathrm{Load}(T')) \cap \mathrm{Load}(T')$, which is empty.

Note also that $\mathrm{Out}(t < T) \cap \mathrm{Load}(T) = \mathrm{Out}(t < T) \cap ((\mathrm{In}(T) \setminus \mathrm{In}(t < T)) \setminus \mathrm{Out}(t < T)) = \emptyset$, thus Condition (ii) of Definition 1 (valid Load) is satisfied. As this condition is always satisfied for the first tile $T_{\min}$ and as the load operator is unique, given $\mathrm{Load}(T_{\min})$, this proves that an exact load operator is always valid. There is no need to add Condition (ii) of Definition 1 in Definition 3.

Now, let us prove that Store is valid and exact. It is clear that $\mathrm{Store}(t \leq T_{\max}) = \mathrm{Out}(t \leq T_{\max})$ and that $\mathrm{Store}(T) \cap \mathrm{Store}(T') = \emptyset$ given $T' < T$, which proves Condition (ii) of Definition 4 (exact Store) and Condition (i) of Definition 2 (valid Store). It remains to verify Condition (ii) of Definition 2. This is also clear, as, by definition, $\mathrm{Out}(t > T)$ is subtracted from $\mathrm{Store}(T)$. ■

Provided the sets $\mathrm{In}(t)$ and $\mathrm{Out}(t)$, this gives a method to compute the exact sets of data to load from and to store to the DDR for every tile. This nicely works whenever the sets $\mathrm{In}(t)$ and $\mathrm{Out}(t)$ can be computed. This is not always possible and we usually have two options.

(i) Identify a subset of programs making possible an exact computation. This is the option we chose in our current implementation. The detail of the algorithm and the implementation considerations will be given later.

(ii) Deal with approximation. In this case, we need to express the validity conditions relating the operators Load and Store to the approximated In and Out, and then to exhibit such operators. We now discuss this second option.

A conservative approximation

In general, it is not always possible to compute the sets In and Out. We now give a sufficient condition for the validity of Load and Store dealing with the approximation. We also exhibit a definition of the operators Load and Store, which we prove to be valid provided that some additional conditions are met. The key simplifying idea is to keep a scheme in which any dataflow dependence within the tile band corresponds, after load and store insertions, to a dataflow dependence in the local memory, i.e., no data is stored to the DDR and then read from it in the same tile band.

In the following, we assume the set $\mathrm{In}(t)$ to be over-approximated by the set $\overline{\mathrm{In}}(t)$, i.e., $\mathrm{In}(t) \subseteq \overline{\mathrm{In}}(t)$ for each valid tile t, and the set $\mathrm{Out}(t)$ to be under- and over-approximated by the sets $\underline{\mathrm{Out}}(t)$ and $\overline{\mathrm{Out}}(t)$: $\underline{\mathrm{Out}}(t) \subseteq \mathrm{Out}(t) \subseteq \overline{\mathrm{Out}}(t)$, for each valid tile t. We want to adapt the validity conditions given by Definitions 1 and 2. The two following theorems give a *sufficient* (but not necessary) condition on the sets $\overline{\mathrm{In}}$, $\underline{\mathrm{Out}}$, $\overline{\mathrm{Out}}$ for Load and Store to be valid.

Theorem 2 (Valid approximated Load**)** *The function* $t \mapsto \mathrm{Load}(t)$ *is valid if it verifies the following conditions, for any tile* T*:*

(i) $\cup_{t \leq T} \left\{ \overline{\mathrm{In}}(t) \setminus \underline{\mathrm{Out}}(t' < t) \right\} \subseteq \mathrm{Load}(t \leq T)$.

(ii) $\overline{\mathrm{Out}}(t < T) \cap \mathrm{Load}(T) = \emptyset$.

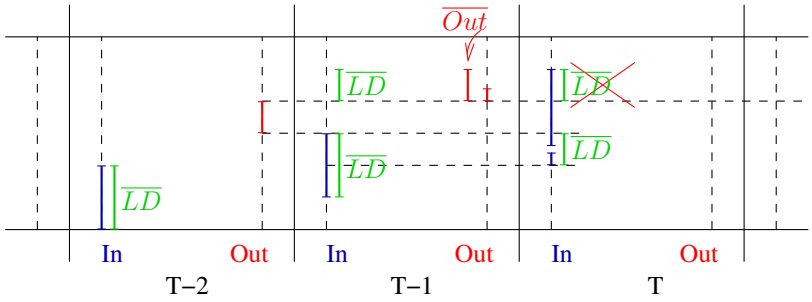


Figure 5.9: Valid approximated load formalization for a two dimensional rectangular tiling example.

For a two dimensional rectangular tiling the formalization can be expressed as shown in Figure 5.9.

Proof. It is sufficient to prove that these conditions imply the correctness conditions given by Definition 1. Let us first check that Condition (i) of Definition 1 holds. This is clear because $\mathrm{In}(T) \setminus \mathrm{Out}(t' < t)$ is included in $\overline{\mathrm{In}}(t) \setminus \underline{\mathrm{Out}}(t' < t)$, thus in $\mathrm{Load}(t \leq T)$. Similarly, since, for each tile t, $\mathrm{Out}(t) \subseteq \overline{\mathrm{Out}}(t)$, Condition (ii) of Definition 1 is verified. ■

As for Definition 1, Condition (i) means that Load contains all the data read by the current tile, without those already written in previous tiles. The term $\overline{\mathrm{In}}(t)$ can cause useless data to be read, but the term $\underline{\mathrm{Out}}(t' < t)$ cannot cause already-written data to be loaded, because of Condition (ii). This condition filters and removes from $\mathrm{Load}(t)$ the data previously written, and possibly more, depending on the approximation, but the loads are guaranteed to contain the data read, thanks to the term $\overline{\mathrm{In}}(t)$ in Condition (i). *In fine*, the approximation can cause loads of useless data, as well as redundant loads along the tile band, but any data modified in the tile band and read after is read from the local memory. Of course, the exact conditions of Definition 3 are no longer guaranteed.

The conditions for Store are more tricky as, at first glance, Store does not accept any approximation. Indeed, if Store is over-approximated, extra data may be stored to the DDR, causing

useful data in the DDR to be crushed. Conversely, if Store is under-approximated, we may forget to store useful outputs defined by the tile band. Both approximations may cause an incorrect execution. Actually, an over-approximation of the Store operator can be correct if the extra data to be stored are exactly those already present in the DDR. This would not crush any data and would keep the program semantics. This condition holds when $\text{Store}(T) \subseteq \text{Load}(t \leq T) \cup \text{Out}(t \leq T)$ for each tile T, i.e., any data stored from the local memory to the DDR and not defined by the computations of the previous tiles (T included) has been previously loaded from the DDR. Then, a sufficient condition for the Store operator to be an over-approximation can be given as follows.

Theorem 3 (Valid approximated Store**)** *The function* $t \mapsto \text{Store}(t)$ *is valid if:*

(i) $\overline{\text{Out}}(t \leq T_{\max}) \subseteq \text{Store}(t \leq T_{\max})$ *for the last tile* $T_{\max}$ *of the tile band.*

(ii) $\text{Store}(T) \cap \overline{\text{Out}}(t > T) = \emptyset$ *for any tile* T*.*

(iii) $\text{Store}(T) \subseteq \overline{\text{In}}(t \leq T) \cup \underline{\text{Out}}(t \leq T)$ *for any tile* T*, when used with a valid approximated* Load*.*

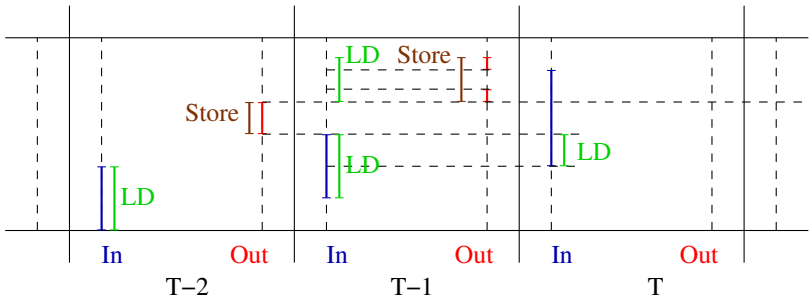


Figure 5.10: Valid approximated store formalization for a two dimensional rectangular tiling example.

For a two dimensional rectangular tiling the formalization can be expressed as shown in Figure 5.10.

Proof. The two first conditions show that $\text{Store}(T)$ is an over-approximation that verifies Conditions (i) and (ii) of Definition 2 (valid Store), because of the approximation $\text{Out}(t) \subseteq \overline{\text{Out}}(t)$ for each tile t. Condition (iii) ensures the correctness of Store, as the previous discussion explained, if it implies $\text{Store}(T) \subseteq \text{Load}(t \leq T) \cup \text{Out}(t \leq T)$ for each tile T. If the function $t \mapsto \text{Load}(t)$ is a valid approximated Load, then $\cup_{t \leq T} \left\{\overline{\text{In}}(t) \setminus \underline{\text{Out}}(t' < t)\right\} \subseteq \text{Load}(t \leq T)$ (Condition (i) of Theorem 2). Furthermore, $\cup_{t \leq T} \left\{\overline{\text{In}}(t) \setminus \underline{\text{Out}}(t' < t)\right\} \cup \text{Out}(t \leq T) = \cup_{t \leq T} \overline{\text{In}}(t) \cup \text{Out}(t \leq T) = \overline{\text{In}}(t \leq T) \cup \text{Out}(t \leq T)$. Thus, if $\text{Store}(T) \subseteq \overline{\text{In}}(t \leq T) \cup \underline{\text{Out}}(t \leq T)$, then $\text{Store}(T) \subseteq \overline{\text{In}}(t \leq T) \cup \text{Out}(t \leq T)$, and finally $\text{Store}(T) \subseteq \text{Load}(t \leq T) \cup \text{Out}(t \leq T)$. The Store operator is valid too. ■

We now exhibit Load and Store operators verifying these conditions. The worst-case solution is to load, before the first tile, all data potentially read in the tile band and to store, after the last tile, all data potentially written in the tile band, in other words: $\text{Load}(T_{\min}) = \overline{\text{In}}(t \leq T_{\max})$ and $\text{Load}(t) = \emptyset$ if $t \neq T_{\min}$, $\text{Store}(T_{\max}) = \overline{\text{Out}}(t \leq T_{\max})$ and $\text{Store}(t) = \emptyset$ if $t \neq T_{\max}$. According to Theorems 2 and 3, this scheme is valid if $\overline{\text{Out}}(t \leq T_{\max}) \subseteq \overline{\text{In}}(t \leq T_{\max}) \cup \underline{\text{Out}}(t \leq T_{\max})$. To make it always valid, we should thus also pre-load all data that cannot be proved to be defined in the tile band, i.e., $\text{Load}(T_{\min}) = \overline{\text{In}}(t \leq T_{\max}) \cup \left\{\overline{\text{Out}}(t \leq T_{\max}) \setminus \underline{\text{Out}}(t \leq T_{\max})\right\}$.

More generally, consider the operators Load and Store defined as follows.

$$\begin{aligned}\mathrm{Load}(T) &= \overline{\mathrm{In}}(T) \setminus \left\{\overline{\mathrm{In}}(t<T) \cup \overline{\mathrm{Out}}(t<T)\right\}\\ \mathrm{Store}(T) &= \overline{\mathrm{Out}}(T) \setminus \overline{\mathrm{Out}}(t>T)\end{aligned}$$

First note that, if the sets $\mathrm{Out}(T)$ are not approximated, i.e., if $\underline{\mathrm{Out}}(T) = \mathrm{Out}(T) = \overline{\mathrm{Out}}(T)$ for any tile T, then there is no problem: the operators Load and Store are both valid (and even exact with respect to the sets $\overline{\mathrm{In}}(T)$) with the same proof as in Theorem 1. The difficulty arises only when the sets $\mathrm{Out}(T)$ are not exact, as we now show.

It is easy to see that $\mathrm{Store}(t \leq T_{max}) = \overline{\mathrm{Out}}(t \leq T_{max})$ and that $\mathrm{Store}(T) \cap \overline{\mathrm{Out}}(t>T) = \emptyset$ for any tile T, hence Conditions (i) and (ii) of Theorem 3 are satisfied. The function $t \mapsto \mathrm{Store}(t)$ is valid if Condition (iii) is fulfilled, i.e., if $\mathrm{Store}(T) \subseteq \overline{\mathrm{In}}(t \leq T) \cup \underline{\mathrm{Out}}(t \leq T)$ holds for any tile T, which is equivalent to $\overline{\mathrm{Out}}(T) \subseteq \overline{\mathrm{In}}(t \leq T) \cup \underline{\mathrm{Out}}(t \leq T) \cup \overline{\mathrm{Out}}(t>T)$. This condition can always be fulfilled by increasing the over-approximation $\overline{\mathrm{In}}$ of In, i.e., with extra pre-loads. Unfortunately, this is not as simple with the Load operator. Indeed, the over-approximation on $\mathrm{Out}(t<T)$ can prevent effective input data to be loaded. As is, this definition does not directly comply with the validity conditions of Theorem 2. We provide additional constraints on $\overline{\mathrm{In}}$, $\overline{\mathrm{Out}}$, and $\underline{\mathrm{Out}}$, which, if respected, are sufficient to ensure the validity of the operators Load and Store. What is needed is again to pre-load values that are needed but that cannot be proved to be defined in the tile band.

Theorem 4 *The* Load *and* Store *operators defined by* $\mathrm{Load}(T) = \overline{\mathrm{In}}(T) \setminus \left\{\overline{\mathrm{In}}(t<T) \cup \overline{\mathrm{Out}}(t<T)\right\}$ *and* $\mathrm{Store}(T) = \overline{\mathrm{Out}}(T) \setminus \overline{\mathrm{Out}}(t>T)$ *are valid if the following constraints hold for any tile* T*:*

(i) $\overline{\mathrm{Out}}(T) \subseteq \overline{\mathrm{In}}(t \leq T) \cup \overline{\mathrm{Out}}(t>T) \cup \underline{\mathrm{Out}}(t \leq T)$.

(ii) $\overline{\mathrm{In}}(T) \cap \left\{\overline{\mathrm{Out}}(t<T) \setminus \underline{\mathrm{Out}}(t<T)\right\} \subseteq \overline{\mathrm{In}}(t<T)$.

Proof. We already proved the validity of Store provided Constraint (i), which can be easily interpreted: it means that if a data appears to be defined in a tile T, then either it can be proved to be defined in T or earlier (set $\underline{\mathrm{Out}}(t \leq T)$), or it will appear to be defined again later (and will be stored later, set $\overline{\mathrm{Out}}(t>T)$), or it has been accessed in T or earlier (thus either loaded or defined earlier, if Load is valid). So, let us prove now the validity of Load. Condition (ii) of Theorem 2 is satisfied as $\overline{\mathrm{Out}}(t<T)$ is subtracted. It remains to consider Condition (i) of Theorem 2.

The proof is similar to the proof of Theorem 1. First, it is immediate to show that the set $\mathrm{Load}(t \leq T)$ is a subset of $\cup_{t \leq T} \left\{\overline{\mathrm{In}}(t) \setminus \underline{\mathrm{Out}}(t'<t)\right\}$ since $\underline{\mathrm{Out}}(t'<t) \subseteq \overline{\mathrm{Out}}(t'<t)$. Now, let x be in this latter union and let t_0 be the smallest tile index such that $x \in \overline{\mathrm{In}}(t_0) \setminus \underline{\mathrm{Out}}(t<t_0)$. By construction, for all $t<t_0$, $x \notin \underline{\mathrm{Out}}(t)$. This implies $x \notin \overline{\mathrm{In}}(t)$ for all $t<t_0$, otherwise this would contradict the minimality of t_0. Thus x belongs to $\left\{\overline{\mathrm{In}}(t_0) \setminus \underline{\mathrm{Out}}(t<t_0)\right\} \setminus \overline{\mathrm{In}}(t<t_0)$. Now, if $x \notin \mathrm{Load}(t_0)$ then, by definition of $\mathrm{Load}(t_0)$, x must belong to $\overline{\mathrm{Out}}(t<t_0) \setminus \underline{\mathrm{Out}}(t<t_0)$. We can now use the last condition of the theorem, which states that $x \in \overline{\mathrm{In}}(t<t_0)$, a contradiction. Thus $x \in \mathrm{Load}(t_0)$, which proves that Condition (i) of Theorem 2 is satisfied with an equality. ■

This construction could be used to design a conservative analysis that would cope with every kind of programs, providing that Constraints (i) and (ii) of Theorem 4 are met. How to compute the sets $\overline{\mathrm{In}}$, $\underline{\mathrm{Out}}$, and $\overline{\mathrm{Out}}$ verifying these constraints is beyond the scope of this thesis, and is left for future work. But, still, one can already noticed that a direct approach consisting in over-approximating the sets $\overline{\mathrm{In}}(T)$ into sets $\overline{\overline{\mathrm{In}}}(T)$ will work. Indeed, the constraints on sets $\overline{\mathrm{In}}(T)$ always give a "lower bound", but no "upper bound". For example, a simple solution to solve Constraint (i) is to add $\overline{\mathrm{Out}}(T) \setminus \underline{\mathrm{Out}}(T)$ to $\overline{\mathrm{In}}(T)$. Then, Constraint (ii), starting from these new sets $\overline{\mathrm{In}}(T)$,

define new lower bounds, by induction, for decreasing values of T. The induction can be avoided and approximated by defining directly $\overline{\overline{\text{In}}}(T) = \cup_{t>T} \left\{\overline{\text{In}}(t) \cap (\overline{\text{Out}}(t' < t) \setminus \underline{\text{Out}}(t' < t))\right\} \cup \overline{\text{In}}(T)$.

Note also that any over-approximation of Load(T) is valid as long as Condition (ii) of Theorem 2 is satisfied. All these theoretical results give opportunities for handling cases where program analysis cannot be performed exactly or when approximating Load and Store sets allows a better packing of data to be transferred. Again, this optimization is left for future work. We believe also that abstract interpretation techniques [46] could be used to compute the sets $\overline{\text{In}}$, $\underline{\text{Out}}$, and $\overline{\text{Out}}$, as done in [48]. Then, a post-processing is needed to meet Constraints (i) and (ii), as explained above with the sets $\overline{\overline{\text{In}}}(T)$.

5.2.2 An exact solution for computing the Load and Store sets.

We now give an algorithm that implements the exact Load and Store operators specified in the previous section. We want to compute the functions Load and Store, or equivalently, the expressions of Load(T) and Store(T) for a *parametric* (symbolic) tile T. These expressions for Load and Store will be used later to generate the C functions for the Load and Store accelerators.

Our algorithm to specify the function Load does not directly rely on the sets In and Out, as defined in the previous section, but it is based on the computation of FirstRead, the set of operations responsible for a *first read* within the *parameterized tile band* being considered, i.e., a read from a memory (array) location that has never been read before in the tile band. We will see later (in Theorem 5) the link with the exact Load as defined in Theorem 1. If values read in the tile band are not written earlier in the tile, we can define Load(T) to be the first reads that belong to the tile T, i.e., FirstRead $\cap\, T$. Actually, FirstRead $\cap\, T$ is a set of operations, not a set of data as Load(T), so we cannot rigorously write Load(T) = FirstRead $\cap\, T$. Rather, we should write Load(T) = Input(FirstRead $\cap\, T$) where Input gives the data read by a set of operations. Here, of course, an operation is defined at the granularity of a memory access, i.e., we only pick the input that is of interest. FirstRead $\cap\, T$ actually contains more information than Load(T), and will be helpful during the code generation phase.

Store(T) is obtained the same way, from LastWrite, the set of operations responsible for a *last write* within the tile band. The intersection of LastWrite with a tile T, LastWrite$\cap T$, gives the last writes on that tile and can be used to define Store(T). We define Store(T) = Output(LastWrite$\cap T$), where Output gives the data written by a set of operations.

Matrix multiply example (continued). Let us consider the matrix multiply example again. We write (I, J, K, i, j, k) the iteration vector of the final tiled loop nest. (I, J, K) iterates over the tiles, while (i, j, k) iterates into the tile specified by (I, J, K). As we tile along the canonical directions, the tile executed at the iteration (I, J, K) of the tile loops is defined by the set of iteration vectors such that:

$$0 \leq i, j, k \leq N \wedge 32I \leq i \leq 32I + 31 \wedge 32J \leq j \leq 32J + 31 \wedge 32K \leq k \leq 32K + 31$$

Here, we used tiles equal to squares of size 32. The tile size can be specified as an input by the user, but (so far) cannot be parameterized, as this would lead to non-affine constraints. Similarly, the tile band defined by the tile iteration (I, J) is the set of iteration vectors (I, J, K, i, j, k) verifying:

$$0 \leq i, j, k \leq N \wedge 32I \leq i \leq 32I + 31 \wedge 32J \leq j \leq 32J + 31$$

We actually just removed the last constraints, which will be added later to isolate a tile of the tile band. We assume I and J to be *parameters*. Then, the solution will be given in terms of I and J, *i.e.*, will be *parameterized by the tile band*. This feature is essential.

For (i, j, k) in the tile band (I, J) given above, we get the following expressions for FirstRead and LastWrite. They are coherent with the sets FirstRead and LastWrite given in Figure 5.3.

Array	FirstRead()	LastWrite()
$a(i, j)$	$(S, i, 32J, j)$	$\emptyset$
$b(i, j)$	$(S, 32I, j, i)$	$\emptyset$
$c(i, j)$	$(S, i, j, 0)$	(S, i, j, N)

In a more general situation, defining the operator Load this way is not correct. Indeed, as we explained in the previous section, the first reads of $a(\vec{i})$ that are preceded by a write of $a(\vec{i})$ should not be loaded, otherwise, this will cause a load to overwrite the value of $a(\vec{i})$. To avoid this situation, one can define the *first read* of $a(\vec{i})$ to be the first executed operation among all the reads of $a(\vec{i})$ and the writes of $a(\vec{i})$. Then, to define Load(T), it suffices to restrict to the reads, the writes corresponding to the forbidden case. This way, the operators Load and Store are exact in the meaning of Definitions 3 and 4 as shown by the following theorem.

Theorem 5 *The operators* Load *and* Store *are valid and exact if they are defined as follows.*

$$\begin{aligned} \text{Load}(T) &= \text{Input}(\text{FirstRead} \cap T) \\ \text{Store}(T) &= \text{Output}(\text{LastWrite} \cap T) \end{aligned}$$

Proof. By construction, Load(T) contains all the first reads (with the definition that takes into account previous writes) in T, thus all data that are read in T, not read earlier, and not defined earlier. Thus, Load(T) is exactly defined as in Theorem 1 by $\{\text{In}(T) \setminus \text{In}(t < T)\} \setminus \text{Out}(t < T)$. Similarly, Store$(T)$ contains all the last writes in T, which means all the data written in T and not written later again. In other words, Load$(T) = \text{Out}(T) \setminus \text{Out}(t > T)$. Theorem 1 then shows that the sets Load and Store are valid and exact. ■

We now describe how to compute the sets FirstRead and LastWrite. For the sake of clarity, the next subsection explains how to compute FirstRead among reads only, then we explain how to modify the algorithm to get the first read among reads and writes, and how to filter the reads.

Computing the first reads

For each array c, FirstRead(c) is obtained by extracting the set of operations reading a given $c(\vec{i_0})$, where $\vec{i_0}$ is a *parameter*. Then, we compute the read whose schedule is minimum. As the kernel fits into the polyhedral model, this basically boils down to compute the lexicographic minimum for a union of polytopes. The final result, FirstRead(c), is given as a discussion on the parameters value, including the array cell $\vec{i_0}$ being considered.

As the kernel is assumed to fit into the polyhedral model, all assignments reading c can be written:

$$S_\ell : \vec{i} \in D_\ell : \ldots = \ldots c[u_\ell(\vec{i})] \ldots$$

where D_ℓ is the iteration domain of the statement S_ℓ, $\vec{i}$ is an iteration vector, and u_ℓ is an *affine function*. The reads of $c(\vec{i_0})$ in S_ℓ is the set of operations $(S_\ell, \vec{i})$ that read $c(\vec{i_0})$, i.e., $u_\ell(\vec{i}) = \vec{i_0}$, and that are actually executed, i.e., $(\vec{i} \in D_\ell)$. In other words:

$$\text{Read}(c, S_\ell) = \{\vec{i} \in D_\ell \mid u_\ell(\vec{i}) = \vec{i_0}\}$$

For convenience, we just put iteration vectors $\vec{i}$ in $\mathrm{Read}(c, S_\ell)$ instead of operations $(S_\ell, \vec{i})$. If only one read of c occurs in S_ℓ, as u_ℓ is affine, $\mathrm{Read}(c, S_\ell)$ is a polytope (actually the integer points in a polytope). Otherwise, in general, the result of Read is (the integer points in) a *union of polytopes*.

Every statement S is given an affine schedule θ_S, assigning an execution date to every iteration vector. This schedule is obtained from the tiling specified by the user, as discussed in Section 5.1.1. We extend the definition of Read by providing the execution date of $\vec{i}$, $\vec{t} = \theta_{S_\ell}(\vec{i})$ together with $\vec{i}$:

$$\mathrm{Read}(c, S_\ell) = \{(\vec{t}, \vec{i}) \mid \vec{t} = \theta_{S_\ell}(\vec{i}) \wedge u_\ell(\vec{i}) = \vec{i_0} \wedge \vec{i} \in D_\ell\}$$

The first instance of S_ℓ reading the cell $c(\vec{i_0})$ is defined by the iteration vector of $\mathrm{Read}(c, S_\ell)$ whose schedule is minimum. In other words, we need to get the couple $(\vec{t}, \vec{i})$ of $\mathrm{Read}(c, S_\ell)$ which minimizes the execution date $\vec{t}$. This is simply the lexicographic minimum of $\mathrm{Read}(c, S_\ell)$: $(\vec{t}_{\mathrm{FirstRead}}, \vec{i}_{\mathrm{FirstRead}}) = \min_{\ll} \mathrm{Read}(c, S_\ell)$. Then, the first instance of S_ℓ reading $c(\vec{i_0})$ is $(S_\ell, \vec{i}_{\mathrm{FirstRead}})$. As a bonus, we get the execution date of the first read $\vec{t}_{\mathrm{FirstRead}}$.

To compute $\mathrm{FirstRead}(c)$, it is sufficient to proceed in the same way for every assignment reading c, getting as many local first instances, and to compute the global minimum, which is still a lexicographic minimum. In other words, if $S_1, \ldots, S_n$ denote the assignments reading c, the global $\mathrm{FirstRead}(c)$ is:

$$\mathrm{FirstRead}(c) = \min_{\ll}(\mathrm{Read}(c, S_1) \cup \ldots \cup \mathrm{Read}(c, S_n))$$

When $\mathrm{Read}(c, S_1) \cup \ldots \cup \mathrm{Read}(c, S_n)$ is a non-convex union of polytopes, the minimum cannot be computed directly with integer linear programming. It is better to use the equivalent form:

$$\mathrm{FirstRead}(c) = \min_{\ll}(\min_{\ll} \mathrm{Read}(c, S_1), \ldots, \min_{\ll} \mathrm{Read}(c, S_n))$$

Thus, to get $\mathrm{FirstRead}(c)$, we compute the local first reads $\min_{\ll} \mathrm{Read}(c, S_i)$, as described above. Then, we get the lexicographic minimum of these first reads. The inner lexicographic minima apply on polytopes that depend on parameters such as $\vec{i_0}$ (the array cell whose first read is searched). As $\vec{i_0}$ is unknown, the result is a discussion on $\vec{i_0}$, giving for such or such domain the corresponding pair $(\vec{t}_{\min}, \vec{i}_{\min})$. We actually get a set of clauses:

$$\left[\vec{n} \in D'_1 : (\vec{t}_{\min_1}, \vec{i}_{\min_1})\right] \vee \ldots \vee \left[\vec{n} \in D'_p : (\vec{t}_{\min_p}, \vec{i}_{\min_p})\right]$$

where $\vec{n}$ is the vector of parameters (including $\vec{i_0}$). The $\vec{t}_{\min_k}$ and $\vec{i}_{\min_k}$ are *affine expressions of the parameters*. Standard integer linear programming techniques cannot handle such problems. We use the technique of parametric integer programming [66], which give the result as a selection tree on $\vec{n}$, the QUAST (quasi-affine selection tree).

The outer lexicographic minimum is actually the lexicographic minimum for a set of clauses. Standard combination techniques [67] are used there. We just need to tag each set of clauses with the corresponding assignment (S_ℓ), so it will be remembered during the combination. Finally, the result is a set of tagged clauses:

$$\mathrm{FirstRead}(c) = \left[\vec{n} \in D'_1 : (S_{\ell_1}, \vec{i}_{\min_1})\right] \vee \ldots \vee \left[\vec{n} \in D'_p : (S_{\ell_p}, \vec{i}_{\min_p})\right]$$

Again, $\vec{n}$ is the vector of parameters, which, in particular, includes $\vec{i_0}$, the array cell whose first read is searched, and I, J, the parameters defining the current tile band in which the first read is searched. Also, our method works with any affine schedule, which makes it very general.

This explains how to compute FirstRead(t) among reads only. Now, a simple modification allows to compute FirstRead(t) among reads and writes. Each time S_ℓ writes the array cell $c[v_\ell(\vec{i})]$, it suffices to complete the set $\text{Read}(c, S_\ell)$ with the write to c by means of the constraint $v_\ell(\vec{i}) = \vec{i_0}$:

$$\text{Read}(c, S_\ell) = \{(\vec{t}, \vec{i}) \mid \vec{t} = \theta_{S_\ell}(\vec{i}) \wedge u_\ell(\vec{i}) = \vec{i_0} \wedge \vec{i} \in D_\ell\} \cup \{(\vec{t}, \vec{i}) \mid \vec{t} = \theta_{S_\ell}(\vec{i}) \wedge v_\ell(\vec{i}) = \vec{i_0} \wedge \vec{i} \in D_\ell\}$$

This is a union of polyhedra, and the lexicographic minima $\min_{\ll} \text{Read}(c, S_\ell)$ must be computed separately. Here, we will get two sets of clauses $R_{\min}(k)$ and $W_{\min}(k)$. We tag $W_{\min}(k)$, then we compute the global minimum with the method described above. Then, it suffices to remove from the final system of clauses, the tagged clauses coming from a $W_{\min}(k)$.

Computing the last writes

The method for computing the last writes is exactly the symmetric of the method to compute the first reads. We use the same notations as in the previous section. We thus let the reader read the previous section for more details. Let S_ℓ be an assignment writing the array c:

$$S_\ell : \vec{i} \in D_\ell : c[v_\ell(\vec{i})] = ...$$

The writes of $c(\vec{i_0})$ in S_ℓ is the set of operations $(S_\ell, \vec{i})$ that write $c(\vec{i_0})$, i.e., $v_\ell(\vec{i}) = \vec{i_0}$, and that are actually executed, i.e., $(\vec{i} \in D_\ell)$. In other words:

$$\text{Write}(c, S_\ell) = \{\vec{i} \mid v_\ell(\vec{i}) = \vec{i_0} \wedge \vec{i} \in D_\ell\}$$

This is a polytope for the reasons discussed above. Similarly, we extend the definition of $\text{Write}(c, S_\ell)$ by adding the execution of the iteration vector $\vec{i}$:

$$\text{Write}(c, S_\ell) = \{(\vec{t}, \vec{i}) \mid \vec{t} = \theta_{S_\ell}(\vec{i}) \wedge v_\ell(\vec{i}) = \vec{i_0} \wedge \vec{i} \in D_\ell\}$$

Similarly, the last instance of S_ℓ writing the cell $c(\vec{i_0})$ is the *lexicographic maximum* $\max_{\ll} \text{Write}(c, S_\ell)$. If $S_1, \ldots, S_n$ denote the assignments writing c, the global $\text{LastWrite}(c)$ is:

$$\text{LastWrite}(c) = \max_{\ll}(\text{Write}(c, S_1) \cup \ldots \cup \text{Write}(c, S_n))$$

It is again better to use the equivalent form:

$$\text{LastWrite}(c) = \max_{\ll}(\max_{\ll} \text{Write}(c, S_1), \ldots, \max_{\ll} \text{Write}(c, S_n))$$

Thus, to get $\text{LastWrite}(c)$, we compute the local last writes $\min_{\ll} \text{Write}(c, S_i)$, then we compute the lexicographic maximum of these last writes. Finally, applying the same techniques as for $\text{FirstRead}(c)$ (and substituting $\min_{\ll}$ by $\max_{\ll}$), we end up with a set of tagged clauses:

$$\text{LastWrite}(c) = \left[\vec{n} \in D'_1 : (S_{\ell_1}, \vec{i}_{\min_1})\right] \vee \ldots \vee \left[\vec{n} \in D'_p : (S_{\ell_p}, \vec{i}_{\min_p})\right]$$

Again, $\vec{n}$ is the vector of parameters, which, in particular, includes $\vec{i_0}$, the array cell whose last write is searched, and I, J, the parameters defining the current tile band in which the last write is searched. Also, our method works with any affine schedule, which makes it very general.

5.3 Local memory management

With our method, the variables are loaded in the local memory just before needed and then stored back to the DDR at the expiration of their lifetime in the tiled band. Meanwhile, the computations are done in the local memory, using the temporary images of the variables. This section presents a method to decide *how* to store the variables in the local memory. Each variable must be mapped to the local memory in such a way that (i) two data live at the same time cannot be mapped to the same local address, and (ii) the local memory size must be as small as possible.

One could organize the local memory by mapping every used memory cell @ to a local address $\sigma(@)$. Rather, we propose a weaker method where each variable is mapped to its own local variable with a possibly reduced size. In other words, each scalar variable is mapped to a local scalar variable with the same size and each array variable a is mapped to a local array variable a_tmp. Each original array cell $a(\vec{i})$ is mapped to the local array cell $a\_tmp(\sigma(\vec{i}))$, with a mapping σ verifying the conditions (i) and (ii) mentioned above.

Matrix multiply example (continued). Let us see what happens with the array a in the matrix multiply example. Communication coalescing informs us that every tile (I, J, K) must load in a_tmp the following region (with Fortran notations) for a complete tile:

$$a\_tmp \longleftarrow \text{Load}_a(I, J, K) = a[bI : bI + b - 1][bK : bK + b - 1]$$

where b denotes the tile size (here $b = 32$). Meanwhile, the double buffering process needs to load in a_tmp the data for the next tile in the band, $(I, J, K + 1)$:

$$a\_tmp \longleftarrow \text{Load}_a(I, J, K + 1) = a[bI : bI + b - 1][bK + b : bK + 2b - 1]$$

This means that *at the same time*, a_tmp needs at most $b \times 2b$ array cells: b cells in the first dimension and $2b$ cells in the second dimension. Now, the issue is to map the array a to the local array a_tmp. This can be done thanks to the mapping:

$$\sigma : a[i][k] \mapsto a\_tmp[i \bmod b][k \bmod 2b]$$

The mapping σ can be found thanks to array contraction techniques. Here, it corresponds exactly to two blocks, used in a double-buffering manner. We now present the general principles of an array contraction technique developed by Darte et al. [55] for the theoretical part, by Alias et al. [23] for the program analysis part, and which produces such mappings σ.

5.3.1 Array contraction

In many cases, the array index functions are *uniform* (*i.e.*, they are translations with respect to the loop indices as in a[i][j-1]), and the program reads and writes consecutive array cells. The set of live array cells is a window sliding during a tiled program execution, allowing some memory optimizations [36]. This also makes possible to fold the array thanks to a so-called modular mapping $\sigma(\vec{i}) = A\vec{i} \bmod \vec{b}$, where A is an integer matrix and $\vec{b}$ is an integral vector defining a modulo operation component-wise. The framework presented in [55], called "lattice-based memory allocation", can find such a mapping, even for more general cases, given an analysis of live array cells.

Theoretical framework: critical lattices

For any input program, a conflict relation $\bowtie$ is defined, relating array cells whose lifetime conflict: $a(\vec{i}) \bowtie a(\vec{j})$ if an only if the lifetime intervals of $a(\vec{i})$ and $a(\vec{j})$ are not disjoint. The method assumes the conflict relation $\bowtie$ to be given, and proceeds into three steps:

Step a. Conflict polyhedron From the conflict relation $\bowtie$, we derive a conflict polyhedron $DS = \{\vec{i}-\vec{j} \mid a(\vec{i}) \bowtie a(\vec{j})\}$ (DS stands for difference set). The vertices of the conflict polyhedron can be viewed as vectors linking two extremal conflicting array cells. In a way, the conflict polyhedron represents the sliding window mentioned above. To find the mapping, we need to bound this sliding window. This is the purpose of the next step.
Matrix multiply example (continued). The region $[bI : bI+b-1][bK : bK+b-1]$ of array a is used while the region $[bI : bI+b-1][bK+b : bK+2b-1]$ is loaded. Then, the conflict polyhedron for array a is $DS = [-b+1; b-1] \times [-2b+1; 2b-1]$.

Step b. Critical lattice A mapping σ is *valid* if two conflicting array cells $a(\vec{i})$ and $a(\vec{j})$ cannot be mapped to the same location, i.e., $\sigma(\vec{i}) = \sigma(\vec{j}) \Rightarrow \vec{i} = \vec{j}$. As σ is linear, it is valid iff $\vec{d} \in DS$, $\sigma(\vec{d}) = 0$ implies $\vec{d} = 0$, i.e., $DS \cap \ker \sigma = \{0\}$ where $\ker \sigma = \{\vec{i} \mid \sigma(\vec{i}) = 0\}$. The set $\ker \sigma$ is an integer lattice. Furthermore, for any integer lattice L, a modular mapping σ can be built such that $\ker \sigma = L$. Thus, to find a *valid* mapping σ, it suffices to find an integer lattice L such that $DS \cap L = \{0\}$ (L is called an *admissible lattice* for DS). The mapping σ is then derived from L.

As mentioned above, DS can be viewed as a live window and L as a kind [2] of bounding box for DS. Intuitively, the smaller the bounding box is, the smaller the final size of the contracted array will be. More precisely, the determinant $\det L$ of the lattice L is exactly the size of the contracted array. Thus, we additionally need to minimize $\det L$. In number theory, a lattice L admissible for DS and with smallest determinant is called a *critical lattice* for DS.

The authors of [55] present an exhaustive search to find a critical lattice. The method is expensive if DS is large, and not always applicable in our context if DS is parameterized. Hopefully, fast heuristics are proposed to solve a relaxed version of the problem, in which $\det L$ tends to be reasonably small. Extensions to handle parameterized sets DS are possible.
Matrix multiply example (continued). The integer lattice L generated by the vectors $(b, 0)$ and $(0, 2b)$: $(b, 0)\mathbb{Z}+(0, 2b)\mathbb{Z}$ is clearly a critical lattice for DS. Indeed, it is admissible, with determinant $2b^2$ and, during program execution, $2b^2$ array cells are simultaneously live. We point out that this situation is not a general one. It may happen that the best modular mapping leads to a memory size larger than the maximal number of simultaneously-live array cells.

Step c. Mapping extraction Given a "good" lattice L, the mapping σ can be extracted thanks to the Smith normal form. The reader is referred to the [55] for more details.
Matrix multiply example (continued). From the lattice $L = (b, 0)\mathbb{Z} + (0, 2b)\mathbb{Z}$, we can derive the modular mapping σ defined by $\sigma(i, j) = (i \bmod b, j \bmod 2b)$. It is such that $L = \ker \sigma$.

The exhaustive search and the heuristics are implemented in a tool called Clak. Clak takes as input a polyhedron (the set DS) and produces an admissible lattice for DS, according to the construction method that is selected.

Practical framework: lifetime analysis for arrays

The theoretical framework formalizes the array contraction problem as finding a "good" integer lattice for a polyhedron, where the polyhedron summarizes the lifetime information of the array cells, and the integer lattice represents the array contraction mapping. This section presents a method due to Alias et al [23] to compute this polyhedron from a static control program.

The method focuses on programs fitting into the polyhedral model, the so-called static control programs, for which exact analysis can be computed. This lifetime analysis can be parameterized

2. A bounding box does define an admissible lattice, but the converse is not true.

by the program schedule, which is sequential by default. This feature allows to contract arrays for parallel programs as well as for programs with subtle *fine-grain* parallelism as software pipelining. This is fundamental to be able to handle programs in the double-buffering context.

To make the discussion simpler, any array cell that is read is assumed to be previously written. If it is not the case, this means that there is an implicit write at the start of the code region being analyzed, i.e., it is an input data. With this assumption, if an array element $a(\vec{i})$ is read by an operation R_i, it is *live* from the last (for the schedule θ) operation that previously wrote it, to R_i. Then, two array cells $a(\vec{i})$ and $a(\vec{j})$ conflict iff there exist a read R_i of $a(\vec{i})$ and a read R_j of $a(\vec{j})$ such that the time intervals $[\theta(\ell_i), \theta(R_i)]$ and $[\theta(\ell_j), \theta(R_j)]$ overlap, where ℓ_i is the last write to $a(\vec{i})$ before R_i and ℓ_j is the last write to $a(\vec{j})$ before R_j. In other words,

$$a(\vec{i}) \bowtie a(\vec{j}) \text{ if and only if } \exists R_i, R_j, \ell_i, \ell_j \text{ such that } \theta(\ell_i) \ll \theta(R_j) \wedge \theta(\ell_j) \ll \theta(R_i)$$

The computation of the last writes ℓ_i and ℓ_j before each read is possible, but would require an exact array dataflow analysis, as presented in [67] to be settled. This would involve an heavy and expensive machinery, which can be avoided with the following conservative approximation. Instead of the last writes ℓ_i and ℓ_j of $a(\vec{i})$ and $a(\vec{j})$, the technique of [23] considers any write W_i of $a(\vec{i})$ executed before R_i, and any write W_j of $a(\vec{j})$ executed before R_j (see Figure 5.11):

$$a(\vec{i}) \bowtie a(\vec{j}) \Rightarrow \exists W_i, W_j, R_i, R_j \text{ such that } \theta(W_i) \ll \theta(R_j) \wedge \theta(W_j) \ll \theta(R_i)$$

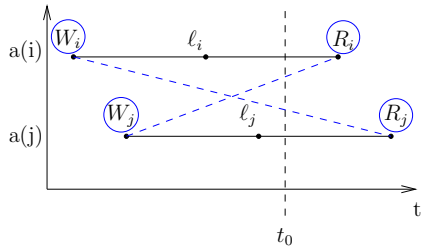


Figure 5.11: Necessary and sufficient condition for $a(i) \bowtie a(j)$

This over-approximation has the effect of considering that an array cell is live from its very first write to its very last read, even when it is written several times and live only on several smaller "intervals". For our specific usage, the optimization of the local memory used to store data from the DDR within a tile band, this means that a local array cell is considered live from the time it is loaded to its last use in the tile band, even if it is overwritten one or more times in the tile band.

Then, the analysis consists simply in enumerating (with four nested loops) all the candidate writes W_i and W_j, and all the candidate reads R_i and R_j. For each such combination of reads and writes, a set P with the following constraints is built, assuming that W_i writes to $a(v_1(\vec{i}_1))$, W_j writes to $a(v_2(\vec{i}_2))$, R_i reads $a(u_1(\vec{i}_3))$, and R_j reads $a(u_2(\vec{i}_4))$.

- $a(\vec{i})$ and $a(\vec{j})$ are accessed: $\vec{i} = v_1(\vec{i}_1) = u_1(\vec{i}_3)$, $\vec{j} = v_2(\vec{i}_2) = u_2(\vec{i}_4)$, and $\vec{i}_1$, $\vec{i}_2$, $\vec{i}_3$, and $\vec{i}_4$ are valid iterations.
- $a(\vec{i})$ and $a(\vec{j})$ conflict: $\theta(W_i, \vec{i}_1) \ll \theta(R_j, \vec{i}_4)$ and $\theta(W_j, \vec{i}_2) \ll \theta(R_i, \vec{i}_3)$. [3]
- differences $\vec{k}$ (to compute DS): $\vec{k} = \vec{i} - \vec{j}$.

3. Adding the constraints $\theta(W_i, \vec{i}_1) \ll \theta(R_i, \vec{i}_3)$ and $\theta(W_j, \vec{i}_2) \ll \theta(R_j, \vec{i}_4)$ are useless. Even if one of them is not satisfied, the live ranges still overlap. And both cannot be wrong at the same time if the other two are satisfied.

Each P built this way is in general a union of polyhedra when the schedule θ is multi-dimensional and thus the order $\ll$ lexicographic. It gives rise to a set DS, which is its projection on $\vec{k}$. After the enumeration, we get a sequence of $DS_1, \ldots, DS_n$ of local results, included in DS. The convex hull of their union finally provides an over-approximation for DS.

A tool called Bee has been developed by Christophe Alias based on these principles. It takes as input a program annotated with pragmas specifying the schedule. Then, Bee analyzes the program, computes a set DS, calls Clak to get an admissible lattice for DS, then computes the mapping, and finally outputs the program with the contracted arrays.

5.3.2 Mapping global memory to local memory

As discussed in the introduction of Section 5.3, the mapping σ relating each array cell $a(\vec{i})$ in the DDR to its image $a\_tmp(\sigma(\vec{i}))$ in local memory can be obtained by applying the previous array contraction technique to a_tmp. To do so, we need to derive from the original kernel a double-bufferized version with the adequate schedules, then to apply the method mentioned above to contract the local arrays. It is actually sufficient to specify the double-bufferized version as a system of clauses expressing the loads and stores, together with a schedule. We now describe how to get these clauses. How to specify the double-buffering execution order with a schedule will be explained later.

Load For each array a, we compute FirstRead(a) as described in Section 5.2. We get a system of clauses:

$$\text{FirstRead}(a) = \left[\vec{i} \in D_1 : S_1(\vec{i})\right] \vee \ldots \vee \left[\vec{i} \in D_n : S_n(\vec{i})\right]$$

To get the restriction on a tile T, we simply intersect each domain D_i with T:

$$\text{FirstRead}(a,t) = \left[\vec{i} \in (D_1 \cap T) : S_1(\vec{i})\right] \vee \ldots \vee \left[\vec{i} \in (D_n \cap T) : S_n(\vec{i})\right]$$

Finally, for further clause manipulations, we rewrite properly each $S_k(\vec{i})$ as a load into a_tmp. If the access to a corresponding to the first read in $S_k(\vec{i})$ is $a(u_k(\vec{i}))$, we redefine $S_k(\vec{i})$ as:

$$S_k(\vec{i}) : a\_tmp(u_k(\vec{i})) = a(u_k(\vec{i}))$$

Then, we add these clauses to the set $\mathcal{L}$ of load clauses. Of course, it remains to specify the double-buffering schedule for each clause. This will be addressed later.

Compute Again, we write the original kernel as a system of clauses. We specify that the execution holds on the tile T by intersecting each clause domain with T. Also, as the computations operate on local arrays, we substitute each array a by its local image a_tmp. We add these clauses to the set $\mathcal{C}$ of compute clauses.

Store As for the load clauses, we compute LastWrite(a). We get a system of clauses:

$$\text{LastWrite}(a) = \left[\vec{i} \in D_1 : S_1(\vec{i})\right] \vee \ldots \vee \left[\vec{i} \in D_n : S_n(\vec{i})\right]$$

This system is restricted to a tile T with a simple intersection:

$$\text{LastWrite}(a,t) = \left[\vec{i} \in (D_1 \cap T) : S_1(\vec{i})\right] \vee \ldots \vee \left[\vec{i} \in (D_n \cap T) : S_n(\vec{i})\right]$$

Finally, we rewrite properly each $S_k(\vec{i})$ as a store into the array a. If the last write is referenced as $a(v_k(\vec{i}))$ in $S_k(\vec{i})$, we define $S_k(\vec{i})$ as:

$$S_k(\vec{i}) : a(v_k(\vec{i})) = a\_tmp(v_k(\vec{i}))$$

Finally, we add these clauses to the set $\mathcal{S}$ of store clauses.

It remains to specify for each clause of $\mathcal{L}$, $\mathcal{C}$, and $\mathcal{S}$ the schedule specifying the double-buffering execution order. As the double-buffering scheme operates on blocks of two tiles, the schedule cannot be specified directly as an affine function. To overcome this issue, we partition each of the sets of clauses $\mathcal{L}$, $\mathcal{C}$, and $\mathcal{S}$ in two parts:

- A restriction to *even* tiles of the tile band: $\mathcal{L}_0$, $\mathcal{C}_0$, and $\mathcal{S}_0$. If K is the innermost tile counter (iterating on the tile band), it suffices to add the constraint $K = 2p$ to each domain of $\mathcal{L}$, $\mathcal{C}$,q and $\mathcal{S}$, where p is a fresh integer variable.
- A restriction to *odd* tiles of the tile band: $\mathcal{L}_1$, $\mathcal{C}_1$, and $\mathcal{S}_1$. If K is the innermost tile counter, it suffices to add the constraint $K = 2p + 1$ to each domain of $\mathcal{L}$, $\mathcal{C}$ and $\mathcal{S}$, where p is a fresh integer variable.

Now, it is easy to specify the double buffering with an affine schedule θ_{db}. For example, the schedule corresponding to the scheme described in Figure 4.43 is given in the following table.

Function	Schedule	Function	Schedule
$\mathcal{L}_0$	$(K, 0)$		
$\mathcal{C}_0$	$(K, 1)$	$\mathcal{L}_1$	$(K-1, 1)$
$\mathcal{S}_0$	$(K, 2)$	$\mathcal{C}_1$	$(K-1, 2)$
		$\mathcal{S}_1$	$(K-1, 3)$

It is easy to see that this schedule is affine and specifies the same double-buffering execution order. This schedule is actually very rough, as each function is assumed to execute its operations in parallel. Of course, this is not the case, but this is actually sufficient to specify the conflicts among the local variables, the only important thing for array contraction. A more accurate schedule will be specified by the code generation step, which is the purpose of the next section. Finally, we get the mapping functions σ for each local array by applying the array contraction technique we described before, to the program defined by the set of clauses $\mathcal{L}_0$, $\mathcal{C}_0$, $\mathcal{S}_0$, $\mathcal{L}_1$, $\mathcal{C}_1$, and $\mathcal{S}_1$, and the schedule θ_{db}. These clauses define the reads and writes to the local arrays.

5.4 Code generation

It remains to generate the final, `C2H`-compliant, C program implementing the double-bufferized input kernel version. The technique is illustrated following the scheme of Figures 4.42 and 4.43.

5.4.1 General organization

The following listing shows the organization of the intermediate program that is generated.

```
1  void Load0() {
2   for(T_1 = ...) {
3    ...
4     for(T_{n-1} = ...) {
5       for(T_n = L(T_1,...,T_{n-1});
6           T_n <= U(T_1,...,T_{n-1}); T_n += 2) {
7        //Synchronize from Store1()
8        //Load(T_1,...,T_n) + sync. to Load1()
```

```
9         //Synchronize to Compute()
10       }
11      }
12     ...
13    }
14   }
15
16   void Load1() {
17    for(T1 = ...) {
18     ...
19      for(Tn−1 = ...) {
20       for(Tn = L(T1,...,Tn−1) + 1;
21           Tn ≤ U(T1,...,Tn−1) + 1; Tn += 2) {
22        //Synchronize from Load0()
23        if(Tn ≤ U(T1,...,Tn))
24          //Load(T1,...,Tn) + sync. to Store0()
25        else //Synchronize to Store0()
26        //Synchronize to Compute()
27       }
28      }
29     ...
30    }
31   }
32
33   void Store0() {
34    for(T1 = ...) {
35     ...
36      for(Tn−1 = ...) {
37       for(Tn = L(T1,...,Tn−1);
38           Tn ≤ U(T1,...,Tn−1); Tn += 2) {
39        //Synchronize from Compute()
40        //Synchronize from Load1()
41        //Store(T1,...Tn) + sync. to Store1()
42       }
43      }
44     ...
45    }
46   }
47
48   void Store1() {
49    for(T1 = ...) {
50     ...
51      for(Tn−1 = ...) {
52       for(Tn = L(T1,...,Tn−1) + 1;
53           Tn ≤ U(T1,...,Tn−1) + 1; Tn += 2) {
54        //Synchronize from Store0()
55        //Synchronize from Compute()
56        if(Tn ≤ U(T1,...,Tn))
57          //Store(T1,...Tn) + sync. to Load0()
58        else //Synchronize to Load0()
59       }
60      }
61     ...
62    }
63   }
64
65   void Compute() {
66    for(T1 = ...) {
67     ...
68      for(Tn−1 = ...) {
69       c = 0;
70       for(Tn = L(T1,...,Tn−1);
71           Tn ≤ U(T1,...,Tn−1); Tn++) {
72        if(c mod 2 == 0) //Synchronize from Load0()
73        else //Synchronize from Load1()
74        //Compute(T1,...,Tn)
75        if(c mod 2 == 0) //Synchronize to Store0()
76        else //Synchronize to Store1()
77        c++;
78       }
79
80       if(c mod 2 == 1) {
81        //Synchronize from Load1()
82        //Synchronize to Store1()
83       }
84      }
85     ...
86    }
87   }
```

We generate an implementation of the double-buffering architecture discussed in Chapter 4. The modules BUFF0_LD, BUFF1_LD, COMP 0/1, STORE0 and STORE1 (see Figure 4.42, Page 101) are implemented as separated functions, and will be translated by `C2H` into separate hardware accelerators. The function Load0 (Line 1) implements BUFF0_LD, the function Load1 (Line 16) implements BUFF1_LD, the function Compute (Line 65) implements COMP0/1, the function Store0 (Line 33) implements STORE0, and the function Store1 (Line 48) implements STORE1.

These functions apply the double-buffering scheme for each tile band, which are processed sequentially. For each tile band, the function Compute() processes all the tiles so as to save hardware resources, whereas the functions Load0(), Load1(), Store0(), and Store1() process the tiles two-by-two, starting from the first tile for Load0() and Store0(), and starting from the second tile for Load1() and Store1(). Notice that Load0() and Store0() are not meant to process exclusively even tiles, or exclusively odd tiles. This slightly differs from the assumptions made in Section 5.3,

but does not change the correctness of the mappings. Each function contains a loop nest iterating over the tiles. For each tile $(T_1, \ldots, T_n)$, a piece of code is executed (shown as a blue comment in the listing), performing the required loads, computations, or stores. We call these pieces of code *kernels*, while we call *drivers* the functions Load0(), Load1(), Compute(), Store0(), and Store1().

The drivers are meant to be run in parallel, and will respect the double-buffering schedule thanks to synchronization signals, as depicted in Figure 4.42, Page 101, and recalled here in Figure 5.12. Dotted arrows represent kernel-level synchronizations used to sequentialize the accesses to the DDR. These synchronizations should be sent as soon as the last request to the DDR within a tile is done, to avoid the penalty due to the loop FSM structure with `C2H`, and thus embedded in the corresponding kernel. Each one is mentioned in the listing in blue with the kernel that contains it with the mention "+ sync. to ...". The remaining synchronizations, represented with blue arrows are not embedded in the kernels but outside, to make sure that DDR latencies are respected.

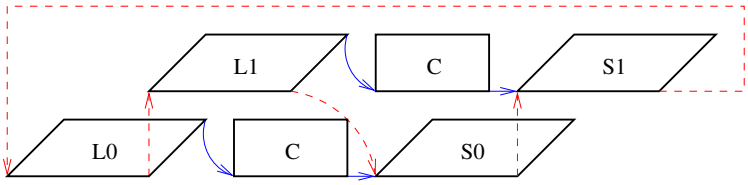


Figure 5.12: Synchronization signals.

The conditions Lines 23 and 56 in Load1() and Store1() capture the case where, in a tile band, the last tile is processed by Load0() and Store0(). In this case, the synchronization signals still need to be propagated properly from Load0() to Load1(), then to Store1(), and finally to Load0() again, which is ready to start the processing of the next tile band. However, the last synchronizations from Store0() and Store1() should be extracted from the kernels so as to guarantee latencies and avoid any overlap with the next tile band (this is not depicted in the code).

The driver program depicted in the previous page can be seen as a template, which will be the same for every input program. Of course, the tricky part is to feed the template with the kernels for loads, computations, and stores. How to generate these kernels is the purpose of the next section.

5.4.2 Generation of load, compute, and store kernels

Code generation in the polyhedral model has been addressed with success leading to powerful methods and tools. Among them, ClooG (`http//www.cloog.org`) implements an improvement of the code-generation method proposed by Fabien Quilleré et al. [97]. A direct approach would be to feed ClooG with the system of clauses $\mathcal{L}$, $\mathcal{C}$, and $\mathcal{S}$ defined in the previous section, together with a sequential schedule. This would give a correct, but very inefficient code. Among the causes of inefficiency, here are the critical ones.

Loop nests. As explained in Chapter 4, `C2H` generates one FSM per loop. Each time a variable is detected (or assumed) to be modified in the loop and referenced later, `C2H` adds a synchronization state which stalls until the variable is written to the DDR or read from the DDR. This results in what we called the data fetch penalty each time the execution flow goes out of a loop. This is even worsen by the fact that the algorithms in `C2H` for dependence analysis and scheduling do not seem very sophisticated: `C2H` sometimes generates poor and unexpected software pipelines, which also causes an unexpected order of memory accesses

thus an undesired spatial locality. A way to reduce these problems is to write the whole kernel as a single "linearized" loop executing one instruction per iteration. This way, the operations (in the polyhedral meaning) will be executed iteration after iteration.

Synchronization. As seen in previous section, synchronizations must be sent to the others accelerators during the last iteration. This can be easily done with a single loop kernel. With an imperfect loop nest with conditionals, deciding when (thus where in the code) to send the signal can be very tricky and can cause a code blow-up.

Spatial data locality of loads and stores. When reading or writing to a different row in the DDR, an additional cost is paid to change the current row, which we called the row change penalty. Thus, it is very important to load and store elements as close as possible, ideally consecutive in the DDR. Thus, one needs to be able to specify in which order the set of loads and stores are scanned.

Scanning polyhedra without nested for loops: the Boulet-Feautrier method

The solution proposed in Chapter 4 is to write the whole kernel as a single loop executing one instruction per iteration. We called this transformation *loop juggling* or *loop linearization*. It can be done in the polyhedral model with the Boulet-Feautrier method [37]. The program to be generated is specified as a system of clauses: $\vec{i} \in D_1 : S_1(\vec{i}) \vee \ldots \vee \vec{i} \in D_n : S_n(\vec{i})$ and a schedule θ. Then, the Boulet-Feautrier algorithm computes and generates two functions, First() and Next():

First() is the first operation to be executed, according to the specified schedule θ. For each clause $\vec{i} \in D_k : S_k(i)$, an iteration $\vec{i}$ with a minimum schedule time $\theta(S, \vec{i})$ is obtained by computing the lexicographic minimum:

$$\text{First}_k = \min_{\ll}\{(\vec{t}, \vec{i}) \mid \vec{t} = \theta(\vec{i}) \wedge \vec{i} \in D_k\}$$

Then, the first operation is obtained by taking the global minimum $\text{First} = \min_{\ll}(\cup_{k=1}^{n} \text{First}_k)$. This computation is similar to those of the FirstRead and LastWrite functions described in Section 5.2. The result is a selection tree giving the first operation depending on parameters.

Next() maps an operation $(S_\ell, \vec{j})$ to its immediate successor in the execution order specified by θ. The computation is similar to First, with the additional constraint that the result must be executed after $(S_\ell, \vec{j})$. For each clause $\vec{i} \in D_k : S_k(\vec{i})$, the first instance of S_k executed after $(S_\ell, \vec{j})$ is computed thanks to the lexicographic minimum:

$$\text{Next}_k = \min_{\ll}\{(\vec{t}, \vec{i}) \mid \vec{t} = \theta(S_k, \vec{i}) \wedge \vec{i} \in D_k \wedge \theta(S_\ell, \vec{j}) \ll \theta(S_k, \vec{i})\}$$

As for First, it remains to compute the global minimum with $\text{Next} = \min_{\ll}(\cup_{k=1}^{n} \text{Next}_k)$. The result is a selection tree giving the next operation depending on parameters value. Notice that the parameters include $\vec{j}$. When the next operation does not exist, the tree leads to a leaf with a special operation $\perp$.

Finally, the code generated will be:

```
ω := First();
while(ω ≠ ⊥) {
    Execute(ω);
    ω := Next(ω);
}
```

where Execute(ω) is a macro in charge of executing the operation ω, typically a single load request.

Load kernel generation

The load kernel will execute the appropriate loads for a given tile t. Remember that the array cells must be accessed in the DDR with a good spatial locality. We choose to schedule the loads so that the arrays are loaded one after the other and, for each array, in the increasing lexicographic order of the indices, thus as much as possible row by row. Section 5.3 gives the system of clauses $\mathcal{L}$ for a load on a tile T. We show how to build the schedule θ_{gen} guaranteeing an optimal data locality. For each array a_ℓ and each tile T, we have:

$$\text{FirstRead}(a_\ell, T) = \left[\vec{i} \in (D_1 \cap T) : S_1(\vec{i})\right] \vee \ldots \vee \left[\vec{i} \in (D_n \cap T) : S_n(\vec{i})\right]$$

Also, let us assume that the read of a_ℓ in $S_k(\vec{i})$ is done with the reference $a_\ell(u_k(\vec{i}))$. We attach, to each clause $\vec{i} \in (D_k \cap t) : S_k(\vec{i})$, the schedule

$$\theta_{\text{gen}}(S_k, \vec{i}) = (\ell, u_k(\vec{i}))$$

Since the schedule follows the lexicographic ordering, the first dimension, equal to ℓ, means that the arrays will be read one after the other. This forbids any interleaving such as a_1, then a_2, then a_1 again. The second dimension, equal to $u_k(\vec{i})$, means that the array cells will be loaded in the increasing lexicographic order of the indices, guaranteeing an optimal spatial locality. Giving $\mathcal{L}$ together with θ_{gen} to the Boulet-Feautrier method gives the desired load kernel. Finally, the synchronization on the last iteration can be added with a simple post-processing:

```
ω := First();
while(ω ≠ ⊥) {
    Execute(ω);
    ω := Next(ω);
    if(ω == ⊥) {//Last iteration ?
        //Send the synchronization signal
    }
}
```

Store kernel generation

We proceed similarly for the stores. Section 5.3 gives the system of clauses $\mathcal{S}$ for the stores on a tile T. For each array a_ℓ, we have:

$$\text{LastWrite}(a_\ell, T) = \left[\vec{i} \in (D_1 \cap T) : S_1(\vec{i})\right] \vee \ldots \vee \left[\vec{i} \in (D_n \cap T) : S_n(\vec{i})\right]$$

Also, let us assume that the write of a_ℓ in $S_k(\vec{i})$ is done with the reference $a_\ell(v_k(\vec{i}))$. We attach to each clause $\vec{i} \in (D_k \cap t) : S_k(\vec{i})$ the schedule:

$$\theta_{\text{gen}}(S_k, \vec{i}) = (\ell, v_k(\vec{i}))$$

Similarly, this schedule will force to store the arrays one after the other and, for each array, to store the required cells in the increasing lexicographic order of indices, ensuring an optimal spatial locality. As well, giving $\mathcal{S}$ together with θ_{gen} to the Boulet-Feautrier method gives the desired store kernel. The synchronization required on the last iteration is added by a post-processing, in the same way as for the load code.

Compute kernel generation

Section 5.3 gives the system of clauses $\mathcal{C}$ for a local computation on a tile T. It suffices to feed the Boulet-Feautrier method with the clauses of $\mathcal{C}$ together with a sequential schedule to get the desired compute kernel.

5.5 Experimental results

Our method has been fully implemented by Christophe Alias in a tool called Chuba. Chuba uses the state-of-the-art polyhedral libraries PIP [66] and Polylib [10] *via* the polyhedral compiler infrastructure PoCo [4] developed by Christophe Alias. The array contraction is done thanks to the tools Bee for the array lifetime analysis part and Clak for the admissible lattices computations [23]. The tool Chuba itself represents more than 1800 lines of C++ code. The reduced size of Chuba is mainly due to the high-level abstractions provided by PoCo.

Chuba takes as input the C source code of the kernel to optimize and generates a C source code, which fully implements a `C2H`-compliant double-bufferized optimization. The input parameters, as the loop tiling, are specified with pragmas in the source code. Figure 5.13 shows the input program for the product of polynomials. The part of the code to be optimized is bounded with the pragmas `begin_scop` and `end_scop`, at Lines 11 and 21. Then, the tiling is specified with the pragma `schedule`, Line 18. The tiles are assumed to be square, and the tile size is specified with the pragma `tile_size`, Line 14. This specifies the tiling hyperplanes depicted in Figure 5.2, Page 123. Finally, the pragma `depth`, at Line 13, gives the depth of the tile loop which will enumerate the tiles inside a tile band. It is possible to specify a depth less or equal to the depth of the loop nest.

We have run Chuba on the kernels DMA and vector sum discussed in chapter 4. We give the synthesis results obtained from the automatically-transformed version. For the sake of comparison, we recall the synthesis results obtained from the manual-transformed version presented in chapter 4.

Figures 5.14 and 5.15 presents the speed-ups obtained for the DMA kernel. The speed-ups of the manually-transformed kernel are depicted with a dotted line. The automatically-transformed version is slightly less efficient than the manual one because of slightly larger synchronization-to-transfer latencies present in the automatically-transformed version. These latencies could be removed by code transformation techniques. However, starting at a tile size of 1K, the differences become very small when compared to the execution time and could be neglected.

Figures 5.16 and 5.17 presents the speed-ups for the vector sum kernel. Again, the speed-up for the automatically-transformed version is slightly smaller than for the manual implementation but, from a block size of 1K, the difference is very small and can be neglected.

Figure 5.18 presents the synthesis results for the DMA and vector sum kernels. The table recalls the synthesis results obtained in the previous chapter for the direct implementation and manually-transformed versions with maximum tile size of 1K. Also, we present in the table the manually-transformed code and the automatically-transformed version with a tile size of 8K. The automatic version uses slightly more ALUTs and registers than the manually-transformed version mostly due to the fact that the automatically-transformed version has two separate FIFOs used for synchronization between `BUFF0_LD` and `BUFF1_LD` to `COMP01`. The automatically-generated version uses also more multipliers to perform tile address calculations. These multipliers could be normally

4. A release and a research report will be available soon.

```
1  int N;
2
3  int main(void)
4  {
5    //Declarations and Initializations
6    // [...]
7
8    for(i=0; i<=2*N; i++)
9      c[i] = 0;
10
11 #pragma begin_scop
12
13 #pragma depth[2]
14 #pragma tile_size[32]
15
16   for(i=0; i<=N; i++)
17     for(j=0; j<=N; j++)
18 #pragma schedule[i+j][i]
19       c[i+j] = c[i+j] + a[i]*b[j];
20
21 #pragma end_scop
22
23   return 0;
24 }
```

Figure 5.13: Product of polynomials: input program for CHUBA

removed by standard strength reduction techniques. Also, the maximum frequency is slightly-slower for the automatically-transformed code. This could be due to more complex code and fewer memory modules due to the use of double-port memories available in the FPGA. The use of double-port memories induces additional synthesis constraints.

5.6 Conclusion

In this chapter, we have proposed a fully-automatic method to optimize a kernel written in C for `C2H`, the C-to-VHDL compiler of Altera. The result is set of C functions implementing the double-buffering scheme, which will be translated to separate hardware accelerators by `C2H`. The double buffering is driven thanks to a loop tiling specified by the user. Then, original techniques are used to analyze, optimize, and generate the final code. The method has been fully implemented, and the first experimental results show the method to be effective and give promising results.

There is room for many improvements. Our prototype is a direct implementation of our method, with no attempt for optimization. For the analysis part (Step 1, communication coalescing, and Step 2, local memory management), this only impacts the analysis time without causing any performance degradation on the target program. However, this is more problematic for the code generation part. The Boulet-Feautrier method involves selection trees and combination thereof. A direct implementation of the combination leads rapidly to a code blow-up. Techniques exist to combine properly selection trees and to avoid this situation. We plan to add them to our tool. It may also be interesting to try to build the sets Load(T) and Store(T), not by computing first reads

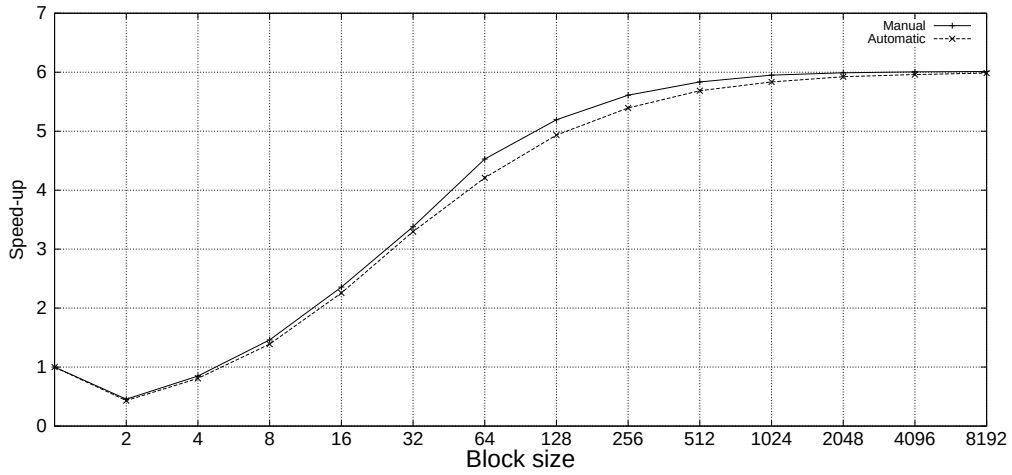


Figure 5.14: Manually- and automatically-transformed code vs. original DMA experimental results

Block size	2	4	8	16	32	64	128
Blocked manual	5481760	2943940	1709070	1059120	738340	551320	480610
Speed-up manual	0.45	0.85	1.46	2.36	3.38	4.53	5.19
Blocked automatic	5769350	3088260	1795440	1105950	756980	592930	505780
Speed-up automatic	0.43	0.80	1.39	2.25	3.29	4.20	4.93
Block size	256	512	1024	2048	4096	8192	
Blocked manual	444930	427740	419410	416730	415570	415110	
Speed-up manual	5.61	5.84	5.95	5.99	6.00	6.01	
Blocked automatic	462770	438980	427770	421390	418730	416960	
Speed-up automatic	5.39	5.68	5.83	5.92	5.96	5.98	

Figure 5.15: Manually- and automatically-transformed code vs. original DMA (Table in ns)

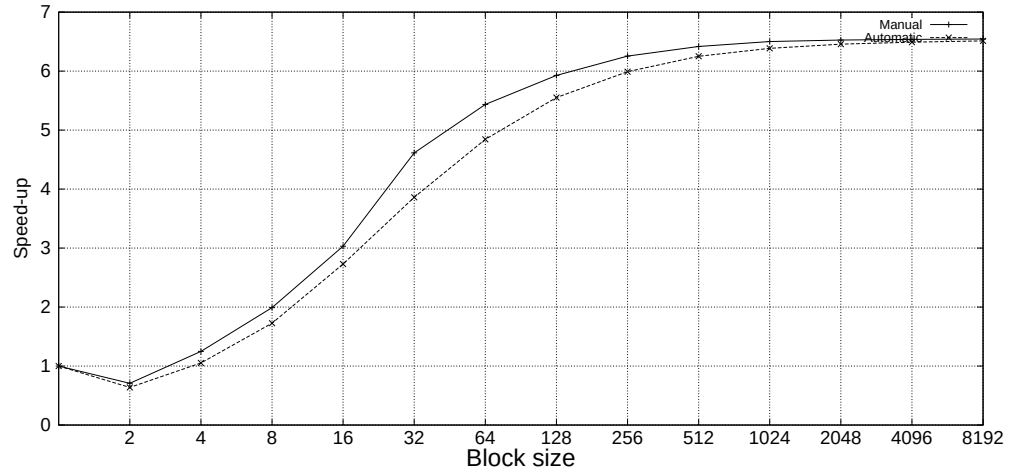


Figure 5.16: Manually- and automatically-transformed code vs. original vector sum results

Block size	2	4	8	16	32	64	128
Blocked manual	5412340	3067580	1924200	1262690	829860	704490	645900
Speed-up manual	0.71	1.25	1.99	3.03	4.61	5.43	5.93
Blocked automatic	6005080	3639320	2217910	1402770	991750	790600	689590
Speed-up automatic	0.63	1.05	1.72	2.72	3.86	4.84	5.55

Block size	256	512	1024	2048	4096	8192	
Blocked manual	612160	596620	588960	586580	585680	584990	
Speed-up manual	6.25	6.48	6.50	6.53	6.54	6.54	
Blocked automatic	639120	612440	599550	592910	589940	587830	
Speed-up automatic	5.99	6.25	6.38	6.45	6.48	6.51	

Figure 5.17: Manually- and automatically-transformed code vs. original vector sum (Table in ns)

Kernel	Speed-up	ALUT	Dedicated registers	Total registers
System alone		4406	3474	3606
DMA direct implementation	1	4598	3612	3744
DMA transformed manually (1K tile)	6.01	9665	10244	10376
DMA transformed manually (8K tile)	6.01	9853	10517	10649
DMA automatic (8K tile)	5.99	11052	12133	12265
Vector sum direct implementation	1	5333	4607	4739
Vector sum transformed manually (1K tile)	6.54	10345	10346	11478
Vector sum transformed manually (8K tile)	6.54	10881	11361	11493
Vector sum automatic (8K tile)	6.51	11632	13127	13259

Kernel	Total block memory bits	DSP block 9-bit elements	Max Freq. (MHz)
System alone	66908	8	205.85
DMA direct implementation	66908	8	200.52
DMA transformed manually (1k tile)	203100	8	167.25
DMA transformed manually (8k tile)	1120604	8	162.55
DMA automatic (8k tile)	1120348	48	167.87
Vector sum direct implementation	68956	8	189.04
Vector sum transformed manually (1k tile)	269148	8	175.93
Vector sum transformed manually (8k tile)	1645404	8	164
Vector sum automatic (8k tile)	1644892	48	159.8

Figure 5.18: Synthesis results of DMA and vector sum examples using direct, manual, and automatic transformation techniques for 1k and 8k maximum tile size

and last writes, but through the computation of the sets $\mathrm{In}(T)$ and $\mathrm{Out}(T)$, or approximations, as suggested in Section 5.2.1. This may lead to faster algorithms and also to sets which are less complex to scan. This has still to be explored.

The parts to be optimized and the loop tiling must be specified by the user. It could be worth to detect automatically the hot spots in the program with profiling techniques and to decide with appropriate criteria which of them deserve to be optimized. Also, the loop tiling could possibly be found automatically. As discussed in Section 5.2, a good solution is formed by tiling hyperplanes that pushes the data flow and input dependences in the innermost loops. Existing approaches can find such a loop tiling [35] and could be connected to our tool. So far, the choice of the tile size is left to the user. Instead, it would be better to let the user specify the maximum size for the local memory, and to let the tool find automatically the corresponding tile size. As well, the tiles are assumed to have the same size along each dimension. This is not always the best solution and the relevant sizes should be found automatically.

Finally, the algorithms we presented are fundamentally restricted to static control programs. For example, how to write a loop tiling for a program with while loops is still an open question. The same is true for programs with non-affine array references. Section 5.2 shows that Step 1 (communication coalescing) could work for general programs providing approximations for the sets In and Out. We believe that such approximations could be found with abstract interpretation techniques [46]. However, how to extend the array contraction techniques (Step 2) and how to specify and to generate the final code (Step 3) are still open questions. All these questions can be interesting to solve, not only for the particular context of optimizing DDR accesses for hardware accelerators automatically generated by `C2H`, but also each time tiling needs to be used to deport a kernel on an accelerator with a smaller local memory, as it is the case for GPGPUs [76, 30].

Chapter 6

Conclusion

In the recent years, an exponential increase in performance demands was observed on the IC market. IC manufacturers have to design very complex circuits in a very small amount of time due to time-to-market constraints. Traditional low-level design methods cannot always guarantee that these constraints can be reached. In order to meet these constraints, new HLS methods were proposed. HLS increases the abstraction level of the design specification and thus decreases the design effort. Unfortunately, higher abstraction level implies a huge design space that needs to be explored. Operation scheduling including parallelism handling, interconnection generation, interface design, data transfer and reuse etc., are supposed to be handled now by the tool rather than the designer. Recent advanced developments in HLS focused mostly on the first two problematics using automatic or semi-automatic methods. The interface design and data availability problem in HLS are still in their infancy and, most of the time, left to the designer.

In this thesis, after a quick introduction (Chapter 1) and an overview of HLS (Chapter 2), we presented, in the next three chapters, several attempts and approaches for integrating an HLS-generated hardware accelerator into a system with performance metrics linked to data availability and external optimal interface transfer in mind. The performance of most multimedia applications is indeed limited by the data availability. By optimizing the data availability, one could increase the performances of these applications.

In Chapter 3, we studied a system where the cache memory of a GPP can be reused by hardware accelerators. The first part of the chapter presented a study done with the HLS tool MMAlpha on the complex matrix-matrix multiplication application. Instead of connecting directly the generated IP using a template interface, we manually built, using the application data access information, a glue logic implementing the interface connection to a GPP and a local memory architecture that promoted data reuse. The local memory architecture controller implemented techniques related to blocking, double buffering, and temporal storage. We demonstrated, on this matrix-matrix multiply example, the difficulty of manually generating such a glue logic and the performance loss due to poor communication in a complete system.

In the second part of Chapter 3, to avoid tedious manual implementations of the glue logic needed around a generated hardware accelerator, we transformed the input code of the HLS tool so as to improve data transfer and availability. Transforming the code by hand is not an easy task. Using languages such as C, modern HLS tools can synthesize from a higher level of abstraction than with traditional hardware-design methodologies. When using such a language, one can easily apply powerful high-level code transformations to improve performances as opposed to low-level

descriptions where it is just impossible to apply them. A possibility is to implement all the required transformations in the HLS tool itself, but because of the IR complexity, it can be very difficult. Also, most of the time, it is even impossible to do so since the HLS tools are provided in binary format. As we demonstrated, a solution is to use a code-preprocessing tool before synthesis. By doing so, the tool can be adapted and reused for different HLS tools, eliminating the re-engineering cost and perpetuating the preprocessing tool. The preprocessing tool will perform the not-so-trivial adaptation together with high-performance code transformations on the algorithmic specification to match the restrictions of the HLS tool with performance metrics in mind. In this context, we presented a semi-automatic source-to-source transformation methodology based on the WRaP-IT polyhedral loop transformation tool. Since it is a source-to-source tool, it can be used for any HLS tool that accepts a specification written in C. In our study, we used the HLS tool `Spark`, and we worked on optimizing the memory communication and data reuse. As `Spark` does not generate local memories, we used the cache of a GPP for local storage. Our study demonstrates that HLS tools can be coupled with a source-to-source preprocessing step to obtain better synthesis results. We believe that a user-friendly tool for such transformations, which would be specialized to HLS both in terms of functionalities and code generation, would be a must for the users of HLS tools. The tool Gecos (developed, among others, by Steven Derrien) has such objectives.

The performance improvements in Chapter 3 were obtained mostly due to data reuse, either by designing a local memory architecture or by reusing the processor cache memory system. Designing such a memory architecture by hand (as for MMAlpha) is not easy and, most importantly, it is not flexible, i.e., it requires a significant designer rethinking if the input specification changes during advanced design stages. More generally, as our study with `Spark` also shows, it is mandatory that the HLS tools provide mechanisms to interface, within the complete system and without degrading performances, the hardware accelerators they generate. We believe that this is a *sine qua non* condition for HLS tools to be usable in a hardware design. In the rest of the thesis, we proposed to go one step further and to use the HLS tools themselves, in a form of "meta-compilation", to synthesize, together with the computational IP, a memory architecture attached to it. We used, in our study, the `C2H` HLS tool from Altera. We analyzed the case where hardware accelerators are connected directly to a DDR memory. Our study was divided into two parts: Chapter 4 was devoted to the design, by hand, of an optimization scheme, while Chapter 5 presented its automation.

More precisely, in Chapter 4, we analyzed the use of various code transformation techniques for HLS, in particular those that can be used to optimize data transfer between the accelerator and the memory, and to promote data reuse. We showed that direct techniques do not suffice to obtain acceptable results. We proposed a technique that consists of generating software-pipelined communicating accelerators using a globally asynchronous locally synchronous (GALS) methodology. One accelerator is performing the actual computations and the other accelerators, using techniques such as double buffering but not only, are responsible for data transfer and memory reuse in the generated memory architecture. The process synchronization of processes are performed at a block level, thanks to loop tiling. In this type of design, the computations can be hidden by communications when communications are longer. Also, if computations require more time than communications, a stall in a communication is damped without stalling the computations, in a true LIS design.

The last chapter (Chapter 5) showed how this scheme can be automated. We proposed a method which takes as input a naive version of the kernel to be optimized, written in C, and which derives automatically a set of C functions corresponding to the pipelined communicating hardware accelerators specified in Chapter 4. The double-buffering (more precisely the overlap

of communications and computations) relies on a loop tiling specified by the user by means of pragmas in the input C program, and on sophisticated code analysis and data reuse techniques. New techniques have been defined to analyze, optimize, and generate the final C code optimized for `C2H`. The method has been fully implemented in a tool called CHUBA and the first experiments give promising results, with communication patterns to the DDR and performances similar to the solutions previously designed by hand in Chapter 4.

So far, CHUBA is a prototype, which incorporates our method without any attempts for optimization. There is still work to improve the back-end code generation part, such as simplifying code expressions, removing redundancy, specializing the code to cheap hardware structures, etc. Also, the algorithms we implemented are inherently limited to static control kernels with affine array references. It could be worth to extend our technique to programs with while loops and non-affine arrays references. A first step in this direction is already given in Chapter 5, which explains how to deal with approximations. Techniques based on abstract interpretation such as [48] could also be helpful. Finally, in our current implementation, the parts to be optimized must be specified by the user with pragmas, together with the loop tiling schedule and the tile size. It may possible to derive these pragmas from more hardware user-friendly high-level hardware directives such as the maximum local memory available, the type and organization of the memory, etc. But, for the moment, it is more reasonable to rely on the user expertise, in a semi-automatic design. Many other high-level hardware directives can be imagined to specify a maximum number of resources such as memory modules or logic elements that the accelerator performing the computation could use. More generally, we believe that designing a language based on C and such compilation directives could be of interest in the context of HLS as it was done with HPF and distributed-memory processors. Similarly, designing advanced data partitioning and parallelization techniques in order to improve the computation and communication performances is important. Indeed, in this thesis, we only optimized one sequential process communicating to one DDR memory. An important extension would be to be able to derive, from a C-level description with compilation directives, a set of computing accelerators, communicating with external memory or with other accelerators through buffers to be optimized, both in terms of size and structure.

We point out that the techniques presented in this thesis for DDR optimizations can be used also for optimizing data transfers over burst-based buses such as PLB or packet-based ones such as `HyperTransport` from AMD (`XtremeData` platform) or `QuickPath Interconnect` from Intel, and even over fast high-performance optical inter-IC network communications. In such communication systems, grouped and sometimes successive transfers are imperative for maximizing data throughput. Further research is required to analyze the impact of these code transformations in a more complex SOC design. As mentioned in the conclusion of Chapter 5, we also found recently some similarities with analysis and optimizations required for GPGPUs.

We also think that all the methods we developed can be used in a context of a more general SOC design. Usually, in a SOC system, multiple hardware accelerators are synthesized independently using various HLS tools or just designed manually. In most cases, these hardware accelerators make use of temporary storage memory elements. In such a system, the accelerators are not necessarily active at the same time. Thus, memory resources of idle accelerators are wasted but could be reused. We are currently considering how to share memory resources in such a system. We propose to design and implement a hardware memory management unit (HMMU). This HMMU represents a hardware module that will integrate the data transfer techniques presented in Chapters 4 and 5 and could serve multiple computation accelerators, instead of just one as presented in this thesis.

Each accelerator will have a wrapper-encapsulating interface to the outside world that will manage all the required memory connections. When an accelerator is launched, the wrapper sends a temporary resource request token containing the required memory size and the number of requested memory modules to the HMMU. The memory unit will allocate the request in its bitmap-based data structure, if possible, and will send the response to the wrapper that will use it to make a proper connection of the accelerator to the allocated memory modules. We also intend to use this technique to perform dynamic memory allocation in hardware (dynamic allocation in local memories). Usually, in a complex SOC system, there is a very limited number of external memory modules such as a DDR. When an accelerator requires to access some data from the external DDR memory, the wrapper will send this request to the HMMU and will remap the accesses to a local storage where the HMMU will fetch or store the required data. Using the techniques presented above, the SOC should use less local memory resources and should have better performances in external memory accesses.

To summarize, we believe that our study contributes to push high-level transformations, in particular loop transformations that were primarily developed in the context of high-performance computing, to the context of HLS. We believe that, to make HLS tools usable, the designer should not have to take care of low-level details linked to the interface of the hardware accelerator in the complete system. The HLS tool should provide mechanisms to optimize this interface (and, in particular, it should provide mechanisms to guarantee memory coherency), it should provide a companion tool to perform source-to-source transformations, and it should be based on a language with directives where the designer can drive the tool to the right communication, computation, and memory optimizations. Our contributions go in these directions.

Bibliography

[1] Agility DK design suite. http://www.agilityds.com.

[2] Altera C2H: Nios II C-to-hardware acceleration compiler. http://www.altera.com/products/ip/processors/nios2/tools/c2h/ni2-c2h.html.

[3] AutoPilot, AutoESL DesignTechnologies INC. http://www.autoesl.com/.

[4] C-to-Silicon Compiler, Cadence. http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx.

[5] Dinero IV trace-driven uniprocessor cache simulator. http://www.cs.wisc.edu/markhill/DineroIV/.

[6] Forte design system: Cynthesizer. http://www.forteds.com.

[7] Impulse-C, accelerate software using FPGAs as coprocessors. http://www.impulseaccelerated.com.

[8] Mentor CatapultC high-level synthesis. http://www.mentor.com/products/esl/high_level_synthesis.

[9] Nisc: No instruction-set computer (C-to-RTL). http://www.ics.uci.edu/nisc.

[10] Polylib – A library of polyhedral functions. http://www.irisa.fr/polylib.

[11] Synfora Pico Express algorithmic synthesis in SoC. http://www.synfora.com/products/picoexpress.html.

[12] Synopsys: Synphony. http://www.synopsys.com/.

[13] Ugh: User-guided high-level synthesis. http://www-asim.lip6.fr/recherche/disydent/disydent_sect_12.html.

[14] Wikipedia. http://wikipedia.org/.

[15] *Verilog - A Language Reference Manual*, 1996. Version 1.0.

[16] *Handel-C Language Reference Manual*, 1998.

[17] *IEEE Standard VHDL Language Reference Manual*, 2000.

[18] Behavioral synthesis crossroad. http://www.eetasia.com/ART_8800338054_480100_NT_4d3932ab.HTM, June 2004.

[19] JESD79E JEdec standard DDR1. 2005.

[20] JESD79-3A JEdec standard DDR3. Technical report, 2007.

[21] *Embedded Design Handbook*, July 2009. http://www.altera.com.

[22] *Nios II C2H Compiler User Guide*, November 2009. Version 9.1. http://www.altera.com.

[23] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+Cl@k: An implementation of lattice-based array contraction in the source-to-source translator Rose. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, San Diego, USA, June 2007.

[24] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *17th International Static Analysis Symposium (SAS'10)*, Perpignan, France, September 2010.

[25] Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing DDR-SDRAM communications at C-level for automatically-generated hardware accelerators. An experience with the Altera C2H HLS tool. In *21st IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP'10)*. IEEE Computer Society, July 2010.

[26] John R. Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[27] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[28] Ivan Augé, Frédéric Pétrot, François Donnet, and Pascal Gomez. Platform-based design from parallel C specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(12):1811–1826, 2005.

[29] David F. Bacon, Susan L. Graham, and Oliver J. S. Harp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[30] Soufiane Baghdadi, Armin Größlinger, and Albert Cohen. Putting automatic polyhedral compilation for GPGPU to work. Presentation at CPC'10, International Workshop on Compilers for Parallel Computers, July 2010.

[31] Uptal Banerjee. *Loop Transformations for Restructuring Compilers - The Foundations*. Kluwer Academic Publishers, 1993.

[32] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *13th International Conference on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 7–16. IEEE Computer Society, 2004.

[33] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting polyhedral loop transformations to work. In Lawrence Rauchwerger, editor, *Languages and Compilers for Parallel Computing (LCPC'03)*, volume 2958 of *Lecture Notes in Computer Science*, pages 209–225. Springer, 2004.

[34] Uday Bondhugula, Muthu Manikandan Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *17th International Conference on Compiler Construction (CC'08)*, pages 132–146, 2008.

[35] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM International Conference on Programming Languages Design and Implementation (PLDI'08)*, pages 101–113, Tucson, Arizona, June 2008.

[36] Youcef Bouchebaba and Fabien Coelho. Tiling and memory reuse for sequences of nested loops. In Burkhard Monien and Rainer Feldmann, editors, *8th International Euro-Par Conference (Euro-Par'02)*, volume 2400 of *Lecture Notes in Computer Science*, pages 255–264. Springer-Verlag, 2002.

[37] Pierre Boulet and Paul Feautrier. Scanning polyhedra without Do-loops. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 4–9, 1998.

[38] Mihai Budiu and Seth Copen Goldstein. Compiling application-specific hardware. In *Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 853–863, London, UK, 2002. Springer-Verlag.

[39] David Cachera, Anne-Claire Guillou, Fabien Quilleré, Patrice Quinton, Sanjay Rajopadhye, and Tanguy Risset. Hardware design methodologies with Alpha. Technical report, IRISA, Campus de Beaulieu, 35042 Rennes, France.

[40] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp architecture and C compiler. *Computer*, 33(4):62–69, April 2000.

[41] João M. P. Cardoso and Pedro C. Diniz. *Compilation Techniques for Reconfigurable Architectures*. 2009.

[42] Francky Catthoor, Sven Wuytack, G. E. de Greef, Florin Balasa, Lode Nachtergaele, and Arnout Vandecappelle. *Custom Memory Management Methodology*. Kluwer Academic Publishers, 1998.

[43] Daniel Chavarría-Miranda and John Mellor-Crummey. Effective communication coalescing for data-parallel applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 14–25, Chicago, IL, USA, 2005.

[44] Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. Platform-based behavior-level and system-level synthesis. In *International SOC Conference*, pages 199–202. IEEE, 2006.

[45] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan-Kaufmann, 2003.

[46] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96, Tucson, January 1978.

[47] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[48] Béatrice Creusillet. *Analyses de régions de tableaux et applications*. PhD thesis, École des mines de Paris, December 1996.

[49] Alain Darte. On the complexity of loop fusion. *Parallel Computing*, 26:1175–1193, 2000.

[50] Alain Darte and Guillaume Huard. Loop shifting for loop compaction. *International Journal of Parallel Programming*, 28:499–534, 2000.

[51] Alain Darte and Guillaume Huard. Complexity of multi-dimensional loop alignment. In 19*th International Symposium on Theoretical Aspects of Computer Science (STACS'02)*, volume 2285, pages 179–191. Springer Verlag, March 2002.

[52] Alain Darte and Guillaume Huard. New complexity results on array contraction and related problems. *Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, 40(1):35–55, May 2005.

[53] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization*. Birkhauser, 2000.

[54] Alain Darte, Robert Schreiber, B. Ramakrishna Rau, and Frédéric Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Transactions on Design Automation of Electronic Systems (ACM TODAES)*, 7(1):159–172, 2002.

[55] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, October 2005.

[56] Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *International Journal of Parallel Programming*, 25(6):447–496, December 1997.

[57] Hugo De Man, Jan M. Rabaey, J. Vanhoof, Gert Goossens, Paul Six, and Luc Claesen. Cathedral II – a computer-aided synthesis system for digital signal processing VLSI systems. *Computer-Aided Engineering Journal*, 5(2):55–66, 1988.

[58] Giovanni De Micheli, David Ku, Frédéric Mailhot, and Thomas Truong. The Olympus synthesis system. *IEEE Design and Test of Computers*, 7(5):37–53, 1990.

[59] Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout, and Dirk Stroobandt. From loop transformation to hardware generation. In *17th ProRISC Workshop*, pages 249–255, November 2006.

[60] Harald Devos, Kristof Beyls, Mark Christiaens, Jan Van Campenhout, Erik D'Hollander, and Dirk Stroobandt. Finding and applying loop transformations for generating optimized FPGA implementations. In Per Stenström, editor, *Transactions on High-Performance Embedded Architectures and Compilers I*, volume 4050 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2007.

[61] Harald Devos, Jan Van Campenhout, and Dirk Stroobandt. Building an application-specific memory hierarchy on FPGA. In *Proceedings of the 2nd HiPEAC Workshop on Reconfigurable Computing*, pages 53–62, Göteborg, 1 2008.

[62] Rainer Dömer, Andreas Gerstlauer, and Daniel Gajski. *SpecC Language Reference Manual*, 2002. Version 2.0. SpecC Technology Open Consortium.

[63] Florent Dupont de Dinechin, Patrice Quinton, Sanjay Rajopadhye, and Tanguy Risset. First steps in Alpha. Technical Report 1244, Irisa, 1999.

[64] Stephen A. Edwards. The challenges of hardware synthesis from C-like languages. In *Design, Automation, and Test in Europe (DATE'05)*, pages 66–67. IEEE Computer Society, 2005.

[65] Paul Feautrier. Array expansion. In *ACM International Conference on Supercomputing (ICS'88)*, pages 429–441, 1988.

[66] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[67] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, February 1991.

[68] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II: Multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[69] Paul Feautrier. *The Data Parallel Programming Model*, volume 1132 of *LNCS Tutorial*, chapter Automatic Parallelization in the Polytope Model, pages 79–103. Springer Verlag, 1996.

[70] Antoine Fraboulet and Tanguy Risset. Master interface for on-chip hardware accelerator burst communications. *Journal of VLSI Signal Processing*, 2(1):73–85, 2007.

[71] David R. Galloway. The Transmogrifier C hardware description language and compiler for FPGAs. *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'95)*, pages 136–144, 1995.

[72] Abhijit Ghosh, Joachim Kunkel, and Stan Liao. Hardware synthesis from C/C++. In *Design, Automation, and Test in Europe (DATE'99)*, Munich, Germany, 1999. ACM. Article No.82.

[73] Sylvain Girbal. *Optimisation d'application - composition de transformations de programme: modèle et outils*. PhD thesis, Universite de Paris XI Orsay, 2005.

[74] Eddy De Greef, Francky Catthoor, and Hugo De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing*, 23:1811–1837, 1997.

[75] Martin Griebl and Chris Lengauer. The loop parallelizer LooPo. In Michael Gerndt, editor, *Workshop on Compilers for Parallel Computers (CPC'96)*, volume 21, pages 311–320. 1996.

[76] Armin Größlinger. Precise management of scratchpad memories for localizing array accesses in scientific codes. In O. de Moor and M. Schwartzbach, editors, *International Conference on Compiler Construction (CC'09)*, volume 5501 of *Lecture Notes in Computer Science*, pages 236–250. Springer-Verlag, 2009.

[77] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Spark: A high-level synthesis framework for applying parallelizing compiler transformations. *International Conference on VLSI Design*, pages 461–466, 2003.

[78] Sumit Gupta, Rajesh Gupta, Nikil Dutt, and Alexandru Nicolau. *Spark: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic Publishers, 2004.

[79] François Irigoin and Rémi Triolet. Supernode partitioning. In *15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*, pages 319–329, San Diego, USA, 1988. ACM Press.

[80] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *5th International Conference on Supercomputing (ICS'91)*, pages 244–251, Köln, West Germany, 1991. ACM Press.

[81] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress 74*, pages 471–475, 1974.

[82] Takashi Kambe, Akihisa Yamada, Koichi Nishida, Kazuhisa Okada, Mitsuhisa Ohnishi, Andrew Kay, Paul Boca, Vince Zammit, and Toshio Nomura. A C-based synthesis system, Bach, and its application. *Asia and South Pacific Design Automation Conference (ASP-DAC'01)*, pages 151–155, 2001.

[83] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr, and Mary E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[84] David Ku and Giovanni De Micheli. HardwareC – A language for hardware design (version 2.0). Technical report, Stanford, CA, USA, 1990.

[85] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24:649–671, 1998.

[86] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[87] Chris Lengaueur. Loop parallelization in the polytope model. In Eike Best, editor, *4th International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *Lecture Notes in Computer Science*, pages 398–416. Springer, Hildesheim, Germany, August 1993.

[88] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PoPL'97)*. ACM Press, January 1997.

[89] Vincent Loechner, Benoît Meister, and Philippe Clauss. Precise data locality optimization of nested loops. *Journal of Supercomputing*, 21(1):37–76, 2002.

[90] Eric Martin, Olivier Sentieys, Hélène Dubois, and Jean-Luc Philippe. Gaut: An architectural synthesis tool for dedicated signal processors. In *Design Automation Conference with EURO-VHDL'93 (EURO-DAC'93)*, pages 14–19, September 1993.

[91] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design and Test of Computers*, 26:18–25, 2009.

[92] Christophe Mauras. *Alpha: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. PhD thesis, Université de Rennes I, December 1989.

[93] Steven S. Muchnick. *Compiler Design Implementation*. Morgan-Kaufmann, 1997.

[94] Guy-René Perrin and Alain Darte, editors. *The Data Parallel Programming Model*, volume 1132 of *LNCS Tutorial*. Springer Verlag, 1996.

[95] Alexandru Plesco and Tanguy Risset. Coupling loop transformations and high-level synthesis. In *SYMPosium en Architectures nouvelles de machines (SYMPA'08)*, Fribourg, Switzerland, February 2008.

[96] Sébastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. GRAPHITE: Polyhedral analyses and optimizations for GCC. In *GCC Developers Summit*, pages 179–198, 2006.

[97] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28:469–498, 2000.

[98] Bob Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *27th Annual International Symposium on Microarchitecture (MICRO'27)*, pages 63–74, San Jose, California, United States, 1994. ACM.

[99] Patrick Schaumont, Serge Vernalde, Luc Rijnders, Marc Engels, and Ivo Bolsens. A programming environment for the design of complex high speed ASICs. In *Design Automation Conference (DAC'98)*, pages 315–320, June 1998.

[100] Rob Schreiber, Shail Aditya, Scott Mahlke, Vinod Kathail, Bob Ramakrishna Rau, Darren Cronquist, and Mukund Sivaraman. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing Systems for Signal Image and Video Technology (VLSISP)*, 31:127–142, 2002.

[101] Donald Soderman and Yuri Panchul. Implementing C algorithms in reconfigurable hardware using C2Verilog. pages 339–342, Napa Valley, CA, 1998. IEEE Computer Society.

[102] Charles E. Stroud, Ronald R. Munoz, and David A. Pierce. Behavioral-model synthesis with cones. *IEEE Design and Test of Computers*, 5:22–30, 1988.

[103] Kalle Tammemäe, Mattias O'Nils, Anders Tornemo, and Hannu Tenhunen. VLSI system level codesign toolkit AKKA. In *14th IEEE Norchip Conference*, pages 196–202, 1996.

[104] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. Translating affine nested-loop programs to process networks. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*, pages 220–229. ACM, 2004.

[105] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *15th International Conference on Compiler Construction (ETAPS CC'06)*, volume 3923 of *Lecture Notes in Computer Science*, pages 185–201. Springer-Verlag, Vienna, Austria, March 2006.

[106] Michael E. Wazlowski, Lalit Agarwal, Tony S. Lee, A. Smith, E. Lam, Peter M. Athanas, Harvey F. Silverman, and Sumit Ghosh. PRISM-II compiler and architecture. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16, April 1993.

[107] Doran K. Wilde. *From Alpha to imperative code: A transformational compiler for an array-based functional language.* PhD thesis, Corvallis, OR, USA, 1996.

[108] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29:31–37, 1994.

[109] Michael Wolfe. A loop restructuring research tool. Technical Report CSE 90-014, Oregon Graduate Institute, August 1990.

[110] Jingling Xue. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, 2000.

Colophon

This thesis was typeset using LaTeX. The images were drawn using XFig and experimental results were plotted using Gnuplot.

Chapter 7

Appendix

7.1 MMAlpha

7.1.1 Reference code XUP

The code presented below represents the code used for communication between the PowerPC processor and the interface of the accelerator.

```
/*
 * This source file contains C code of an example HW/SW interface for the
 * Xilinx XUP board.
 *
 * Redistributions of any form whatsoever must retain and/or include the
 * following acknowledgment, notices and disclaimer:
 *
 * This product includes software developed by Carnegie Mellon University.
 *
 * Copyright (c) 2006 by J. C. Hoe, Carnegie Mellon University
 *
 * You may not use the name "Carnegie Mellon University" or derivations
 * thereof to endorse or promote products derived from this software.
 *
 * If you modify the software you must place a notice on or within any
 * modified version provided or made available to any third party stating
 * that you have modified the software. The notice shall include at least
 * your name, address, phone number, email address and the date and purpose
 * of the modification.
 *
 * THE SOFTWARE IS PROVIDED "AS-IS" WITHOUT ANY WARRANTY OF ANY KIND, EITHER
 * EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY WARRANTY
 * THAT THE SOFTWARE WILL CONFORM TO SPECIFICATIONS OR BE ERROR-FREE AND ANY
 * IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE,
 * TITLE, OR NON-INFRINGEMENT. IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY
 * BE LIABLE FOR ANY DAMAGES, INCLUDING BUT NOT LIMITED TO DIRECT, INDIRECT,
 * SPECIAL OR CONSEQUENTIAL DAMAGES, ARISING OUT OF, RESULTING FROM, OR IN
 * ANY WAY CONNECTED WITH THIS SOFTWARE (WHETHER OR NOT BASED UPON WARRANTY,
 * CONTRACT, TORT OR OTHERWISE).
 */

#include <stdio.h>
#include <assert.h>
#include <xparameters.h>
#include <xio.h>

#define OCM0BASE ((int*)XPAR_DOCM_CNTLR_BASEADDR)
#define OCMMEMSIZE (4*1024) // in bytes

volatile int *OcmInBase = OCM0BASE + (0*OCMMEMSIZE/sizeof(int));
volatile int *OcmOutBase = OCM0BASE + (1*OCMMEMSIZE/sizeof(int));
volatile int *OcmCntrlBase = OCM0BASE + (2*OCMMEMSIZE/sizeof(int));
int *OcmMemBase = OCM0BASE + (3*OCMMEMSIZE/sizeof(int));

#define IBUFHD (0)
#define IBUFTL (8)
#define OBUFHD (16)
#define OBUFTL (24)
#define SFTRST (56)

#define SYNC {asm volatile ("sync"::);}
```

```
54 // cacheblock aligned buffer
   int buff1[4*OCMMEMSIZE/sizeof(int)] __attribute__ ((aligned(32)));
56 int buff2[4*OCMMEMSIZE/sizeof(int)] __attribute__ ((aligned(32)));

58 int __ibufhead=0;
   int __ibuftail =(-1%(OCMMEMSIZE/sizeof(int)));
60 int __obufhead=0;
   int __obuftail =0;
62
   void init() {
64   static int cache=0;
     printf("␣---␣reset␣HW␣---␣\n");
66
     // enable caching, assume 512MByte
68   if (!cache) {
       XCache_EnableICache(0xcc000000);
70     XCache_EnableDCache(0xa0000000);
       cache=1;
72   }

74   SYNC;
     OcmCntrlBase[SFTRST]=-1;
76   SYNC;
     OcmCntrlBase[IBUFHD]=0;
78   OcmCntrlBase[OBUFTL]=0;

80   __ibufhead=0;
      __ibuftail =(-1%(OCMMEMSIZE/sizeof(int)));
82   __obufhead=0;
      __obuftail =0;
84
     SYNC
86   OcmCntrlBase[SFTRST]=0;
     SYNC;
88 }

90 void dumpState(char *str) {
     unsigned int temp;
92   printf("---%s---\n", str);
     // read hardware registers
94   printf("ibufhd=%x␣---␣", OcmCntrlBase[IBUFHD]); // head pointer (next to pop) for SW->HW
     printf("ibuftl=%x␣---␣", OcmCntrlBase[IBUFTL]); // tail pointer (next to push) for SW->HW
96   printf("obufhd=%x␣---␣", OcmCntrlBase[OBUFHD]); // head pointer (next to pop) for HW->SW
     printf("obuftl=%x\n", OcmCntrlBase[OBUFTL]); // tail pointer (next to push) for HW->SW
98   printf("sftrst=%x\n", OcmCntrlBase[SFTRST]);

100  // read software "cached" copies
     printf("softptr␣ibufhd=%x␣---␣", __ibufhead);
102  printf("softptr␣ibuftl=%x␣---␣", __ibuftail );
     printf("softptr␣obufhd=%x␣---␣", __obufhead);
104  printf("softptr␣obuftl=%x\n", __obuftail );
   }
106
   /* Pushing from buffer "num" integer words to the OCM HW.
108 * This is a blocking call.
    */
110 void push(int *buffer, int num) {
     int index=0;
112  int released=0;
     int numx=num;
114
     while(num--) {
116    while (__ibufhead==__ibuftail) {
         if (!released) {
118        SYNC;
           OcmCntrlBase[IBUFHD]=__ibufhead;
120        released=1;
         }
122      SYNC;
         __ibuftail =OcmCntrlBase[IBUFTL];
124      __ibuftail --;
         __ibuftail %=(OCMMEMSIZE/sizeof(int));
126    }
       released=0;
128
       OcmInBase[__ibufhead]=buffer[index++];
130    __ibufhead++;
       __ibufhead%=(OCMMEMSIZE/sizeof(int));
132  }
     SYNC;
134  OcmCntrlBase[IBUFHD]=__ibufhead;
   }
136
   /* Receiving into buffer "num" integer words from the OCM HW.
138 * This is a blocking call.
    */
140 void pop(int *buffer, int num) {
```

```
  int index=0;
142  int released=0;

144  while(num--) {
       while (__obufhead==__obuftail) {
146      if (!released) {
           SYNC;
148        OcmCntrlBase[OBUFTL]=__obuftail;
           released=1;
150      }
         SYNC;
152      __obufhead=OcmCntrlBase[OBUFHD];
       }
154    released=0;

156    buffer[index++]=OcmOutBase[__obuftail];
       __obuftail++;
158    __obuftail %=(OCMMEMSIZE/sizeof(int));
     }
160  SYNC;
     OcmCntrlBase[OBUFTL]=__obuftail;
162 }

164 /* example code to exercise and check the interface */
    void testefifo() {
166   int iter=0, size=0, dataseed=10;

168   printf("---test EFIFO ---\n");

170   for(iter=0;iter<10000;iter++) {
        int idx;
172
        // initialize buff1 with data
174     size=(size+1)%2000;
        idx=size;
176
        while(idx--) {
178       buff1[idx]=dataseed++;
        }
180
        // push buff1 and pop buff2, demo HW is just does a loopback
182     push(buff1,size);
        pop(buff2,size);
184
        // check returned buff2 against original buff1
186     idx=size;
        while(idx--) {
188       if (!(buff1[idx]==buff2[idx])) {
            int k;
190         for(k=-2;k<2;k++) {
              printf("%d: buff1[%x]==buff2[%x]\n",idx-k, buff1[idx-k],buff2[idx-k]);
192         }
            for(k=-2;k<2;k++) {
194           int where=(__ibufhead-size+(idx-k))%(OCMMEMSIZE/sizeof(int));
              assert(__ibufhead==__obuftail);
196           printf("%d %d %d %d\n",__ibufhead,idx,k,size);
              printf("ocmi[%d]=%x::ocmo[%d]=%x\n",
198                   where,
                     OcmInBase[where],
200                  where,
                     OcmOutBase[where]);
202         }
            dumpState("Bailing out\n");
204         printf("size=%d\n",size);
          }
206       assert(buff1[idx]==buff2[idx]);
        }
208
        if (!(size%100)) printf(".");
210   }
      putchar('\n');
212 }

214 extern int test_mmm();

216 int main (void) {
      int iter=0;
218
      init();
220
      test_mmm(); // test the reference implementation of matrix multiplication
222
      while(1) {
224     printf("---Iteration %d ---\n", iter++);
        testefifo();
226   }

228   return(0);
```

}

7.1.2 Alphahard code of the matrix-matrix multiplication

We present below the Alphahard representation of the matrix-matrix multiplication obtained using MMAlpha synthesis.

```
system ControlmatmultModule :{N,NB | NB+1<=N; 2<=NB}
                              ( )
 returns (CImXctl1PPXInit : {t | t=2; NB+1<=N; 2<=NB} | {t | 3<=t<=N+2; NB+1<=N; 2<=NB} of boolean;
          CReXctl1PPXInit : {t | t=2; NB+1<=N; 2<=NB} | {t | 3<=t<=N+2; NB+1<=N; 2<=NB} of boolean);
let
  CImXctl1PPXInit[t] =
      case
        { | t=2; NB+1<=N; 2<=NB} : True[];
        { | 3<=t<=N+2; NB+1<=N; 2<=NB} : False[];
      esac;
  CReXctl1PPXInit[t] =
      case
        { | t=2; NB+1<=N; 2<=NB} : True[];
        { | 3<=t<=N+2; NB+1<=N; 2<=NB} : False[];
      esac;
tel;

system cellmatmultModule1 :{p1,p2,N,NB | p1=1; 2<=p2<=NB; NB+1<=N}
                  (bImXMirr1 : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
                   bReXMirr1 : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
                   AImReg1Xloc : {t | p2-1<=t<=p2+N-2; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
                   AReReg2Xloc : {t | p2-1<=t<=p2+N-2; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
                   CImXctl1PReg7Xloc : {t | p2<=t<=p2+N; p1=1; 2<=p2<=NB; NB+1<=N} of boolean;
                   CReXctl1PReg11Xloc : {t | p2<=t<=p2+N; p1=1; 2<=p2<=NB; NB+1<=N} of boolean)
        returns   (AIm : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
                   ARe : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
                   BIm : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
                   BRe : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
                   CImXctl1P : {t | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of boolean;
                   CReXctl1P : {t | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of boolean;
                   CRe : {t | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
                   CIm : {t | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16]);
var
  CImloc8 : {t | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  CReloc7 : {t | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  CReXctl1Ploc6 : {t | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of boolean;
  CImXctl1Ploc5 : {t | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of boolean;
  BReloc4 : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  BImloc3 : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  AReloc2 : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  AImloc1 : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  AImReg1 : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  AReReg2 : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  CImReg5 : {t | p2+2<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  CImXctl1 : {t | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of boolean;
  CImXctl1PReg7 : {t | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of boolean;
  CReReg9 : {t | p2+2<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  CReXctl1 : {t | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of boolean;
  CReXctl1PReg11 : {t | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of boolean;
  opAIm : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  opARe : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  opBIm : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  opBRe : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resIm : {t | p2+1<=t<=p2+N; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resIm1 : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resIm1Reg13 : {t | p2+1<=t<=p2+N; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resIm2 : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resIm2Reg14 : {t | p2+1<=t<=p2+N; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resImReg6 : {t | p2+2<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resRe : {t | p2+1<=t<=p2+N; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resRe1 : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resRe1Reg15 : {t | p2+1<=t<=p2+N; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resRe2 : {t | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resRe2Reg16 : {t | p2+1<=t<=p2+N; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resReReg10 : {t | p2+2<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  TSep1 : {t | p2+2<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  TSep2 : {t | p2+2<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
let
  CIm[t] = CImloc8[t];
  CRe[t] = CReloc7[t];
  CReXctl1P[t] = CReXctl1Ploc6[t];
  CImXctl1P[t] = CImXctl1Ploc5[t];
  BRe[t] = BReloc4[t];
  BIm[t] = BImloc3[t];
  ARe[t] = AReloc2[t];
  AIm[t] = AImloc1[t];
  AImReg1[t] = AImReg1Xloc[t-1];
```

```
    AReReg2[t] = AReReg2Xloc[t-1];
    CImXctl1PReg7[t] = CImXctl1PReg7Xloc[t-1];
    CReXctl1PReg11[t] = CReXctl1PReg11Xloc[t-1];
    resRe2Reg16[t] = resRe2[t-1];
    resRe1Reg15[t] = resRe1[t-1];
    resIm2Reg14[t] = resIm2[t-1];
    resIm1Reg13[t] = resIm1[t-1];
    resReReg10[t] = resRe[t-1];
    CReReg9[t] = CReloc7[t-1];
    resImReg6[t] = resIm[t-1];
    CImReg5[t] = CImloc8[t-1];
    BImloc3[t] = bImXMirr1[t];
    BReloc4[t] = bReXMirr1[t];
    AImloc1[t] = AImReg1[t];
    AReloc2[t] = AReReg2[t];
    opARe[t] = AReloc2[t];
    opAIm[t] = AImloc1[t];
    opBRe[t] = BReloc4[t];
    opBIm[t] = BImloc3[t];
    use  Mult[] (opARe, opBIm) returns (resRe1) ;
    use  Mult[] (opAIm, opBRe) returns (resRe2) ;
    resRe[t] = resRe1Reg15[t] + resRe2Reg16[t];
    use  Mult[] (opARe, opBRe) returns (resIm1) ;
    use  Mult[] (opAIm, opBIm) returns (resIm2) ;
    resIm[t] = resIm1Reg13[t] - resIm2Reg14[t];
    CReXctl1Ploc6[t] = CReXctl1PReg11[t];
    CReXctl1[t] = CReXctl1Ploc6[t];
    TSep1[t] = CReReg9[t] + resReReg10[t];
    CReloc7[t] =
        case
          { | t=p2+1; p1=1; 2<=p2<=NB; NB+1<=N} : if (CReXctl1[t]) then 0[] else 0[];
          { | p2+2<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} : if (CReXctl1[t]) then 0[] else TSep1[t];
        esac;
    CImXctl1Ploc5[t] = CImXctl1PReg7[t];
    CImXctl1[t] = CImXctl1Ploc5[t];
    TSep2[t] = CImReg5[t] + resImReg6[t];
    CImloc8[t] =
        case
          { | t=p2+1; p1=1; 2<=p2<=NB; NB+1<=N} : if (CImXctl1[t]) then 0[] else 0[];
          { | p2+2<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} : if (CImXctl1[t]) then 0[] else TSep2[t];
        esac;
tel;

system cellmatmultModule2 :{p1,p2,N,NB | p1=1; p2=1; NB+1<=N; 2<=NB}
                  (bImXMirr1 : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
                   bReXMirr1 : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
                   aImXMirr1 : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
                   aReXMirr1 : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
                   CImXctl1PPXInitXIn : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
                   CReXctl1PPXInitXIn : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean)
         returns  (AIm : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
                   ARe : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
                   BIm : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
                   BRe : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
                   CImXctl1PP : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
                   CImXctl1P : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
                   CReXctl1PP : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
                   CReXctl1P : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
                   CRe : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
                   CIm : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16]);
  var
    CImloc10 : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    CReloc9 : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    CReXctl1Ploc8 : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
    CReXctl1PPloc7 : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
    CImXctl1Ploc6 : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
    CImXctl1PPloc5 : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
    BReloc4 : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    BImloc3 : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    AReloc2 : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    AImloc1 : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    CImReg5 : {t | 3<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    CImXctl1 : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
    CReReg9 : {t | 3<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    CReXctl1 : {t | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
    opAIm : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    opARe : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    opBIm : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    opBRe : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    resIm : {t | 2<=t<=N+1; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    resIm1 : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    resIm1Reg13 : {t | 2<=t<=N+1; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    resIm2 : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    resIm2Reg14 : {t | 2<=t<=N+1; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    resImReg6 : {t | 3<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    resRe : {t | 2<=t<=N+1; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    resRe1 : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    resRe1Reg15 : {t | 2<=t<=N+1; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
```

```
    resRe2 : {t | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    resRe2Reg16 : {t | 2<=t<=N+1; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    resReReg10 : {t | 3<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    TSep1 : {t | 3<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
    TSep2 : {t | 3<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
let
    CIm[t] = CImloc10[t];
    CRe[t] = CReloc9[t];
    CReXctl1P[t] = CReXctl1Ploc8[t];
    CReXctl1PP[t] = CReXctl1PPloc7[t];
    CImXctl1P[t] = CImXctl1Ploc6[t];
    CImXctl1PP[t] = CImXctl1PPloc5[t];
    BRe[t] = BReloc4[t];
    BIm[t] = BImloc3[t];
    ARe[t] = AReloc2[t];
    AIm[t] = AImloc1[t];
    resRe2Reg16[t] = resRe2[t-1];
    resRe1Reg15[t] = resRe1[t-1];
    resIm2Reg14[t] = resIm2[t-1];
    resIm1Reg13[t] = resIm1[t-1];
    resReReg10[t] = resRe[t-1];
    CReReg9[t] = CReloc9[t-1];
    resImReg6[t] = resIm[t-1];
    CImReg5[t] = CImloc10[t-1];
    BImloc3[t] = bImXMirr1[t];
    BReloc4[t] = bReXMirr1[t];
    AImloc1[t] = aImXMirr1[t];
    AReloc2[t] = aReXMirr1[t];
    opARe[t] = AReloc2[t];
    opAIm[t] = AImloc1[t];
    opBRe[t] = BReloc4[t];
    opBIm[t] = BImloc3[t];
    use  Mult[] (opARe, opBIm) returns (resRe1) ;
    use  Mult[] (opAIm, opBRe) returns (resRe2) ;
    resRe[t] = resRe1Reg15[t] + resRe2Reg16[t];
    use  Mult[] (opARe, opBRe) returns (resIm1) ;
    use  Mult[] (opAIm, opBIm) returns (resIm2) ;
    resIm[t] = resIm1Reg13[t] - resIm2Reg14[t];
    CReXctl1PPloc7[t] = CReXctl1PPXInitXIn[t];
    CReXctl1Ploc8[t] = CReXctl1PPloc7[t];
    CReXctl1[t] = CReXctl1Ploc8[t];
    TSep1[t] = CReReg9[t] + resReReg10[t];
    CReloc9[t] =
        case
          { | t=2; p1=1; p2=1; NB+1<=N; 2<=NB} : if (CReXctl1[t]) then 0[] else 0[];
          { | 3<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} : if (CReXctl1[t]) then 0[] else TSep1[t];
        esac;
    CImXctl1PPloc5[t] = CImXctl1PPXInitXIn[t];
    CImXctl1Ploc6[t] = CImXctl1PPloc5[t];
    CImXctl1[t] = CImXctl1Ploc6[t];
    TSep2[t] = CImReg5[t] + resImReg6[t];
    CImloc10[t] =
        case
          { | t=2; p1=1; p2=1; NB+1<=N; 2<=NB} : if (CImXctl1[t]) then 0[] else 0[];
          { | 3<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} : if (CImXctl1[t]) then 0[] else TSep2[t];
        esac;
tel ;

system cellmatmultModule3 :{p1,p2,N,NB | 2<=p1<=NB; p2=1; NB+1<=N}
                  (aImXMirr1 : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
                   aReXMirr1 : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
                   BImReg3Xloc : {t | p1-1<=t<=p1+N-2; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
                   BReReg4Xloc : {t | p1-1<=t<=p1+N-2; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
                   CImXctl1PPReg8Xloc : {t | p1<=t<=p1+N; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
                   CReXctl1PPReg12Xloc : {t | p1<=t<=p1+N; 2<=p1<=NB; p2=1; NB+1<=N} of boolean)
        returns  (AIm : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
                   ARe : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
                   BIm : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
                   BRe : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
                   CImXctl1PP : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
                   CImXctl1P : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
                   CReXctl1PP : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
                   CReXctl1P : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
                   CRe : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
                   CIm : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16]);
var
    CImloc10 : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    CReloc9 : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    CReXctl1Ploc8 : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
    CReXctl1PPloc7 : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
    CImXctl1Ploc6 : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
    CImXctl1PPloc5 : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
    BReloc4 : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    BImloc3 : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    AReloc2 : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    AImloc1 : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    BImReg3 : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    BReReg4 : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
```

```
    CImReg5 : {t | p1+2<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    CImXctl1 : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
    CImXctl1PPReg8 : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
    CReReg9 : {t | p1+2<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    CReXctl1 : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
    CReXctl1PPReg12 : {t | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
    opAIm : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    opARe : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    opBIm : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    opBRe : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    resIm : {t | p1+1<=t<=p1+N; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    resIm1 : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    resIm1Reg13 : {t | p1+1<=t<=p1+N; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    resIm2 : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    resIm2Reg14 : {t | p1+1<=t<=p1+N; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    resImReg6 : {t | p1+2<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    resRe : {t | p1+1<=t<=p1+N; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    resRe1 : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    resRe1Reg15 : {t | p1+1<=t<=p1+N; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    resRe2 : {t | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    resRe2Reg16 : {t | p1+1<=t<=p1+N; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    resReReg10 : {t | p1+2<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    TSep1 : {t | p1+2<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
    TSep2 : {t | p1+2<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
  let
    CIm[t] = CImloc10[t];
    CRe[t] = CReloc9[t];
    CReXctl1P[t] = CReXctl1Ploc8[t];
    CReXctl1PP[t] = CReXctl1PPloc7[t];
    CImXctl1P[t] = CImXctl1Ploc6[t];
    CImXctl1PP[t] = CImXctl1PPloc5[t];
    BRe[t] = BReloc4[t];
    BIm[t] = BImloc3[t];
    ARe[t] = AReloc2[t];
    AIm[t] = AImloc1[t];
    BImReg3[t] = BImReg3Xloc[t-1];
    BReReg4[t] = BReReg4Xloc[t-1];
    CImXctl1PPReg8[t] = CImXctl1PPReg8Xloc[t-1];
    CReXctl1PPReg12[t] = CReXctl1PPReg12Xloc[t-1];
    resRe2Reg16[t] = resRe2[t-1];
    resRe1Reg15[t] = resRe1[t-1];
    resIm2Reg14[t] = resIm2[t-1];
    resIm1Reg13[t] = resIm1[t-1];
    resReReg10[t] = resRe[t-1];
    CReReg9[t] = CReloc9[t-1];
    resImReg6[t] = resIm[t-1];
    CImReg5[t] = CImloc10[t-1];
    BImloc3[t] = BImReg3[t];
    BReloc4[t] = BReReg4[t];
    AImloc1[t] = aImXMirr1[t];
    AReloc2[t] = aReXMirr1[t];
    opARe[t] = AReloc2[t];
    opAIm[t] = AImloc1[t];
    opBRe[t] = BReloc4[t];
    opBIm[t] = BImloc3[t];
    use  Mult[] (opARe, opBIm) returns (resRe1) ;
    use  Mult[] (opAIm, opBRe) returns (resRe2) ;
    resRe[t] = resRe1Reg15[t] + resRe2Reg16[t];
    use  Mult[] (opARe, opBRe) returns (resIm1) ;
    use  Mult[] (opAIm, opBIm) returns (resIm2) ;
    resIm[t] = resIm1Reg13[t] - resIm2Reg14[t];
    CReXctl1PPloc7[t] = CReXctl1PPReg12[t];
    CReXctl1Ploc8[t] = CReXctl1PPloc7[t];
    CReXctl1[t] = CReXctl1Ploc8[t];
    TSep1[t] = CReReg9[t] + resReReg10[t];
    CReloc9[t] =
        case
          { | t=p1+1; 2<=p1<=NB; p2=1; NB+1<=N} : if (CReXctl1[t]) then 0[] else 0[];
          { | p1+2<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} : if (CReXctl1[t]) then 0[] else TSep1[t];
        esac;
    CImXctl1PPloc5[t] = CImXctl1PPReg8[t];
    CImXctl1Ploc6[t] = CImXctl1PPloc5[t];
    CImXctl1[t] = CImXctl1Ploc6[t];
    TSep2[t] = CImReg5[t] + resImReg6[t];
    CImloc10[t] =
        case
          { | t=p1+1; 2<=p1<=NB; p2=1; NB+1<=N} : if (CImXctl1[t]) then 0[] else 0[];
          { | p1+2<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} : if (CImXctl1[t]) then 0[] else TSep2[t];
        esac;
  tel ;

  system cellmatmultModule4 :{p1,p2,N,NB | 2<=p1<=NB; 2<=p2<=NB; NB+1<=N}
    (AImReg1Xloc : {t | p1+p2-2<=t<=p1+p2+N-3; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
    AReReg2Xloc : {t | p1+p2-2<=t<=p1+p2+N-3; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
    BImReg3Xloc : {t | p1+p2-2<=t<=p1+p2+N-3; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
    BReReg4Xloc : {t | p1+p2-2<=t<=p1+p2+N-3; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
    CImXctl1PReg7Xloc : {t | p1+p2-1<=t<=p1+p2+N-1; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of boolean;
    CReXctl1PReg11Xloc : {t | p1+p2-1<=t<=p1+p2+N-1; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of boolean)
```

```
returns (AIm : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
         ARe : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
         BIm : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
         BRe : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
         CImXctl1P : {t | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of boolean;
         CReXctl1P : {t | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of boolean;
         CRe : {t | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
         CIm : {t | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16]);
var
  CImloc8 : {t | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  CReloc7 : {t | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  CReXctl1Ploc6 : {t | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of boolean;
  CImXctl1Ploc5 : {t | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of boolean;
  BReloc4 : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  BImloc3 : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  AReloc2 : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  AImloc1 : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  AImReg1 : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  AReReg2 : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  BImReg3 : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  BReReg4 : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  CImReg5 : {t | p1+p2+1<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  CImXctl1 : {t | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of boolean;
  CImXctl1PReg7 : {t | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of boolean;
  CReReg9 : {t | p1+p2+1<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  CReXctl1 : {t | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of boolean;
  CReXctl1PReg11 : {t | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of boolean;
  opAIm : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  opARe : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  opBIm : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  opBRe : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resIm : {t | p1+p2<=t<=p1+p2+N−1; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resIm1 : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resIm1Reg13 : {t | p1+p2<=t<=p1+p2+N−1; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resIm2 : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resIm2Reg14 : {t | p1+p2<=t<=p1+p2+N−1; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resImReg6 : {t | p1+p2+1<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resRe : {t | p1+p2<=t<=p1+p2+N−1; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resRe1 : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resRe1Reg15 : {t | p1+p2<=t<=p1+p2+N−1; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resRe2 : {t | p1+p2−1<=t<=p1+p2+N−2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resRe2Reg16 : {t | p1+p2<=t<=p1+p2+N−1; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  resReReg10 : {t | p1+p2+1<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  TSep1 : {t | p1+p2+1<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  TSep2 : {t | p1+p2+1<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
let
  CIm[t] = CImloc8[t];
  CRe[t] = CReloc7[t];
  CReXctl1P[t] = CReXctl1Ploc6[t];
  CImXctl1P[t] = CImXctl1Ploc5[t];
  BRe[t] = BReloc4[t];
  BIm[t] = BImloc3[t];
  ARe[t] = AReloc2[t];
  AIm[t] = AImloc1[t];
  AImReg1[t] = AImReg1Xloc[t−1];
  AReReg2[t] = AReReg2Xloc[t−1];
  BImReg3[t] = BImReg3Xloc[t−1];
  BReReg4[t] = BReReg4Xloc[t−1];
  CImXctl1PReg7[t] = CImXctl1PReg7Xloc[t−1];
  CReXctl1PReg11[t] = CReXctl1PReg11Xloc[t−1];
  resRe2Reg16[t] = resRe2[t−1];
  resRe1Reg15[t] = resRe1[t−1];
  resIm2Reg14[t] = resIm2[t−1];
  resIm1Reg13[t] = resIm1[t−1];
  resReReg10[t] = resRe[t−1];
  CReReg9[t] = CReloc7[t−1];
  resImReg6[t] = resIm[t−1];
  CImReg5[t] = CImloc8[t−1];
  BImloc3[t] = BImReg3[t];
  BReloc4[t] = BReReg4[t];
  AImloc1[t] = AImReg1[t];
  AReloc2[t] = AReReg2[t];
  opARe[t] = AReloc2[t];
  opAIm[t] = AImloc1[t];
  opBRe[t] = BReloc4[t];
  opBIm[t] = BImloc3[t];
  use  Mult[] (opARe, opBIm) returns (resRe1) ;
  use  Mult[] (opAIm, opBRe) returns (resRe2) ;
  resRe[t] = resRe1Reg15[t] + resRe2Reg16[t];
  use  Mult[] (opARe, opBRe) returns (resIm1) ;
  use  Mult[] (opAIm, opBIm) returns (resIm2) ;
  resIm[t] = resIm1Reg13[t] − resIm2Reg14[t];
  CReXctl1Ploc6[t] = CReXctl1PReg11[t];
  CReXctl1[t] = CReXctl1Ploc6[t];
  TSep1[t] = CReReg9[t] + resReReg10[t];
  CReloc7[t] =
      case
        { | t=p1+p2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} : if (CReXctl1[t]) then 0[] else 0[];
```

```
        { | p1+p2+1<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} : if (CReXctl1[t]) then 0[] else TSep1[t];
431     esac;
  CImXctl1Ploc5[t] = CImXctl1PReg7[t];
433 CImXctl1[t] = CImXctl1Ploc5[t];
  TSep2[t] = CImReg5[t] + resImReg6[t];
435 CImloc8[t] =
      case
437     { | t=p1+p2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} : if (CImXctl1[t]) then 0[] else 0[];
        { | p1+p2+1<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} : if (CImXctl1[t]) then 0[] else TSep2[t];
439     esac;
tel;
441
system matmultModule :{N,NB | NB+1<=N; 2<=NB}
443             (aImXMirr1In : {t,p1,p2 | p1<=t<=p1+N-1; 1<=p1<=NB; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
                aReXMirr1In : {t,p1,p2 | p1<=t<=p1+N-1; 1<=p1<=NB; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
445             bImXMirr1In : {t,p1,p2 | p2<=t<=p2+N-1; p1=1; 1<=p2<=NB; NB+1<=N; 2<=NB} of integer[U,16];
                bReXMirr1In : {t,p1,p2 | p2<=t<=p2+N-1; p1=1; 1<=p2<=NB; NB+1<=N; 2<=NB} of integer[U,16])
447     returns (CImOut : {t,p1,p2 | p1+p2<=t<=p1+p2+N; 1<=p1<=NB; 1<=p2<=NB} of integer[U,16];
                 CReOut : {t,p1,p2 | p1+p2<=t<=p1+p2+N; 1<=p1<=NB; 1<=p2<=NB} of integer[U,16]);
449 var
  CImXctl1PPXInit : {t | 2<=t<=N+2; NB+1<=N; 2<=NB} of boolean;
451 CReXctl1PPXInit : {t | 2<=t<=N+2; NB+1<=N; 2<=NB} of boolean;
  bImXMirr1 : {t,p1,p2 | p2<=t<=p2+N-1; p1=1; 1<=p2<=NB; NB+1<=N; 2<=NB} of integer[U,16];
453 bReXMirr1 : {t,p1,p2 | p2<=t<=p2+N-1; p1=1; 1<=p2<=NB; NB+1<=N; 2<=NB} of integer[U,16];
  AIm1 : {t,p1,p2 | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
455 AImReg1Xloc : {t,p1,p2 | p1+p2-2<=t<=p1+p2+N-3; 1<=p1<=NB; 2<=p2<=NB+1} of integer[U,16];
  ARe1 : {t,p1,p2 | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
457 AReReg2Xloc : {t,p1,p2 | p1+p2-2<=t<=p1+p2+N-3; 1<=p1<=NB; 2<=p2<=NB+1} of integer[U,16];
  BIm1 : {t,p1,p2 | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
459 BImReg3Xloc : {t,p1,p2 | p1+p2-2<=t<=p1+p2+N-3; 2<=p1<=NB+1; 1<=p2<=NB} of integer[U,16];
  BRe1 : {t,p1,p2 | p2<=t<=p2+N-1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
461 BReReg4Xloc : {t,p1,p2 | p1+p2-2<=t<=p1+p2+N-3; 2<=p1<=NB+1; 1<=p2<=NB} of integer[U,16];
  CImXctl1P1 : {t,p1,p2 | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of boolean;
463 CImXctl1PReg7Xloc : {t,p1,p2 | p1+p2-1<=t<=p1+p2+N-1; 1<=p1<=NB; 2<=p2<=NB+1; NB+1<=N; 2<=NB} of boolean;
  CReXctl1P1 : {t,p1,p2 | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of boolean;
465 CReXctl1PReg11Xloc : {t,p1,p2 | p1+p2-1<=t<=p1+p2+N-1; 1<=p1<=NB; 2<=p2<=NB+1; NB+1<=N; 2<=NB} of boolean;
  CRe1 : {t,p1,p2 | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
467 CIm1 : {t,p1,p2 | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  aImXMirr1 : {t,p1,p2 | p1<=t<=p1+N-1; 1<=p1<=NB; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
469 aReXMirr1 : {t,p1,p2 | p1<=t<=p1+N-1; 1<=p1<=NB; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
  AIm2 : {t,p1,p2 | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
471 ARe2 : {t,p1,p2 | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
  BIm2 : {t,p1,p2 | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
473 BRe2 : {t,p1,p2 | 1<=t<=N; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
  CImXctl1PP2 : {t,p1,p2 | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
475 CImXctl1PPReg8Xloc : {t,p1,p2 | p1<=t<=p1+N; 2<=p1<=NB+1; p2=1; NB+1<=N; 2<=NB} of boolean;
  CImXctl1P2 : {t,p1,p2 | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
477 CReXctl1PP2 : {t,p1,p2 | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
  CReXctl1PPReg12Xloc : {t,p1,p2 | p1<=t<=p1+N; 2<=p1<=NB+1; p2=1; NB+1<=N; 2<=NB} of boolean;
479 CReXctl1P2 : {t,p1,p2 | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
  CImXctl1PPXInitXIn : {t,p1,p2 | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
481 CReXctl1PPXInitXIn : {t,p1,p2 | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of boolean;
  CRe2 : {t,p1,p2 | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
483 CIm2 : {t,p1,p2 | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
  AIm3 : {t,p1,p2 | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
485 ARe3 : {t,p1,p2 | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
  BIm3 : {t,p1,p2 | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
487 BRe3 : {t,p1,p2 | p1<=t<=p1+N-1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
  CImXctl1PP3 : {t,p1,p2 | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
489 CImXctl1P3 : {t,p1,p2 | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
  CReXctl1PP3 : {t,p1,p2 | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
491 CReXctl1P3 : {t,p1,p2 | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of boolean;
  CRe3 : {t,p1,p2 | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
493 CIm3 : {t,p1,p2 | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} of integer[U,16];
  AIm4 : {t,p1,p2 | p1+p2-1<=t<=p1+p2+N-2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
495 ARe4 : {t,p1,p2 | p1+p2-1<=t<=p1+p2+N-2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  BIm4 : {t,p1,p2 | p1+p2-1<=t<=p1+p2+N-2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
497 BRe4 : {t,p1,p2 | p1+p2-1<=t<=p1+p2+N-2; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
  CImXctl1P4 : {t,p1,p2 | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of boolean;
499 CReXctl1P4 : {t,p1,p2 | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of boolean;
  CRe4 : {t,p1,p2 | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
501 CIm4 : {t,p1,p2 | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} of integer[U,16];
let
503 AImReg1Xloc[t,p1,p2] =
      case
505     { | p2-1<=t<=p2+N-2; p1=1; 3<=p2<=NB+1; NB+1<=N} : AIm1[t,p1,p2-1];
        { | 1<=t<=N; p1=1; p2=2; NB+1<=N; 2<=NB} : AIm2[t,p1,p2-1];
507     { | p1<=t<=p1+N-1; 2<=p1<=NB; p2=2; NB+1<=N} : AIm3[t,p1,p2-1];
        { | p1+p2-2<=t<=p1+p2+N-3; 2<=p1<=NB; 3<=p2<=NB+1; NB+1<=N} : AIm4[t,p1,p2-1];
509     esac;
  aImXMirr1[t,p1,p2] = aImXMirr1In[t,p1,p2];
511 AReReg2Xloc[t,p1,p2] =
      case
513     { | p2-1<=t<=p2+N-2; p1=1; 3<=p2<=NB+1; NB+1<=N} : ARe1[t,p1,p2-1];
        { | 1<=t<=N; p1=1; p2=2; NB+1<=N; 2<=NB} : ARe2[t,p1,p2-1];
515     { | p1<=t<=p1+N-1; 2<=p1<=NB; p2=2; NB+1<=N} : ARe3[t,p1,p2-1];
        { | p1+p2-2<=t<=p1+p2+N-3; 2<=p1<=NB; 3<=p2<=NB+1; NB+1<=N} : ARe4[t,p1,p2-1];
517     esac;
```

```
    aReXMirr1[t,p1,p2] = aReXMirr1In[t,p1,p2];
519 BImReg3Xloc[t,p1,p2] =
      case
521     { | p2<=t<=p2+N-1; p1=2; 2<=p2<=NB; NB+1<=N} : BIm1[t,p1-1,p2];
        { | 1<=t<=N; p1=2; p2=1; NB+1<=N; 2<=NB} : BIm2[t,p1-1,p2];
523     { | p1-1<=t<=p1+N-2; 3<=p1<=NB+1; p2=1; NB+1<=N} : BIm3[t,p1-1,p2];
        { | p1+p2-2<=t<=p1+p2+N-3; 3<=p1<=NB+1; 2<=p2<=NB; NB+1<=N} : BIm4[t,p1-1,p2];
525   esac;
    bImXMirr1[t,p1,p2] = bImXMirr1In[t,p1,p2];
527 BReReg4Xloc[t,p1,p2] =
      case
529     { | p2<=t<=p2+N-1; p1=2; 2<=p2<=NB; NB+1<=N} : BRe1[t,p1-1,p2];
        { | 1<=t<=N; p1=2; p2=1; NB+1<=N; 2<=NB} : BRe2[t,p1-1,p2];
531     { | p1-1<=t<=p1+N-2; 3<=p1<=NB+1; p2=1; NB+1<=N} : BRe3[t,p1-1,p2];
        { | p1+p2-2<=t<=p1+p2+N-3; 3<=p1<=NB+1; 2<=p2<=NB; NB+1<=N} : BRe4[t,p1-1,p2];
533   esac;
    bReXMirr1[t,p1,p2] = bReXMirr1In[t,p1,p2];
535 CImOut[t,p1,p2] =
      case
537     { | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} : CIm1[t,p1,p2];
        { | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} : CIm2[t,p1,p2];
539     { | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} : CIm3[t,p1,p2];
        { | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} : CIm4[t,p1,p2];
541   esac;
    CImXctl1PPReg8Xloc[t,p1,p2] =
543   case
        { | 2<=t<=N+2; p1=2; p2=1; NB+1<=N; 2<=NB} : CImXctl1PP2[t,p1-1,p2];
545     { | p1<=t<=p1+N; 3<=p1<=NB+1; p2=1; NB+1<=N} : CImXctl1PP3[t,p1-1,p2];
      esac;
547 CImXctl1PPXInitXIn[t,p1,p2] = CImXctl1PPXInit[t];
    CImXctl1PReg7Xloc[t,p1,p2] =
549   case
        { | p2<=t<=p2+N; p1=1; 3<=p2<=NB+1; NB+1<=N} : CImXctl1P1[t,p1,p2-1];
551     { | 2<=t<=N+2; p1=1; p2=2; NB+1<=N; 2<=NB} : CImXctl1P2[t,p1,p2-1];
        { | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=2; NB+1<=N} : CImXctl1P3[t,p1,p2-1];
553     { | p1+p2-1<=t<=p1+p2+N-1; 2<=p1<=NB; 3<=p2<=NB+1; NB+1<=N} : CImXctl1P4[t,p1,p2-1];
      esac;
555 CReOut[t,p1,p2] =
      case
557     { | p2+1<=t<=p2+N+1; p1=1; 2<=p2<=NB; NB+1<=N} : CRe1[t,p1,p2];
        { | 2<=t<=N+2; p1=1; p2=1; NB+1<=N; 2<=NB} : CRe2[t,p1,p2];
559     { | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=1; NB+1<=N} : CRe3[t,p1,p2];
        { | p1+p2<=t<=p1+p2+N; 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} : CRe4[t,p1,p2];
561   esac;
    CReXctl1PPReg12Xloc[t,p1,p2] =
563   case
        { | 2<=t<=N+2; p1=2; p2=1; NB+1<=N; 2<=NB} : CReXctl1PP2[t,p1-1,p2];
565     { | p1<=t<=p1+N; 3<=p1<=NB+1; p2=1; NB+1<=N} : CReXctl1PP3[t,p1-1,p2];
      esac;
567 CReXctl1PPXInitXIn[t,p1,p2] = CReXctl1PPXInit[t];
    CReXctl1PReg11Xloc[t,p1,p2] =
569   case
        { | p2<=t<=p2+N; p1=1; 3<=p2<=NB+1; NB+1<=N} : CReXctl1P1[t,p1,p2-1];
571     { | 2<=t<=N+2; p1=1; p2=2; NB+1<=N; 2<=NB} : CReXctl1P2[t,p1,p2-1];
        { | p1+1<=t<=p1+N+1; 2<=p1<=NB; p2=2; NB+1<=N} : CReXctl1P3[t,p1,p2-1];
573     { | p1+p2-1<=t<=p1+p2+N-1; 2<=p1<=NB; 3<=p2<=NB+1; NB+1<=N} : CReXctl1P4[t,p1,p2-1];
      esac;
575 use  ControlmatmultModule[N,NB] () returns (CImXctl1PPXInit, CReXctl1PPXInit) ;
    use {p1,p2 | p1=1; 2<=p2<=NB; NB+1<=N} cellmatmultModule1[p1,p2,N,NB] (bImXMirr1, bReXMirr1, AImReg1Xloc, AReReg2Xloc,
        CImXctl1PReg7Xloc, CReXctl1PReg11Xloc) returns (AIm1, ARe1, BIm1, BRe1, CImXctl1P1, CReXctl1P1, CRe1, CIm1) ;
577 use {p1,p2 | p1=1; p2=1; NB+1<=N; 2<=NB} cellmatmultModule2[p1,p2,N,NB] (bImXMirr1, bReXMirr1, aImXMirr1, aReXMirr1,
        CImXctl1PPXInitXIn, CReXctl1PPXInitXIn) returns (AIm2, ARe2, BIm2, BRe2, CImXctl1PP2, CImXctl1P2, CReXctl1PP2,
        CReXctl1P2, CRe2, CIm2) ;
    use {p1,p2 | 2<=p1<=NB; p2=1; NB+1<=N} cellmatmultModule3[p1,p2,N,NB] (aImXMirr1, aReXMirr1, BImReg3Xloc, BReReg4Xloc,
        CImXctl1PPReg8Xloc, CReXctl1PPReg12Xloc)
579     returns  (AIm3, ARe3, BIm3, BRe3, CImXctl1PP3, CImXctl1P3, CReXctl1PP3, CReXctl1P3, CRe3, CIm3) ;
    use {p1,p2 | 2<=p1<=NB; 2<=p2<=NB; NB+1<=N} cellmatmultModule4[p1,p2,N,NB] (AImReg1Xloc, AReReg2Xloc, BImReg3Xloc,
        BReReg4Xloc, CImXctl1PReg7Xloc, CReXctl1PReg11Xloc) returns (AIm4, ARe4, BIm4, BRe4, CImXctl1P4, CReXctl1P4, CRe4, CIm4
        ) ;
581 tel ;

583 system matmult :{N,NB | NB+1<=N; 2<=NB}
                (aRe : {i,j  | 1<=i<=NB; 1<=j<=N} of integer[U,16];
585              aIm : {i,j  | 1<=i<=NB; 1<=j<=N} of integer[U,16];
                 bRe : {i,j  | 1<=i<=N; 1<=j<=NB} of integer[U,16];
587              bIm : {i,j  | 1<=i<=N; 1<=j<=NB} of integer[U,16])
        returns (cRe : {i,j  | 1<=i<=NB; 1<=j<=NB} of integer[U,16];
589              cIm : {i,j  | 1<=i<=NB; 1<=j<=NB} of integer[U,16]);
  var
591   aImXMirr1 : {t,p1,p2 | p1<=t<=p1+N-1; 1<=p1<=NB; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
      aReXMirr1 : {t,p1,p2 | p1<=t<=p1+N-1; 1<=p1<=NB; p2=1; NB+1<=N; 2<=NB} of integer[U,16];
593   bImXMirr1 : {t,p1,p2 | p2<=t<=p2+N-1; p1=1; 1<=p2<=NB; NB+1<=N; 2<=NB} of integer[U,16];
      bReXMirr1 : {t,p1,p2 | p2<=t<=p2+N-1; p1=1; 1<=p2<=NB; NB+1<=N; 2<=NB} of integer[U,16];
595   CIm : {t,p1,p2 | p1+p2<=t<=p1+p2+N; 1<=p1<=NB; 1<=p2<=NB} of integer[U,16];
      CRe : {t,p1,p2 | p1+p2<=t<=p1+p2+N; 1<=p1<=NB; 1<=p2<=NB} of integer[U,16];
597 let
      bImXMirr1[t,p1,p2] = bIm[t-p2+1,p1+p2-1];
599   bReXMirr1[t,p1,p2] = bRe[t-p2+1,p1+p2-1];
```

```
    aImXMirr1[t,p1,p2] = aIm[p1,t-p1+1];
601 aReXMirr1[t,p1,p2] = aRe[p1,t-p1+1];
    cRe[i,j] = CRe[i+j+N,i,j];
603 cIm[i,j] = CIm[i+j+N,i,j];
    use matmultModule[N,NB] (aImXMirr1, aReXMirr1, bImXMirr1, bReXMirr1) returns (CIm, CRe) ;
605 tel;
```

7.2 Spark + WRaP-IT

In this section, we present different sources used in our Spark + WRaP-IT experimentation, in particular the sources that were used in the YUV to RGB format conversion acceleration from the `H263` CODEC.

7.2.1 Oprofile results

Here are the results obtained by running the Oprofile linux profiler. We used this information to choose parts of the code that are interesting to accelerate in hardware.

Statistics per run (over 30 runs) of the number of times each procedure was sampled OProfile:

App/Library	Procedure	Average	Std Dev	95% Confidence Interval
(no	symbols)	0.00	0.000	[0.000, 0.000]
_dl_determine_tlsoffset	1	0.00	0.000	[0.000, 0.000]
_dl_fixup	1	0.00	0.000	[0.000, 0.000]
_dl_lookup_symbol_x	1	0.00	0.000	[0.000, 0.000]
_dl_relocate_object	1	0.00	0.000	[0.000, 0.000]
_dl_runtime_resolve	1	0.00	0.000	[0.000, 0.000]
_dl_sort_fini	1	0.00	0.000	[0.000, 0.000]
anon	(tgid:17429	0.03	0.000	[0.033, 0.033]
anon	(tgid:2417	0.03	0.000	[0.033, 0.033]
anon	(tgid:26094	0.03	0.000	[0.033, 0.033]
anon	(tgid:9107	0.03	0.000	[0.033, 0.033]
close	1	0.00	0.000	[0.000, 0.000]
do_lookup_x	1	0.00	0.000	[0.000, 0.000]
getCBPY	1	0.00	0.000	[0.000, 0.000]
getMCBPC	1	0.00	0.000	[0.000, 0.000]
getbits1	1	0.00	0.000	[0.000, 0.000]
getheader	1	0.00	0.000	[0.000, 0.000]
ld-2.4.so	___fxstat64	0.07	0.000	[0.067, 0.067]
ld-2.4.so	__i686.get_pc_thunk.bx	0.07	0.000	[0.067, 0.067]
ld-2.4.so	__i686.get_pc_thunk.cx	0.03	0.000	[0.033, 0.033]
ld-2.4.so	__libc_memalign	0.03	0.000	[0.033, 0.033]
ld-2.4.so	_dl_allocate_tls_init	0.03	0.000	[0.033, 0.033]
ld-2.4.so	_dl_cache_libcmp	0.10	0.000	[0.100, 0.100]
ld-2.4.so	_dl_check_map_versions	0.10	0.000	[0.100, 0.100]
ld-2.4.so	_dl_elf_hash	0.53	0.000	[0.533, 0.533]
ld-2.4.so	_dl_fini	0.03	0.000	[0.033, 0.033]
ld-2.4.so	_dl_fixup	0.23	0.000	[0.233, 0.233]
ld-2.4.so	_dl_important_hwcaps	0.10	0.000	[0.100, 0.100]
ld-2.4.so	_dl_load_cache_lookup	0.10	0.000	[0.100, 0.100]
ld-2.4.so	_dl_lookup_symbol_x	0.40	0.000	[0.400, 0.400]
ld-2.4.so	_dl_map_object	0.13	0.000	[0.133, 0.133]
ld-2.4.so	_dl_map_object_deps	0.10	0.000	[0.100, 0.100]
ld-2.4.so	_dl_map_object_from_fd	0.23	0.000	[0.233, 0.233]
ld-2.4.so	_dl_name_match_p	0.13	0.000	[0.133, 0.133]
ld-2.4.so	_dl_new_object	0.03	0.000	[0.033, 0.033]
ld-2.4.so	_dl_relocate_object	1.77	1.000	[1.409, 2.125]
ld-2.4.so	_dl_start	0.03	0.000	[0.033, 0.033]
ld-2.4.so	do_lookup_x	4.27	1.414	[3.761, 4.773]
ld-2.4.so	fillin_rpath	0.03	0.000	[0.033, 0.033]
ld-2.4.so	init_tls	0.03	0.000	[0.033, 0.033]
ld-2.4.so	match_symbol	0.03	0.000	[0.033, 0.033]
ld-2.4.so	memcpy	0.03	0.000	[0.033, 0.033]
ld-2.4.so	mempcpy	0.13	0.000	[0.133, 0.133]
ld-2.4.so	memset	0.10	0.000	[0.100, 0.100]
ld-2.4.so	mmap	0.03	0.000	[0.033, 0.033]
ld-2.4.so	open	0.17	0.000	[0.167, 0.167]
ld-2.4.so	open_path	0.17	0.000	[0.167, 0.167]
ld-2.4.so	open_verify	0.10	0.000	[0.100, 0.100]
ld-2.4.so	read	0.03	0.000	[0.033, 0.033]
ld-2.4.so	strcmp	1.43	1.000	[1.075, 1.791]
ld-2.4.so	strsep	0.03	0.000	[0.033, 0.033]
libX11.so.6.2.0	(no symbols)	0.10	0.000	[0.100, 0.100]
libc-2.4.so	(no symbols)	12.73	3.742	[11.394, 14.072]
libm-2.4.so	(no symbols)	0.03	0.000	[0.033, 0.033]
main	1	0.00	0.000	[0.000, 0.000]

mempcpy	1	0.00	0.000	[0.000, 0.000]
memset	1	0.00	0.000	[0.000, 0.000]
no-vmlinux	(no symbols)	693.87	92.855	[660.639, 727.094]
open_path	1	0.00	0.000	[0.000, 0.000]
strcmp	1	0.00	0.000	[0.000, 0.000]
symbols)	1	0.00	0.000	[0.000, 0.000]
tmndec	.plt	0.30	0.000	[0.300, 0.300]
tmndec	addblock	86.63	12.845	[82.037, 91.230]
tmndec	clearblock	27.90	5.831	[25.813, 29.987]
tmndec	conv420to422	1102.90	132.514	[1055.480, 1150.320]
tmndec	conv422to444	1795.37	243.173	[1708.348, 1882.385]
tmndec	fillbfr	10.37	3.317	[9.180, 11.554]
tmndec	find_pmv	15.03	4.359	[13.474, 16.593]
tmndec	flushbits	34.90	8.246	[31.949, 37.851]
tmndec	getCBPY	1.93	1.732	[1.314, 2.553]
tmndec	getMCBPC	2.17	1.414	[1.661, 2.673]
tmndec	getMCBPCintra	0.30	0.000	[0.300, 0.300]
tmndec	getTMNMV	5.23	2.646	[4.287, 6.180]
tmndec	getbits	14.63	4.472	[13.033, 16.234]
tmndec	getbits1	1.20	1.000	[0.842, 1.558]
tmndec	getblock	50.47	9.899	[46.924, 54.009]
tmndec	getheader	0.10	0.000	[0.100, 0.100]
tmndec	getpicture	62.50	12.166	[58.147, 66.853]
tmndec	idct	284.33	40.596	[269.806, 298.860]
tmndec	init_idct	0.13	0.000	[0.133, 0.133]
tmndec	main	0.17	0.000	[0.167, 0.167]
tmndec	motion_decode	2.83	1.414	[2.327, 3.339]
tmndec	putbyte	623.50	83.540	[593.605, 653.395]
tmndec	recon_comp	313.07	52.125	[294.414, 331.719]
tmndec	reconstruct	8.33	3.464	[7.094, 9.573]
tmndec	showbits	62.93	11.269	[58.901, 66.966]
tmndec	store_ppm_tga	952.73	141.605	[902.061, 1003.406]
tmndec	storeframe	0.13	0.000	[0.133, 0.133]

Statistics per run (over 30 runs) of the percentage of execution time spent in each procedure, as measured by OProfile:

App/Library	Procedure	Average	Std Dev	95% Confidence Interval
(no	symbols)	0.00%	0.000%	[0.000%, 0.000%]
_dl_determine_tlsoffset	1	0.00%	0.000%	[0.000%, 0.000%]
_dl_fixup	1	0.00%	0.000%	[0.000%, 0.000%]
_dl_lookup_symbol_x	1	0.00%	0.000%	[0.000%, 0.000%]
_dl_relocate_object	1	0.00%	0.000%	[0.000%, 0.000%]
_dl_runtime_resolve	1	0.00%	0.000%	[0.000%, 0.000%]
_dl_sort_fini	1	0.00%	0.000%	[0.000%, 0.000%]
anon	(tgid:17429	0.00%	0.000%	[0.001%, 0.001%]
anon	(tgid:2417	0.00%	0.000%	[0.001%, 0.001%]
anon	(tgid:26094	0.00%	0.000%	[0.001%, 0.001%]
anon	(tgid:9107	0.00%	0.000%	[0.001%, 0.001%]
close	1	0.00%	0.000%	[0.000%, 0.000%]
do_lookup_x	1	0.00%	0.000%	[0.000%, 0.000%]
getCBPY	1	0.00%	0.000%	[0.000%, 0.000%]
getMCBPC	1	0.00%	0.000%	[0.000%, 0.000%]
getbits1	1	0.00%	0.000%	[0.000%, 0.000%]
getheader	1	0.00%	0.000%	[0.000%, 0.000%]
ld-2.4.so	___fxstat64	0.00%	0.000%	[0.001%, 0.001%]
ld-2.4.so	__i686.get_pc_thunk.bx	0.00%	0.000%	[0.001%, 0.001%]
ld-2.4.so	__i686.get_pc_thunk.cx	0.00%	0.000%	[0.001%, 0.001%]
ld-2.4.so	__libc_memalign	0.00%	0.000%	[0.001%, 0.001%]
ld-2.4.so	_dl_allocate_tls_init	0.00%	0.000%	[0.001%, 0.001%]
ld-2.4.so	_dl_cache_libcmp	0.00%	0.000%	[0.002%, 0.002%]
ld-2.4.so	_dl_check_map_versions	0.00%	0.000%	[0.002%, 0.002%]
ld-2.4.so	_dl_elf_hash	0.01%	0.000%	[0.009%, 0.009%]
ld-2.4.so	_dl_fini	0.00%	0.000%	[0.001%, 0.001%]
ld-2.4.so	_dl_fixup	0.00%	0.000%	[0.004%, 0.004%]
ld-2.4.so	_dl_important_hwcaps	0.00%	0.000%	[0.002%, 0.002%]
ld-2.4.so	_dl_load_cache_lookup	0.00%	0.000%	[0.002%, 0.002%]
ld-2.4.so	_dl_lookup_symbol_x	0.01%	0.000%	[0.006%, 0.006%]
ld-2.4.so	_dl_map_object	0.00%	0.000%	[0.002%, 0.002%]
ld-2.4.so	_dl_map_object_deps	0.00%	0.000%	[0.002%, 0.002%]
ld-2.4.so	_dl_map_object_from_fd	0.00%	0.000%	[0.004%, 0.004%]
ld-2.4.so	_dl_name_match_p	0.00%	0.000%	[0.002%, 0.002%]
ld-2.4.so	_dl_new_object	0.00%	0.000%	[0.001%, 0.001%]
ld-2.4.so	_dl_relocate_object	0.03%	0.016%	[0.023%, 0.034%]
ld-2.4.so	_dl_start	0.00%	0.000%	[0.001%, 0.001%]
ld-2.4.so	do_lookup_x	0.07%	0.023%	[0.061%, 0.077%]
ld-2.4.so	fillin_rpath	0.00%	0.000%	[0.001%, 0.001%]
ld-2.4.so	init_tls	0.00%	0.000%	[0.001%, 0.001%]
ld-2.4.so	match_symbol	0.00%	0.000%	[0.001%, 0.001%]
ld-2.4.so	memcpy	0.00%	0.000%	[0.001%, 0.001%]
ld-2.4.so	mempcpy	0.00%	0.000%	[0.002%, 0.002%]
ld-2.4.so	memset	0.00%	0.000%	[0.002%, 0.002%]
ld-2.4.so	mmap	0.00%	0.000%	[0.001%, 0.001%]
ld-2.4.so	open	0.00%	0.000%	[0.003%, 0.003%]
ld-2.4.so	open_path	0.00%	0.000%	[0.003%, 0.003%]
ld-2.4.so	open_verify	0.00%	0.000%	[0.002%, 0.002%]
ld-2.4.so	read	0.00%	0.000%	[0.001%, 0.001%]
ld-2.4.so	strcmp	0.02%	0.016%	[0.017%, 0.029%]
ld-2.4.so	strsep	0.00%	0.000%	[0.001%, 0.001%]

```
  libX11.so.6.2.0       (no symbols)                  0.00%          0.000%    [    0.002%,     0.002%]
  libc-2.4.so           (no symbols)                  0.21%          0.061%    [    0.184%,     0.228%]
  libm-2.4.so           (no symbols)                  0.00%          0.000%    [    0.001%,     0.001%]
  main                  1                             0.00%          0.000%    [    0.000%,     0.000%]
  mempcpy               1                             0.00%          0.000%    [    0.000%,     0.000%]
  memset                1                             0.00%          0.000%    [    0.000%,     0.000%]
  no-vmlinux            (no symbols)                 11.23%          1.503%    [   10.694%,    11.769%]
  open_path             1                             0.00%          0.000%    [    0.000%,     0.000%]
  strcmp                1                             0.00%          0.000%    [    0.000%,     0.000%]
  symbols)              1                             0.00%          0.000%    [    0.000%,     0.000%]
  tmndec                .plt                          0.00%          0.000%    [    0.005%,     0.005%]
  tmndec                addblock                      1.40%          0.208%    [    1.328%,     1.477%]
  tmndec                clearblock                    0.45%          0.094%    [    0.418%,     0.485%]
  tmndec                conv420to422                 17.85%          2.145%    [   17.085%,    18.620%]
  tmndec                conv422to444                 29.06%          3.936%    [   27.653%,    30.470%]
  tmndec                fillbfr                       0.17%          0.054%    [    0.149%,     0.187%]
  tmndec                find_pmv                      0.24%          0.071%    [    0.218%,     0.269%]
  tmndec                flushbits                     0.56%          0.133%    [    0.517%,     0.613%]
  tmndec                getCBPY                       0.03%          0.028%    [    0.021%,     0.041%]
  tmndec                getMCBPC                      0.04%          0.023%    [    0.027%,     0.043%]
  tmndec                getMCBPCintra                 0.00%          0.000%    [    0.005%,     0.005%]
  tmndec                getTMNMV                      0.08%          0.043%    [    0.069%,     0.100%]
  tmndec                getbits                       0.24%          0.072%    [    0.211%,     0.263%]
  tmndec                getbits1                      0.02%          0.016%    [    0.014%,     0.025%]
  tmndec                getblock                      0.82%          0.160%    [    0.760%,     0.874%]
  tmndec                getheader                     0.00%          0.000%    [    0.002%,     0.002%]
  tmndec                getpicture                    1.01%          0.197%    [    0.941%,     1.082%]
  tmndec                idct                          4.60%          0.657%    [    4.367%,     4.838%]
  tmndec                init_idct                     0.00%          0.000%    [    0.002%,     0.002%]
  tmndec                main                          0.00%          0.000%    [    0.003%,     0.003%]
  tmndec                motion_decode                 0.05%          0.023%    [    0.038%,     0.054%]
  tmndec                putbyte                      10.09%          1.352%    [    9.609%,    10.576%]
  tmndec                recon_comp                    5.07%          0.844%    [    4.766%,     5.370%]
  tmndec                reconstruct                   0.13%          0.056%    [    0.115%,     0.155%]
  tmndec                showbits                      1.02%          0.182%    [    0.953%,     1.084%]
  tmndec                store_ppm_tga                15.42%          2.292%    [   14.602%,    16.242%]
  tmndec                storeframe                    0.00%          0.000%    [    0.002%,     0.002%]

Total Samples per run (over 30 runs):

                                                    Average          Std Dev    95% Confidence Interval
                                                    -------          -------    -----------------------

  total_samples                                     6177.83          700.537    [  5927.149,   6428.517]
```

7.2.2 Original source code

Here is the original inlined source code used as the input to the synthesis and code transformation framework.

```
#include <stdio.h>
#define MAXVAL 1000
unsigned char srcY[MAXVAL*MAXVAL],srcU[MAXVAL*MAXVAL/4],srcV[MAXVAL*MAXVAL/4];
unsigned char u422[MAXVAL*MAXVAL/2], v422[MAXVAL*MAXVAL/2], u444[MAXVAL*MAXVAL], v444[MAXVAL*MAXVAL],rgb[
    MAXVAL*MAXVAL*3];
unsigned char clp[1024];
int convmat[8 * 4]=
{
   117504, 138453, 13954, 34903, /* no sequence_display_extension */
   117504, 138453, 13954, 34903, /* ITU-R Rec. 709 (1990) */
   104597, 132201, 25675, 53279, /* unspecified */
   104597, 132201, 25675, 53279, /* reserved */
   104448, 132798, 24759, 53109, /* FCC */
   104597, 132201, 25675, 53279, /* ITU-R Rec. 624-4 System B, G */
   104597, 132201, 25675, 53279, /* SMPTE 170M */
   117579, 136230, 16907, 35559  /* SMPTE 240M (1987) */
};

/*
 * decompress from yuv420 to RGB in order to store in TGA file(not implemented)
 */
void yuvrgb(int advance)
{
  int y, u, v, r, g, b;
  int crv, cbu, cgu, cgv;
  int py, pu, pv;
  int width,height;
  int incr = width;
  int w,h,w1,h1;
  int matrix_coefficients = 5;
  int i = 0;
  int j = 0;
  int jm30, jm31, jm20, jm21, jm10, jm11, jp10, jp11, jp20, jp21, jp30, jp31;
  int i20, i21, im30, im31, im20, im21, im10, im11, ip10, ip11,ip20, ip21,ip30, ip31;
  int j20,j21;
  int s;
```

```
    int src_index00 = 0;
37  int dst_index00 = 0;
    int src_index01 = 0;
39  int dst_index01 = 0;
    int src_index10 = 0;
41  int dst_index10 = 0;
    int src_index11 = 0;
43  int dst_index11 = 0;
    int src_indexRGB = 0;
45  int dst_indexRGB = 0;
    int wRGB,hRGB;
47  width = 1000;
    height = 1000;
49  w = width/2;
    h = height/2;
51  w1 = width/2;
    h1 = height;
53


55
    /* intra frame */
57  __URUK_BEG: for (i=0; i<w; i++) {
    __URUK_BEG1: for (j=0; j<h; j++) {
59  __URUK_IF01:
            j20 = j*2;
61  __URUK_IF02:if (j<3)
    __URUK_IF021: jm30 = 0; else
63  __URUK_IF022: jm30 = j-3;
    __URUK_IF03:if(j<2)
65  __URUK_IF031: jm20 = 0; else
    __URUK_IF032: jm20 = j-2;
67  __URUK_IF04:if(j<1)
    __URUK_IF041: jm10 = 0; else
69  __URUK_IF042: jm10 = j-1;
    __URUK_IF05:if(j<h-1)
71  __URUK_IF051: jp10 = j+1; else
    __URUK_IF052: jp10 = h-1;
73  __URUK_IF06:if(j<h-2)
    __URUK_IF061: jp20 = j+2; else
75  __URUK_IF062: jp20 = h-1;
    __URUK_IF07:if(j<h-3)
77  __URUK_IF071:jp30 = j+3; else
    __URUK_IF072:jp30 = h-1;
79
    __URUK_IF08:    u422[dst_index00 + w*j20] =    clp[(int)(  3*srcU[src_index00 + w*jm30]
81              -16*srcU[src_index00 + w*jm20]
                +67*srcU[src_index00 + w*jm10]
83              +227*srcU[src_index00 + w*j]
                -32*srcU[src_index00 + w*jp10]
85              +7*srcU[src_index00 + w*jp20]+128)/256];

87  __URUK_IF09:    u422[dst_index00 + w*(j20+1)] = clp[(int)(  3*srcU[src_index00 + w*jp30]
                -16*srcU[src_index00 + w*jp20]
89              +67*srcU[src_index00 + w*jp10]
                +227*srcU[src_index00 + w*j]
91              -32*srcU[src_index00 + w*jm10]
                +7*srcU[src_index00 + w*jm20]+128)/256];
93      }
    __URUK_LBL1: src_index00 = src_index00 + 1;
95  __URUK_LBL2: dst_index00 = dst_index00 + 1;
      }
97
    /* conv420to422(srcV,v422); */
99
    /* intra frame */
101   for (i=0; i<w; i++) {
    __URUK_BEG2: for (j=0; j<h; j++) {
103 __URUK_IF11:    j21 = j*2;
    __URUK_IF12:    if(j<3)
105 __URUK_IF121: jm31 = 0; else
    __URUK_IF122: jm31 = j-3;
107 __URUK_IF13:    if(j<2)
    __URUK_IF131: jm21 = 0;else
109 __URUK_IF132: jm21 = j-2;
    __URUK_IF14:    if(j<1)
111 __URUK_IF141: jm11 = 0; else
    __URUK_IF142: jm11 = j-1;
113 __URUK_IF15:    if(j<h-1)
    __URUK_IF151:jp11 = j+1; else
115 __URUK_IF152:jp11 = h-1;
    __URUK_IF16:    if(j<h-2)
117 __URUK_IF161:jp21 = j+2; else
    __URUK_IF162:jp21 = h-1;
119 __URUK_IF17:    if(j<h-3)
    __URUK_IF171:jp31 = j+3; else
121 __URUK_IF172:jp31 = h-1;

123 __URUK_IF18:    v422[dst_index01 + w*j21] =    clp[(int)(  3*srcV[src_index01 + w*jm31]
```

```
            −16∗srcV[src_index01+ w∗jm21]
125         +67∗srcV[src_index01 + w∗jm11]
            +227∗srcV[src_index01 + w∗j]
127         −32∗srcV[src_index01 + w∗jp11]
            +7∗srcV[src_index01 + w∗jp21]+128)/256];
129
    __URUK_IF19:    v422[dst_index01 + w∗(j21+1)] = clp[(int)( 3∗srcV[src_index01 + w∗jp31]
131         −16∗srcV[src_index01 + w∗jp21]
            +67∗srcV[src_index01 + w∗jp11]
133         +227∗srcV[src_index01 + w∗j]
            −32∗srcV[src_index01 + w∗jm11]
135         +7∗srcV[src_index01 + w∗jm21]+128)/256];
      }
137   src_index01 = src_index01 + 1;
      dst_index01 = dst_index01 + 1;
139 }

141

143 /∗ conv422to444(u422,u444); ∗/

145   for (j=0; j<h1; j++) {
        for (i=0; i<w1; i++) {
147
    __URUK_IIF01: i20 = i∗2;
149 __URUK_IIF02: if(i<3)
    __URUK_IIF021: im30 = 0; else
151 __URUK_IIF022: im30 = i−3;
    __URUK_IIF03: if(i<2)
153 __URUK_IIF031: im20 = 0; else
    __URUK_IIF032: im20 = i−2;
155 __URUK_IIF04: if(i<1)
    __URUK_IIF041: im10 = 0; else
157 __URUK_IIF042: im20 = i−1;
    __URUK_IIF05: if(i<w1−1)
159 __URUK_IIF051: ip10 = i+1; else
    __URUK_IIF052: ip10 = w1−1;
161 __URUK_IIF06: if(i<w1−2)
    __URUK_IIF061: ip20 = i+2; else
163 __URUK_IIF062: ip20 = w1−1;
    __URUK_IIF07: if(i<w1−3)
165 __URUK_IIF071: ip30 = i+3; else
    __URUK_IIF072: ip30 = w1−1;
167
    __URUK_IIF08:    u444[dst_index10 + i20] =   clp[(int)(  5∗u422[src_index10 + im30]
169             −21∗u422[src_index10 + im20]
                +70∗u422[src_index10 + im10]
171             +228∗u422[src_index10 + i]
                −37∗u422[src_index10 + ip10]
173             +11∗u422[src_index10 + ip20]+128)/256];

175 __URUK_IIF09:    u444[dst_index10 + i20+1] = clp[(int)(  5∗u422[src_index10 + ip30]
                −21∗u422[src_index10 + ip20]
177             +70∗u422[src_index10 + ip10]
                +228∗u422[src_index10 + i]
179             −37∗u422[src_index10 + im10]
                +11∗u422[src_index10 + im20]+128)/256];
181     }
    __URUK_END01: src_index10 = src_index10 + w1;
183 __URUK_END02: dst_index10 = dst_index10 + width;
      }
185
    /∗   conv422to444(v422,v444); ∗/
187   for (j=0; j<h1; j++) {
        for (i=0; i<w1; i++) {
189
    __URUK_IIF11: i21 = i∗2;
191 __URUK_IIF12: if(i<3)
    __URUK_IIF121: im31 = 0; else
193 __URUK_IIF122: im31 = i−3;
    __URUK_IIF13: if(i<2)
195 __URUK_IIF131: im21 = 0; else
    __URUK_IIF132: im21 = i−2;
197 __URUK_IIF14: if(i<1)
    __URUK_IIF141: im11 = 0; else
199 __URUK_IIF142: im21 = i−1;
    __URUK_IIF15: if(i<w1−1)
201 __URUK_IIF151: ip11 = i+1; else
    __URUK_IIF152: ip11 = w1−1;
203 __URUK_IIF16: if(i<w1−2)
    __URUK_IIF161: ip21 = i+2; else
205 __URUK_IIF162: ip21 = w1−1;
    __URUK_IIF17: if(i<w1−3)
207 __URUK_IIF171: ip31 = i+3; else
    __URUK_IIF172: ip31 = w1−1;
209
    __URUK_IIF18:    v444[dst_index11 + i21] =   clp[(int)(  5∗v422[src_index11 + im31]
211             −21∗v422[src_index11 + im21]
```

```
                +70*v422[src_index11 + im11]
213             +228*v422[src_index11 + i]
                -37*v422[src_index11 + ip11]
215             +11*v422[src_index11 + ip21]+128)/256];

217 __URUK_IIF19:    v444[dst_index11 + i21+1] = clp[(int)(  5*v422[src_index11 + ip31]
                -21*v422[src_index11 + ip21]
219             +70*v422[src_index11 + ip11]
                +228*v422[src_index11 + i]
221             -37*v422[src_index11 + im11]
                +11*v422[src_index11 + im21]+128)/256];
223     }
    __URUK_END11: src_index11 = src_index11 + w1;
225 __URUK_END12: dst_index11 = dst_index11 + width;
    }
227 __URUK_CMAT1: crv = convmat[matrix_coefficients * 4 + 0];
    __URUK_CMAT2: cbu = convmat[matrix_coefficients * 4 + 1];
229 __URUK_CMAT3: cgu = convmat[matrix_coefficients * 4 + 2];
    __URUK_CMAT4: cgv = convmat[matrix_coefficients * 4 + 3];
231   /* matrix coefficients */
    for (i=0; i<height; i++)
233         for (j=0; j< width; j++) {
    __URUK_RGB6: u = u444[src_indexRGB] - 128;
235 __URUK_RGB7: v = v444[src_indexRGB] - 128;
    __URUK_RGB8: y = 76309 * (srcY[src_indexRGB] - 16);
237 __URUK_RGB9: rgb[dst_indexRGB] = clp[(y + crv*v + 32768)>>16];
    __URUK_RGB10: rgb[dst_indexRGB+1] = clp[(y - cgu*u - cgv*v + 32768)>>16];
239 __URUK_RGB11: rgb[dst_indexRGB+2] = clp[(y + cbu*u + 32786)>>16];
    __URUK_RGB12: if(advance>0){
241 __URUK_RGB13: dst_indexRGB = dst_indexRGB + 3;
    __URUK_RGB14: src_indexRGB = src_indexRGB + 3;
243                 }
        }
245 }
```

7.2.3 WRaP-IT code transformations

Here is the source file that contains the list of transformations to apply on the YUV-to-RGB code.

```
1  shift(BEG   ,{[0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,-2]})
   shift(BEG2  ,{[0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,-2]})
3  fusion(enclose(IF08,2))
   fusion(enclose(IF08,1))
5  interchange(IF02,1)
   interchange(IF022,1)
7  interchange(IF03,1)
   interchange(IF032,1)
9  interchange(IF04,1)
   interchange(IF042,1)
11 interchange(IF05,1)
   interchange(IF052,1)
13 interchange(IF06,1)
   interchange(IF062,1)
15 interchange(IF07,1)
   interchange(IF072,1)
17 interchange(IF08,1)
   interchange(IF09,1)
19 interchange(IF12,1)
   interchange(IF122,1)
21 interchange(IF13,1)
   interchange(IF132,1)
23 interchange(IF14,1)
   interchange(IF142,1)
25 interchange(IF15,1)
   interchange(IF152,1)
27 interchange(IF16,1)
   interchange(IF162,1)
29 interchange(IF17,1)
   interchange(IF172,1)
31 interchange(IF18,1)
   interchange(IF19,1)
33 fusion(enclose(IIF09,2))
   fusion(enclose(IIF09))
35 stripmine(enclose(IIF08,2),2)
   interchange(IIF02,2)
37 interchange(IIF022,2)
   interchange(IIF03,2)
39 interchange(IIF032,2)
   interchange(IIF04,2)
41 interchange(IIF042,2)
   interchange(IIF05,2)
43 interchange(IIF052,2)
```

```
interchange(IIF06,2)
interchange(IIF062,2)
interchange(IIF07,2)
interchange(IIF072,2)
interchange(IIF08,2)
interchange(IIF09,2)
interchange(IIF12,2)
interchange(IIF122,2)
interchange(IIF13,2)
interchange(IIF132,2)
interchange(IIF14,2)
interchange(IIF142,2)
interchange(IIF15,2)
interchange(IIF152,2)
interchange(IIF16,2)
interchange(IIF162,2)
interchange(IIF17,2)
interchange(IIF172,2)
interchange(IIF18,2)
interchange(IIF19,2)
fusion(enclose(IF19,2))
fusion(enclose(IF19,1))
addContext(RGB6,'advance>0')
motion_block(CMAT4,BEG)
motion_block(CMAT3,CMAT4)
motion_block(CMAT2,CMAT3)
motion_block(CMAT1,CMAT2)
stripmine(enclose(RGB6,2),2)
stripmine(enclose(RGB6,1),2)
interchange(RGB6,2)
interchange(RGB7,2)
interchange(RGB8,2)
interchange(RGB9,2)
interchange(RGB10,2)
interchange(RGB11,2)
interchange(RGB12,2)
interchange(RGB14,2)
fusion(enclose(IIF19,3))
fusion(enclose(IIF19,2))
fusion(enclose(IIF19))
```

7.3 Altera C2H

7.3.1 Manually-transformed DMA double-buffering implementation

We give here the source code that implements the double-buffering block transfer of the DMA source code. It can be directly synthesized using the Altera `C2H` tool. The code was obtained by manual transformations of the original code.

```
#include "system.h"


#define BLOCK 100
#define MAX 1000

/* #pragma altera_accelerate enable_interrupt_for_function  accelerator */
#pragma altera_accelerate enable_interrupt_for_function buff0_acc
#pragma altera_accelerate enable_interrupt_for_function buff1_acc
#pragma altera_accelerate enable_interrupt_for_function st0_acc
//#pragma altera_accelerate  enable_interrupt_for_function  st1_acc
#pragma altera_accelerate enable_interrupt_for_function c01_acc

/* buff0_acc */
/* fifo  connection*/
#pragma altera_accelerate enable_flow_control_for_pointer buff0_acc/st1_buff0_read
#pragma altera_accelerate enable_flow_control_for_pointer buff0_acc/ buff0_buff1_write
#pragma altera_accelerate enable_flow_control_for_pointer buff0_acc/buff01_c01_write
#pragma altera_accelerate connect_variable buff0_acc/st1_buff0_read to st1_buff0/out
#pragma altera_accelerate connect_variable buff0_acc/buff0_buff1_write to buff0_buff1/in
#pragma altera_accelerate connect_variable buff0_acc/buff01_c01_write to buff01_c01/in

/* data connection */
#pragma altera_accelerate connect_variable buff0_acc/mem_pointer to altmemddr_0
#pragma altera_accelerate connect_variable buff0_acc/buff0 to buff0

/* buff1_acc */
/* fifo  connection*/
#pragma altera_accelerate enable_flow_control_for_pointer buff1_acc/ buff1_st0_write
#pragma altera_accelerate enable_flow_control_for_pointer buff1_acc/buff0_buff1_read
#pragma altera_accelerate enable_flow_control_for_pointer buff1_acc/buff01_c01_write
```

```
#pragma altera_accelerate connect_variable buff1_acc/buff1_st0_write to buff1_st0/in
#pragma altera_accelerate connect_variable buff1_acc/buff0_buff1_read to buff0_buff1/out
#pragma altera_accelerate connect_variable buff1_acc/buff01_c01_write to buff01_c01/in

/* data connection */
#pragma altera_accelerate connect_variable buff1_acc/mem_pointer to altmemddr_0
#pragma altera_accelerate connect_variable buff1_acc/buff1 to buff1

/* st0_acc */
/* fifo connection*/
#pragma altera_accelerate enable_flow_control_for_pointer st0_acc/buff1_st0_read
#pragma altera_accelerate enable_flow_control_for_pointer st0_acc/c01_st0_read
#pragma altera_accelerate enable_flow_control_for_pointer st0_acc/st0_st1_write
#pragma altera_accelerate connect_variable st0_acc/buff1_st0_read to buff1_st0/out
#pragma altera_accelerate connect_variable st0_acc/c01_st0_read to c01_st0/out
#pragma altera_accelerate connect_variable st0_acc/st0_st1_write to st0_st1/in

/* data connection */
#pragma altera_accelerate connect_variable st0_acc/mem_pointer to altmemddr_0
#pragma altera_accelerate connect_variable st0_acc/st0 to st0

/* st1_acc */
/* fifo connection*/
#pragma altera_accelerate enable_flow_control_for_pointer st1_acc/st0_st1_read
#pragma altera_accelerate enable_flow_control_for_pointer st1_acc/c01_st1_read
#pragma altera_accelerate enable_flow_control_for_pointer st1_acc/st1_buff0_write
#pragma altera_accelerate connect_variable st1_acc/st0_st1_read to st0_st1/out
#pragma altera_accelerate connect_variable st1_acc/c01_st1_read to c01_st1/out
#pragma altera_accelerate connect_variable st1_acc/st1_buff0_write to st1_buff0/in

/* data connection */
#pragma altera_accelerate connect_variable st1_acc/mem_pointer to altmemddr_0
#pragma altera_accelerate connect_variable st1_acc/st1 to st1

/* c01_acc */
/* fifo connection*/
#pragma altera_accelerate enable_flow_control_for_pointer c01_acc/c01_st0_write
#pragma altera_accelerate enable_flow_control_for_pointer c01_acc/c01_st1_write
#pragma altera_accelerate enable_flow_control_for_pointer c01_acc/buff01_c01_read
#pragma altera_accelerate connect_variable c01_acc/c01_st0_write to c01_st0/in
#pragma altera_accelerate connect_variable c01_acc/c01_st1_write to c01_st1/in
#pragma altera_accelerate connect_variable c01_acc/buff01_c01_read to buff01_c01/out

/* data connection */
#pragma altera_accelerate connect_variable c01_acc/buff0 to buff0
#pragma altera_accelerate connect_variable c01_acc/buff1 to buff1
#pragma altera_accelerate connect_variable c01_acc/st0 to st0
#pragma altera_accelerate connect_variable c01_acc/st1 to st1


int buff0_acc(int* __restrict__ mem_pointer)
{
    int i,j;
    int dummy_read = 0, tmp;
    /* fifo communication declaration section */
    volatile int* __restrict__ st1_buff0_read = (int *) ST1_BUFF0_OUT_BASE;
    volatile int* __restrict__ buff0_buff1_write = (int *) BUFF0_BUFF1_IN_BASE;
    volatile int* __restrict__ buff01_c01_write = (int *) BUFF01_C01_IN_BASE;
    /* local buffer declaration section */
    int* __restrict__ buff0 = (int *) BUFF0_BASE;

    for (i = 0; i < MAX; i = i + 2 * BLOCK)
    {

        dummy_read += *st1_buff0_read;
        for (j = 0, tmp = dummy_read; j < BLOCK; j++)
        {
            if (j == BLOCK-37) //35+1+1
            {
                tmp = 0;
                *buff0_buff1_write = 0xdeadbee0;
            }
            buff0[j] = mem_pointer[i + j];
        }
        *buff01_c01_write = tmp; /* buffer 0 is ready */
    }
    return dummy_read;
}

int buff1_acc(int* __restrict__ mem_pointer)
{
    int i,j;
    /* fifo communication declaration section */
    int dummy_read = 0, tmp;
    volatile int* __restrict__ buff1_st0_write = (int *) BUFF1_ST0_IN_BASE;
    volatile int* __restrict__ buff0_buff1_read = (int *) BUFF0_BUFF1_OUT_BASE;
    volatile int* __restrict__ buff01_c01_write = (int *) BUFF01_C01_IN_BASE;
    /* local buffer declaration section */
```

```
    int* __restrict__ buff1 = (int *) BUFF1_BASE;

    for (i = BLOCK; i < MAX; i = i + 2 * BLOCK)
    {
        dummy_read += *buff0_buff1_read;
        for (j = 0, tmp = dummy_read; j < BLOCK; j++)
        {
            if (j == BLOCK - 37)
            {
                tmp = 1;
                *buff1_st0_write = 0xdeadbee2;
            }
            buff1[j] = mem_pointer[i + j];
        }
        *buff01_c01_write = tmp; /* buffer 1 is ready */
    }
    return dummy_read;
}

int st0_acc(int* __restrict__ mem_pointer)
{
    int i, j;
    /* fifo communication declaration section */
    int dummy_read = 0, dummy_read1 = 0, tmp;
    volatile int* __restrict__ buff1_st0_read = (int *) BUFF1_ST0_OUT_BASE;
    volatile int* __restrict__ c01_st0_read = (int *) C01_ST0_OUT_BASE;
    volatile int* __restrict__ st0_st1_write = (int *) ST0_ST1_IN_BASE;
    /* local buffer declaration section */
    int* __restrict__ st0 = (int *) ST0_BASE;

    for (i = 0; i < MAX; i = i + 2 * BLOCK)
    {
        dummy_read += *buff1_st0_read;
        dummy_read1 += *c01_st0_read;
        for (j = 0, tmp = dummy_read + dummy_read1; j < BLOCK+1; j++)
        {
            if (j == BLOCK) *st0_st1_write = 0xdeadbee4;
            else mem_pointer[i + j] = st0[j];

        }

    }
    return dummy_read + dummy_read1;
}

int st1_acc(int* __restrict__ mem_pointer)
{
    int i, j;
    /* fifo communication declaration section */
    int dummy_read = 0, dummy_read1 = 0, tmp;
    volatile int* __restrict__ st0_st1_read = (int *) ST0_ST1_OUT_BASE;
    volatile int* __restrict__ c01_st1_read = (int *) C01_ST1_OUT_BASE;
    volatile int* __restrict__ st1_buff0_write = (int *) ST1_BUFF0_IN_BASE;
    /* local buffer declaration section */
    int* __restrict__ st1 = (int *) ST1_BASE;

    /* Send the token to buff0 in order to start the execution */
    *st1_buff0_write = 0xdeadbee5;

    for (i = BLOCK; i < MAX; i = i + 2 * BLOCK)
    {
        dummy_read += *st0_st1_read;
        dummy_read1 += *c01_st1_read;
        for (j = 0, tmp = dummy_read + dummy_read1; j < BLOCK + 1; j++)
        {
            if (j == BLOCK) *st1_buff0_write = 0xdeadbee6;
            else mem_pointer[i + j] = st1[j];
        }
    }
    return dummy_read + dummy_read1;
}

int c01_acc()
{
    int i,j;
    /* fifo communication declaration section */
    int dummy_read = 0, tmp;
    volatile int* __restrict__ c01_st0_write = (int *) C01_ST0_IN_BASE;
    volatile int* __restrict__ c01_st1_write = (int *) C01_ST1_IN_BASE;
    volatile int* __restrict__ buff01_c01_read = (int *) BUFF01_C01_OUT_BASE;
    /* local buffer declaration section */
    int* __restrict__ buff0 = (int *) BUFF0_BASE;
    int* __restrict__ buff1 = (int *) BUFF1_BASE;
    int* __restrict__ st0 = (int *) ST0_BASE;
    int* __restrict__ st1 = (int *) ST1_BASE;

```

```
    for (i = 0; i < MAX; i = i + BLOCK)
    {
        dummy_read = *buff01_c01_read;
        if (dummy_read == 0)
        {
            for (j = 0, tmp = dummy_read +1; j < BLOCK; j++)
            {
                st0[j] = buff0[j];
            }

            *c01_st0_write = st0[j];
        }
        if (dummy_read == 1)
        {
            for (j = 0, tmp = dummy_read + 2; j < BLOCK; j++)
            {
                st1[j] = buff1[j];
            }

            *c01_st1_write = st1[j];
        }

    }

    return dummy_read;
}

int accelerator (int* __restrict__ mem_pointer0, int* __restrict__ mem_pointer1)
{
    buff0_acc(mem_pointer0);
    buff1_acc(mem_pointer0);
    c01_acc();
    st0_acc(mem_pointer1);
    st1_acc(mem_pointer1);
    return 0;
}
```

7.3.2 Manually-transformed vector-sum double-buffering implementation

We give here the source code that implements the double-buffering block transfer of the vector sum example. It can be directly synthesized using the Altera `C2H` tool. The code was obtained by manual transformations of the original code.

```
#include "system.h"
#include "global.h"

#define BLOCK 100
#define MAX 1000

/* #pragma altera_accelerate enable_interrupt_for_function accelerator */
#pragma altera_accelerate enable_interrupt_for_function buff0_acc
#pragma altera_accelerate enable_interrupt_for_function buff1_acc
#pragma altera_accelerate enable_interrupt_for_function st0_acc
//#pragma altera_accelerate enable_interrupt_for_function st1_acc
#pragma altera_accelerate enable_interrupt_for_function c01_acc

/* buff0_acc */
/* fifo connection*/
#pragma altera_accelerate enable_flow_control_for_pointer buff0_acc/st1_buff0_read
#pragma altera_accelerate enable_flow_control_for_pointer buff0_acc/ buff0_buff1_write
#pragma altera_accelerate enable_flow_control_for_pointer buff0_acc/buff01_c01_write
#pragma altera_accelerate connect_variable buff0_acc/st1_buff0_read to st1_buff0/out
#pragma altera_accelerate connect_variable buff0_acc/buff0_buff1_write to buff0_buff1/in
#pragma altera_accelerate connect_variable buff0_acc/buff01_c01_write to buff01_c01/in

/* data connection */
//#pragma altera_accelerate connect_variable buff0_acc/ext_memp to altmemddr_0
#pragma altera_accelerate connect_variable buff0_acc/mem_pointers to altmemddr_0
#pragma altera_accelerate connect_variable buff0_acc/itrs_arg to altmemddr_0
#pragma altera_accelerate connect_variable buff0_acc/ememp to altmemddr_0
#pragma altera_accelerate connect_variable buff0_acc/mem_pointersp to altmemddr_0
#pragma altera_accelerate connect_variable buff0_acc/itrs_argitr to altmemddr_0
#pragma altera_accelerate connect_variable buff0_acc/itrs_argblks to altmemddr_0

/* local data connection */
#pragma altera_accelerate connect_variable buff0_acc/buff0_0 to buff0_0
#pragma altera_accelerate connect_variable buff0_acc/buff0_1 to buff0_1
#pragma altera_accelerate connect_variable buff0_acc/lmemp to buff0_0
#pragma altera_accelerate connect_variable buff0_acc/lmemp to buff0_1

/* buff1_acc */
/* fifo connection*/
```

```
#pragma altera_accelerate enable_flow_control_for_pointer buff1_acc/buff1_st0_write
#pragma altera_accelerate enable_flow_control_for_pointer buff1_acc/buff0_buff1_read
#pragma altera_accelerate enable_flow_control_for_pointer buff1_acc/buff01_c01_write
#pragma altera_accelerate connect_variable buff1_acc/buff1_st0_write to buff1_st0/in
#pragma altera_accelerate connect_variable buff1_acc/buff0_buff1_read to buff0_buff1/out
#pragma altera_accelerate connect_variable buff1_acc/buff01_c01_write to buff01_c01/in

/* data connection */
#pragma altera_accelerate connect_variable buff1_acc/ext_memp to altmemddr_0
#pragma altera_accelerate connect_variable buff1_acc/mem_pointers to altmemddr_0
#pragma altera_accelerate connect_variable buff1_acc/itrs_arg to altmemddr_0
#pragma altera_accelerate connect_variable buff1_acc/ememp to altmemddr_0
#pragma altera_accelerate connect_variable buff1_acc/mem_pointersp to altmemddr_0
#pragma altera_accelerate connect_variable buff1_acc/itrs_argitr to altmemddr_0
#pragma altera_accelerate connect_variable buff1_acc/itrs_argblks to altmemddr_0

/* local data connection */
#pragma altera_accelerate connect_variable buff1_acc/buff1_0 to buff1_0
#pragma altera_accelerate connect_variable buff1_acc/buff1_1 to buff1_1
#pragma altera_accelerate connect_variable buff1_acc/lmemp to buff1_0
#pragma altera_accelerate connect_variable buff1_acc/lmemp to buff1_1

/* st0_acc */
/* fifo connection*/
#pragma altera_accelerate enable_flow_control_for_pointer st0_acc/buff1_st0_read
#pragma altera_accelerate enable_flow_control_for_pointer st0_acc/c01_st0_read
#pragma altera_accelerate enable_flow_control_for_pointer st0_acc/st0_st1_write
#pragma altera_accelerate connect_variable st0_acc/buff1_st0_read to buff1_st0/out
#pragma altera_accelerate connect_variable st0_acc/c01_st0_read to c01_st0/out
#pragma altera_accelerate connect_variable st0_acc/st0_st1_write to st0_st1/in

/* data connection */
#pragma altera_accelerate connect_variable st0_acc/ext_memp to altmemddr_0
#pragma altera_accelerate connect_variable st0_acc/mem_pointers to altmemddr_0
#pragma altera_accelerate connect_variable st0_acc/itrs_arg to altmemddr_0
#pragma altera_accelerate connect_variable st0_acc/st0 to st0

/* st1_acc */
/* fifo connection*/
#pragma altera_accelerate enable_flow_control_for_pointer st1_acc/st0_st1_read
#pragma altera_accelerate enable_flow_control_for_pointer st1_acc/c01_st1_read
#pragma altera_accelerate enable_flow_control_for_pointer st1_acc/st1_buff0_write
#pragma altera_accelerate connect_variable st1_acc/st0_st1_read to st0_st1/out
#pragma altera_accelerate connect_variable st1_acc/c01_st1_read to c01_st1/out
#pragma altera_accelerate connect_variable st1_acc/st1_buff0_write to st1_buff0/in

/* data connection */
#pragma altera_accelerate connect_variable st1_acc/ext_memp to altmemddr_0
#pragma altera_accelerate connect_variable st1_acc/mem_pointers to altmemddr_0
#pragma altera_accelerate connect_variable st1_acc/itrs_arg to altmemddr_0
#pragma altera_accelerate connect_variable st1_acc/st1 to st1

/* c01_acc */
/* fifo connection*/
#pragma altera_accelerate enable_flow_control_for_pointer c01_acc/c01_st0_write
#pragma altera_accelerate enable_flow_control_for_pointer c01_acc/c01_st1_write
#pragma altera_accelerate enable_flow_control_for_pointer c01_acc/buff01_c01_read
#pragma altera_accelerate connect_variable c01_acc/c01_st0_write to c01_st0/in
#pragma altera_accelerate connect_variable c01_acc/c01_st1_write to c01_st1/in
#pragma altera_accelerate connect_variable c01_acc/buff01_c01_read to buff01_c01/out

/* data connection */
#pragma altera_accelerate connect_variable c01_acc/buff0_0 to buff0_0
#pragma altera_accelerate connect_variable c01_acc/buff0_1 to buff0_1
#pragma altera_accelerate connect_variable c01_acc/buff1_0 to buff1_0
#pragma altera_accelerate connect_variable c01_acc/buff1_1 to buff1_1
#pragma altera_accelerate connect_variable c01_acc/st0 to st0
#pragma altera_accelerate connect_variable c01_acc/st1 to st1
#pragma altera_accelerate connect_variable c01_acc/itrs_arg to altmemddr_0


int buff0_acc(memory_pointers* __restrict__ mem_pointers, iterators* __restrict__ itrs_arg )
{
    int i,j,ii,k;
    int dummy_read = 0, tmp;
    /* fifo communication declaration section */
    volatile int* __restrict__ st1_buff0_read = (int *) ST1_BUFF0_OUT_BASE;
    volatile int* __restrict__ buff0_buff1_write = (int *) BUFF0_BUFF1_IN_BASE;
    volatile int* __restrict__ buff01_c01_write = (int *) BUFF01_C01_IN_BASE;
    /* local buffer declaration section */
    int* __restrict__ lbuffs[10];
    int* __restrict__ buff0_0 = (int *) BUFF0_0_BASE;
    int* __restrict__ buff0_1 = (int *) BUFF0_1_BASE;

    int* __restrict__ ememp;
    int* __restrict__ lmemp;
    int offset_ext_memp, offset_bufflocal;
    int* ext_memp[10];
```

```
    int* __restrict__ *mem_pointersp;
129 int* __restrict__ itrs_argitr ;
    int* __restrict__ itrs_argblks ;
131 int nmp;
    int itrs [10];
133 int blks [10];
    int nb_iterators , nb_blocks;
135 int itr0 , blk0;
    int* __restrict__ lbuff ;
137 int* __restrict__ extmemp;
    int jn , ij ;
139 lbuffs [0] = buff0_0;
    lbuffs [1] = buff0_1;
141
    /* copy memory pointers to local memory */
143 nmp = mem_pointers->nmp;
    mem_pointersp = mem_pointers->p;
145 itrs_argitr = itrs_arg->itr;
    itrs_argblks = itrs_arg->blks;
147 for (i = 0; i < nmp; i++) ext_memp[i] = mem_pointersp[i];
    /* copy iterators to local memory */
149 nb_iterators = itrs_arg->nb_iterators;
    nb_blocks = itrs_arg->nb_blocks;
151 for (i = 0; i < nb_iterators; i++) itrs[i] = itrs_argitr [i];
    for (i = 0; i < nb_blocks; i++) blks[i] = itrs_argblks [i];
153 itr0 = itrs [0];
    blk0 = blks [0];
155 lbuff = lbuffs [0];
    extmemp = ext_memp[0];
157 for (ii = 0; ii < itr0; ii = ii + 2 * blk0)
    {
159     dummy_read += *st1_buff0_read;
        j = 0; i = 0; jn = 0; ij = 0;
161     for (k = 0, tmp = dummy_read; k < blk0 * 2; k++) //*2 for a and b
        {
163         offset_bufflocal = i;
            offset_ext_memp = ii + i;
165
            if (i < blk0 - 1) {i++; }
167         else i = 0;

169         if (ij == blk0) {j++; ij = 0;}
            else ij++;
171


173
            /* sync code */
175         if (k == blk0 * 2 - 1) //35+1+1
            {
177             tmp = 0;
                *buff0_buff1_write = 0xdeadbee2;
179         }
            /* transfer code */
181
            lmemp = (int *) lbuffs[j] + offset_bufflocal ;
183
        }
185     *buff01_c01_write = tmp; /* buffer 0 is ready */
    }
187 return dummy_read;
}
189
int buff1_acc(memory_pointers* __restrict__ mem_pointers, iterators* __restrict__ itrs_arg )
191 {
    int i , j , ii , k;
193 int dummy_read = 0, tmp;
    /* fifo communication declaration section */
195 volatile int* __restrict__ buff1_st0_write = (int *) BUFF1_ST0_IN_BASE;
    volatile int* __restrict__ buff0_buff1_read = (int *) BUFF0_BUFF1_OUT_BASE;
197 volatile int* __restrict__ buff01_c01_write = (int *) BUFF01_C01_IN_BASE;
    /* local buffer declaration section */
199 int* __restrict__ lbuffs [10];
    int* __restrict__ buff1_0 = (int *) BUFF1_0_BASE;
201 int* __restrict__ buff1_1 = (int *) BUFF1_1_BASE;

203 int* __restrict__ ememp;
    int* __restrict__ lmemp;
205 int offset_ext_memp, offset_bufflocal ;

207 int* __restrict__ ext_memp[10];
    int** __restrict__ mem_pointersp;
209 int* __restrict__ itrs_argitr ;
    int* __restrict__ itrs_argblks ;
211 int nmp;
    int itrs [10];
213 int blks [10];
    int nb_iterators , nb_blocks;
215 int* __restrict__ base_addr_ext;
```

```
    int* __restrict__ base_addr_loc;
217 int jn, ij;
    lbuffs[0] = buff1_0;
219 lbuffs[1] = buff1_1;

221 /* copy memory pointers to local memory */
    nmp = mem_pointers->nmp;
223 mem_pointersp = mem_pointers->p;
    itrs_argitr = itrs_arg->itr;
225 itrs_argblks = itrs_arg->blks;
    for (i = 0; i < nmp; i++) ext_memp[i] = mem_pointersp[i];
227 /* copy iterators to local memory */
    nb_iterators = itrs_arg->nb_iterators;
229 nb_blocks = itrs_arg->nb_blocks;
    for (i = 0; i < nb_iterators; i++) itrs[i] = itrs_argitr[i];
231 for (i = 0; i < nb_blocks; i++) blks[i] = itrs_argblks[i];

233 for (ii = blks[0]; ii < itrs[0] + blks[0]; ii = ii + 2 * blks[0])
    {
235     dummy_read += *buff0_buff1_read;
        j = 0; i = 0; ij = 0;
237     for (k = 0, tmp = dummy_read; k < blks[0] * 2; k++) //*2 for a and b
        {
239         offset_bufflocal = i;
            offset_ext_memp = ii + i;
241         if (i < blks[0] - 1) {i++; }
            else i = 0;
243         if (ij == blks[0]) {j++; ij = 0;}
            else ij++;
245         /* sync code */
            if (k == blks[0] - 1) //35+1+1
247         {
                tmp = 1;
249             *buff1_st0_write = 0xdeadbee2;
            }
251         /* transfer code */
            ememp = (int *) ext_memp[j] + offset_ext_memp;
253         lmemp = (int *) lbuffs[j] + offset_bufflocal;
            *lmemp = *ememp;
255     }
        *buff01_c01_write = tmp; /* buffer 1 is ready */
257 }
    return dummy_read;
259 }

261 int st0_acc (memory_pointers* __restrict__ mem_pointers, iterators* __restrict__ itrs_arg)
    {
263 int i, j;
    /* fifo communication declaration section */
265 int dummy_read = 0, dummy_read1 = 0, tmp;
    volatile int* __restrict__ buff1_st0_read = (int *) BUFF1_ST0_OUT_BASE;
267 volatile int* __restrict__ c01_st0_read = (int *) C01_ST0_OUT_BASE;
    volatile int* __restrict__ st0_st1_write = (int *) ST0_ST1_IN_BASE;
269 /* local buffer declaration section */
    int* __restrict__ st0 = (int *) ST0_BASE;
271 int nmp;
    int itrs[10];
273 int blks[10];
    int* __restrict__ ext_memp;
275 ext_memp = mem_pointers->p[2];
    itrs[0] = itrs_arg->itr[0];
277 blks[0] = itrs_arg->blks[0];

279
    for (i = 0; i < itrs[0]; i = i + 2 * blks[0])
281 {
        dummy_read += *buff1_st0_read;
283     dummy_read1 += *c01_st0_read;
        for (j = 0, tmp = dummy_read + dummy_read1; j < blks[0]; j++)
285     {
            if (j == blks[0] - 1) *st0_st1_write = 0xdeadbee4;
287         ext_memp[i + j] = st0[j];

289     }

291 }
    return dummy_read + dummy_read1;
293 }

295
    int st1_acc (memory_pointers* __restrict__ mem_pointers, iterators* __restrict__ itrs_arg)
297 {
    int i, j;
299 /* fifo communication declaration section */
    int dummy_read = 0, dummy_read1 = 0, tmp;
301 volatile int* __restrict__ st0_st1_read = (int *) ST0_ST1_OUT_BASE;
    volatile int* __restrict__ c01_st1_read = (int *) C01_ST1_OUT_BASE;
303 volatile int* __restrict__ st1_buff0_write = (int *) ST1_BUFF0_IN_BASE;
```

```
    /* local buffer declaration section */
305 int* __restrict__ st1 = (int *) ST1_BASE;
    int nmp;
307 int itrs[10];
    int blks[10];
309 int* ext_memp;
    ext_memp = mem_pointers->p[2];
311 itrs[0] = itrs_arg->itr[0];
    blks[0] = itrs_arg->blks[0];
313
    /* Send the token to buff0 in order to start the execution */
315 *st1_buff0_write = 0xdeadbee5;

317 for (i = blks[0]; i < itrs[0] + blks[0]; i = i + 2 * blks[0])
    {
319     dummy_read += *st0_st1_read;
        dummy_read1 += *c01_st1_read;
321     for (j = 0, tmp = dummy_read + dummy_read1; j < blks[0]; j++)
        {
323         if (j == blks[0] - 1) *st1_buff0_write = 0xdeadbee6;
            ext_memp[i + j] = st1[j];
325
        }
327
    }
329 return dummy_read + dummy_read1;
}
331

333 int c01_acc(iterators* __restrict__ itrs_arg)
{
335 int i,j;
    /* fifo communication declaration section */
337 int dummy_read = 0, tmp;
    volatile int* __restrict__ c01_st0_write = (int *) C01_ST0_IN_BASE;
339 volatile int* __restrict__ c01_st1_write = (int *) C01_ST1_IN_BASE;
    volatile int* __restrict__ buff01_c01_read = (int *) BUFF01_C01_OUT_BASE;
341 /* local buffer declaration section */
    int* __restrict__ buff0_0 = (int *) BUFF0_0_BASE;
343 int* __restrict__ buff0_1 = (int *) BUFF0_1_BASE;
    int* __restrict__ buff1_0 = (int *) BUFF1_0_BASE;
345 int* __restrict__ buff1_1 = (int *) BUFF1_1_BASE;
    int* __restrict__ st0 = (int *) ST0_BASE;
347 int* __restrict__ st1 = (int *) ST1_BASE;

349 int itrs[10];
    int blks[10];
351 int* ext_memp;
    itrs[0] = itrs_arg->itr[0];
353 blks[0] = itrs_arg->blks[0];

355 for (i = 0; i < itrs[0]; i = i + blks[0])
    {
357     dummy_read = *buff01_c01_read;
        if (dummy_read == 0)
359     {
            for (j = 0, tmp = dummy_read +1; j < blks[0]; j++)
361         {
                st0[j] = buff0_0[j] + buff0_1[j];
363         }

365         *c01_st0_write = st0[j];
        }
367     if (dummy_read == 1)
        {
369         for (j = 0, tmp = dummy_read + 2; j < blks[0]; j++)
            {
371             st1[j] = buff1_0[j] + buff1_1[j];
            }
373
            *c01_st1_write = st1[j];
375     }

377 }

379 return dummy_read;
}
```

7.3.3 Manually-transformed matrix-multiply double-buffering implementation

Transformation steps

We give here the source code that implements the double-buffering block transfer for the matrix-matrix multiplication. It can be directly synthesized using the Altera `C2H` tool. The code was obtained by manual transformations of the original code. The array size and the block size are sent by parameters to the code, i.e., are not hard-coded.

```
1
  //=========================================================================================
3 for (i = 0; i < n0; i++)
    for (j = 0; j < m1; j++)
5     {
        *(c + (i * m1 + j)) = 0;
7       for (k = 0; k < m0; k++)
          {
9           *(c + (i * m1 + j)) += *(a + (i * m0 + k)) * *(b + (k * m1 + j));
          }
11    }

13 //=========================================================================================
   //strip-mine on i,j,k;
15 //=========================================================================================
   for (ii = 0; ii < n0; ii+=block_ii)
17   for (i = ii; i < min(ii + block_ii,n0); i++)
       for (jj = 0; jj < m1; jj+=block_jj)
19       for (j = jj; j < min(jj + block_jj,m1); j++)
           {
21           *(c + (i * m1 + j)) = 0;
             for (kk = 0; kk < m0; kk+=block_kk)
23             for (k = kk; k < min(kk + block_kk,m0); k++)
                 {
25                 *(c + (i * m1 + j)) += *(a + (i * m0 + k)) * *(b + (k * m1 + j));
                 }
27         }

29 //=========================================================================================
   // interchange (i,jj) (kk,j) (kk,jj)
31 //=========================================================================================
   for (ii = 0; ii < n0; ii+=block_ii)
33   for (jj = 0; jj < m1; jj+=block_jj)
       for (kk = 0; kk < m0; kk+=block_kk)
35       for (i = ii; i < min(ii + block_ii,n0); i++)
           for (j = jj; j < min(jj + block_jj,m1); j++)
37           {
               if (kk == 0) *(c + (i * m1 + j)) = 0;
39
               for (k = kk; k < min(kk + block_kk,m0); k++)
41               {
                   *(c + (i * m1 + j)) += *(a + (i * m0 + k)) * *(b + (k * m1 + j));
43               }
             }
45
   //=========================================================================================
47 // insert temporary variables c_tmp, a_tmp, b_tmp
   //=========================================================================================
49 for (ii = 0; ii < n0; ii+=block_ii)
     for (jj = 0; jj < m1; jj+=block_jj)
51     for (kk = 0; kk < m0; kk+=block_kk)
         for (i = ii; i < min(ii + block_ii,n0); i++)
53         for (j = jj; j < min(jj + block_jj,m1); j++)
             {
55             *(c_tmp + (i * block_jj  + j)) = *(c + (i * m1 + j));
               if (kk == 0) *(c_tmp + (i * block_jj + j)) = 0;
57
               for (k = kk; k < min(kk + block_kk,m0); k++)
59               {
                   *(a_tmp + (i * block_kk + k)) = *(a + (i * m0 + k));
61                 *(b_tmp + (k * block_jj + j)) = *(b + (k * m1 + j));
                   *(c_tmp + (i * block_jj  + j)) += *(a_tmp + (i * block_kk + k)) * *(b_tmp + (k * block_jj + j));
63               }
               if ( kk >= m0) *(c + (i * m1 + j)) = *(c_tmp + (i * block_jj + j));
65           }

67 //=========================================================================================
   // adjust domains to the c_tmp sizes and addresses (rename variables i' = i - ii,
69 // i = i' + ii and shift tmp address space by ii, remove WW copy)
   //=========================================================================================
71 for (ii = 0; ii < n0; ii+=block_ii)
     for (jj = 0; jj < m1; jj+=block_jj)
73     for (kk = 0; kk < m0; kk+=block_kk)
         for (i = 0; i < min(block_ii,n0 - ii); i++)
```

```
75          for (j = 0; j < min(block_jj,m1 - jj); j++)
              {
77              if (kk == 0) *(c_tmp + (i * block_jj + j)) = 0;
                for (k = 0; k < min(kk + block_kk, m0 - kk); k++)
79                {
                    *(a_tmp + (i * block_kk + k)) = *(a + ((i + ii) * m0 + k + kk));
81                  *(b_tmp + (k * block_jj + j)) = *(b + ((k + kk) * m1 + j + jj));
                    *(c_tmp + (i * block_jj + j)) += *(a_tmp + (i * block_kk + k)) * *(b_tmp + (k * block_jj + j));
83                }
                if ( kk >= m0) *(c + ((i + ii) * m1 + j + jj)) = *(c_tmp + (i * block_jj + j));
85            }

87  //=============================================================================================
    //loop distribution on _tmp fetch and computation
89  //=============================================================================================

91  for (ii = 0; ii < n0; ii+= block_ii)
      for (jj = 0; jj < m1; jj+=block_jj)
93      for (kk = 0; kk < m0; kk+=block_kk)
          {
95          if (kk == 0)
              for (i = 0; i < min(block_ii, n0 - ii); i++)
97              for (j = 0; j < min(block_jj, m1 - jj); jj++)
                  *(c_tmp + (i * block_jj + j)) = 0;
99
            for (i = 0; i < min(block_ii, n0 - ii); i++)
101           for (k = 0; k < min(block_kk, m0 - kk); k++)
                *(a_tmp + (i * block_kk + k)) = *(a + ((i + ii) * m0 + k + kk));
103         for (j = 0; j < min(block_jj, m1 - jj); jj++)
              for (k = 0; k < min(block_kk, m0 - kk); k++)
105             *(b_tmp + (k * block_jj + j)) = *(b + ((k + kk) * m1 + j + jj));
            for (i = 0; i < min(block_ii, n0 - ii); i++)
107           for (j = 0; j < min(block_jj, m1 - jj); jj++)
                for (k = 0; k < min(block_kk, m0 - kk); k++)
109               *(c_tmp + (i * block_jj + j)) += *(a_tmp + (i * block_kk + k)) * *(b_tmp + (k * block_jj + j));
            if (kk >= m0)
111           for (i = 0; i < min(block_ii, n0 - ii); i++)
                for (j = 0; j < min(block_jj, m1 - jj); jj++)
113               *(c + ((i + ii) * m1 + j + jj)) = *(c_tmp + (i * block_jj + j));
        }
115 }
    //=============================================================================================
117 //unroll by 2 the parallel loop in this case ii, and fuse ii, jj, kk loops
    //=============================================================================================
119
    for (ii = 0; ii < n0; ii+= 2 * block_ii)
121   for (jj = 0; jj < m1; jj+=block_jj)
        for (kk = 0; kk < m0; kk+=block_kk)
123       {
            if (kk == 0)
125           for (i = 0; i < min(block_ii, n0 - ii); i++)
                for (j = 0; j < min(block_jj, m1 - jj); jj++)
127               *(c_tmp + (i * block_jj + j)) = 0;
            for (i = 0; i < min(block_ii, n0 - ii); i++)
129           for (k = 0; k < min(block_kk, m0 - kk); k++)
                *(a_tmp + (i * block_kk + k)) = *(a + ((i + ii) * m0 + k + kk));
131         for (j = 0; j < min(block_jj, m1 - jj); jj++)
              for (k = 0; k < min(block_kk, m0 - kk); k++)
133             *(b_tmp + (k * block_jj + j)) = *(b + ((k + kk) * m1 + j + jj));
            for (i = 0; i < min(block_ii, n0 - ii); i++)
135           for (j = 0; j < min(block_jj, m1 - jj); jj++)
                for (k = 0; k < min(block_kk, m0 - kk); k++)
137               *(c_tmp + (i * block_jj + j)) += *(a_tmp + (i * block_kk + k)) * *(b_tmp + (k * block_jj + j));
            if (kk >= m0)
139           for (i = 0; i < min(block_ii, n0 - ii); i++)
                for (j = 0; j < min(block_jj, m1 - jj); jj++)
141               *(c + ((i + ii) * m1 + j + jj)) = *(c_tmp + (i * block_jj + j));

143         if (kk == 0)
              for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
145             for (j = 0; j < min(block_jj, m1 - jj); jj++)
                  if (ii + block_ii < n0) *(c_tmp + (i * block_jj + j)) = 0;
147         for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
              for (k = 0; k < min(block_kk, m0 - kk); k++)
149             if (ii + block_ii < n0) *(a_tmp + (i * block_kk + k)) = *(a + ((i + ii) * m0 + k + kk));
            for (j = 0; j < min(block_jj, m1 - jj); jj++)
151           for (k = 0; k < min(block_kk, m0 - kk); k++)
                if (ii + block_ii < n0) *(b_tmp + (k * block_jj + j)) = *(b + ((k + kk) * m1 + j + jj));
153         for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
              for (j = 0; j < min(block_jj, m1 - jj); jj++)
155             for (k = 0; k < min(block_kk, m0 - kk); k++)
                  if (ii + block_ii < n0) *(c_tmp + (i * block_jj + j)) += *(a_tmp + (i * block_kk + k)) * *(b_tmp + (k * block_jj + j));
157         if (kk >= m0)
              for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
159             for (j = 0; j < min(block_jj, m1 - jj); jj++)
                  if (ii + block_ii < n0) *(c + ((i + ii) * m1 + j + jj)) = *(c_tmp + (i * block_jj + j));
161       }
    }
```

```
//=============================================================================================
// perform code motion to close up double-buffering transfers
//=============================================================================================
for (ii = 0; ii < n0; ii+= 2 * block_ii)
  for (jj = 0; jj < m1; jj+=block_jj)
    for (kk = 0; kk < m0; kk+=block_kk)
      {
        //buff0_load
        for (i = 0; i < min(block_ii, n0 - ii); i++)
          for (k = 0; k < min(block_kk, m0 - kk); k++)
            *(a_tmp + (i * block_kk + k)) = *(a + ((i + ii) * m0 + k + kk));
        for (j = 0; j < min(block_jj, m1 - jj); jj++)
          for (k = 0; k < min(block_kk, m0 - kk); k++)
            *(b_tmp + (k * block_jj + j)) = *(b + ((k + kk) * m1 + j + jj));
        //buff1_load
        for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
          for (k = 0; k < min(block_kk, m0 - kk); k++)
            if (ii + block_ii < n0) *(a_tmp + (i * block_kk + k)) = *(a + ((i + ii) * m0 + k + kk));
        for (j = 0; j < min(block_jj, m1 - jj); jj++)
          for (k = 0; k < min(block_kk, m0 - kk); k++)
            if (ii + block_ii < n0) *(b_tmp + (k * block_jj + j)) = *(b + ((k + kk) * m1 + j + jj));
        //perform computation on buff0
        if (kk == 0)
          for (i = 0; i < min(block_ii, n0 - ii); i++)
            for (j = 0; j < min(block_jj, m1 - jj); jj++)
              *(c_tmp + (i * block_jj + j)) = 0;
        for (i = 0; i < min(block_ii, n0 - ii); i++)
          for (j = 0; j < min(block_jj, m1 - jj); jj++)
            for (k = 0; k < min(block_kk, m0 - kk); k++)
              *(c_tmp + (i * block_jj + j)) += *(a_tmp + (i * block_kk + k)) * *(b_tmp + (k * block_jj + j));
        //store st0 buffer
        if (kk >= m0)
          for (i = 0; i < min(block_ii, n0 - ii); i++)
            for (j = 0; j < min(block_jj, m1 - jj); jj++)
              *(c + ((i + ii) * m1 + j + jj)) = *(c_tmp + (i * block_jj + j));
        //perform computation on buff1
        if (kk == 0)
          for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
            for (j = 0; j < min(block_jj, m1 - jj); jj++)
              if (ii + block_ii < n0) *(c_tmp + (i * block_jj + j)) = 0;
        for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
          for (j = 0; j < min(block_jj, m1 - jj); jj++)
            for (k = 0; k < min(block_kk, m0 - kk); k++)
              if (ii + block_ii < n0) *(c_tmp + (i * block_jj + j)) += *(a_tmp + (i * block_kk + k)) * *(b_tmp + (k * block_jj + j));
        //store st1 buffer
        if (kk >= m0)
          for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
            for (j = 0; j < min(block_jj, m1 - jj); jj++)
              if (ii + block_ii < n0) *(c + ((i + ii) * m1 + j + jj)) = *(c_tmp + (i * block_jj + j));
      }
}
//=============================================================================================
//perform array privatization on tmp_a, tmp_b, tmp_c (no data dependence between buff0 and buff1
//=============================================================================================
for (ii = 0; ii < n0; ii+= 2 * block_ii)
  for (jj = 0; jj < m1; jj+=block_jj)
    for (kk = 0; kk < m0; kk+=block_kk)
      {
        printf("ii = %d, jj = %d, kk = %d, i = %d, j = %d, k = %d\n", ii, jj, kk,i, j, k);
        //buff0_load
        for (i = 0; i < min(block_ii, n0 - ii); i++)
          for (k = 0; k < min(block_kk, m0 - kk); k++)
            *(a_tmp_0 + (i * block_kk + k)) = *(a + ((i + ii) * m0 + k + kk));
        for (j = 0; j < min(block_jj, m1 - jj); j++)
          for (k = 0; k < min(block_kk, m0 - kk); k++)
            *(b_tmp_0 + (k * m1 + j)) = *(b + ((k + kk) * m1 + j + jj));
        //buff1_load
        for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
          for (k = 0; k < min(block_kk, m0 - kk); k++)
            if (ii + block_ii < n0) *(a_tmp_1 + (i * block_kk + k)) = *(a + ((i + ii) * m0 + k + kk));
        for (j = 0; j < min(block_jj, m1 - jj); j++)
          for (k = 0; k < min(block_kk, m0 - kk); k++)
            if (ii + block_ii < n0) *(b_tmp_1 + (k * block_jj + j)) = *(b + ((k + kk) * m1 + j + jj));
        //perform computation on buff0
        if (kk == 0)
          for (i = 0; i < min(block_ii, n0 - ii); i++)
            for (j = 0; j < min(block_jj, m1 - jj); j++)
              *(c_tmp + (i * block_jj + j)) = 0;
        for (i = 0; i < min(block_ii, n0 - ii); i++)
          for (j = 0; j < min(block_jj, m1 - jj); j++)
            for (k = 0; k < min(block_kk, m0 - kk); k++)
              *(c_tmp + (i * block_jj + j)) += *(a_tmp_0 + (i * block_kk + k)) * *(b_tmp_0 + (k * block_jj + j));
        //store st0 buffer
        if (kk + block_kk >= m0)
          for (i = 0; i < min(block_ii, n0 - ii); i++)
            for (j = 0; j < min(block_jj, m1 - jj); j++)
              *(c + ((i + ii) * m1 + j + jj)) = *(c_tmp + (i * block_jj + j));
        //perform computation on buff1
```

```
251           if (kk == 0)
                for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
253               for (j = 0; j < min(block_jj, m1 - jj); j++)
                    if (ii + block_ii < n0) *(c_tmp + (i * block_jj + j)) = 0;
255           for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
                for (j = 0; j < min(block_jj, m1 - jj); j++)
257               for (k = 0; k < min(block_kk, m0 - kk); k++)
                    if (ii + block_ii < n0) *(c_tmp + (i * block_jj + j)) += *(a_tmp_1 + (i * block_kk + k)) * *(b_tmp_1 + (k * block_jj + j));
259           //store st1 buffer
              if (kk + block_kk>= m0)
261             for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
                  for (j = 0; j < min(block_jj, m1 - jj); j++)
263                 if (ii + block_ii < n0) *(c + ((i + ii) * m1 + j + jj)) = *(c_tmp + (i * block_jj + j));
            }
265
    //=============================================================================================
267 //transform multiple nested loops into 2 loops format ( linearize the loops)
    //=============================================================================================
269
    ii = 0;
271 jj = 0;
    kk = 0;
273 for (t = 0; t < n0 * m1 * m0; t+= 2 * block_ii + block_jj + block_kk)
      {
275
        if (ii < n0) ii+= 2 * block_ii;
277     if (jj < m1) jj+= block_jj;
        if (kk < m0) kk+= block_kk;
279     min_ii = min(block_ii, n0 - ii);
        min_jj = min(block_jj, m1 - jj);
281     min_kk = min(block_kk, m0 - kk);
        i = 0;
283     j = 0;
        k = 0;
285     s = 0;
        for (r = 0; r < min_ii * min_kk; r++)
287       {
            if (s == 0){
289           offset_bufflocal = i * block_kk + k;
              offset_ext_memp = (ii + i) * m0 + k + kk;
291           if (k < min_kk) k++;
              else{
293             k = 0;
                if (i < min_ii - 1) i++;
295             else {i = 0; s++;}
              }
297           lmemp = a_tmp_0 + offset_bufflocal;
              ememp = a + offset_ext_memp;
299
            }else if (s == 1){
301           offset_bufflocal = k * block_jj + j;
              offset_ext_memp = (k + kk) * m1 + j + jj;
303           if (k < min_kk) k++;
              else{
305             k = 0;
                if (j < min_jj - 1) j++;
307             else {j = 0; s++;}
                  lmemp = b_tmp_0 + offset_bufflocal;
309               ememp = b + offset_ext_memp;
              }
311         }
            /*transfer code */
313         *lmemp = *ememp;
          }
315
        // buff1_load
317     if (ii + block_ii < n0) {
          i = block_ii;
319       for (r = 0; r < min_ii * min_kk; r++)
            {
321           if (s == 0){
                offset_bufflocal = i * block_kk + k;
323             offset_ext_memp = (ii + i) * m0 + k + kk;
                if (k < min_kk) k++;
325             else{
                  k = 0;
327               if (i < min_ii - 1) i++;
                  else {i = 0; s++;}
329             }
                lmemp = a_tmp_1 + offset_bufflocal;
331             ememp = a + offset_ext_memp;

333         }else if (s == 1){
                offset_bufflocal = k * block_jj + j;
335             offset_ext_memp = (k + kk) * m1 + j + jj;
                if (k < min_kk) k++;
337             else{
                  k = 0;
```

```
339                 if (j < min_jj - 1) j++;
                    else {j = 0; s++;}
341                 lmemp = b_tmp_1 + offset_bufflocal;
                    ememp = b + offset_ext_memp;
343               }
                }
345             /*transfer code */
                *lmemp = *ememp;
347           }
        }
349
        //perform computation on buff0
351     for (i = 0; i < min(block_ii, n0 - ii); i++)
          for (j = 0; j < min(block_jj, m1 - jj); jj++)
353         *(c_tmp + (i * block_jj + j)) = 0;
        for (i = 0; i < min(block_ii, n0 - ii); i++)
355       for (j = 0; j < min(block_jj, m1 - jj); jj++)
            for (k = 0; k < min(block_kk, m0 - kk); k++)
357           *(c_tmp + (i * block_jj + j)) += *(a_tmp_0 + (i * block_kk + k)) * *(b_tmp_0 + (k * block_jj + j));
        //store st0 buffer
359     for (r = 0; r < min_ii * min_jj; r++)
          {
361         offset_bufflocal = i * block_jj + j;
            offset_ext_memp = (ii + i) * m1 + j + jj;
363         if (k < min_kk) k++;
            else{
365           k = 0;
              if (i < min_ii - 1) i++;
367           else {i = 0; s++;}
            }
369         lmemp = c_tmp + offset_bufflocal;
            ememp = c + offset_ext_memp;
371         /*transfer code */
            *ememp = *lmemp;
373       }

375     //perform computation on buff1
        for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
377       for (j = 0; j < min(block_jj, m1 - jj); jj++)
            if (ii + block_ii < n0) *(c_tmp + (i * block_jj + j)) = 0;
379     for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
          for (j = 0; j < min(block_jj, m1 - jj); jj++)
381         for (k = 0; k < min(block_kk, m0 - kk); k++)
              if (ii + block_ii < n0) *(c_tmp + (i * block_jj + j)) += *(a_tmp_1 + (i * block_kk + k)) * *(b_tmp_1 + (k * block_jj + j));
383     //store st1 buffer
        if (ii + block_ii < n0){
385       i = block_ii;
          for (r = 0; r < min_ii * min_jj; r++)
387         {
              offset_bufflocal = i * block_jj + j;
389           offset_ext_memp = (ii + i) * m1 + j + jj;
              if (k < min_kk) k++;
391           else{
                k = 0;
393             if (i < min_ii - 1) i++;
                else {i = 0; s++;}
395           }
              lmemp = c_tmp + offset_bufflocal;
397           ememp = c + offset_ext_memp;
              /*transfer code */
399           *ememp = *lmemp;
            }
401     }

403   }
    }
```

Bug testing setup

This code represents an intermediate representation of the matrix-matrix multiplication code. It was compiled and executed in order to eliminate all the possible bugs obtained when transforming the code by hand.

```
1   #include <stdio.h>
    #include <math.h>
3
    #define MAXN0 20
5   #define MAXM0 20
    #define MAXM1 20
7
    #define BL_II 1
9   #define BL_JJ 1
```

```
#define BL_KK 1

#define BL_II_MAX MAXN0
#define BL_JJ_MAX MAXM0
#define BL_KK_MAX MAXM1

//#define DEBUG

int a[MAXN0 * MAXM0];
int b[MAXM0 * MAXM1];
int c[MAXN0 * MAXM1];
int c_test[MAXN0 * MAXM1];

int a_tmp[BL_II_MAX * BL_KK_MAX];
int b_tmp[BL_JJ_MAX * BL_KK_MAX];
int a_tmp_0[BL_II_MAX * BL_KK_MAX];
int b_tmp_0[BL_JJ_MAX * BL_KK_MAX];
int a_tmp_1[BL_II_MAX * BL_KK_MAX];
int b_tmp_1[BL_JJ_MAX * BL_KK_MAX];
int c_tmp[BL_II_MAX * BL_JJ_MAX * 2];

int dbg0, dbg1, dbg2;

int min(int min0, int min1)
{
  if (min0 < min1) return min0;
  else return min1;
}

int matrix_multiply_hw_inter_0 (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int
      block_jj,int block_kk)
{
  int i, j, k;
  int ii, jj, kk;
  int dbg0, dbg1;

  int tmp;
  i = 0;
  j = 0;
  k = 0;
  for (ii = 0; ii < n0; ii+= block_ii)
    for (jj = 0; jj < m1; jj+=block_jj)
      for (kk = 0; kk < m0; kk+=block_kk)
        {
          printf("ii=%d,jj=%d,kk=%d,i=%d,j=%d,k=%d\n", ii, jj, kk, i, j, k);
          if (kk == 0)
            for (i = 0; i < min(block_ii, n0 - ii); i++)
              for (j = 0; j < min(block_jj, m1 - jj); j++)
                *(c_tmp + (i * block_jj + j)) = 0;

          for (i = 0; i < min(block_ii, n0 - ii); i++)
            for (k = 0; k < min(block_kk, m0 - kk); k++)
              {
                *(a_tmp + (i * block_kk + k)) = *(a + ((i + ii) * m0 + k + kk));
                printf("Read a=%d value\n", *(a + ((i + ii) * m0 + k + kk)));
              }
          for (k = 0; k < min(block_kk, m0 - kk); k++)
            for (j = 0; j < min(block_jj, m1 - jj); j++)
              {
                *(b_tmp + (k * block_jj + j)) = *(b + ((k + kk) * m1 + j + jj));
                printf("Read b=%d value\n", *(b + ((k + kk) * m1 + j + jj)));
              }
          printf("A_tmp\n");
          for (dbg0 = 0; dbg0 < min(block_ii, n0 - ii); dbg0++)
            {
              for (dbg1 = 0; dbg1 < min(block_kk, m0 - kk); dbg1++)
                printf("%d ", *(a_tmp + (dbg0 * block_kk + dbg1)));
              printf("\n");
            }
          printf("B_tmp\n");
          for (dbg1 = 0; dbg1 < min(block_kk, m0 - kk); dbg1++)
            {
              for (dbg0 = 0; dbg0 < min(block_jj, m1 - jj); dbg0++)
                {
                  printf("%d ", *(b_tmp + (dbg1 * block_jj + dbg0)));

                }
              printf("\n");
            }

          for (i = 0; i < min(block_ii, n0 - ii); i++)
            for (j = 0; j < min(block_jj, m1 - jj); j++)
              for (k = 0; k < min(block_kk, m0 - kk); k++)
                {
                  *(c_tmp + (i * block_jj + j)) += *(a_tmp + (i * block_kk + k)) * *(b_tmp + (k * block_jj + j));
                  printf("Compute c_tmp(%d) += a_tmp(%d) * b_tmp(%d) \t| i=%d, j=%d, k=%d\n",*(c_tmp + (i * block_jj + j)), *(a_tmp
                         + (i * block_kk + k)), *(b_tmp + (k * block_jj + j)), i, j, k);
                }
```

```
            if (kk + block_kk >= m0)
              {
                printf("Writing c\n");
                for (i = 0; i < min(block_ii, n0 - ii); i++)
                  for (j = 0; j < min(block_jj, m1 - jj); j++)
                    *(c + ((i + ii) * m1 + j + jj)) = *(c_tmp + (i * block_jj + j));
              }
          }
      return 0;
}

int matrix_multiply_hw_inter_1 (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii ,int
    block_jj ,int block_kk)
{
  int i, j, k;
  int ii, jj, kk;
  int min_ii, min_jj, min_kk;
  int tmp;

  i = 0;
  j = 0;
  k = 0;

  for (ii = 0; ii < n0; ii+= 2 * block_ii)
    for (jj = 0; jj < m1; jj+=block_jj)
      for (kk = 0; kk < m0; kk+=block_kk)
        {

          min_ii = min(block_ii, n0 - ii);
          min_jj = min(block_jj, m1 - jj);
          min_kk = min(block_kk, m0 - kk);

          printf("ii = %d, jj = %d, kk = %d, i = %d, j = %d, k = %d\n", ii, jj, kk,i, j, k);
          // buff0_load
          for (i = 0; i < min(block_ii, n0 - ii); i++)
            for (k = 0; k < min(block_kk, m0 - kk); k++)
              *(a_tmp_0 + (i * block_kk + k)) = *(a + ((i + ii) * m0 + k + kk));

          for (k = 0; k < min(block_kk, m0 - kk); k++)
            for (j = 0; j < min(block_jj, m1 - jj); j++)
              *(b_tmp_0 + (k * block_jj + j)) = *(b + ((k + kk) * m1 + j + jj));

          printf("Buff0 transfer:\n");
          printf("A_tmp_0\n");
          for (dbg0 = 0; dbg0 < min_ii; dbg0++)
            {
              for (dbg1 = 0; dbg1 < min_kk; dbg1++)
                printf("%d ", *(a_tmp_0 + (dbg0 * block_kk + dbg1)));
              printf("\n");
            }
          printf("B_tmp_0\n");
          for (dbg1 = 0; dbg1 < min_kk; dbg1++)
            {
              for (dbg0 = 0; dbg0 < min_jj; dbg0++)
                {
                  printf("%d ", *(b_tmp_0 + (dbg1 * block_jj + dbg0)));

                }
              printf("\n");
            }

          // buff1_load
          for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
            for (k = 0; k < min(block_kk, m0 - kk); k++)
              {
                if (ii + i < n0) {
                  *(a_tmp_1 + (i * block_kk + k)) = *(a + ((i + ii) * m0 + k + kk));
                  printf("Read a = %d value\n", *(a + ((i + ii) * m0 + k + kk)));
                }
              }
          for (k = 0; k < min(block_kk, m0 - kk); k++)
            for (j = 0; j < min(block_jj, m1 - jj); j++)
              if (ii + block_ii < n0) *(b_tmp_1 + (k * block_jj + j)) = *(b + ((k + kk) * m1 + j + jj));

          printf("Buff1 transfer:\n");
          printf("A_tmp_1\n");
          for (dbg0 = block_ii; dbg0 < block_ii + min_ii; dbg0++)
            {
              for (dbg1 = 0; dbg1 < min_kk; dbg1++)
                printf("%d ", *(a_tmp_1 + (dbg0 * block_kk + dbg1)));
              printf("\n");
            }
          printf("B_tmp_1\n");
          for (dbg1 = 0; dbg1 < min_kk; dbg1++)
            {
              for (dbg0 = 0; dbg0 < min_jj; dbg0++)
                {
                  printf("%d ", *(b_tmp_1 + (dbg1 * block_jj + dbg0)));
```

```
183
                    }
185                 printf("\n");
                  }
187
              //perform computation on buff0
189           if (kk == 0)
                for (i = 0; i < min(block_ii, n0 - ii); i++)
191               for (j = 0; j < min(block_jj, m1 - jj); j++)
                    *(c_tmp + (i * block_jj + j)) = 0;
193           for (i = 0; i < min(block_ii, n0 - ii); i++)
                for (j = 0; j < min(block_jj, m1 - jj); j++)
195               for (k = 0; k < min(block_kk, m0 - kk); k++)
                    *(c_tmp + (i * block_jj + j)) += *(a_tmp_0 + (i * block_kk + k)) * *(b_tmp_0 + (k * block_jj + j));
197           //store st0 buffer
              if (kk + block_kk >= m0)
199             for (i = 0; i < min(block_ii, n0 - ii); i++)
                  for (j = 0; j < min(block_jj, m1 - jj); j++)
201                 *(c + ((i + ii) * m1 + j + jj)) = *(c_tmp + (i * block_jj + j));
              //perform computation on buff1
203           if (kk == 0)
                for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
205               for (j = 0; j < min(block_jj, m1 - jj); j++)
                    if (ii + i < n0) *(c_tmp + (i * block_jj + j)) = 0;
207           for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
                for (j = 0; j < min(block_jj, m1 - jj); j++)
209               for (k = 0; k < min(block_kk, m0 - kk); k++)
                    if (ii + i < n0) *(c_tmp + (i * block_jj + j)) += *(a_tmp_1 + (i * block_kk + k)) * *(b_tmp_1 + (k * block_jj + j));
211           //store st1 buffer
              if (kk + block_kk>= m0)
213             for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
                  for (j = 0; j < min(block_jj, m1 - jj); j++)
215                 if (ii + i < n0) *(c + ((i + ii) * m1 + j + jj)) = *(c_tmp + (i * block_jj + j));
            }
217
    return 0;
219 }

221
    int matrix_multiply_hw (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int block_jj,int
        block_kk)
223 {
    int i, j, k, ii, jj, kk, t, r, s;
225 int min_ii, min_jj, min_kk;
    int offset_bufflocal, offset_ext_memp;
227 int* lmemp, * ememp;
    int iter_space;
229 int db_iter;
    ii = 0;
231 jj = 0;
    kk = 0;
233 iter_space = (int) (ceil(((double) n0) / block_ii) * ceil( ((double) m1) / block_jj) * ceil( ((double) m0) / block_kk));

235 #ifdef DEBUG
      printf("\nIter space = %d\n", iter_space);
237 #endif
    if (ceil ( ((double) n0) / block_ii) > 1) db_iter = 2;
239 else db_iter = 1;
    for (t = 0; t < iter_space; t+= db_iter)
241   {

243     min_ii = min(block_ii, n0 - ii);
        min_jj = min(block_jj, m1 - jj);
245     min_kk = min(block_kk, m0 - kk);
        i = 0;
247     j = 0;
        k = 0;
249     s = 0;
    #ifdef DEBUG
251     printf ("Buff0\n");
    #endif
253     for (r = 0; r < min_ii * min_kk + min_jj * min_kk; r++)
        {
255 #ifdef DEBUG
          printf("i = %d, j = %d, k = %d, ii = %d, jj = %d, kk = %d, s = %d\n", i, j, k, ii, jj, kk, s);
257 #endif
          if (s == 0){
259         offset_bufflocal = i * block_kk + k;
            offset_ext_memp = (ii + i) * m0 + k + kk;
261         //      printf("Transfering a_tmp[%d][%d] = a[%d][%d] = ", i, k, i + ii, k + kk);
            if (k < min_kk - 1) k++;
263         else{
              k = 0;
265           if (i < min_ii - 1) i++;
              else {i = 0; s++;}
267         }
            lmemp = a_tmp_0 + offset_bufflocal;
269         ememp = a + offset_ext_memp;
```

```
271
            }else {
273           if (s == 1){
                offset_bufflocal  = k * block_jj + j;
275             offset_ext_memp = (kk + k) * m1 + j + jj;

277             if (k < min_kk - 1) k++;
                else{
279               k = 0;
                  if (j < min_jj - 1) j++;
281               else {j = 0; s = 0;}
                }
283             lmemp = b_tmp_0 + offset_bufflocal;
                ememp = b + offset_ext_memp;
285
                //                printf("%d\n",*(b + offset_ext_memp));
287
              }
289         }
            /*transfer code */
291         *lmemp = *ememp;
          }
293 #ifdef DEBUG
        printf("Buff0 transfer:\n");
295     printf("A_tmp_0\n");
        for (dbg0 = 0; dbg0 < min_ii; dbg0++)
297       {
          for (dbg1 = 0; dbg1 < min_kk; dbg1++)
299         printf("%d ", *(a_tmp_0 + (dbg0 * block_kk + dbg1)));
          printf("\n");
301     }
        printf("B_tmp_0\n");
303     for (dbg1 = 0; dbg1 < min_kk; dbg1++)
        {
305       for (dbg0 = 0; dbg0 < min_jj; dbg0++)
          {
307         printf("%d ", *(b_tmp_0 + (dbg1 * block_jj + dbg0)));

309       }
          printf("\n");
311     }

313     printf("Buff1\n");
    #endif
315     if (ii + block_ii < n0) {
          i = block_ii;
317       for (r = 0; r < min_ii * min_kk + min_jj * min_kk; r++)
          {
319 #ifdef DEBUG
            printf("i =%d,j =%d,k =%d,ii =%d,jj =%d,kk =%d,s =%d\n", i, j, k, ii, jj, kk, s);
321 #endif
            if (s == 0){
              offset_bufflocal  = (i - block_ii) * block_kk + k;
              offset_ext_memp = (ii + i) * m0 + k + kk;
325           if (k < min_kk - 1) k++;
              else{
327             k = 0;
                if (i < min_ii + block_ii - 1) i++;
329             else {i = 0; s++;}
              }
331           lmemp = a_tmp_1 + offset_bufflocal;
              ememp = a + offset_ext_memp;
333
            }else if (s == 1){
335           offset_bufflocal  = k * block_jj + j;
              offset_ext_memp = (k + kk) * m1 + j + jj;
337           if (k < min_kk - 1) k++;
              else{
339             k = 0;
                if (j < min_jj - 1) j++;
341             else {j = 0; s = 0;}
              }
343           lmemp = b_tmp_1 + offset_bufflocal;
              ememp = b + offset_ext_memp;
345         }
            /*transfer code */
347         *lmemp = *ememp;
          }
349 #ifdef DEBUG
        printf("Buff1 transfer:\n");
351     printf("A_tmp_1\n");
        for (dbg0 = 0; dbg0 < min_ii; dbg0++)
353       {
          for (dbg1 = 0; dbg1 < min_kk; dbg1++)
355         printf("%d ", *(a_tmp_1 + (dbg0 * block_kk + dbg1)));
          printf("\n");
357       }
```

```
          printf("B_tmp_1\n");
359       for (dbg1 = 0; dbg1 < min_kk; dbg1++)
            {
361           for (dbg0 = 0; dbg0 < min_jj; dbg0++)
                {
363               printf("%d ", *(b_tmp_1 + (dbg1 * block_jj + dbg0)));

365             }
                printf("\n");
367         }
    #endif
369     }

371     //perform computation on buff0
        if (kk == 0){
373       for (i = 0; i < min_ii; i++)
            for (j = 0; j < min_jj; j++)
375           *(c_tmp + (i * block_jj + j)) = 0;
        }
377     for (i = 0; i < min_ii; i++)
          for (j = 0; j < min_jj; j++)
379         for (k = 0; k < min_kk; k++)
              {
381             *(c_tmp + (i * block_jj + j)) += *(a_tmp_0 + (i * block_kk + k)) * *(b_tmp_0 + (k * block_jj + j));
    #ifdef DEBUG
383             printf("Compute c_tmp (%d) += a_tmp_0 (%d) * b_tmp_0 (%d) \t | i = %d, j = %d, k = %d\n",*(c_tmp + (i * block_jj + j)), *(
                    a_tmp_0 + (i * block_kk + k)), *(b_tmp_0 + (k * block_jj + j)), i, j, k);
    #endif
385           }
        //store st0 buffer
387     i = 0;
        j = 0;
389
        if (kk + block_kk >= m0)
391       {
            for (r = 0; r < min_ii * min_jj; r++)
393           {

395             offset_bufflocal = i * block_jj + j;
                offset_ext_memp = (ii + i) * m1 + j + jj;
397 #ifdef DEBUG
                printf("Store st0 j = %d, i = %d\n", j, i );
399 #endif
                if (j < min_jj - 1) j++;
401             else{
                  j = 0;
403               if (i < min_ii - 1) i++;
                  else {i = 0; s++;}
405           }
              lmemp = c_tmp + offset_bufflocal;
407           ememp = c + offset_ext_memp;

409           /*transfer code */
              *ememp = *lmemp;
411 #ifdef DEBUG
              printf("%d (add = %x) <= %d (add = %x)\n", *ememp, ememp, *lmemp, lmemp);
413 #endif
            }
415 #ifdef DEBUG
          printf ("Storing c0[%d][%d]\n", ii , jj );
417       for (dbg0 = 0; dbg0 < min_ii; dbg0++)
            {
419           for (dbg1 = 0; dbg1 < min_jj; dbg1++)
                {
421               printf("%d ", *(c + (ii + dbg0) * m1 + dbg1 + jj));
                }
423           printf("\n");
            }
425 #endif
        }
427
        //perform computation on buff1
429     if (kk == 0){
          for (i = block_ii; i < block_ii + min_ii; i++)
431         for (j = 0; j < min_jj; j++)
              if (ii + i < n0) *(c_tmp + (i * block_jj + j)) = 0;
433     }
        for (i = block_ii; i < block_ii + min_ii; i++)
435       for (j = 0; j < min_jj; j++)
            for (k = 0; k < min_kk; k++)
437           if (ii + i < n0)
                {
439               *(c_tmp + (i * block_jj + j)) += *(a_tmp_1 + ((i - block_ii) * block_kk + k)) * *(b_tmp_1 + (k * block_jj + j));
    #ifdef DEBUG
441               printf("Compute c_tmp (%d) += a_tmp_1 (%d) * b_tmp_1 (%d) \t | i = %d, j = %d, k = %d\n",*(c_tmp + (i * block_jj + j)), *(
                      a_tmp_1 + ((i - block_ii) * block_kk + k)), *(b_tmp_1 + (k * block_jj + j)), i, j, k);
    #endif
443           }
```

```
445       //store st1 buffer
          i = block_ii;
447       j = 0;
          if (kk + block_kk >= m0)
449         {
              for (r = 0; r < min_ii * min_jj; r++)
451             {
                  offset_bufflocal  = i * block_jj + j;
453               offset_ext_memp = (ii + i) * m1 + j + jj;
                  if (j < min_jj - 1) j++;
455               else{
                    j = 0;
                    if ((i < block_ii + min_ii - 1) && (ii + i < n0 - 1)) i++;
                    else {i = 0; s++;}
459               }
                  lmemp = c_tmp + offset_bufflocal;
461               ememp = c + offset_ext_memp;
                  /*transfer code */
463               *ememp = *lmemp;
                }
465 #ifdef DEBUG
              printf ("Storing c1[%d][%d]\n", ii + block_ii, jj);
467           for (dbg0 = block_ii; dbg0 < block_ii + min_ii; dbg0++)
                {
469               for (dbg1 = 0; dbg1 < min_jj; dbg1++)
                    {
471                   printf("%d", *(c + (ii + dbg0) * m1 + dbg1 + jj));
                      printf("(%d) ", *(c_tmp + (dbg0 * block_jj + dbg1)));
473                 }
                  printf("\n");
475             }
    #endif
477       }
          if (kk + block_kk < m0) kk+= block_kk;
479       else {
            kk = 0;
            if (jj + block_jj < m1) jj+= block_jj;
            else {
483           jj = 0;
              if (ii + 2 * block_ii < n0) ii+= 2 * block_ii;
485         }
          }
487


489     }
      return 0;
491 }

493 int matrix_multiply_hw_orig (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1)
    {
495   int i, j, k;
      int tmp;
497


499   for (i = 0; i < n0; i++)
        for (j = 0; j < m1; j++)
501       {
            tmp = 0;
503         for (k = 0; k < m0; k++)
              {
505             tmp += *(a + (i * m0 + k)) * *(b + (k * m1 + j));

507           }
            *(c + (i * m1 + j)) = tmp;
509       }
      return 0;
511 }

513
    int main()
515 {
      int error_found = 0;
517   int i,j,k;
      int n0, m0, n1, m1;
519   int block_ii, block_jj, block_kk;
      int tmp;
521   n0 = MAXN0;
      m0 = MAXM0;
523   n1 = MAXM0;
      m1 = MAXM1;
525   block_ii = BL_II;
      block_jj = BL_JJ;
527   block_kk = BL_KK;
      printf ("Main\n");
529   for (n0 = 1; n0 < MAXN0; n0++)
        for (m0 = 1; m0 < MAXM0; m0++)
531       for (m1 = 1; m1 < MAXM1; m1++)
```

```
        {
533
#ifdef DEBUG
535           printf("\n ==========================================================================\n");
              printf("Testing␣with␣n0␣=␣%d,␣m0␣=␣%d,␣m1␣=␣%d\n", n0, m0, m1);
537 #endif
              n1 = m0;
539 #ifdef DEBUG
              printf ("matrix␣A\n");
541           for (i = 0; i < n0; i++){
                for (j = 0; j < m0; j++)
                  {
                    *(a + i * m0 + j) = i * m0 + j + 1;
545                 printf ("%d\t",*(a + i * m0 + j));
                  }
547             printf("\n");
              }
549
              printf( "matrix␣B\n");
551           for (i = 0; i < m0; i++){
                for (j = 0; j < m1; j++)
                  {
                    *(b + i * m1 + j) = i * m1 + j + 1;
555                 printf("%d\t",  *(b + (i * m1 + j)));

557               }
                printf("\n");
559           }
    #endif /*DEBUG*/
561           for (i = 0; i < n0; i++)
                for (j = 0; j < m1; j++)
563               c[i * m1 + j] = 0;

565           for (i = 0; i < n0; i++)
                for (j = 0; j < m1; j++)
567               {
                    c_test[i * m1 + j] = 0;
569                 for (k = 0; k < m0; k++)
                      c_test[i * m1 + j] += a[i * m0 + k] * b[k * m1 + j];
571               }
    #ifdef DEBUG
573           printf("matrix␣C\n");
              for (i = 0; i < n0; i++){
575             for (j = 0; j < m1; j++)
                  printf("%d\t", c_test[i * m1 + j]);
577           printf("\n");
              }
579
              printf("Testing␣original␣version\n");
581 #endif /*DEBUG*/
              matrix_multiply_hw_orig ((int *) a, (int *) b, (int *)c, n0, m0, m1);
583
              for (i = 0; i < n0; i++)
                for ( j =0; j < m1; j++)
                  {
587                 if ( c_test[i * m1 + j] != c[i * m1 + j]) printf ("Error␣on␣c_test[%d][%d]␣=␣%d␣!=␣%d␣=␣c[%d][%d]\n", i,j,c_test[i * m1 + j],c
                          [i * m1 + j], i , j);

589               }
              for ( block_ii = 1; block_ii <= n0; block_ii++)
591             for ( block_jj = 1; block_jj <= m0; block_jj++)
                  for (block_kk = 1; block_kk <= m1; block_kk++)
593                 {

595
    #ifdef DEBUG
597                   printf("\n␣block_ii␣=␣%d,␣block_jj␣=␣%d,␣block_kk␣=␣%d\n\n", block_ii , block_jj , block_kk);

599

601                   printf("Testing␣transformed␣version\n");
    #endif
603
                      for (i = 0; i < n0; i++)
605                     for (j = 0; j < m1; j++)
                          c[i * m1 + j] = 0;
607                   //          matrix_multiply_hw_inter_1 ((int *) a, (int *) b, (int *)c, n0, m0, m1, block_ii , block_jj , block_kk );

609

611                   matrix_multiply_hw ((int *) a, (int *) b, (int *)c, n0, m0, m1, block_ii , block_jj , block_kk);
    #ifdef DEBUG
613                   printf("matrix␣C␣(computed)\n");
                      printf("x\t");
615                   for (j = 0; j < m1; j++)
                        printf("%d\t", j);
617                   printf("\n");
                      for (j = 0; j < m1 + 1; j++)
```

```
619                     printf("-\t");
                      printf("\n");
621                   for (i = 0; i < n0; i++){
                        printf("%d |\t", i);
623                     for (j = 0; j < m1; j++)
                          printf("%d\t", c[i * m1 + j]);
625                     printf("\n");
                      }
627 #endif /*DEBUG*/
                    for (i = 0; i < n0; i++)
629                   for ( j =0; j < m1; j++)
                        {
631                       if ( c_test[i * m1 + j] != c[i * m1 + j])
                          {
633                         printf ("Error on c_test[%d][%d] = %d != c[%d][%d] (add = %x)\n", i,j,c_test[i * m1 + j],c[i * m1 + j], i , j
                                , (c + i * m1 + j));
                          printf ("Error found on block_ii = %d, block_jj = %d, block_kk = %d\n", block_ii, block_jj, block_kk);
635                       printf("and with n0 = %d, m0 = %d, m1 = %d\n\n", n0, m0, m1);
                          error_found = 1;
637                       }

639                     }
    #ifdef DEBUG
641                   printf("\nFinish testing with block_ii = %d, block_jj = %d, block_kk = %d\n", block_ii, block_jj, block_kk);
                      printf("and with n0 = %d, m0 = %d, m1 = %d\n\n", n0, m0, m1);
643 #endif
                      if (error_found == 1) return -1;
645                 }
          }
647   printf ("End\n");

649
    return 0;
651 }
```

Linux with shared memory and processes simulation

This code represents the almost final version of the matrix-matrix multiplication code. It was transformed by hand and divided into multiple functions that will be synthesized as an accelerator. The FIFO communications are simulated using Linux named pipes and local memories using Linux shared memories. It was compiled and executed on a multiprocessor system so as to find possible bugs obtained during manual code transformations. SystemC could be used for system simulation, however the presented here C code is closer to the code that is synthesized using `C2H`.

```
1
  /* int a_tmp[BL_II_MAX * BL_KK_MAX]; */
3 /* int b_tmp[BL_JJ_MAX * BL_KK_MAX]; */
  /* int a_tmp_0[BL_II_MAX * BL_KK_MAX]; */
5 /* int b_tmp_0[BL_JJ_MAX * BL_KK_MAX]; */
  /* int a_tmp_1[BL_II_MAX * BL_KK_MAX]; */
7 /* int b_tmp_1[BL_JJ_MAX * BL_KK_MAX]; */
  /* int c_tmp[BL_II_MAX * BL_JJ_MAX * 2]; */
9
  int *a_tmp;
11 int *b_tmp;
  int *a_tmp_0;
13 int *b_tmp_0;
  int *a_tmp_1;
15 int *b_tmp_1;
  int *c_tmp;
17

19 int dbg0, dbg1, dbg2;

21 /* Pipes declarations */
  int st1_buff0 [2];
23 int buff0_buff1 [2];
  int buff1_st0 [2];
25 int st0_st1 [2];
  int c01_st0 [2];
27 int c01_st1 [2];
  int buff01_c01 [2];
29


31
  int min(int min0, int min1)
33 {
    if (min0 < min1) return min0;
35  else return min1;
  }
```

```
37
int matrix_multiply_hw_inter_0 (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int
      block_jj,int block_kk)
39 {
  int i, j, k;
41  int ii, jj, kk;
  int dbg0, dbg1;
43
  int tmp;
45  i = 0;
  j = 0;
47  k = 0;
  for (ii = 0; ii < n0; ii+= block_ii)
49    for (jj = 0; jj < m1; jj+=block_jj)
      for (kk = 0; kk < m0; kk+=block_kk)
51      {
        printf("ii = %d, jj = %d, kk = %d, i = %d, j = %d, k = %d\n", ii, jj, kk, i, j, k);
53      if (kk == 0)
          for (i = 0; i < min(block_ii, n0 - ii); i++)
55          for (j = 0; j < min(block_jj, m1 - jj); j++)
              *(c_tmp + (i * block_jj + j)) = 0;
57
        for (i = 0; i < min(block_ii, n0 - ii); i++)
59        for (k = 0; k < min(block_kk, m0 - kk); k++)
            {
61            *(a_tmp + (i * block_kk + k)) = *(a + ((i + ii) * m0 + k + kk));
              printf("Read a = %d value\n", *(a + ((i + ii) * m0 + k + kk)));
63          }
        for (k = 0; k < min(block_kk, m0 - kk); k++)
65        for (j = 0; j < min(block_jj, m1 - jj); j++)
            {
67            *(b_tmp + (k * block_jj + j)) = *(b + ((k + kk) * m1 + j + jj));
              printf("Read b = %d value\n", *(b + ((k + kk) * m1 + j + jj)));
69          }
        printf("A_tmp\n");
71      for (dbg0 = 0; dbg0 < min(block_ii, n0 - ii); dbg0++)
          {
73          for (dbg1 = 0; dbg1 < min(block_kk, m0 - kk); dbg1++)
              printf("%d ", *(a_tmp + (dbg0 * block_kk + dbg1)));
75          printf("\n");
          }
77      printf("B_tmp\n");
        for (dbg1 = 0; dbg1 < min(block_kk, m0 - kk); dbg1++)
79        {
            for (dbg0 = 0; dbg0 < min(block_jj, m1 - jj); dbg0++)
81            {
                printf("%d ", *(b_tmp + (dbg1 * block_jj + dbg0)));
83
              }
85          printf("\n");
          }
87
        for (i = 0; i < min(block_ii, n0 - ii); i++)
89        for (j = 0; j < min(block_jj, m1 - jj); j++)
            for (k = 0; k < min(block_kk, m0 - kk); k++)
91            {
                *(c_tmp + (i * block_jj + j)) += *(a_tmp + (i * block_kk + k)) * *(b_tmp + (k * block_jj + j));
93              printf("Compute c_tmp (%d) += a_tmp (%d) * b_tmp (%d) \t | i = %d, j = %d, k = %d\n",*(c_tmp + (i * block_jj + j)), *(a_tmp
                    + (i * block_kk + k)), *(b_tmp + (k * block_jj + j)), i, j, k);
              }
95      if (kk + block_kk >= m0)
          {
97          printf("Writing c\n");
            for (i = 0; i < min(block_ii, n0 - ii); i++)
99            for (j = 0; j < min(block_jj, m1 - jj); j++)
                *(c + ((i + ii) * m1 + j + jj)) = *(c_tmp + (i * block_jj + j));
101       }
      }
103 return 0;
}
105
int matrix_multiply_hw_inter_1 (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int
      block_jj,int block_kk)
107 {
  int i, j, k;
109 int ii, jj, kk;
  int min_ii, min_jj, min_kk;
111 int tmp;

113 i = 0;
  j = 0;
115 k = 0;

117 for (ii = 0; ii < n0; ii+= 2 * block_ii)
      for (jj = 0; jj < m1; jj+=block_jj)
119     for (kk = 0; kk < m0; kk+=block_kk)
        {
121
```

```
    min_ii = min(block_ii, n0 - ii);
    min_jj = min(block_jj, m1 - jj);
    min_kk = min(block_kk, m0 - kk);

    printf("ii = %d, jj = %d, kk = %d, i = %d, j = %d, k = %d\n", ii, jj, kk,i, j, k);
    // buff0_load
    for (i = 0; i < min(block_ii, n0 - ii); i++)
      for (k = 0; k < min(block_kk, m0 - kk); k++)
        *(a_tmp_0 + (i * block_kk + k)) = *(a + ((i + ii) * m0 + k + kk));

    for (k = 0; k < min(block_kk, m0 - kk); k++)
      for (j = 0; j < min(block_jj, m1 - jj); j++)
        *(b_tmp_0 + (k * block_jj + j)) = *(b + ((k + kk) * m1 + j + jj));

    printf("Buff0 transfer:\n");
    printf("A_tmp_0\n");
    for (dbg0 = 0; dbg0 < min_ii; dbg0++)
      {
        for (dbg1 = 0; dbg1 < min_kk; dbg1++)
          printf("%d ", *(a_tmp_0 + (dbg0 * block_kk + dbg1)));
        printf("\n");
      }
    printf("B_tmp_0\n");
    for (dbg1 = 0; dbg1 < min_kk; dbg1++)
      {
        for (dbg0 = 0; dbg0 < min_jj; dbg0++)
          {
            printf("%d ", *(b_tmp_0 + (dbg1 * block_jj + dbg0)));

          }
        printf("\n");
      }

    // buff1_load
    for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
      for (k = 0; k < min(block_kk, m0 - kk); k++)
        {
          if (ii + i < n0) {
            *(a_tmp_1 + (i * block_kk + k)) = *(a + ((i + ii) * m0 + k + kk));
            printf("Read a = %d value\n", *(a + ((i + ii) * m0 + k + kk)));
          }
        }
    for (k = 0; k < min(block_kk, m0 - kk); k++)
      for (j = 0; j < min(block_jj, m1 - jj); j++)
        if (ii + block_ii < n0) *(b_tmp_1 + (k * block_jj + j)) = *(b + ((k + kk) * m1 + j + jj));

    printf("Buff1 transfer:\n");
    printf("A_tmp_1\n");
    for (dbg0 = block_ii; dbg0 < block_ii + min_ii; dbg0++)
      {
        for (dbg1 = 0; dbg1 < min_kk; dbg1++)
          printf("%d ", *(a_tmp_1 + (dbg0 * block_kk + dbg1)));
        printf("\n");
      }
    printf("B_tmp_1\n");
    for (dbg1 = 0; dbg1 < min_kk; dbg1++)
      {
        for (dbg0 = 0; dbg0 < min_jj; dbg0++)
          {
            printf("%d ", *(b_tmp_1 + (dbg1 * block_jj + dbg0)));

          }
        printf("\n");
      }

    //perform computation on buff0
    if (kk == 0)
      for (i = 0; i < min(block_ii, n0 - ii); i++)
        for (j = 0; j < min(block_jj, m1 - jj); j++)
          *(c_tmp + (i * block_jj + j)) = 0;
    for (i = 0; i < min(block_ii, n0 - ii); i++)
      for (j = 0; j < min(block_jj, m1 - jj); j++)
        for (k = 0; k < min(block_kk, m0 - kk); k++)
          *(c_tmp + (i * block_jj + j)) += *(a_tmp_0 + (i * block_kk + k)) * *(b_tmp_0 + (k * block_jj + j));
    //store st0 buffer
    if (kk + block_kk >= m0)
      for (i = 0; i < min(block_ii, n0 - ii); i++)
        for (j = 0; j < min(block_jj, m1 - jj); j++)
          *(c + ((i + ii) * m1 + j + jj)) = *(c_tmp + (i * block_jj + j));
    //perform computation on buff1
    if (kk == 0)
      for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
        for (j = 0; j < min(block_jj, m1 - jj); j++)
          if (ii + i < n0) *(c_tmp + (i * block_jj + j)) = 0;
    for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
      for (j = 0; j < min(block_jj, m1 - jj); j++)
        for (k = 0; k < min(block_kk, m0 - kk); k++)
          if (ii + i < n0) *(c_tmp + (i * block_jj + j)) += *(a_tmp_1 + (i * block_kk + k)) * *(b_tmp_1 + (k * block_jj + j));
```

```
            //store st1 buffer
211         if (kk + block_kk>= m0)
              for (i = block_ii; i < block_ii + min(block_ii, n0 - ii); i++)
213             for (j = 0; j < min(block_jj, m1 - jj); j++)
                  if (ii + i < n0) *(c + ((i + ii) * m1 + j + jj)) = *(c_tmp + (i * block_jj + j));
215       }

217   return 0;
    }
219
    int read_pipe_one_element (int* pipe_fd_read)
221 {
      char readbuffer;
223   int nbytes;
      nbytes = read(*pipe_fd_read, &readbuffer, sizeof(readbuffer));
225   return (int) readbuffer;
    }
227
    int write_pipe_one_element(int* pipe_fd_write, int data)
229 {
      char data_char = (char) data;
231   write(*pipe_fd_write, &data_char, sizeof(data_char));
      return 0;
233 }

235 int matrix_multiply_hw_buff0 (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii ,int block_jj
        ,int block_kk, int iter_space, int db_iter)
    {
237   int i, j, k, ii, jj, kk, t, r, s;
      int min_ii, min_jj, min_kk;
239   int offset_bufflocal , offset_ext_memp;
      int* lmemp, * ememp;
241   int dummy_read = 0;
      int r_sup;
243   int bool_nsent = 1;
      int tmp;
245   int* buff0_buff1_write = &buff0_buff1[1];
      int* buff01_c01_write = &buff01_c01[1];
247   int* st1_buff0_read = &st1_buff0[0];
      ii = 0;
249   jj = 0;
      kk = 0;
251
      for (t = 0; t < iter_space; t+= db_iter)
253     {

255       min_ii = min(block_ii, n0 - ii);
          min_jj = min(block_jj, m1 - jj);
257       min_kk = min(block_kk, m0 - kk);
          i = 0;
259       j = 0;
          k = 0;
261       s = 0;

263       /* read from the st1_buff0 pipe */
          dummy_read += read_pipe_one_element (st1_buff0_read);
265 #ifdef DEBUG
          printf("Read from st1_buff0 pipe\n");
267       printf ("Buff0\n");
    #endif
269       r_sup = min_ii * min_kk + min_jj * min_kk;
          bool_nsent = 1;
271       for (r = 0, tmp = dummy_read; r < r_sup; r++)
            {
273 #ifdef DEBUG
            printf("i = %d, j = %d, k = %d, ii = %d, jj = %d, kk = %d, s = %d\n", i, j, k, ii, jj, kk, s);
275 #endif
            if (s == 0){
277           offset_bufflocal = i * block_kk + k;
              offset_ext_memp = (ii + i) * m0 + k + kk;
279
              if (k < min_kk - 1) k++;
281           else{
                k = 0;
283             if (i < min_ii - 1) i++;
              else {i = 0; s++;}
285           }
              lmemp = a_tmp_0 + offset_bufflocal;
287           ememp = a + offset_ext_memp;

289
            }else {
291           if (s == 1){
                offset_bufflocal = k * block_jj + j;
293             offset_ext_memp = (kk + k) * m1 + j + jj;

295             if (j < min_jj - 1) j++;
                else{
```

```
297                   j = 0;
                      if (k < min_kk - 1) k++;
299                   else {k = 0; s = 0;}
                    }
301
                  lmemp = b_tmp_0 + offset_bufflocal;
303               ememp = b + offset_ext_memp;

305


307             }
              }
309           /*transfer code */
              *lmemp = *ememp;
311           /* sync code */
              if ((r == r_sup - 1) & bool_nsent)
313             {
                bool_nsent = 0;
315             tmp = 0;
                write_pipe_one_element(buff0_buff1_write, 0);
317 #ifdef DEBUG
                printf("Write to buff0_buff1 pipe\n");
319 #endif

321           }
            }
323       write_pipe_one_element(buff01_c01_write, tmp);
  #ifdef DEBUG
325       printf("Write to buff01_c01 pipe\n");
  #endif
327
  #ifdef DEBUG
329       printf("Buff0 transfer:\n");
          printf("A_tmp_0\n");
331       for (dbg0 = 0; dbg0 < min_ii; dbg0++)
            {
333           for (dbg1 = 0; dbg1 < min_kk; dbg1++)
                printf("%d ", *(a_tmp_0 + (dbg0 * block_kk + dbg1)));
335         printf("\n");
            }
337       printf("B_tmp_0\n");
          for (dbg1 = 0; dbg1 < min_kk; dbg1++)
339         {
              for (dbg0 = 0; dbg0 < min_jj; dbg0++)
341             {
                  printf("%d ", *(b_tmp_0 + (dbg1 * block_jj + dbg0)));
343
                }
345           printf("\n");
            }
347 #endif
          if (kk + block_kk < m0) kk+= block_kk;
349       else {
            kk = 0;
351         if (jj + block_jj < m1) jj+= block_jj;
            else {
353           jj = 0;
              if (ii + 2 * block_ii < n0) ii+= 2 * block_ii;
355         }
          }
357     }

359   /* read from the st1_buff0 pipe */
      dummy_read += read_pipe_one_element (st1_buff0_read);
361 #ifdef DEBUG
      printf("Read (clear)  st1_buff0 pipe\n");
363 #endif

365   return 0;
  }
367

369 int matrix_multiply_hw_buff1 (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii ,int block_jj
        ,int block_kk, int iter_space, int db_iter)
    {
371   int i, j, k, ii, jj, kk, t, r, s;
      int min_ii, min_jj, min_kk;
373   int offset_bufflocal , offset_ext_memp;
      int* lmemp, * ememp;
375   int dummy_read = 0;
      int r_sup;
377   int bool_nsent = 1;
      int tmp;
379   int* buff0_buff1_read = &buff0_buff1[0];
      int* buff1_st0_write = &buff1_st0[1];
381   int* buff01_c01_write = &buff01_c01[1];

383    ii = 0;
```

```
    jj = 0;
385 kk = 0;

387 for (t = 0; t < iter_space; t+= db_iter)
      {
389
        min_ii = min(block_ii, n0 - ii);
391     min_jj = min(block_jj, m1 - jj);
        min_kk = min(block_kk, m0 - kk);
393     i = 0;
        j = 0;
395     k = 0;
        s = 0;
397     /* read from the buff0_buff1 pipe */
        dummy_read += read_pipe_one_element (buff0_buff1_read);
399 #ifdef DEBUG
        printf("Read from buff0_buff1 pipe\n");
401 #endif

403 #ifdef DEBUG
        printf("Buff1\n");
405 #endif
        if (ii + block_ii < n0) {
407       i = block_ii;
          r_sup = min_ii * min_kk + min_jj * min_kk;
409       bool_nsent = 1;
          for (r = 0, tmp = dummy_read; r < r_sup; r++)
411         {
    #ifdef DEBUG
413           printf("i = %d, j = %d, k = %d, ii = %d, jj = %d, kk = %d, s = %d\n", i, j, k, ii, jj, kk, s);
    #endif
415           if (s == 0){
              offset_bufflocal = (i - block_ii) * block_kk + k;
417           offset_ext_memp = (ii + i) * m0 + k + kk;
              if (k < min_kk - 1) k++;
419           else{
                k = 0;
                if (i < min_ii + block_ii - 1) i++;
421             else {i = 0; s++;}
              }
423           lmemp = a_tmp_1 + offset_bufflocal;
              ememp = a + offset_ext_memp;
425
427           }else if (s == 1){
              offset_bufflocal = k * block_jj + j;
429           offset_ext_memp = (k + kk) * m1 + j + jj;
              if (j < min_jj - 1) j++;
431           else{
                j = 0;
                if (k < min_kk - 1) k++;
433             else {k = 0; s = 0;}
              }
435
              lmemp = b_tmp_1 + offset_bufflocal;
437           ememp = b + offset_ext_memp;
            }
439         /*transfer code */
            *lmemp = *ememp;
441         if ((r == r_sup - 1) & bool_nsent)
              {
443             bool_nsent = 0;
                tmp = 1;
445             write_pipe_one_element(buff1_st0_write, 0);
    #ifdef DEBUG
447             printf("Write to buff1_st0 pipe\n");
    #endif
449
              }
451       }
        }
453
      write_pipe_one_element(buff01_c01_write, tmp);
455 #ifdef DEBUG
      printf("Write to buff01_c01 (buff1) pipe\n");
457 #endif

459 #ifdef DEBUG
      printf("Buff1 transfer:\n");
461   printf("A_tmp_1\n");
      for (dbg0 = 0; dbg0 < min_ii; dbg0++)
463     {
        for (dbg1 = 0; dbg1 < min_kk; dbg1++)
465       printf("%d ", *(a_tmp_1 + (dbg0 * block_kk + dbg1)));
        printf("\n");
467   }
      printf("B_tmp_1\n");
469   for (dbg1 = 0; dbg1 < min_kk; dbg1++)
      {
471     for (dbg0 = 0; dbg0 < min_jj; dbg0++)
```

```
                {
473                 printf("%d␣", *(b_tmp_1 + (dbg1 * block_jj + dbg0)));

475               }
                  printf("\n");
477             }
    #endif
479       }
          else
481         {

483           write_pipe_one_element(buff1_st0_write, 0);
              write_pipe_one_element(buff01_c01_write, 1);
485 #ifdef DEBUG
              printf("Write␣to␣buff1_st0␣pipe\n");
487           printf("Write␣to␣buff01_c01␣(buff1)␣pipe\n");
    #endif
489

            }
491       if (kk + block_kk < m0) kk+= block_kk;
          else {
493         kk = 0;
            if (jj + block_jj < m1) jj+= block_jj;
495         else {
              jj = 0;
497           if (ii + 2 * block_ii < n0) ii+= 2 * block_ii;
            }
499       }
        }
501   return 0;
    }
503
    int matrix_multiply_hw_st0 (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int block_jj,
          int block_kk, int iter_space, int db_iter)
505 {
      int i, j, k, ii, jj, kk, t, r, s;
507   int min_ii, min_jj, min_kk;
      int offset_bufflocal, offset_ext_memp;
509   int* lmemp, * ememp;
      int r_sup;
511   int dummy_read = 0;
      int dummy_read1 = 0;
513   int tmp;
      int bool_nsent = 1;
515   int* buff1_st0_read = &buff1_st0[0];
      int* c01_st0_read = &c01_st0[0];
517   int* st0_st1_write = &st0_st1[1];
      ii = 0;
519   jj = 0;
      kk = 0;
521   for (t = 0; t < iter_space; t+= db_iter)
        {
523

          min_ii = min(block_ii, n0 - ii);
525       min_jj = min(block_jj, m1 - jj);
          min_kk = min(block_kk, m0 - kk);
527       i = 0;
          j = 0;
529       k = 0;
          s = 0;
531       //store st0 buffer
          i = 0;
533       j = 0;


535


537       dummy_read += read_pipe_one_element (buff1_st0_read);
    #ifdef DEBUG
539         printf("Read␣from␣buff1_st0␣pipe\n");
    #endif
541       dummy_read1 += read_pipe_one_element (c01_st0_read);
    #ifdef DEBUG
543         printf("Read␣from␣c01_st0␣pipe\n");
    #endif
545

          if (kk + block_kk >= m0)
547         {
              r_sup = min_ii * min_jj;
549           bool_nsent = 1;
              for (r = 0, tmp = dummy_read + dummy_read1; r < r_sup; r++)
551             {

553               offset_bufflocal = i * block_jj + j;
                  offset_ext_memp = (ii + i) * m1 + j + jj;
555 #ifdef DEBUG
                  printf("Store␣st0␣j␣=␣%d,␣i␣=␣%d\n", j, i );
557 #endif
                  if (j < min_jj - 1) j++;
```

```
559                 else{
                      j = 0;
561                   if (i < min_ii - 1) i++;
                      else {i = 0; s++;}
563                 }
                    lmemp = c_tmp + offset_bufflocal;
565                 ememp = c + offset_ext_memp;

567                 /*transfer code */
                    *ememp = *lmemp;
569                 if ((r == r_sup - 1) & bool_nsent)
                      {
571                     bool_nsent = 0;
                        write_pipe_one_element(st0_st1_write, 0);
573 #ifdef DEBUG
                        printf("Write to st0_st1 pipe\n");
575 #endif

577                   }

579 #ifdef DEBUG
                    printf("%d (add = %x) <= %d (add = %x)\n", *ememp, ememp, *lmemp, lmemp);
581 #endif
                  }
583 #ifdef DEBUG
              printf ("Storing c0[%d][%d]\n", ii, jj);
585           for (dbg0 = 0; dbg0 < min_ii; dbg0++)
                {
587               for (dbg1 = 0; dbg1 < min_jj; dbg1++)
                    {
589                   printf("%d ", *(c + (ii + dbg0) * m1 + dbg1 + jj));
                    }
591               printf("\n");
                }
593 #endif
            }
595       else
            {
597
              write_pipe_one_element(st0_st1_write, 0);
599 #ifdef DEBUG
              printf("Write to st0_st1\n");
601 #endif

603         }
          if (kk + block_kk < m0) kk+= block_kk;
605       else {
            kk = 0;
607         if (jj + block_jj < m1) jj+= block_jj;
            else {
609           jj = 0;
              if (ii + 2 * block_ii < n0) ii+= 2 * block_ii;
611         }
          }
613     }
    return 0;
615 }

617 int matrix_multiply_hw_st1 (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii, int block_jj,
        int block_kk,  int iter_space, int db_iter)
    {
619   int i, j, k, ii, jj, kk, t, r, s;
      int min_ii, min_jj, min_kk;
621   int offset_bufflocal, offset_ext_memp;
      int* lmemp, * ememp;
623   int r_sup;
      int dummy_read = 0;
625   int dummy_read1 = 0;
      int tmp;
627   int bool_nsent = 1;
      int* st0_st1_read = &st0_st1[0];
629   int* c01_st1_read = &c01_st1[0];
      int* st1_buff0_write = &st1_buff0[1];
631   int min_iii;
      ii = 0;
633   jj = 0;
      kk = 0;
635
      write_pipe_one_element(st1_buff0_write, 0);
637 #ifdef DEBUG
      printf("Write to st1_buff0 pipe\n");
639 #endif

641   for (t = 0; t < iter_space; t+= db_iter)
        {
643
          min_ii = min(block_ii, n0 - ii);
645       min_jj = min(block_jj, m1 - jj);
```

```
        min_kk = min(block_kk, m0 - kk);
647     i = 0;
        j = 0;
649     k = 0;
        s = 0;
651     //store st1 buffer
        i = block_ii;
653     j = 0;
        dummy_read += read_pipe_one_element (st0_st1_read);
655 #ifdef DEBUG
        printf("Read from st0_st1 pipe\n");
657 #endif

659     dummy_read1 += read_pipe_one_element (c01_st1_read);
    #ifdef DEBUG
661     printf("Read from c01_st1 pipe\n");
    #endif
663
        if ((kk + block_kk >= m0) && (ii + i < n0))
665       {
            min_iii = min(min_ii, n0 - (ii + block_ii));
667         r_sup = min_iii * min_jj;
            bool_nsent = 1;
669         for (r = 0, tmp = dummy_read + dummy_read1; r < r_sup; r++)
              {
671             offset_bufflocal = i * block_jj + j;
                offset_ext_memp = (ii + i) * m1 + j + jj;
673             if (j < min_jj - 1) j++;
                else{
675               j = 0;
                  if ((i < block_ii + min_ii - 1) && (ii + i < n0 - 1)) i++;
677               else {i = 0; s++;}
                }
679             lmemp = c_tmp + offset_bufflocal;
                ememp = c + offset_ext_memp;
681             /*transfer code */
                *ememp = *lmemp;
683             if ((r == r_sup - 1) & bool_nsent)
                  {
685                 bool_nsent = 0;
                    write_pipe_one_element(st1_buff0_write, 0);
687 #ifdef DEBUG
                    printf("Write to st1_buff0 pipe\n");
689 #endif

691                 }

693           }
    #ifdef DEBUG
695         printf ("Storing c1[%d][%d]\n", ii + block_ii, jj);
            for (dbg0 = block_ii; dbg0 < block_ii + min_ii; dbg0++)
697           {
              for (dbg1 = 0; dbg1 < min_jj; dbg1++)
699             {
                printf("%d", *(c + (ii + dbg0) * m1 + dbg1 + jj));
701             printf("(%d) ", *(c_tmp + (dbg0 * block_jj + dbg1)));
              }
703           printf("\n");
            }
705 #endif
        }
707     else
        {
709       write_pipe_one_element(st1_buff0_write, 0);
    #ifdef DEBUG
711         printf("Write to st1_buff0 pipe in else\n");
    #endif
713     }
        if (kk + block_kk < m0) kk+= block_kk;
715     else {
          kk = 0;
717       if (jj + block_jj < m1) jj+= block_jj;
          else {
719         jj = 0;
            if (ii + 2 * block_ii < n0) ii+= 2 * block_ii;
721       }
        }
723   }
  return 0;
725 }

727 int matrix_multiply_hw_c01 (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int block_jj,
        int block_kk, int iter_space, int db_iter)
    {
729   int i, j, k, ii, jj, kk, t, r, s;
      int min_ii, min_jj, min_kk;
731   int offset_bufflocal, offset_ext_memp;
      int* lmemp, * ememp;
```

```
733   int dummy_read;
      int tmp = 0;
735   int* c01_st0_write = &c01_st0[1];
      int* c01_st1_write = &c01_st1[1];
737   int* buff01_c01_read = &buff01_c01[0];
      int cswitch;
739   int iter_int ;
      ii = 0;
741   jj = 0;
      kk = 0;
743   cswitch = 0;

745   for (t = 0; t < iter_space * 2; t+=db_iter)
        {
747
    #ifdef DEBUG
749       printf(" ----------------------------------------------------------------------------------------\n");
          printf("II=%d, JJ=%d, KK=%d, cswitch=%d\n", ii, jj, kk, cswitch);
751 #endif
          min_ii = min(block_ii, n0 - ii);
753       min_jj = min(block_jj, m1 - jj);
          min_kk = min(block_kk, m0 - kk);
755       i = 0;
          j = 0;
757       k = 0;
          s = 0;
759
          dummy_read = read_pipe_one_element (buff01_c01_read);
761 #ifdef DEBUG
          printf("Read from buff01_c01 pipe = %d\n", dummy_read);
763 #endif

765       if (dummy_read == 0)
            {
767           a_tmp = a_tmp_0;
              b_tmp = b_tmp_0;
769
            }
771       else
            {
773           a_tmp = a_tmp_1;
              b_tmp = b_tmp_1;
775
            }
777       i = 0; j = 0; k = 0;
          for (r = 0; r < min_ii * min_jj * min_kk; r++)
779         {
              if ( ii + i < n0 - block_ii){
781
                if (k == 0) lc_tmp = 0;
783             else  lc_tmp += *(a_tmp_0 + (i * block_kk + k)) * *(b_tmp_0 + (k * block_jj + j));

785             if (k == min_kk - 1)
                  {
787                 *(c_tmp_0_w + (i * block_jj + j))  = ((kk == 0) && (k == 0)) ? lc_tmp : *(c_tmp_0_r + (i * block_jj + j)) + lc_tmp;
                  }
789             if (k < min_kk - 1) k++;
                else {
791               k = 0;
                  if (j < min_jj - 1) j++;
793               else {
                    j = 0;
795                 i++;
                  }
797             }
              }
799         }

801       if (dummy_read == 0)
            {
803           write_pipe_one_element (c01_st0_write, tmp);
    #ifdef DEBUG
805           printf("Write to c01_st0 pipe\n");
    #endif
807
            }
809       else
            {
811           write_pipe_one_element (c01_st1_write, tmp);
    #ifdef DEBUG
813           printf("Write c01_st1 pipe\n");
    #endif
815         }

817
          if (cswitch < 1) cswitch++;
819       else {
            cswitch = 0;
```

```
821           if (kk + block_kk < m0) kk+= block_kk;
              else {
823             kk = 0;
                if (jj + block_jj < m1) jj+= block_jj;
825             else {
                  jj = 0;
827               if (ii + 2 * block_ii < n0) ii+= 2 * block_ii;
                }
829           }
            }
831       }
      return 0;
833   }

835   int matrix_multiply_hw_buff0_process (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int
          block_jj,int block_kk, int iter_space, int db_iter)
      {
837     int childpid;
        if ((childpid = fork()) == -1)
839       {
          perror("fork");
841       exit(1);
        }
843     if (childpid == 0)
        {
845       /* child process executes buff0 and exits immediatly */
          close (buff0_buff1[0]);
847       close (buff01_c01[0]);
          close (st1_buff0[1]);
849
          matrix_multiply_hw_buff0 (a, b, c, n0, m0, m1, block_ii, block_jj, block_kk, iter_space, db_iter);
851       exit (0);
        }
853     return 0;
      }
855   int matrix_multiply_hw_buff1_process (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int
          block_jj,int block_kk, int iter_space, int db_iter)
      {
857     int childpid;
        if ((childpid = fork()) == -1)
859       {
          perror("fork");
861       exit(1);
        }
863     if (childpid == 0)
        {
865       /* child process executes buff0 and exits immediatly */
          close (buff1_st0[0]);
867       close (buff01_c01[0]);
          close (buff0_buff1[1]);
869
          matrix_multiply_hw_buff1 (a, b, c, n0, m0, m1, block_ii, block_jj, block_kk, iter_space, db_iter);
871       exit(0);
        }
873     return 0;
      }
875
      int matrix_multiply_hw_st0_process (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int
          block_jj,int block_kk, int iter_space, int db_iter)
877   {
        int childpid;
879     if ((childpid = fork()) == -1)
        {
881       perror("fork");
          exit(1);
883     }
        if (childpid == 0)
885     {
          /* child process executes buff0 and exits immediatly */
887       close (buff1_st0[1]);
          close (c01_st0[1]);
889       close (st0_st1[0]);
          matrix_multiply_hw_st0 (a, b, c, n0, m0, m1, block_ii, block_jj, block_kk, iter_space, db_iter);
891       exit(0);
        }
893     return 0;
      }
895
      int matrix_multiply_hw_st1_process (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int
          block_jj,int block_kk, int iter_space, int db_iter)
897   {
        int childpid;
899     if ((childpid = fork()) == -1)
        {
901       perror("fork");
          exit(1);
903     }
        if (childpid == 0)
```

```
905     {
          /* child process executes buff0 and exits immediatly */
907       close (st0_st1[1]);
          close (c01_st1[1]);
909       close (st1_buff0[0]);
          matrix_multiply_hw_st1(a, b, c, n0, m0, m1, block_ii, block_jj, block_kk, iter_space, db_iter);
911       exit(0);
        }
913   return 0;
    }
915
    int matrix_multiply_hw_c01_process(int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int
          block_jj,int block_kk, int iter_space, int db_iter)
917 {
      int childpid;
919   if ((childpid = fork()) == -1)
        {
921       perror("fork");
          exit(1);
923     }
      if (childpid == 0)
925     {
          /* child process executes buff0 and exits immediatly */
927       close (buff01_c01[1]);
          close (c01_st0[0]);
929       close (c01_st1[0]);
          matrix_multiply_hw_c01(a, b, c, n0, m0, m1, block_ii, block_jj, block_kk, iter_space, db_iter);
931       exit(0);
        }
933   return 0;
    }
935
    int matrix_multiply_hw_modul(int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int block_jj
          ,int block_kk)
937 {
      int iter_space;
939   int db_iter;
      int status;
941   pid_t done;
      iter_space = (int)(ceil(((double) n0) / (block_ii * 2)) * ceil( ((double) m1) / block_jj) * ceil( ((double) m0) / block_kk));
943
      db_iter = 1;
945 #ifdef DEBUG
      printf("\nIter space = %d, db_iter = %d\n", iter_space, db_iter);
947 #endif

949   /* Call accelerators */
      /* buff0 */
951   matrix_multiply_hw_buff0_process(a, b, c, n0, m0, m1, block_ii, block_jj, block_kk, iter_space, db_iter);
      /* buff1 */
953   matrix_multiply_hw_buff1_process(a, b, c, n0, m0, m1, block_ii, block_jj, block_kk, iter_space, db_iter);
      /* c01 */
955   matrix_multiply_hw_c01_process(a, b, c, n0, m0, m1, block_ii, block_jj, block_kk, iter_space, db_iter);
      /* st0 */
957   matrix_multiply_hw_st0_process(a, b, c, n0, m0, m1, block_ii, block_jj, block_kk, iter_space, db_iter);
      /* st1 */
959
      matrix_multiply_hw_st1_process(a, b, c, n0, m0, m1, block_ii, block_jj, block_kk, iter_space, db_iter);
961
      while (1){
963     if ((done = wait(&status)) == -1)
          {
965       if (errno == ECHILD) break; /* no more child processes */
          }
967     else
          {
969       if (!WIFEXITED(status) || WEXITSTATUS(status) != 0)
            {
971           printf("error status on exit of the child %d", done);
              exit(1);
973         }
          }
975   }

977   return 0;

979 }

981
    int matrix_multiply_hw(int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int block_jj,int
          block_kk)
983 {
      int i, j, k, ii, jj, kk, t, r, s;
985   int min_ii, min_jj, min_kk;
      int offset_bufflocal, offset_ext_memp;
987   int* lmemp, * ememp;
      int iter_space;
989   int db_iter;
```

```
  int min_iii;
991  ii = 0;
  jj = 0;
993  kk = 0;
  iter_space = (int) (ceil(((double) n0) / (block_ii * 2)) * ceil( ((double) m1) / block_jj) * ceil( ((double) m0) / block_kk));
995
#ifdef DEBUG
997   printf("\nIter space = %d\n", iter_space);
#endif
999
  db_iter = 1;
1001  for (t = 0; t < iter_space; t+= db_iter)
    {
1003
      min_ii = min(block_ii, n0 - ii);
1005  min_jj = min(block_jj, m1 - jj);
      min_kk = min(block_kk, m0 - kk);
1007  i = 0;
      j = 0;
1009  k = 0;
      s = 0;
1011 #ifdef DEBUG
      printf ("Buff0\n");
1013 #endif
      for (r = 0; r < min_ii * min_kk + min_jj * min_kk; r++)
1015    {
#ifdef DEBUG
1017        printf("i = %d, j = %d, k = %d, ii = %d, jj = %d, kk = %d, s = %d\n", i, j, k, ii, jj, kk, s);
#endif
1019        if (s == 0){
             offset_bufflocal = i * block_kk + k;
1021        offset_ext_memp = (ii + i) * m0 + k + kk;

1023        if (k < min_kk - 1) k++;
            else{
1025          k = 0;
              if (i < min_ii - 1) i++;
1027          else {i = 0; s++;}
            }
1029        lmemp = a_tmp_0 + offset_bufflocal;
            ememp = a + offset_ext_memp;
1031

1033        }else {
              if (s == 1){
1035            offset_bufflocal = k * block_jj + j;
                offset_ext_memp = (kk + k) * m1 + j + jj;
1037
                if (j < min_jj - 1) j++;
1039            else{
                  j = 0;
                  if (k < min_kk - 1) k++;
1041              else {k = 0; s = 0;}
                }
1043            lmemp = b_tmp_0 + offset_bufflocal;
                ememp = b + offset_ext_memp;
1045


1047

1049          }
            }
1051        /*transfer code */
            *lmemp = *ememp;
1053      }
#ifdef DEBUG
1055    printf("Buff0 transfer:\n");
      printf("A_tmp_0\n");
1057  for (dbg0 = 0; dbg0 < min_ii; dbg0++)
      {
1059    for (dbg1 = 0; dbg1 < min_kk; dbg1++)
          printf("%d ", *(a_tmp_0 + (dbg0 * block_kk + dbg1)));
1061    printf("\n");
      }
1063  printf("B_tmp_0\n");
      for (dbg1 = 0; dbg1 < min_kk; dbg1++)
1065    {
        for (dbg0 = 0; dbg0 < min_jj; dbg0++)
1067      {
            printf("%d ", *(b_tmp_0 + (dbg1 * block_jj + dbg0)));
1069
          }
1071    printf("\n");
      }
1073
      printf("Buff1\n");
1075 #endif
      if (ii + block_ii < n0) {
1077    i = block_ii;
```

```
        for (r = 0; r < min_ii * min_kk + min_jj * min_kk; r++)
1079      {
#ifdef DEBUG
1081        printf("i = %d, j = %d, k = %d, ii = %d, jj = %d, kk = %d, s = %d\n", i, j, k, ii, jj, kk, s);
#endif
1083        if (s == 0){
             offset_bufflocal = (i - block_ii) * block_kk + k;
1085          offset_ext_memp = (ii + i) * m0 + k + kk;
            if (k < min_kk - 1) k++;
1087        else{
              k = 0;
1089          if (i < min_ii + block_ii - 1) i++;
            else {i = 0; s++;}
1091        }
            lmemp = a_tmp_1 + offset_bufflocal;
1093        ememp = a + offset_ext_memp;

1095        }else if (s == 1){
             offset_bufflocal = k * block_jj + j;
1097          offset_ext_memp = (k + kk) * m1 + j + jj;
            if (j < min_jj - 1) j++;
1099        else{
              j = 0;
1101          if (k < min_kk - 1) k++;
            else {k = 0; s = 0;}
1103        }
            lmemp = b_tmp_1 + offset_bufflocal;
1105        ememp = b + offset_ext_memp;
          }
1107      /*transfer code */
          *lmemp = *ememp;
1109      }
#ifdef DEBUG
1111    printf("Buff1 transfer:\n");
        printf("A_tmp_1\n");
1113    for (dbg0 = 0; dbg0 < min_ii; dbg0++)
          {
1115        for (dbg1 = 0; dbg1 < min_kk; dbg1++)
              printf("%d ", *(a_tmp_1 + (dbg0 * block_kk + dbg1)));
1117        printf("\n");
          }
1119    printf("B_tmp_1\n");
        for (dbg1 = 0; dbg1 < min_kk; dbg1++)
1121      {
            for (dbg0 = 0; dbg0 < min_jj; dbg0++)
1123          {
                printf("%d ", *(b_tmp_1 + (dbg1 * block_jj + dbg0)));
1125

              }
1127        printf("\n");
          }
1129 #endif
      }
1131 /*      //perform computation on buff0 */
      i = 0; j = 0; k = 0;
1133  for (r = 0; r < min_ii * min_jj * min_kk; r++)
        {
1135      if ((kk == 0) && (k == 0)) *(c_tmp + (i * block_jj + j)) = *(a_tmp_0 + (i * block_kk + k)) * *(b_tmp_0 + (k * block_jj + j));
          else *(c_tmp + (i * block_jj + j)) += *(a_tmp_0 + (i * block_kk + k)) * *(b_tmp_0 + (k * block_jj + j));
1137 #ifdef DEBUG
          printf("Compute c_tmp (%d) += a_tmp_0 (%d) * b_tmp_0 (%d) \t| i = %d, j = %d, k = %d\n",*(c_tmp + (i * block_jj + j)), *(a_tmp_0 +
              (i * block_kk + k)), *(b_tmp_0 + (k * block_jj + j)), i, j, k);
1139 #endif

1141      if ( k < min_kk - 1) k++;
          else {
1143        k = 0;
            if (j < min_jj - 1) j++;
1145        else {
              j = 0;
1147          i++;
            }
1149      }
        }
1151

      //store st0 buffer
1153  i = 0;
      j = 0;
1155
      if (kk + block_kk >= m0)
1157    {
        for (r = 0; r < min_ii * min_jj; r++)
          {
1159

1161        offset_bufflocal = i * block_jj + j;
            offset_ext_memp = (ii + i) * m1 + j + jj;
1163 #ifdef DEBUG
            printf("Store st0 j = %d, i = %d\n", j, i );
```

```
#endif
                if (j < min_jj - 1) j++;
                else{
                  j = 0;
                  if (i < min_ii - 1) i++;
                  else {i = 0; s++;}
                }
                lmemp = c_tmp + offset_bufflocal;
                ememp = c + offset_ext_memp;

                /*transfer code */
                *ememp = *lmemp;
#ifdef DEBUG
                printf("%d (add = %x) <= %d (add = %x)\n", *ememp, ememp, *lmemp, lmemp);
#endif
              }
#ifdef DEBUG
            printf ("Storing c0[%d][%d]\n", ii, jj);
            for (dbg0 = 0; dbg0 < min_ii; dbg0++)
              {
                for (dbg1 = 0; dbg1 < min_jj; dbg1++)
                  {
                    printf("%d ", *(c + (ii + dbg0) * m1 + dbg1 + jj));
                  }
                printf("\n");
              }
#endif
          }

        //perform computation on buff1
        i = block_ii; j = 0; k = 0;
        for (r = 0; r < min_ii * min_jj * min_kk; r++)
          {
            if (ii + i < n0)
              {
                if ((kk == 0)&&(k == 0)) *(c_tmp + (i * block_jj + j)) = 0;
                *(c_tmp + (i * block_jj + j)) += *(a_tmp_1 + ((i - block_ii) * block_kk + k)) * *(b_tmp_1 + (k * block_jj + j));
#ifdef DEBUG
                printf("Compute c_tmp (%d) += a_tmp_1 (%d) * b_tmp_1 (%d) \t| i = %d, j = %d, k = %d\n",*(c_tmp + (i * block_jj + j)), *(
                      a_tmp_1 + ((i - block_ii) * block_kk + k)), *(b_tmp_1 + (k * block_jj + j)), i, j, k);
#endif
              }
            if (k < min_kk - 1) k++;
            else {
              k = 0;
              if (j < min_jj - 1) j++;
              else {
                j = 0;
                i++;
              }
            }
          }


        i = block_ii;
        j = 0;
        if (kk + block_kk >= m0)
          {
            min_iii = min(min_ii, n0 - (ii + block_ii));
            for (r = 0; r < min_iii * min_jj; r++)
              {
                offset_bufflocal = i * block_jj + j;
                offset_ext_memp = (ii + i) * m1 + j + jj;
                if (j < min_jj - 1) j++;
                else{
                  j = 0;
                  if ((i < block_ii + min_ii - 1) && (ii + i < n0 - 1)) i++;
                  else {i = 0; s++;}
                }
                lmemp = c_tmp + offset_bufflocal;
                ememp = c + offset_ext_memp;
                /*transfer code */
                *ememp = *lmemp;
              }
#ifdef DEBUG
            printf ("Storing c1[%d][%d]\n", ii + block_ii, jj);
            for (dbg0 = block_ii; dbg0 < block_ii + min_ii; dbg0++)
              {
                for (dbg1 = 0; dbg1 < min_jj; dbg1++)
                  {
                    printf("%d", *(c + (ii + dbg0) * m1 + dbg1 + jj));
                    printf("(%d) ", *(c_tmp + (dbg0 * block_jj + dbg1)));
                  }
                printf("\n");
              }
#endif
          }
        if (kk + block_kk < m0) kk+= block_kk;
```

```
        else {
          kk = 0;
          if (jj + block_jj < m1) jj+= block_jj;
          else {
            jj = 0;
            if (ii + 2 * block_ii < n0) ii+= 2 * block_ii;
          }
        }


      }
    return 0;
  }

  int matrix_multiply_hw_orig (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1)
  {
    int i, j, k;
    int tmp;


    for (i = 0; i < n0; i++)
      for (j = 0; j < m1; j++)
        {
          tmp = 0;
          for (k = 0; k < m0; k++)
            {
              tmp += *(a + (i * m0 + k)) * *(b + (k * m1 + j));

            }
          *(c + (i * m1 + j)) = tmp;
        }
    return 0;
  }


  /* int a_tmp[BL_II_MAX * BL_KK_MAX]; */
  /* int b_tmp[]; */
  /* int a_tmp_0[]; */
  /* int b_tmp_0[]; */
  /* int a_tmp_1[BL_II_MAX * BL_KK_MAX]; */
  /* int b_tmp_1[BL_JJ_MAX * BL_KK_MAX]; */
  /* int c_tmp[]; */

  int alloc_shmem(int* sid_a_tmp,int* sid_b_tmp,int* sid_a_tmp_0,int* sid_b_tmp_0, int* sid_a_tmp_1, int* sid_b_tmp_1, int* sid_c_tmp, int*
        sid_a, int* sid_b, int* sid_c)
  {
    int errsv;
    int size_a_tmp = BL_II_MAX * BL_KK_MAX;
    int size_b_tmp = BL_JJ_MAX * BL_KK_MAX;
    int size_a_tmp_0 = BL_II_MAX * BL_KK_MAX;
    int size_b_tmp_0 = BL_JJ_MAX * BL_KK_MAX;
    int size_a_tmp_1 = BL_II_MAX * BL_KK_MAX;
    int size_b_tmp_1 = BL_II_MAX * BL_KK_MAX;
    int size_c_tmp = BL_II_MAX * BL_JJ_MAX * 2;
    int size_a = MAXN0 * MAXM0;
    int size_b = MAXM0 * MAXM1;
    int size_c = MAXN0 * MAXM1;
    /* create */

    *sid_a_tmp = shmget(IPC_PRIVATE, size_a_tmp, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR | S_IRGRP);
    if (*sid_a_tmp == -1)
      {
        perror("Creating␣a_tmp␣shared␣memory␣failed\n");
        exit(EXIT_FAILURE);
      }

    *sid_b_tmp = shmget(IPC_PRIVATE, size_b_tmp, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR | S_IRGRP);
    if (*sid_b_tmp == -1)
      {
        perror("Creating␣a_tmp␣shared␣memory␣failed\n");
        exit(EXIT_FAILURE);
      }
    *sid_a_tmp_0 = shmget(IPC_PRIVATE, size_a_tmp_0, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR | S_IRGRP);
    if (*sid_a_tmp_0 == -1)
      {
        perror("Creating␣a_tmp␣shared␣memory␣failed\n");
        exit(EXIT_FAILURE);
      }
    *sid_b_tmp_0 = shmget(IPC_PRIVATE, size_b_tmp_0, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR | S_IRGRP);
    if (*sid_b_tmp_0 == -1)
      {
        perror("Creating␣a_tmp␣shared␣memory␣failed\n");
        exit(EXIT_FAILURE);
      }
    *sid_a_tmp_1 = shmget(IPC_PRIVATE, size_a_tmp_1, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR | S_IRGRP);
    if (*sid_a_tmp_1 == -1)
      {
        perror("Creating␣a_tmp␣shared␣memory␣failed\n");
```

```
1339        exit(EXIT_FAILURE);
          }
1341      *sid_b_tmp_1 = shmget(IPC_PRIVATE, size_b_tmp_1, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR | S_IRGRP);
          if (*sid_b_tmp_1 == -1)
1343        {
            perror("Creating a_tmp shared memory failed\n");
1345        exit(EXIT_FAILURE);
          }
1347      *sid_c_tmp = shmget(IPC_PRIVATE, size_c_tmp, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR | S_IRGRP);
          if (*sid_c_tmp == -1)
1349        {
            perror("Creating a_tmp shared memory failed\n");
1351        exit(EXIT_FAILURE);
          }
1353      *sid_a = shmget(IPC_PRIVATE, size_a, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR | S_IRGRP);
          if (*sid_c_tmp == -1)
1355        {
            perror("Creating a shared memory failed\n");
1357        exit(EXIT_FAILURE);
          }
1359      *sid_b = shmget(IPC_PRIVATE, size_b, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR | S_IRGRP);
          if (*sid_c_tmp == -1)
1361        {
            perror("Creating a shared memory failed\n");
1363        exit(EXIT_FAILURE);
          }
1365      *sid_c = shmget(IPC_PRIVATE, size_c, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR | S_IRGRP);
          if (*sid_c_tmp == -1)
1367        {
            perror("Creating a shared memory failed\n");
1369        exit(EXIT_FAILURE);
          }
1371

1373      /* attach */
          a_tmp = (int *) shmat(*sid_a_tmp, 0, 0);
1375      if (a_tmp == -1)
            {
1377          perror("Error attaching a_tmp\n");
            exit(EXIT_FAILURE);
1379        }
          b_tmp = (int *) shmat(*sid_b_tmp, 0, 0);
1381      if (b_tmp == -1)
            {
1383          perror("Error attaching b_tmp\n");
            exit(EXIT_FAILURE);
1385        }
          a_tmp_0 = (int *) shmat(*sid_a_tmp_0, 0, 0);
1387      if (a_tmp_0 == -1)
            {
1389          perror("Error attaching a_tmp_0\n");
            exit(EXIT_FAILURE);
1391        }
          b_tmp_0 = (int *) shmat(*sid_b_tmp_0, 0, 0);
1393      if (b_tmp_0 == -1)
            {
1395          perror("Error attaching b_tmp_0\n");
            exit(EXIT_FAILURE);
1397        }
          a_tmp_1 = (int *) shmat(*sid_a_tmp, 0, 0);
1399      if (a_tmp_1 == -1)
            {
1401          perror("Error attaching a_tmp_1\n");
            exit(EXIT_FAILURE);
1403        }
          b_tmp_1 = (int *) shmat(*sid_b_tmp, 0, 0);
1405      if (b_tmp_1 == -1)
            {
1407          perror("Error attaching b_tmp_1\n");
            exit(EXIT_FAILURE);
1409        }
          c_tmp = (int *) shmat(*sid_c_tmp, 0, 0);
1411      if (c_tmp == -1)
            {
1413          perror("Error attaching c_tmp\n");
            exit(EXIT_FAILURE);
1415        }
          a = (int *) shmat(*sid_a, 0, 0);
1417      if (a == -1)
            {
1419          perror("Error attaching a\n");
            exit(EXIT_FAILURE);
1421        }
          b = (int *) shmat(*sid_b, 0, 0);
1423      if (b == -1)
            {
1425          perror("Error attaching b\n");
            exit(EXIT_FAILURE);
```

```
1427      }
      c = (int *) shmat(*sid_c, 0, 0);
1429    if (c == -1)
          {
1431        perror("Error attaching c\n");
            exit(EXIT_FAILURE);
1433      }

1435    return 0;
      }
1437
      int dealloc_shmem(int sid_a_tmp,int sid_b_tmp,int sid_a_tmp_0,int sid_b_tmp_0, int sid_a_tmp_1, int sid_b_tmp_1, int sid_c_tmp, int sid_a, int
            sid_b, int sid_c)
1439  {
        /* Detach first */
1441    if (shmdt(a_tmp) == -1)
          {
1443        perror("Error detaching a_tmp");
            exit(EXIT_FAILURE);
1445      }
        if (shmdt(b_tmp) == -1)
1447      {
            perror("Error detaching b_tmp");
1449        exit(EXIT_FAILURE);
          }
1451    if(shmdt(a_tmp_0) == -1)
          {
1453        perror("Error detaching a_tmp_0");
            exit(EXIT_FAILURE);
1455      }
        if(shmdt(b_tmp_0) == -1)
1457      {
            perror("Error detaching b_tmp_0");
1459        exit(EXIT_FAILURE);
          }
1461    if(shmdt(a_tmp_1) == -1)
          {
1463        perror("Error detaching a_tmp_1");
            exit(EXIT_FAILURE);
1465      }
        if(shmdt(b_tmp_1) == -1)
1467      {
            perror("Error detaching b_tmp_1");
1469        exit(EXIT_FAILURE);
          }
1471    if(shmdt(c_tmp) == -1)
          {
1473        perror("Error detaching c_tmp");
            exit(EXIT_FAILURE);
1475      }
        if(shmdt(a) == -1)
1477      {
            perror("Error detaching a");
1479        exit(EXIT_FAILURE);
          }
1481    if(shmdt(b) == -1)
          {
1483        perror("Error detaching b");
            exit(EXIT_FAILURE);
1485      }

1487    if(shmdt(c) == -1)
          {
1489        perror("Error detaching c");
            exit(EXIT_FAILURE);
1491      }

1493    /* Dealoc */
        if (shmctl (sid_a_tmp, IPC_RMID, 0) == -1)
1495      {
            printf("Identifier error %d\n", sid_a_tmp);
1497        perror("Error destroing a_tmp");
            exit(EXIT_FAILURE);
1499      }
        if(shmctl (sid_b_tmp, IPC_RMID, 0) == -1)
1501      {
            perror("Error destroing b_tmp");
1503        exit(EXIT_FAILURE);
          }
1505    if(shmctl (sid_a_tmp_0, IPC_RMID, 0) == -1)
          {
1507        perror("Error destroing a_tmp_0");
            exit(EXIT_FAILURE);
1509      }
        if(shmctl (sid_b_tmp_0, IPC_RMID, 0) == -1)
1511      {
            perror("Error destroing b_tmp_0");
1513        exit(EXIT_FAILURE);
```

```
    }
  if (shmctl (sid_a_tmp_1, IPC_RMID, 0) == -1)
    {
      perror("Error destroing a_tmp_1");
      exit(EXIT_FAILURE);
    }
  if(shmctl (sid_b_tmp_1, IPC_RMID, 0) == -1)
    {
      perror("Error destroing b_tmp_1");
      exit(EXIT_FAILURE);
    }
  if(shmctl (sid_c_tmp, IPC_RMID, 0) == -1)
    {
      perror("Error destroing c_tmp");
      exit(EXIT_FAILURE);
    }
  if(shmctl (sid_a, IPC_RMID, 0) == -1)
    {
      perror("Error destroing a");
      exit(EXIT_FAILURE);
    }
  if(shmctl (sid_b, IPC_RMID, 0) == -1)
    {
      perror("Error destroing b");
      exit(EXIT_FAILURE);
    }
  if(shmctl (sid_c, IPC_RMID, 0) == -1)
    {
      perror("Error destroing c");
      exit(EXIT_FAILURE);
    }

  return 0;

}
int main()
{
  int sid_a_tmp, sid_b_tmp, sid_a_tmp_0, sid_b_tmp_0, sid_a_tmp_1, sid_b_tmp_1, sid_c_tmp, sid_a, sid_b, sid_c;
  int error_found = 0;
  int i,j,k;
  int n0, m0, n1, m1;
  int block_ii, block_jj, block_kk;
  int tmp;
  int prcts = 0;
  /* pipe */
  pipe (st1_buff0);
  pipe (buff0_buff1);
  pipe (buff1_st0);
  pipe (st0_st1);
  pipe (c01_st0);
  pipe (c01_st1);
  pipe (buff01_c01);
  /* shared memory */
  alloc_shmem( &sid_a_tmp, &sid_b_tmp, &sid_a_tmp_0, &sid_b_tmp_0, &sid_a_tmp_1, &sid_b_tmp_1, &sid_c_tmp, &sid_a, &sid_b, &sid_c);

  n0 = MAXN0;
  m0 = MAXM0;
  n1 = MAXM0;
  m1 = MAXM1;
  block_ii = BL_II;
  block_jj = BL_JJ;
  block_kk = BL_KK;
  printf ("Main\n");

  for (m0 = 1; m0 <= MAXM0; m0++)
    for (m1 = 1; m1 <= MAXM1; m1++)
      for (n0 = 1; n0 <= MAXN0; n0++)
        {

          printf("%d percents done\n", (int)( prcts * 100.0 / (MAXN0 * MAXM0 * MAXM1)));
          prcts++;

          printf("\n ================================================================================\ n");
          printf("Testing with n0 = %d, m0 = %d, m1 = %d\n", n0, m0, m1);

          n1 = m0;
#ifdef DEBUG
          printf ("matrix A\n");
#endif
          for (i = 0; i < n0; i++){
            for (j = 0; j < m0; j++)
              {
                *(a + i * m0 + j) = i * m0 + j + 1;
#ifdef DEBUG
                printf ("%d\t",*(a + i * m0 + j));
#endif
              }
#ifdef DEBUG
```

```
            printf("\n");
#endif
          }
#ifdef DEBUG
          printf( "matrix B\n");
#endif
          for (i = 0; i < m0; i++){
            for (j = 0; j < m1; j++)
              {
                *(b + i * m1 + j) = i * m1 + j + 1;
#ifdef DEBUG
                printf("%d\t",  *(b + (i * m1 + j)));
#endif

              }
#ifdef DEBUG
            printf("\n");
#endif
          }

          for (i = 0; i < n0; i++)
            for (j = 0; j < m1; j++)
              c[i * m1 + j] = 0;

          for (i = 0; i < n0; i++)
            for (j = 0; j < m1; j++)
              {
                c_test[i * m1 + j] = 0;
                for (k = 0; k < m0; k++)
                  c_test[i * m1 + j] += a[i * m0 + k] * b[k * m1 + j];
              }
#ifdef DEBUG
          printf("matrix C\n");
          for (i = 0; i < n0; i++){
            for (j = 0; j < m1; j++)
              printf("%d\t", c_test[i * m1 + j]);
            printf("\n");
          }

          printf("Testing original version\n");
#endif /*DEBUG*/
          matrix_multiply_hw_orig ((int *) a, (int *) b, (int *)c, n0, m0, m1);

          for (i = 0; i < n0; i++)
            for ( j =0; j < m1; j++)
              {
                if ( c_test[i * m1 + j] != c[i * m1 + j]) printf ("Error on c_test[%d][%d] = %d != %d = c[%d][%d]\n", i,j,c_test[i * m1 + j],c
                    [i * m1 + j], i , j);

              }
          for ( block_ii = 1; block_ii <= n0; block_ii++)
            for ( block_jj = 1; block_jj <= m0; block_jj++)
              for (block_kk = 1; block_kk <= m1; block_kk++)
                {

#ifdef DEBUG
                  printf("\n block_ii = %d, block_jj = %d, block_kk = %d\n\n", block_ii , block_jj , block_kk);


                  printf("Testing transformed version\n");
#endif

                  for (i = 0; i < n0; i++)
                    for (j = 0; j < m1; j++)
                      c[i * m1 + j] = 0;
                  //          matrix_multiply_hw_inter_1  ((int *) a, (int *) b, (int *)c, n0, m0, m1, block_ii , block_jj , block_kk );

                  //matrix_multiply_hw ((int *) a, (int *) b, (int *)c, n0, m0, m1, block_ii , block_jj , block_kk );
                  matrix_multiply_hw_modul ((int *) a, (int *) b, (int *)c, n0, m0, m1, block_ii , block_jj , block_kk);
#ifdef DEBUG
                  printf("matrix C (computed)\n");
                  printf("x\t");
                  for (j = 0; j < m1; j++)
                    printf("%d\t", j);
                  printf("\n");
                  for (j = 0; j < m1 + 1; j++)
                    printf("-\t");
                  printf("\n");
                  for (i = 0; i < n0; i++){
                    printf("%d |\t", i);
                    for (j = 0; j < m1; j++)
                      printf("%d\t", c[i * m1 + j]);
                    printf("\n");
                  }
#endif /*DEBUG*/
```

```
1689                    for (i = 0; i < n0; i++)
                          for ( j =0; j < m1; j++)
1691                        {
                              if ( c_test[i * m1 + j] != c[i * m1 + j])
1693                            {
                                  printf ("Error on c_test[%d][%d] = %d != %d = c[%d][%d] (add = %x)\n", i,j,c_test[i * m1 + j],c[i * m1 + j], i , j
                                        , (c + i * m1 + j));
1695                              printf ("Error found on block_ii = %d, block_jj = %d, block_kk = %d\n", block_ii, block_jj, block_kk);
                                  printf("and with n0 = %d, m0 = %d, m1 = %d\n\n", n0, m0, m1);
1697                              error_found = 1;
                                }
1699
                            }
1701 #ifdef DEBUG
                        printf("\nFinish testing with block_ii = %d, block_jj = %d, block_kk = %d\n", block_ii, block_jj, block_kk);
1703                    printf("and with n0 = %d, m0 = %d, m1 = %d\n\n", n0, m0, m1);
    #endif
1705                    if (error_found == 1)
                          {
1707                        dealloc_shmem( sid_a_tmp, sid_b_tmp, sid_a_tmp_0, sid_b_tmp_0, sid_a_tmp_1, sid_b_tmp_1, sid_c_tmp, sid_a, sid_b,
                                sid_c);

1709                        exit(EXIT_FAILURE);

1711                      }
                      }
1713        }
      printf ("End\n");
1715
      dealloc_shmem( sid_a_tmp, sid_b_tmp, sid_a_tmp_0, sid_b_tmp_0, sid_a_tmp_1, sid_b_tmp_1, sid_c_tmp, sid_a, sid_b, sid_c);
1717
      return 0;
1719 }
```

Final implementation synthesized with Altera C2H

This code represents the final code that contains the code of all the accelerators. It is ready to be synthesized by the Altera C2H HLS tool.

```
    #include "system.h"
 2  #include "global.h"

 4  #define BUFF0_SS 1
    #define BUFF1_SS 1
 6  #define ST0_SS 1
    #define ST1_SS 1
 8
    /* #pragma altera_accelerate enable_interrupt_for_function accelerator */
10  #pragma altera_accelerate enable_interrupt_for_function matrix_multiply_hw_buff0
    #pragma altera_accelerate enable_interrupt_for_function matrix_multiply_hw_buff1
12  #pragma altera_accelerate enable_interrupt_for_function matrix_multiply_hw_st0
    //#pragma altera_accelerate enable_interrupt_for_function matrix_multiply_hw_st1
14  #pragma altera_accelerate enable_interrupt_for_function matrix_multiply_hw_c01

16  /* buff0_acc */
    /* fifo connection*/
18  #pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_buff0/st1_buff0_read
    #pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_buff0/buff0_buff1_write
20  #pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_buff0/buff01_c01_write
    #pragma altera_accelerate connect_variable matrix_multiply_hw_buff0/st1_buff0_read to st1_buff0/out
22  #pragma altera_accelerate connect_variable matrix_multiply_hw_buff0/buff0_buff1_write to buff0_buff1/in
    #pragma altera_accelerate connect_variable matrix_multiply_hw_buff0/buff01_c01_write to buff01_c01/in
24
    /* data connection */
26  #pragma altera_accelerate connect_variable matrix_multiply_hw_buff0/ememp to altmemddr_0 arbitration_share 16

28  /* local data connection */
    #pragma altera_accelerate connect_variable matrix_multiply_hw_buff0/lmemp to buff0_0
30  #pragma altera_accelerate connect_variable matrix_multiply_hw_buff0/lmemp to buff0_1

32  /* matrix_multiply_hw_buff1 */
    /* fifo connection*/
34  #pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_buff1/buff1_st0_write
    #pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_buff1/buff0_buff1_read
36  #pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_buff1/buff01_c01_write
    #pragma altera_accelerate connect_variable matrix_multiply_hw_buff1/buff1_st0_write to buff1_st0/in
38  #pragma altera_accelerate connect_variable matrix_multiply_hw_buff1/buff0_buff1_read to buff0_buff1/out
    #pragma altera_accelerate connect_variable matrix_multiply_hw_buff1/buff01_c01_write to buff01_c01/in
40
    /* data connection */
42  #pragma altera_accelerate connect_variable matrix_multiply_hw_buff1/ememp to altmemddr_0

44  /* local data connection */
```

```
#pragma altera_accelerate connect_variable matrix_multiply_hw_buff1/lmemp to buff1_0
#pragma altera_accelerate connect_variable matrix_multiply_hw_buff1/lmemp to buff1_1

/* matrix_multiply_hw_st0 */
/* fifo connection*/
#pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_st0/buff1_st0_read
#pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_st0/c01_st0_read
#pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_st0/st0_st1_write
#pragma altera_accelerate connect_variable matrix_multiply_hw_st0/buff1_st0_read to buff1_st0/out
#pragma altera_accelerate connect_variable matrix_multiply_hw_st0/c01_st0_read to c01_st0/out
#pragma altera_accelerate connect_variable matrix_multiply_hw_st0/st0_st1_write to st0_st1/in

/* data connection */
#pragma altera_accelerate connect_variable matrix_multiply_hw_st0/ememp to altmemddr_0 arbitration_share 16
#pragma altera_accelerate connect_variable matrix_multiply_hw_st0/lmemp to st0

/* matrix_multiply_hw_st1 */
/* fifo connection*/
#pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_st1/st0_st1_read
#pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_st1/c01_st1_read
#pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_st1/st1_buff0_write
#pragma altera_accelerate connect_variable matrix_multiply_hw_st1/st0_st1_read to st0_st1/out
#pragma altera_accelerate connect_variable matrix_multiply_hw_st1/c01_st1_read to c01_st1/out
#pragma altera_accelerate connect_variable matrix_multiply_hw_st1/st1_buff0_write to st1_buff0/in

/* data connection */
#pragma altera_accelerate connect_variable matrix_multiply_hw_st1/ememp to altmemddr_0
#pragma altera_accelerate connect_variable matrix_multiply_hw_st1/lmemp to st1

/* matrix_multiply_hw_c01 */
/* fifo connection*/
#pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_c01/c01_st0_write
#pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_c01/c01_st1_write
#pragma altera_accelerate enable_flow_control_for_pointer matrix_multiply_hw_c01/buff01_c01_read
#pragma altera_accelerate connect_variable matrix_multiply_hw_c01/c01_st0_write to c01_st0/in
#pragma altera_accelerate connect_variable matrix_multiply_hw_c01/c01_st1_write to c01_st1/in
#pragma altera_accelerate connect_variable matrix_multiply_hw_c01/buff01_c01_read to buff01_c01/out

/* data connection */
#pragma altera_accelerate connect_variable matrix_multiply_hw_c01/a_tmp_0 to buff0_0
#pragma altera_accelerate connect_variable matrix_multiply_hw_c01/b_tmp_0 to buff0_1
#pragma altera_accelerate connect_variable matrix_multiply_hw_c01/a_tmp_1 to buff1_0
#pragma altera_accelerate connect_variable matrix_multiply_hw_c01/b_tmp_1 to buff1_1

#pragma altera_accelerate connect_variable matrix_multiply_hw_c01/c_tmp_0_r to st0 // st0 size * 2
#pragma altera_accelerate connect_variable matrix_multiply_hw_c01/c_tmp_0_w to st0 // st0 size * 2
#pragma altera_accelerate connect_variable matrix_multiply_hw_c01/c_tmp_1_r to st1 // st0 size * 2
#pragma altera_accelerate connect_variable matrix_multiply_hw_c01/c_tmp_1_w to st1 // st0 size * 2




int matrix_multiply_hw_buff0 (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii ,int block_jj
      ,int block_kk, int iter_space , int db_iter)
{
  int i , j , k, ii , jj , kk, t , r , s;
  int min_ii , min_jj , min_kk;
  int  offset_bufflocal , offset_ext_memp;
  int* lmemp, * ememp;
  int dummy_read = 0;
  int r_sup;
  int bool_nsent = 1;
  int tmp;
  volatile int* __restrict__ st1_buff0_read = (int *) ST1_BUFF0_OUT_BASE;
  volatile int* __restrict__ buff0_buff1_write = (int *) BUFF0_BUFF1_IN_BASE;
  volatile int* __restrict__ buff01_c01_write = (int *) BUFF01_C01_IN_BASE;
  int* __restrict__ a_tmp_0 = (int *) BUFF0_0_BASE;
  int* __restrict__ b_tmp_0 = (int *) BUFF0_1_BASE;
  ii = 0;
  jj = 0;
  kk = 0;

  for (t = 0; t < iter_space; t+= db_iter)
    {

      if ( block_ii < n0 - ii) min_ii = block_ii;
      else min_ii = n0 - ii;
      if ( block_jj < m1 - jj) min_jj = block_jj;
      else min_jj = m1 - jj;
      if (block_kk < m0 - kk) min_kk = block_kk;
      else min_kk = m0 - kk;
      i = 0;
      j = 0;
      k = 0;
      s = 0;
```

```
132         /* read from the st1_buff0 pipe */
            dummy_read += *st1_buff0_read;
134
            r_sup = min_ii * min_kk + min_jj * min_kk;
136         bool_nsent = 1;
            for (r = 0, tmp = dummy_read; r < r_sup; r++)
138       {
140         if (s == 0){
              offset_bufflocal  = i * block_kk + k;
142           offset_ext_memp = (ii + i) * m0 + k + kk;
144           if (k < min_kk - 1) k++;
              else{
146             k = 0;
                if (i < min_ii - 1) i++;
148             else {i = 0; s++;}
              }
150           lmemp = a_tmp_0 + offset_bufflocal;
              ememp = a + offset_ext_memp;
152
154         }else {
              if (s == 1){
156             offset_bufflocal  = k * block_jj + j;
                offset_ext_memp = (kk + k) * m1 + j + jj;
158
                if (j < min_jj - 1) j++;
160             else{
              j = 0;
162           if (k < min_kk - 1) k++;
              else {k = 0; s = 0;}
164             }
166           lmemp = b_tmp_0 + offset_bufflocal;
              ememp = b + offset_ext_memp;
168
170         }
          }
172       /*transfer code */
          *lmemp = *ememp;
174       /* sync code */
          if (((r == r_sup - 1) | (r == r_sup - BUFF0_SS))& bool_nsent)
176         {
              bool_nsent = 0;
178           tmp = 0;
              *buff0_buff1_write = 0;
180         }
        }
182       *buff01_c01_write = tmp;
184       if (kk + block_kk < m0) kk+= block_kk;
          else {
186       kk = 0;
          if (jj + block_jj < m1) jj+= block_jj;
188       else {
            jj = 0;
190         if (ii + 2 * block_ii < n0) ii+= 2 * block_ii;
        }
192       }
        }
194     dummy_read += *st1_buff0_read;  /* void the fifo */
      return 0;
196 }
198
    int matrix_multiply_hw_buff1 (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii ,int block_jj
          ,int block_kk, int iter_space , int db_iter)
200 {
      int i, j, k, ii, jj, kk, t, r, s;
202   int min_ii, min_jj, min_kk;
      int  offset_bufflocal , offset_ext_memp;
204   int* lmemp, * ememp;
      int dummy_read = 0;
206   int r_sup;
      int bool_nsent = 1;
208   int tmp;

210    volatile int* __restrict__ buff1_st0_write = (int *) BUFF1_ST0_IN_BASE;
       volatile int* __restrict__ buff0_buff1_read = (int *) BUFF0_BUFF1_OUT_BASE;
212    volatile int* __restrict__ buff01_c01_write = (int *) BUFF01_C01_IN_BASE;
       /* local buffer declaration section */
214    int* __restrict__ a_tmp_1 = (int *) BUFF1_0_BASE;
       int* __restrict__ b_tmp_1 = (int *) BUFF1_1_BASE;
216   ii = 0;
      jj = 0;
218   kk = 0;
```

```
220   for (t = 0; t < iter_space; t+= db_iter)
        {
222
          if (block_ii < n0 - ii) min_ii = block_ii;
224       else min_ii = n0 - ii;
          if (block_jj < m1 - jj) min_jj = block_jj;
226       else min_jj = m1 - jj;
          if (block_kk < m0 - kk) min_kk = block_kk;
228       else min_kk = m0 - kk;
          i = 0;
230       j = 0;
          k = 0;
232       s = 0;
          /* read from the st1_buff0 pipe */
234       dummy_read += *buff0_buff1_read;

236       if (ii + block_ii < n0) {
        i = block_ii;
238     r_sup = min_ii * min_kk + min_jj * min_kk;
        bool_nsent = 1;
240     for (r = 0, tmp = dummy_read; r < r_sup; r++)
          {
242         if (s == 0){
              offset_bufflocal = (i - block_ii) * block_kk + k;
244           offset_ext_memp = (ii + i) * m0 + k + kk;
              if (k < min_kk - 1) k++;
246           else{
            k = 0;
248         if (i < min_ii + block_ii - 1) i++;
            else {i = 0; s++;}
250           }
              lmemp = a_tmp_1 + offset_bufflocal;
252           ememp = a + offset_ext_memp;

254         }else if (s == 1){
              offset_bufflocal = k * block_jj + j;
256           offset_ext_memp = (k + kk) * m1 + j + jj;

258           if (j < min_jj - 1) j++;
              else{
260         j = 0;
            if (k < min_kk - 1) k++;
262         else {k = 0; s = 0;}
              }
264


266           lmemp = b_tmp_1 + offset_bufflocal;
              ememp = b + offset_ext_memp;
268         }
            /*transfer code */
270         *lmemp = *ememp;
            /* mettre dans une fonction ?*/
272         if (((r == r_sup - 1) | (r == r_sup - BUFF1_SS)) & bool_nsent)
            {
274           bool_nsent = 0;
              tmp = 1;
276           *buff1_st0_write = 0; // = *lmemp
            }
278       }
        *buff01_c01_write = tmp;
280
          }
282       /* else forward the token to st0 */
          else {
284         *buff1_st0_write = 0;
            *buff01_c01_write = 1;
286       }
          if (kk + block_kk < m0) kk+= block_kk;
288       else {
        kk = 0;
290     if (jj + block_jj < m1) jj+= block_jj;
        else {
292       jj = 0;
          if (ii + 2 * block_ii < n0) ii+= 2 * block_ii;
294     }
          }
296     }
    return 0;
298 }

300 int matrix_multiply_hw_st0 (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int block_jj,
        int block_kk, int iter_space, int db_iter)
    {
302   int i, j, k, ii, jj, kk, t, r, s;
      int min_ii, min_jj, min_kk;
304   int offset_bufflocal, offset_ext_memp;
      int* __restrict__ lmemp;
```

```
306   int* __restrict__ ememp;
      int r_sup;
308   int dummy_read = 0;
      int dummy_read1 = 0;
310   int tmp;
      int bool_nsent = 1;
312   volatile int* __restrict__ buff1_st0_read = (int *) BUFF1_ST0_OUT_BASE;
      volatile int* __restrict__ c01_st0_read = (int *) C01_ST0_OUT_BASE;
314   volatile int* __restrict__ st0_st1_write = (int *) ST0_ST1_IN_BASE;
      /* local buffer declaration section */
316   int* __restrict__ c_tmp = (int *) ST0_BASE;

318   ii = 0;
      jj = 0;
320   kk = 0;
      for (t = 0; t < iter_space; t+= db_iter)
322     {

324       if (block_ii < n0 - ii) min_ii = block_ii;
        else min_ii = n0 - ii;
326       if (block_jj < m1 - jj) min_jj = block_jj;
        else min_jj = m1 - jj;
328       if (block_kk < m0 - kk) min_kk = block_kk;
        else min_kk = m0 - kk;
330       i = 0;
        j = 0;
332       k = 0;
        s = 0;
334       //store st0 buffer
        i = 0;
336       j = 0;

338

        dummy_read += *buff1_st0_read;
340       dummy_read1 += *c01_st0_read;
      if (kk + block_kk >= m0)
342   {
        r_sup = min_ii * min_jj;
344     bool_nsent = 1;
          for (r = 0, tmp = dummy_read + dummy_read1; r < r_sup; r++)
346         {

348         offset_bufflocal = i * block_jj + j;
          offset_ext_memp = (ii + i) * m1 + j + jj;
350       if (j < min_jj - 1) j++;
          else{
352         j = 0;
            if (i < min_ii - 1) i++;
354         else {i = 0; s++;}
          }
356       lmemp = c_tmp + offset_bufflocal;
          ememp = c + offset_ext_memp;
358
          /*transfer code */
360       *ememp = *lmemp;
          if (((r == r_sup - 1) | (r == r_sup - ST0_SS)) & bool_nsent)
362         {
              bool_nsent = 0;
364           *st0_st1_write = 0;
            }
366


368       }

370   }else
          {
372           *st0_st1_write = 0;
          }
374     if (kk + block_kk < m0) kk+= block_kk;
        else {
376   kk = 0;
      if (jj + block_jj < m1) jj+= block_jj;
378   else {
        jj = 0;
380     if (ii + 2 * block_ii < n0) ii+= 2 * block_ii;
      }
382     }
      }
384 return 0;
}
386
int matrix_multiply_hw_st1 (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int block_jj,
      int block_kk, int iter_space, int db_iter)
388 {
  int i, j, k, ii, jj, kk, t, r, s;
390 int min_ii, min_jj, min_kk;
  int offset_bufflocal, offset_ext_memp;
392 int* __restrict__ lmemp;
```

```
    int* __restrict__ ememp;
394 int r_sup;
    int dummy_read = 0;
396 int dummy_read1 = 0;
    int tmp;
398 int bool_nsent = 1;
    volatile int* __restrict__ st0_st1_read = (int *) ST0_ST1_OUT_BASE;
400 volatile int* __restrict__ c01_st1_read = (int *) C01_ST1_OUT_BASE;
    volatile int* __restrict__ st1_buff0_write = (int *) ST1_BUFF0_IN_BASE;
402 /* local buffer declaration section */
    int* __restrict__ c_tmp = (int *) ST1_BASE;
404 int min_iii;

406 ii = 0;
    jj = 0;
408 kk = 0;

410 *st1_buff0_write = 0;

412 for (t = 0; t < iter_space; t+= db_iter)
      {
414
        if (block_ii < n0 - ii) min_ii = block_ii;
416     else min_ii = n0 - ii;
        if (block_jj < m1 - jj) min_jj = block_jj;
418     else min_jj = m1 - jj;
        if (block_kk < m0 - kk) min_kk = block_kk;
420     else min_kk = m0 - kk;
        i = 0;
422     j = 0;
        k = 0;
424     s = 0;
        //store st1 buffer
426
        i = block_ii;
428     j = 0;

430     dummy_read += *st0_st1_read;
        dummy_read1 += *c01_st1_read;
432   if ((kk + block_kk >= m0) && (ii + i < n0))
      {
434     if (min_ii < n0 - (ii + block_ii)) min_iii = min_ii;
        else min_iii = n0 - (ii + block_ii);
436     r_sup = min_iii * min_jj;
        bool_nsent = 1;
438     for (r = 0, tmp = dummy_read + dummy_read1; r < r_sup; r++)
          {
440         offset_bufflocal = (i - block_ii) * block_jj + j;
            offset_ext_memp = (ii + i) * m1 + j + jj;
442         if (j < min_jj - 1) j++;
            else{
444           j = 0;
              if ((i < block_ii + min_ii - 1) && (ii + i < n0 - 1)) i++;
446           else {i = 0; s++;}
            }
448         lmemp = c_tmp + offset_bufflocal;
            ememp = c + offset_ext_memp;
450         /*transfer code */
            *ememp = *lmemp;
452       if (((r == r_sup - 1) | (r == r_sup - ST1_SS)) & bool_nsent)
          {
454         bool_nsent = 0;
            *st1_buff0_write = 0;
456       }

458       }

460   }else
        {
462         *st1_buff0_write = 0;
        }
464     if (kk + block_kk < m0) kk+= block_kk;
        else {
466     kk = 0;
        if (jj + block_jj < m1) jj+= block_jj;
468     else {
          jj = 0;
470       if (ii + 2 * block_ii < n0) ii+= 2 * block_ii;
      }
472     }
      }
474 return 0;
}
476
int matrix_multiply_hw_c01 (int* __restrict__ a, int* __restrict__ b, int* __restrict__ c, int n0, int m0, int m1, int block_ii,int block_jj,
      int block_kk, int iter_space, int db_iter)
478 {
  int i, j, k, ii, jj, kk;
```

```
480   int t, s;
      int min_ii, min_jj, min_kk;
482   int lc_tmp;
      int r;
484
      int dummy_read;
486   int tmp = 0;
      volatile int* __restrict__ c01_st0_write = (int *) C01_ST0_IN_BASE;
488   volatile int* __restrict__ c01_st1_write = (int *) C01_ST1_IN_BASE;
      volatile int* __restrict__ buff01_c01_read = (int *) BUFF01_C01_OUT_BASE;
490   /* local buffer declaration section */
      int* __restrict__ a_tmp_0 = (int *) BUFF0_0_BASE;
492   int* __restrict__ b_tmp_0 = (int *) BUFF0_1_BASE;
      int* __restrict__ a_tmp_1 = (int *) BUFF1_0_BASE;
494   int* __restrict__ b_tmp_1 = (int *) BUFF1_1_BASE;

496   int* __restrict__ c_tmp_0_r = (int *) ST0_BASE;
      int* __restrict__ c_tmp_0_w = (int *) ST0_BASE;
498   int* __restrict__ c_tmp_1_r = (int *) ST1_BASE;
      int* __restrict__ c_tmp_1_w = (int *) ST1_BASE;
500   int cswitch = 0;

502   ii = 0;
      jj = 0;
504   kk = 0;
      for (t = 0; t < iter_space * 2; t+= db_iter)
506     {

508       if ( block_ii < n0 - ii) min_ii = block_ii;
        else min_ii = n0 - ii;
510       if ( block_jj < m1 - jj) min_jj = block_jj;
        else min_jj = m1 - jj;
512       if (block_kk < m0 - kk) min_kk = block_kk;
        else min_kk = m0 - kk;
514
        i = 0;
516     j = 0;
        k = 0;
518     s = 0;

520 /* if (dummy_read == 0) b_tmp = b_tmp_0, else b_tmp = b_tmp_1;... */

522       dummy_read = *buff01_c01_read;
        if (dummy_read == 0)
524       {
          i = 0; j = 0; k = 0;
526         for (r = 0; r < min_ii * min_jj * min_kk; r++)
            {
528
                if (k == 0) lc_tmp = 0;
530             else lc_tmp += *(a_tmp_0 + (i * block_kk + k)) * *(b_tmp_0 + (k * block_jj + j));

532             if (k == min_kk - 1)
                {
534                 *(c_tmp_0_w + (i * block_jj + j))  = ((kk == 0) && (k == 0)) ? lc_tmp : *(c_tmp_0_r + (i * block_jj + j)) + lc_tmp;
                }
536             if (k < min_kk - 1) k++;
                else {
538                 k = 0;
                    if (j < min_jj - 1) j++;
540                 else {
                        j = 0;
542                     i++;
                    }
544             }
            }
546         *c01_st0_write = lc_tmp;
        }
548   if (dummy_read == 1)
      {
550           //perform computation on buff1
        i = block_ii; j = 0; k = 0;
552     for (r = 0; r < min_ii * min_jj * min_kk; r++)
        {
554       if ( ii + i < n0)
          {
556           if ((kk == 0)&&(k == 0)) lc_tmp = 0;
              lc_tmp += *(a_tmp_1 + ((i - block_ii) * block_kk + k)) * *(b_tmp_1 + (k * block_jj + j));
558           if (k == min_kk - 1)
              {
560               *(c_tmp_1_w + ((i - block_ii) * block_jj + j)) = *(c_tmp_1_r + ( (i - block_ii) * block_jj + j)) + lc_tmp;
              }
562
          }
564       if (k < min_kk - 1) k++;
          else {
566           k = 0;
              if (j < min_jj - 1) j++;
```

```
568                     else {
                            j = 0;
570                         i++;
                        }
572                 }
            }
574         *c01_st1_write = lc_tmp;
        }
576     if (cswitch < 1) cswitch++;
        else {
578         cswitch = 0;
            if (kk + block_kk < m0) kk+= block_kk;
580           else {
              kk = 0;
582           if (jj + block_jj < m1) jj+= block_jj;
              else {
584             jj = 0;
                if (ii + 2 * block_ii < n0) ii+= 2 * block_ii;
586           }
            }
588       }
    }
590   return 0;
}
```