High-level synthesis (HLS) on FPGA for inverse problems : application to tomography and radioastronomy

Daouda Diakité

université
PARIS-SACLAY

High-Level Synthesis (HLS) on FPGA for Inverse Problems: application to tomography and radioastronomy

Synthèse de haut niveau sur FPGA pour problèmes inverses: application à la tomographie et à la radioastronomie

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n° 580, SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION (STIC)
Spécialité de doctorat: Sciences du traitement du signal et des images
Graduate School : Informatique et sciences du numérique. Référent: Faculté des sciences d'Orsay

Thèse préparée dans l'unité de recherche **Laboratoire des signaux et systèmes** (Université Paris-Saclay, CNRS, CentraleSupélec), sous la direction de **Nicolas GAC**, Maître de conférences

Thèse soutenue à Paris-Saclay, le 15 Décembre 2022, par

Daouda DIAKITE

Composition du jury:
Membres du jury avec voix délibérative

Frédéric CHAMPAGNAT Maître de recherche 2, ONERA	Président
Emmanuel CASSEAU Professeur, Université de Rennes 1	Rapporteur & Examinateur
Florent DE DINECHIN Professeur, INSA Lyon	Rapporteur & Examinateur
Mickaël DARDAILLON Maître de conférences, INSA Rennes	Examinateur
Virginie FRESSE Maîtresse de conférences, Université Jean Monnet	Examinatrice

Thèse de doctorat

NNT: 2022UPASG080

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre: Synthèse de haut niveau sur FPGA pour problèmes inverses: application à la tomographie et à la radioastronomie

Mots clés: Adéquation Algorithme Architecture, FPGA, Synthèse de Haut Niveau, Problème Inverse, Radioastronomie, Tomographie.

Résumé: Le besoin croissant de puissance de calcul imposé par la complexité des algorithmes de traitement et la taille des problèmes nécessite l'utilisation d'accélérateurs matériels pour répondre aux contraintes de temps et d'énergie. Les architectures FPGA sont connues pour être parmi les plateformes les plus économes en énergie, notamment pour les systèmes embarqués à travers les langages de description matérielle. L'apparition des nouveaux outils de synthèse de haut niveau a été un facteur majeur dans la prise en compte des FPGAs pour les applications complexes, comme c'est le cas avec les processeurs manycores. Les outils de synthèse de haut niveau génèrent une conception de description matérielle à partir de langages de haut niveau tels que C, C++ ou OpenCL. Les derniers FPGAs sont équipés de plusieurs unités de calcul à virgule flottante capables de répondre aux exigences de précision d'un large éventail d'applications. Cependant, l'exploitation du plein potentiel de ces architectures a toujours été une préoccupation majeure. Cette thèse vise à explorer les méthodologies d'accélération des algorithmes de problèmes inverses mal posés sur les architectures FPGA grâce à de nouveaux outils de synthèse de haut niveau appliqués à la reconstruction tomographique et à la radioastronomie. En effet, de nombreux algorithmes pour ces applications sont limités par la mémoire. Une architecture sur-mesure dérivée d'une méthodologie d'adéquation algorithme-architecture a été proposée pour surmonter le goulot d'étranglement de la mémoire. Nous avons appliqué cette méthodologie à l'opérateur de rétroprojection 3D dans le contexte de la reconstruction itérative. L'architecture du rétroprojecteur 3D tire parti d'une stratégie d'accès à la mémoire pour atteindre un débit de calcul élevé. Ensuite, nous prenons en compte la parallélisation de l'algorithme d'optimisation complet sur FPGA. Nous discutons également de la place des FPGAs en radioastronomie, notamment pour le système d'imagerie du pipeline SKA.

Title: High-Level Synthesis (HLS) on FPGA for Inverse Problems: application to Tomography and Radioastronomy

Keywords: Algorithm Architecture Co-design, FPGA, High-Level Synthesis (HLS), Inverse Problem, Radioastronomy, Tomography.

Abstract: The increasing need for computing power imposed by the complexity of processing algorithms and the size of problems requires using hardware accelerators to meet time and energy constraints. FPGA architectures are known to be among the most power-efficient platforms, especially for embedded systems using hardware description languages. The appearance of the new high-level synthesis tools has been a major factor in the consideration of FPGAs for complex applications, as is the case with the many-core processors. The high-level synthesis tools generate a hardware description design from high-level languages such as C, C++, or OpenCL. The recent FPGAs are equipped with several floating-point computing units capable of meeting the precision requirements of a wide range of applications. However, exploiting the full potential of these architectures has always been a major concern. This thesis aims to explore methodologies for accelerating inverse problem algorithms on FPGA architectures through new high-level synthesis tools applied to tomographic reconstruction and radioastronomy. Indeed, many algorithms for these applications are memory-bound. A custom architecture derived from an algorithm-architecture co-design methodology has been proposed to overcome the memory bottleneck. We applied this methodology to the 3D back-projection operator in the context of iterative reconstruction. The 3D back-projector architecture takes advantage of a custom memory access strategy to reach a full computational throughput. Then we consider the parallelization of the complete optimization algorithm on FPGA. We also discuss the position of FPGAs in radio astronomy, particularly for the SKA pipeline imaging system.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

Acknowledgments

I would like to thank God for allowing me to go through this project with courage and commitment.

This thesis would not have been possible without several key people.

I am deeply grateful to my supervisor Nicolas Gac for his constant encouragement, support, and unfailing guidance throughout this project. I was honored to learn from his knowledge of computer architecture, parallel computing, and inverse problem algorithms. I also thank Nicolas Gac for his research interest, enthusiasm, patience, and feedback. His expertise and valuable suggestions were of significant importance in the completion of this thesis. I have been fortunate to have Nicolas Gac as my thesis supervisor.

I would like to thank the reviewers of this thesis, Emmanuel Casseau and Florent de Dinechin, for agreeing to review this manuscript and for their time, effort, and valuable comments. I also like to thank the examiners of the doctoral committee, Frédéric Champagnat, Mickaël Dardaillon, and Virginie Fresse, for evaluating my work and the insightful discussion on my research topic.

I would like to thank Thomas Rodet for having accepted to be my thesis supervisor at the beginning before Nicolas Gac got his HDR. I also thank him for his valuable insights and for being part of the doctoral committee.

I would like to thank Daniel Charlet for giving me access to one of their FPGA boards at the IJCLab and his interest in promoting hardware acceleration through high-level synthesis tools. I also thank Intel Corporation for granting me access to their DevCloud to use Stratix 10 PAC with the oneAPI framework.

I sincerely thank my dear colleagues and friends Nicolas, Marine, Ralph, Mickaël, Pierre, Agathe, Stefano, Dan, Sunrise, Eduardo, Jean-Batiste, and Josee, for the warm environment and dynamics within the team that we have maintained despite the pandemic. I am grateful for the coffee, lunch breaks, and interesting discussions. Furthermore, I thank François Orieux, Charles Soussen, Patrice Brault, and the GPI team.

To Dah, Ousmane (N'diaye), Ousmane (Dao), Bocar, and Ibrahim (thanks for the re-reading), I want to thank you for all your support and encouragement throughout these years of the thesis. Last but not least, I would like to express my profound gratitude to all my family, especially my parents, siblings, and uncles, for their patience and constant support. The successful completion of this thesis would certainly not have been possible without them.

Contents

List of Figures

List of Tables

Acronyms

ALM Adaptive Logic Module . . . 9
AOC Altera Offline Compiler . . . 17
ASICs Application Specific Integrated Circuits . . . 9
BSP Board Support Package . . . 17
BRAM Block RAM . . . 10
BSP Board Support Package . . . 17
CI Computational Intensity . . . 66
CT Computed Tomography . . . 42
CPU Central Processing Unit . . . 7
CUDA Common Unified Device Architecure . . . 7
CLB Configurable Logic Blocks . . . 9
CC Correlation Coefficient . . . 99
CP Computational Performance . . . 66
DSL Domain-Specific Language . . . 12
DSP Digital Signal Processing . . . 10
DD Distance-Driven . . . 52
FPGAs Field Programmable Gate Arrays . . . 9
FOD Focus-Object Distance . . . 58
FDD Focus-Detector Distance . . . 58
FBP Filtered Back-Projection . . . 42
GPGPU General Purpose GPU . . . 7
GPUs Graphical Processing Units . . . 7
GUPS Giga Updates Per Second . . . 60
GBOPS Giga Bytes Operations Per Second . . . 92

Introduction

The increasing need for computing power imposed by the complexity of processing algorithms and the size of these problems requires the use of hardware accelerators to meet time and energy constraints. However, as a consequence of the decline of Moore's Law, it is impossible to reduce the processing size of transistors at a certain limit. Currently hardware accelerators such as GPUs or FPGAs are being use to increase performance. For example, since the appearance of CUDA language, GPUs have been the preferred architecture over the past decade for HPC applications because of their massively parallel computing pattern. At the same time, FPGAs have recently experienced multiple technological advances and can be considered an alternative for computationally-intensive applications. These advances concern both emergence of more mature high-level synthesis tools, and the technology and number of recent DSPs present in FPGAs capable of performing floating point operations as required in HPC applications.

FPGAs have been used for several decades and have proven themselves in the field of embedded systems, particularly for their energy efficiency and meeting real-time constraints of the applications. When FPGAs are considered for intensive computing, these same advantages should be preserved while meeting the computing needs of these applications. In the context of this thesis, the considered HPC applications are computed tomography and radioastronomy. Indeed, for a long time FPGA technology has been only reserved for hardware designers using hardware description languages. This level of abstraction requires a strong knowledge of FPGA architectures and their programming tools. Also, FPGA development is very time-consuming using these languages compared to general-purpose processor programming models. The necessity to reduce the development time and alleviate the complexity of FPGA programming has motivated the emergence of high-level synthesis tools. The main vendors of FPGAs have contributed considerably by proposing more mature tools that allow the use of FPGAs through high-level languages. Thanks to high-level synthesis tools, the development time of FPGAs has been considerably reduced and made accessible to a wide audience of developers and complex applications.

Motivation

Reducing the processing time for large-scale problems is a topic of interest in high-performance computing and image processing domains. In computed tomography, it is essential to reduce the radiation dose for the patient during the acquisition without affecting the image quality. Iterative reconstruction methods are used to reconstruct the image from this low-dose tomography. According to the literature, iterative reconstruction algorithms take several minutes to several hours. For instance, the computation time of the Expectation Maximization (EM) algorithm for 500 projections of 736×64 pixels of projection data in the CPU is 1.52 hours [Chen 2012a]. An important axis to accelerate these computations relies on the exploration and the use of hardware accelerators. GPUs have been the preferred architecture for the past decade due to their massively parallel computing pattern. Our objective is to emphasize the position of FPGAs for compute-intensive applications since the emergence of high-level synthesis tools. FPGA architectures can be an alternative to the GPU thanks to their cycle accuracy and power efficiency. However, systematic use of software languages on FPGAs does not guarantee good performance. A specific focus is then required to perform FPGA-specific optimizations in order to assist the high-level compiler in synthesizing an efficient pipeline. User expertise is essential to exploit FPGAs' full potential and make them competitive with GPUs for HPC acceleration. Indeed, despite recent advances in FPGA technology, this architecture is still behind GPU architecture in a wide range of applications, mainly because of the substantial computing power of GPUs as well as their high global memory bandwidth.

Radioastronomical imaging algorithms could also benefit from the high energy-efficiency of FPGA devices through HLS tools. The supercomputer for the Square Kilometre Array (SKA) radio telescope should use hardware accelerators to meet the different constraints of this project. SKA is the largest radio telescope in the world, using streaming data to generate multidimensional images of the sky at 7.2 Terabit/s without any storage capabilities. The Science Data Processor (SDP) generates the sky images in the SKA pipeline, and this SDP is highly energy-constrained. Currently, GPUs are the candidate with the most potential to meet the computing needs of SDP. However, these many-core GPUs are very power-hungry in order to meet the energy consumption budget. Another alternative must be explored and since the emergence of the new high-level synthesis tools, FPGAs have proven to be a power-efficient platform. Thus, FPGAs could be a more practical alternative for the SKA SDP system.

Thesis Outline

This thesis aims to explore methodologies for accelerating inverse problem algorithms on FPGAs through new high-level synthesis tools with an application to the field of tomographic reconstruction and radioastronomy. The main contribution of

this dissertation is the design of a custom architecture of the 3D back-projection algorithm. This architecture takes advantage of a specific methodology based on offline memory access analysis to alleviate the memory bottleneck.

The manuscript is organized as follows:

- Chapter 1 introduces computer architectures and the acceleration requirement for compute-intensive applications. The position of FPGAs is discussed as an accelerator and the evolution of HLS tools for their programming. Then, particular attention is paid to Intel's HLS tools used in this study.
- Chapter 2 presents the imaging system of the SKA pipeline and the iterative algorithms used to reconstruct the sky image. The chapter uses the deconvolution use-case to illustrate the design of the algorithm on FPGA using OpenCL HLS tool.
- Chapter 3 highlights the tomography reconstruction problem as an inverse problem. We present the operators used in iterative reconstruction algorithms and the related work about their acceleration on CPU, GPU, FPGA. The first hardware acceleration results of the projector and back-projector operators are also discussed in this chapter.
- Chapter 4 presents our adopted methodology for algorithm-architecture co-design in order to accelerate computationally-intensive applications to overcome the memory bottleneck. We also present several advanced FPGA-aware optimizations using high-level tools to fully harness the device. Then, the adopted methodology is applied to a simple use-case of 2D convolution in order to illustrate the prefetching impact on the design performance using OpenCL HLS.
- Chapter 5 proposes a custom architecture of the 3D back-projection operator used in iterative image reconstruction. The memory access strategy that increased the algorithm's data reuse rate is presented. The roofline model is used to profile and guide the optimization steps of the design, and the scalability strategy to higher-end FPGA is discussed. This chapter also compares the OpenCL and oneAPI tools for the back-projection use case. Furthermore, this Chapter discusses the different strategies for porting this iterative algorithm to the FPGA board. The gradient descent algorithm is considered in the context of CT reconstruction.

Chapter 1

High-Level Synthesis for FPGAs

Contents

Processor architectures have undergone enormous advances over time to meet large-scale applications' increasing demands for computing power. These advances in internal processor architectures have improved the raw performance of these machines over many years by adding more transistors while decreasing process sizes, as predicted by *Moore's Law*. However, Moore's Law is reaching its end with increasing technological complexity. Thus, increasing processor performance by increasing chip density was no longer a viable solution. Other strategies have been employed

to fill the gap created by stagnant performance. These strategies included increasing the operating frequencies and expressing parallelism (ILP, pipelining, ...) of general-purpose processors. However, improving internal architectures with out-of-order execution or deeper pipelines has become counterproductive. In addition, increasing the clock frequency has come up against issues with power dissipation in the circuits. The multiplication of computing cores within a single processor has been adopted to address the concern of energy dissipation caused by increased frequency. Multi-core and manycore processors, e.g., GPUs, were thus developed. In addition, other types of architectures continued to evolve in parallel, such as FPGAs and ASICs, both used for custom application-based design. The most significant difference between the two is that ASICs are not reprogrammable, and their development cost is higher than FPGAs. The increasing maturity of high-level synthesis tools for FPGAs and ASICs and the evolution of their internal architectures with more computational resources make them potential hardware accelerators for large-scale applications.

However, the multiplication of computational units in general-purpose processors and specific circuits has added a layer of complexity to these circuits' programming to exploit better the computational potential offered by these architectures. The programming paradigm is not the same for these distinctive processors. Some will excel better in applications with more control than computation, while others do better with the expression of parallelism on extensive data. For example, FPGAs are well known for their efficiency in executing control- or data-flow applications in a well-elaborated pipeline. Thus, the choice of architecture should depend on the underlying application in order to perform an algorithm-architecture co-design approach to leverage better the full computational capacity offered by the device.

This chapter will present an overview of the most commonly used computing accelerators, from general-purpose processors to specific circuits, before focusing on FPGAs, which are at the heart of this thesis work. We will then present the high-level synthesis tools on FPGAs by presenting their evolution and the different tools developed since the emergence of this technology. The FPGA devices used in our work come from Intel corporation. Therefore, we will discuss the Intel-specific tools and detail those used during our experiments. The OpenCL tool will be presented in detail, starting with its programming model and ending with the various basic optimizations offered by this tool to take advantage of the FPGAs.

1.1 Computer architectures

The requirement of sustainable computing power relies on the constant growth of high-performance application complexity. The development of hardware architecture has been evolving to address compute-intensive problems from single-core processors to parallel processors. The well-known Moore's Law [Moore 1965] and Dennard scaling [Dennard 1974] have driven the advances in computer technology regarding performance and energy efficiency.

1.1.1 General-purpose processors

1.1.1.1 CPU

Central Processing Unit (CPU) is the reference architecture in computer systems. The first generation of CPUs was presented with sequential architecture and executed software as a set of instructions. This architecture is well established for its versatility and ease of use through software programming languages. Therefore, to push the limit of Moore's Law, the operating frequency was increased, and multiple enhancements were provided to the architecture, such as pipelining, out-of-order execution, speculative execution, etc. Multi-core microprocessors have emerged to improve performance and overcome the limits of architectural improvement to CPUs. The multi-core processors use two or more compute cores to execute a set of instructions. These multi-core processors delivered better performance than single-core ones thanks to the exploitation of all the compute cores via parallel programming languages. The versatility of CPUs lies in the management of control in their architecture, which leaves less room for calculations. This makes CPUs dominant in multitasking but hard for them to excel in compute-intensive applications.

1.1.2 Specific-purpose co-processors

Hardware accelerators are used as co-processors in order to meet the need for computing power. These accelerators offer high computing power compared to traditional CPUs for a large number of applications. We will briefly look at these hardware accelerators in this section.

1.1.2.1 GPU

Used for 3D rendering since the 1990s, Graphical Processing Units (GPUs) architectures have been democratized as General Purpose GPU (GPGPU) and are now adopted as the most privileged architecture for compute-intensive applications in various domains. Their massively parallel computing model using thousands of cores explains this preference [Xu 2007]. Indeed, GPUs are considered as *many-core* processors and can perform thousands of computations simultaneously. The compute cores of GPUs are grouped into Streaming Multiprocessors (SM) with a reduced instruction set compared to CPU, and each core in the same SM executes the same program in parallel.

In 2006, Nvidia introduced its Common Unified Device Architecure (CUDA) API to leverage their GPU architectures as hardware accelerators through a software language based on C. GPUs are based on massive parallelism of elementary tasks called *threads*, where each thread represents a set of instruction sequences. Threads execute the same instructions for the same CUDA kernel. GPU programming model organizes threads into warps consisting of a group of 32 threads. Threads in the same warp have the same context of execution. A group of warps constitutes a block, and multiple blocks form a grid. Each block is mapped to a GPU SM during

execution.

CUDA is a proprietary framework only supported on Nvidia's GPUs. OpenCL, initiated by Apple and maintained by KHRONOS GROUP since 2008, has been released in order to provide a programming framework supported by all vendors' devices [Khronos 2009]. OpenCL enables cross-vendor portability for heterogeneous platforms and devices. The programming paradigm of OpenCL and CUDA are relatively similar. However, OpenCL is supposed to support devices other than GPU, such as CPU, DSP, or FPGA.

1.1.2.2 MPPA manycore processor

The Massively Parallel Processor Array (MPPA), launched by Kalray in 2012, is an embedded manycore architecture for high performance and low power purposes. The processor consists of 16 compute clusters communicating through a Network on Chip (NoC). Each cluster comprises 16 VLIW application cores sharing the same local memory. The compute core in MPPA has a fully pipelined 32-bit VLIW processor that performs up to five instructions per cycle. The MPPA processor is connected to a host CPU using high-speed interfaces such as PCIe. Multiple programming models based on standard C/C++ are available to program the heterogeneous MPPA processor, such as OpenCL, OpenMP, POSIX, etc.

1.1.2.3 Digital Signal Processor

Digital Signal Processors are specialized circuits designed to perform specific computations in digital signal processing such as Discrete Fourier Transform (DFT), Fast Fourier Transform (FFT), filtering, correlation, convolution, etc. Like traditional microprocessors, a DSP device contains several units to perform arithmetic or logical operations and on-chip memories object. DSP devices also contain multipliers and Multiply and ACcumulate (MAC) units which are required for most signal processing applications. In addition, they are also equipped with several address generators to handle separate memory spaces. While general-purpose processors are based on Von Neumann architecture, DSPs use Harvard architecture so that they can fetch data and program instructions at the same time.

1.1.3 Specific circuits

Other types of hardware accelerators based on specific circuits are also used to speed up applications. These accelerators are renowned for their energy efficiency and computing performance, particularly in embedded systems and real-time systems. However, the development of these circuits is not accessible to the general public, as is the case with general-purpose and specific-purpose processors.

1.1.3.1 ASIC

Application Specific Integrated Circuits (ASICs) are an integrated circuit designed for a particular application. The ASIC technology offers the best trade-off between performance and power efficiency. ASICs are not reprogrammable, and their design cycle is time-consuming. Thus, their manufacturing process is costly, making them viable only from large volumes of production.

1.1.3.2 FPGA

Field Programmable Gate Arrays (FPGAs) have been developed since the mid-1980s. In contrast to general-purpose fixed architectures (CPUs and GPUs), FPGAs are reconfigurable circuits that offer designers the ability to build a custom architecture driven by the underlying application. The strength of FPGAs lies in the flexibility they offer to build an efficient pipeline architecture. This flexibility positions FPGAs as a mid-range architecture between general-purpose processors and ASICs. In addition, the energy and cyclic efficiency of FPGAs make them excellent candidates for embedded system applications with real-time constraints and a limited power budget.

FPGA architecture:

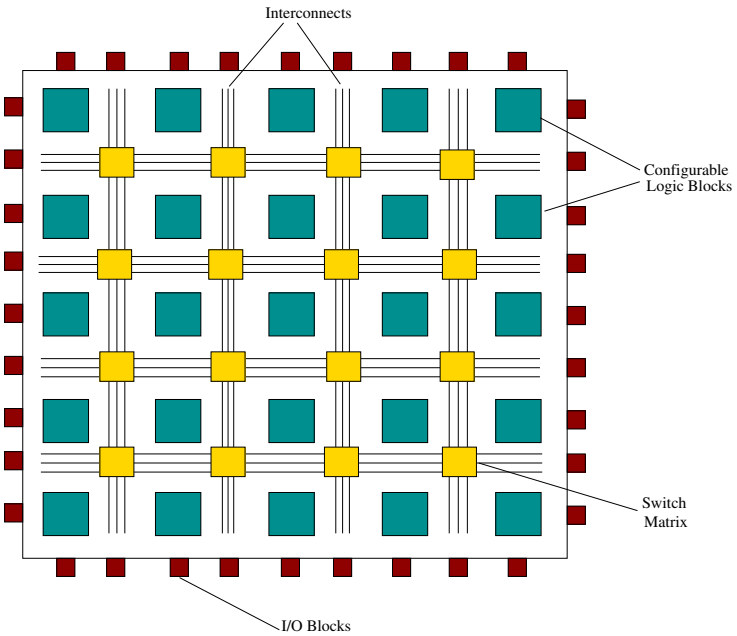


Figure 1.1 – FPGA architecture

The internal architecture of FPGAs is presented in Fig. 1.1. FPGAs are composed of several Adaptive Logic Module (ALM) (or Configurable Logic Blocks

(CLB) in Xilinx terminology), routing switch matrix, and different I/O blocks. An ALM is made of:

- Several Look-up Tables (LUT) with four to six inputs.
- Several registers
- Multiple adders and multiplexers

The interconnection of ALMs and configurable memory cells (e.g., SRAM) can perform several arithmetical and logic operations. The switch matrix is used to interconnect the selected logic block to implement a given application. The I/O blocks are for communication between the FPGA chip and the external components.

In addition to more logic elements, modern Intel FPGAs provides several hard-IPs primitives such as dedicated memory blocks, e.g., Block RAM (BRAM), Digital Signal Processing (DSP) units, etc. These floating-point units provide high design flexibility and are optimized to support high-performance DSP applications in IEEE 754 compliant floating-point single precision. One such example is the Intel Stratix 10 architecture presented in Fig. 1.2. This FPGA device is a 14 nm technology with millions of logic cells and more than five thousand DSP based on Hyperflex core architecture [Intel Corporation 2021a]. The Hyperflex device contains registers everywhere throughout the FPGA core to enable aggressive optimizations, retiming, or pipelining. However, hyperflex architecture comes with several problems. Firstly, routing on these devices is tedious and extremely slow, which is accentuated when using HLS tools. Secondly, a fairly deep pipeline through multiple registers is required to achieve correct frequencies.

FPGA Synthesis:

Basically, an FPGA application is described using Hardware Description Languages (HDL) such as VHDL or Verilog. The use of these languages allows the designer to control the generated pipeline but requires strong knowledge of the FPGA architecture and the synthesis tools. The design flow includes different steps before the bitstream generation for the target FPGA, as illustrated in Fig. 1.3. The first step is to specify the Register Transfer Level (RTL) specifications of the design using HDLs. This step contains several behavioral simulations for the correctness of the application. The RTL description is then translated into a *netlist* made of registers and logic cells. The netlist is mapped into FPGA primitives and placed on the FPGA fabric. The placement step is iterative to find the best combination that minimizes the propagation delay and maximizes the operating frequency. The final step before bitstream generation consists to route the placed netlist which ensures that all resources are connected accordingly to meet all the timing constraints. The place-and-route steps are the most time-consuming ones and are likely to fail when there are unmet constraints.

This traditional design flow is time-consuming and only accessible by strong-skilled hardware designers. In order to reduce the development time of FPGAs

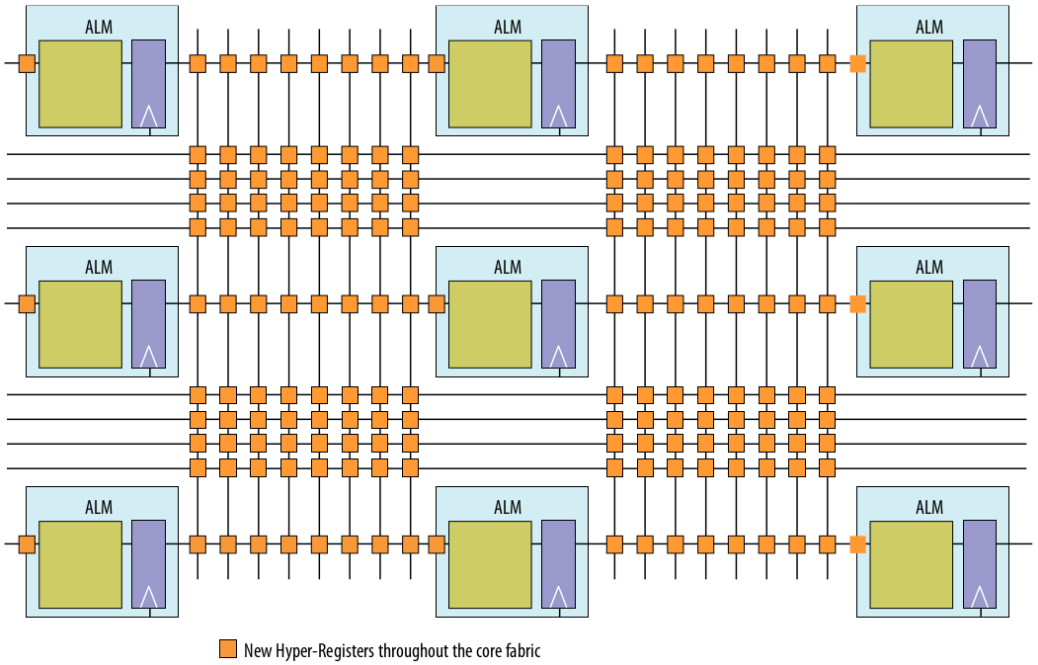


Figure 1.2 – Stratix 10 Hyperflex core architecture from [Intel Corporation 2021c]

and make them accessible to a wide audience of programmers, High-Level Synthesis (HLS) tools have been developed for decades and have only recently reached maturity.

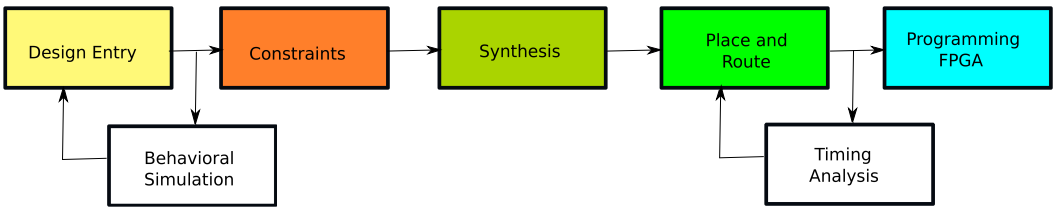


Figure 1.3 – FPGA Design flow with HDL languages

1.2 High-Level Synthesis

HLS is a discipline that attempts to provide more productivity to designers by offering synthesis tools at a high abstraction level than the traditional HDLs. The principle is to automatically generate a hardware circuit from the behavioral descriptions of a given application. A high-level design is entitled to meet several criteria to be effective. The design efficiency, the quality of implementation and the Quality of Results (QoR) in terms of area, performance, and accuracy, should be as good as those obtained with HDL tools. HLS tools provides high design productivity than HDL languages. As illustrated in figure 1.4, the polygon should

be as small as possible to provide both good design efficiency and implementation quality [Pelcat 2016]. Several HLS tools have been developed over four decades, driven by substantial research results and industrial works.

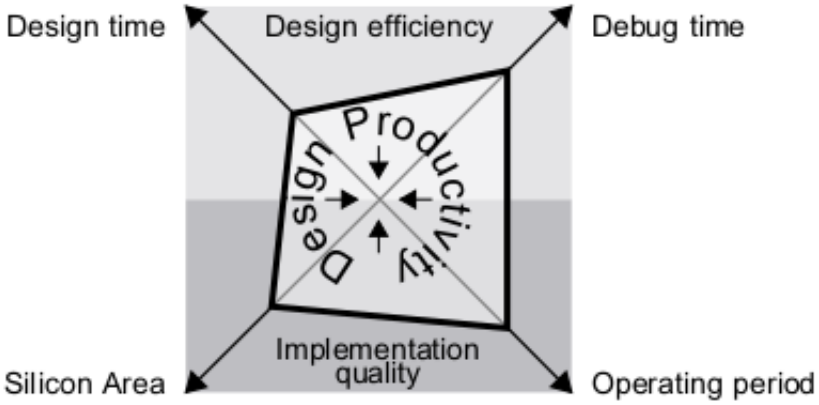


Figure 1.4 – Design efficiency from [Pelcat 2016]

1.2.1 HLS history

The early research in HLS has taken place since the 1970s [Barbacci 1973]. Academic research has led to many advances and tools development, as it can be found in [Gupta 2008, Coussy 2008]. An overview of HLS evolution has been presented in [Martin 2009], and they provided a categorization of the evolution of HLS in three generations preceded by a prehistoric period. A global view of different HLS tools was detailed in [Nane 2015, Numan 2020], including academic and commercial ones. As explained by the authors, the first and second generations (till the early 2000s) of HLS have failed, mainly because the expectations placed on them were too ambitious. Among those expectations, we can find the choice of the right input languages, the guarantee of good QoR, high productivity, etc. The developed tools hardly addressed software developers for these generations and became a commercial failure.

Nevertheless, the third generation of HLS tools succeeds on many levels compared to the previous ones. Many major vendors proposed several tools with a strong focus on the domain of application, the input languages, the target users, the synthesis process, etc. The design entry of an HLS tool could be a Domain-Specific Language (DSL) or a General Purpose Language (GPL). The QoR has seen a significant increase for this generation allowed by the exploitation of most compiler-based optimizations, as is the case for general-purpose languages.

Table 1.1 – An overview of HLS tools

Availability	Tool	Design entry	Target architecture	Year	Owner
Commercial tools (License required)	Bluespec Compiler	BSV	ASIC, FPGA	2007	BlueSpec
	Catapult-C	C/C++, SystemC	ASIC, FPGA	2004	Calypto
	CHC Compiler	C/C++, SystemC	FPGA	2008	Altium
	CoDeveloper	Impulse-C	FPGA	2003	Impulse Accelerated
	C-to-Silicon	C/C++, SystemC	ASIC, FPGA	2008	Cadence
	CyberWorkBench	C, SystemC	ASIC, FPGA	2011	NEC
	Cynthesizer	SystemC	ASIC, FPGA	2004	FORTE
	DK Design Suite	Handel-C	ASIC, FPGA	2009	Mentor Graphics
	HDL Coder	Matlab, Simulink	ASIC, FPGA	2012	MathWorks
	HerculeS	C/NAC	ASIC, FPGA	2012	Ajax Compiler
	Intel HLS Compiler	C/C++	FPGA	2017	Intel
	Intel oneAPI	DPC++	FPGA	2019	Intel
	Intel SDK for OpenCL	OpenCL	FPGA	2013	Intel
	LabView FPGA	G	FPGA	2003	National Instruments
	MaxCompiler	MaxJ, OpenSPL	FPGA	2010	Maxeler Tech.
	Merlin Compiler	C/C++	FPGA	2017	Falcon Comp Sol.
	SDAccel	C/C++, OpenCL	FPGA	2014	Xilinx
	SPIRAL	SPL	ASIC, FPGA	2008	U. Carnegie Mellon
	Synphony C	C/C++	ASIC, FPGA	2010	Synopsys
	Stratus HLS	C++, SystemC	ASIC, FPGA	2015	Cadence
	Vivado HLS	C/C++, SystemC	FPGA	2013	Xilinx
Academic tools (Open-source)	Bambu	C	ASIC, FPGA	2012	Politec. Milano
	DEFACTO	C	FPGA	1999	U. South California
	DWARV	C subset	FPGA	2012	TU. Delft
	CHiMPS	C	FPGA	2008	U. Washington
	CtoVerilog	C	ASIC, FPGA	2008	U. Haifa
	Garp	C subset	FPGA	2000	U. Berkeley
	GAUT	C	ASIC, FPGA	2010	U. Bretagne-Sud
	gcc2verilog	C	FPGA	2011	U. Korea
	HIPAcc	C	FPGA	2014	U. Erlangen-Nürnberg
	Kiwi	C#	FPGA	2008	U. Cambridge
	LegUp	C	FPGA	2011	U. Toronto
	MATCH	MATLAB	FPGA	2000	U. Northwest
	Napa-C	C subset	FPGA	1998	Sarnoff Corp.
	PARO	PAULA	FPGA	2008	U. Erlangen-Nürnberg
	PipeRench Compiler	DIL	PipeRench	2000	U.Carnegie M.
	ROCCCC	C	FPGA	2010	U. California
	SA-C	SA-C	FPGA	2003	U. Colorado
	Sea Cucumber	Java	FPGA	2002	U. Brigham Y.
	SPARK	C	ASIC, FPGA	2003	U. Cal. Irvine
	Trident	C	FPGA	2007	Los Alamos NL

1.2.2 High-Level Synthesis Tools

The emergence of the new high-level synthesis tools for FPGA has been a significant factor in their consideration in complex applications. We present in Table 1.1 a non-exhaustive list of HLS tools, updated with the new tools developed, including academic and commercial ones. Several research works [Koch 2016, Nane 2015, Numan 2020] have provided a complete description of each tool in this table.

Among these tools, the *CHiMPS* compiler is an academic HLS framework, developed for hybrid CPU-FPGA platform for High-Performance Computing (HPC)

applications [Putnam 2008]. The tool takes an input described in C language to generate a synthesizable VHDL for FPGAs. *GAUT* is also an academic tool from Université de Bretagne-Sud designed for DSP applications. The *GAUT* compiler generates the VHDL description from an input C language and also handles the communication and memory management. The *ROCCC* compiler [Villarreal 2010], an open-source HLS framework, is developed at the University of California for compute-intensive applications. Another academic tool is *LegUp* [Canis 2011] based on Low-Level Virtual Machine (LLVM) compiler. From an input C language, *LegUp* can generate the desired custom accelerators as well as the general purpose processor. It is worth mentioning other academic frameworks depending on the domain of application or target architecture, such as *Kiwi* [Singh 2008], *DWARV* [Nane 2012], *BAMBU* [Pilato 2013], etc. In addition to open-source frameworks, several commercial HLS tools have emerged. *Catapult-C* [Calypto 2004] was developed by Mentor Graphics[1] to generate HDL descriptions to target ASICs and FPGAs from a High-Level programming Language (HLL). *C-to-Silicon*, developed by Candence, is a commercial HLS tool aimed for ASICs and FPGAs.This tool generates Verilog code by using a HLL for the target devices mentioned above.

There was a massive surge of interest when major vendors started offering more mature tools for different audiences. *Vivado HLS*, formerly *AutoPilot*, developed initially by AutoESL, was proposed by Xilinx in 2011. *Vivado HLS* allows to synthesize a custom design from an input C, C++ or SystemC to target Xilinx FPGAs. The output of *Vivado HLS* compiler will be Verilog or VHDL IP cores ready to be programmed into the FPGA. Xilinx also proposed their OpenCL-based HLS tool *SDAccel* for software programmers. OpenCL Software Development Kit (SDK) enables FPGA programming for software developers through HLLs such as C, C++, or OpenCL without a strong knowledge of FPGA architectures. The Xilinx *SDAccel* was motivated by the Altera OpenCL released one year early. Indeed, the Intel FPGA (formerly Altera) SDK for OpenCL was developed by Altera in 2013[2]. Both Intel OpenCL SDK and Xilinx SDAccel are LLVM-based compilers that produce RTL level design from HLLs and then perform the full synthesis. In 2017, Intel released a new HLS tool called *Intel HLS Compiler* which is an IP-driven synthesis tool for Intel FPGAs addressed to hardware designers like *Vivado HLS*. This tool allows traditional HDL designers to describe their applications using C or C++ languages allowing them to gain productivity while having the control to tune the design. The high-level design is converted to a LLVM Intermediate Representation (IR), which will be subject to multiple aggressive optimizations by expressing parallelism, pipelining, etc.

However, exploiting the full potential of these architectures using high-level tools has always been a significant concern. Therefore, an algorithm-architecture co-design approach is necessary in order to use these parallel architectures better. For FPGAs with custom architecture, HLS tools provide the guidelines required

[1]Before its acquisition by Calypto Design Systems
[2]Intel acquired Altera in 2015

to implement applications efficiently and thus produce an efficient pipeline. The efficiency of these pipelines has not yet reached that obtained through HDL descriptions, but their development time is drastically reduced.

Nevertheless, a strong knowledge of the architecture of the FPGAs and the functioning of these compilers are essential to help the compiler efficiently synthesize the design and ensure good performance. Besides, an algorithm analysis is critical to choose the architecture that best matches the application. In this work, we only focus on HLS tools from Intel corporation to accelerate our applications.

1.3 Intel HLS Tools

Intel has proposed several tools depending on the target audience to open up the field of FPGA development to software developers and allow hardware designers to gain productivity. Among these HLS tools, we have Intel HLS Compiler, Intel FPGA SDK for OpenCL, and Intel oneAPI, which all use a HLL to synthesize a design on Intel FPGAs.

1.3.1 Intel HLS Compiler

The Intel HLS Compiler is an HLS tool that produces an RTL level design optimized for Intel FPGAs from an input of C/C++ languages. The tool aims to quicken the design for traditional FPGA designers and allow them to integrate their accelerator into a larger system. The tool is IP-driven which means that it synthesizes a C/C++ function into an RTL design as an IP that can be used along with HDL designed IPs in a more complex system. Intel HLS Compiler design flow includes an *emulation* step to verify the component functionality and allow a quick hardware verification of the accelerator by generating a testbench in the *cosimulation* step. In summary, the design flow of an Intel HLS Compiler application consists of multiple iterations of functional correctness and architectural verification followed by the placement and routing of the generated IP and then the integration into the FPGA system. The HLS design can benefit from several basic compiler optimizations. The designer may explore these optimizations to better harness the FPGA through multiple pragmas and attributes.

1.3.2 Intel FPGA SDK for OpenCL

Open Computing Language (OpenCL) is an open royalty-free standard parallel programming API for heterogeneous processing platform (CPU, GPU, FPGA) [Khronos 2009]. The purpose is to give software developers the ability to exploit the parallelism potential of modern processors with code portability. In 2013, Altera presented their OpenCL SDK to allow software developers to target their FPGAs using HLLs.

1.3.2.1 Programming model

OpenCL allows cross-vendor and cross-platform portability, which was a reaction to CUDA. Therefore, the programming model OpenCL is similar to CUDA based on massive threads parallelism. Each thread is called a *work-item* in OpenCL terminology, and multiple work-items are organized to form a *work-group*. Work-items within the same work-group are synchronized using *barriers* to ensure memory consistency. The local memory allows work-items within the same work-group to share data. Data stored into global memory is visible by all work-items within the work-groups.

Based on the C99 standard, OpenCL supports both data- and task-parallel programming models known as Single Work-Item (SWI) and NDRange (NDR) kernels [Khronos 2009] in OpenCL terminology. The former, called SWI kernel, models the design as a deep pipeline to exploit the parallelism at the finest granularity. This execution model uses only one work-item within a single work-group to execute the program through the pipeline. The challenge is to feed the pipeline with incoming data efficiently. A new piece of result is produced, as long as the pipeline stages are full, in a regular interval called **Initiation Interval (II)**. The latter, NDR kernel, exploits thread-level data parallelism to achieve better throughput. The NDR model on FPGA is different from the GPU-like model. Indeed, to exploit the multiple compute cores available on GPU devices, multiple work-items are created to run simultaneously on these cores. However, the compiler generates, for an NDR design on FPGAs, a single pipeline as a compute unit, and all the work-item passes through it to complete the execution. The attempt, in this case, is to launch a new thread at regular II cycles. One can infer SIMD unit or compute unit replication to achieve thread-level parallelism.

The programming model of OpenCL is based on the host and devices paradigm. The application is composed of two programs: a host part and the kernel compiled separately using Just-In-Time (JIT). The JIT compilation is not supported due to the long-time placement and routing step for bitstream generation for FPGAs. Therefore the OpenCL kernel is compiled offline using a vendor-specific compiler followed by the full synthesis flow. The host program is run on the host device and the kernel by the accelerator device. The OpenCL API handles the communications between the host and the device.

An overview of the OpenCL memory model is presented in Fig. 1.5. OpenCL has four memory models regardless of the underlying device. The first one is the **global memory** which is a large-sized long latency off-chip memory visible to all work-items. Then, a small part of global memory is configured as read-only memory with a high cache hit rate called the **constant memory**. This read-only memory acts like cache memory, and data stored in it should fit in the constant memory once. A cache miss in constant memory is extremely expensive. The third memory is **local memory** with low access latency and small in size compared to the external memory. The local memory is used to prefetch data and is visible to all work-items within a work-group. The last one is **private memory** only visible

by one work-item. After OpenCL kernel synthesis, the global memory space resides in the external off-chip memory of the board (DDR or HBM memory), while the local memory space is mapped to the on-chip memory of the FPGA (made of M20K BRAM). Private memory can be implemented using on-chip BRAMs or registers.

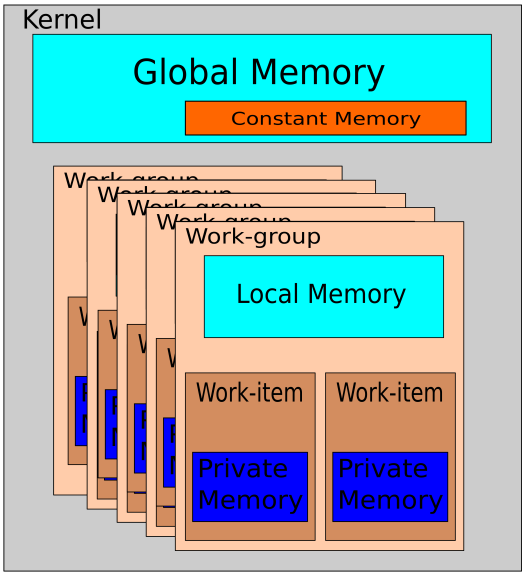


Figure 1.5 – OpenCL Memory Model

The major FPGA manufacturers have proposed their OpenCL SDK to program their FPGAs. This work only deals with Intel FPGA SDK for OpenCL. This Intel OpenCL SDK provides all necessary components to run an application on Intel FPGA devices, including Altera Offline Compiler (AOC) to compile the kernel, host libraries (for host applications), and all utilities related to the Board Support Package (BSP).

1.3.2.2 Board Support Package

The generated bitstream programs the FPGA after compiling the compute kernel using a vendor-specific compiler. A BSP is used to interface the FPGA with the external devices to run the OpenCL kernel. Therefore, the BSP provides all necessary interfaces between the FPGA and external components (memory, host, etc.). The BSP, as shown in Fig. 1.6, provided by the board manufacturers, allows programmers to run the kernel executable on the target FPGA. It packages features such as IP Cores, DDR controllers, PCIe controllers, and DMA drivers to establish communication between the host and the FPGA device. Many manufacturers of boards based on Intel FPGAs provide boards with their specific BSP to quickly design and run applications using Intel FPGA SDK for OpenCL. The main concern is that the vendor updates of BSPs do not necessarily follow OpenCL compiler updates. This prevents the programmer from taking advantage of some of the new compiler features. However, Intel ensures a minimum of two years of backward compatibility

between BSP and the compiler. In addition, designers can implement their custom BSP and, in particular, add additional functions to their application. OpenCL SDK also provides, besides OpenCL directives, many FPGA-specific optimizations to fully harness the device's potential.

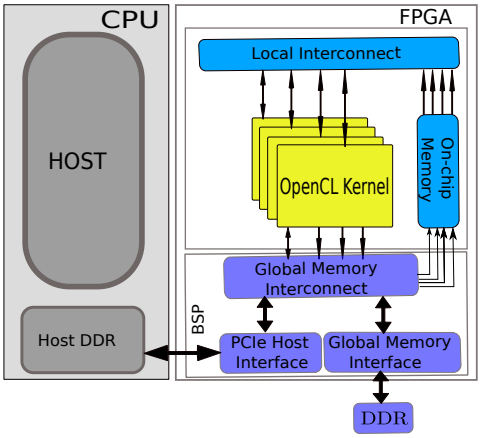


Figure 1.6 – Intel FPGA SDK for OpenCL platform

1.3.3 Intel oneAPI

OneAPI [Intel Corporation 2021b] is Intel's new unified and standardized toolkit that allows targeting multiple platforms using a single-source C++ programming language. Intel oneAPI is based on the DPC++ (Data Parallel C++) language, which is an extension of standard C++ incorporating the SYCL standard [Keryell 2020]. DPC++ is a modern C++ extended HLL based on the OpenCL programming model, and the purpose is to target different architectures (such as CPU, GPU, FPGA, etc.) at a higher level abstraction than OpenCL. The main advantage of the oneAPI framework is that it alleviates programming complexity and allows code reuse across different platforms while ensuring performance. The DPC++ compiler builds the oneAPI project for the target architecture. The JIT compilation can be used when targeting CPU or GPU devices, as is the case for OpenCL. However, JIT compilation is not supported on FPGAs as explained in Section 1.3.2.1. Unlike OpenCL, oneAPI supports single-source compilation for the host and the device application. A BSP, to ensure the interface and communications between the FPGA chip and external hard-IP, is required to run oneAPI application on FPGAs.

Intel plans to replace OpenCL with oneAPI in the next few years. In this case, this tool must be able to offer the same level of performance as the OpenCL tool or offer better performance for FPGAs to make it into the HPC world. This manuscript will also focus on the oneAPI framework to accelerate our compute-intensive algorithms and compare the two languages for the same types of applications.

OneAPI is a superset of OpenCL in terms of kernel programming directives. The problems and optimization solutions are based on the same concepts. However,

the names of the attributes and pragmas are usually different. In the rest of this document, the optimizations presented are universal to both tools, but we will present them using OpenCL terminology.

1.4 OpenCL Optimizations

This section deals with OpenCL kernel optimization techniques to leverage the FPGA architecture. In fact, the use of the tools allows reducing the development time on FPGA considerably without however any guarantee of performance. The HLS compilers implement different optimizations based on their particular IR, such as pipelining or expressing data-level parallelism. However, they suffer from a lack of support for many other unexploited FPGA-specific optimizations for arithmetic operations [Uguen 2019] for instance. In order to exploit their full potential, it is essential to apply some advanced optimizations in addition to the basic compiler optimizations. The OpenCL tool has been evaluated for many HPC applications such as the Rodinia benchmark [Zohouri 2019] or computed tomography [Martelli 2019]. In this section, we will present the optimizations applicable to an FPGA compute kernel to have an efficient pipeline.

1.4.1 Compiler-aided Optimizations

The use of FPGAs through HLS tools such as OpenCL requires careful attention to optimizations for efficient pipeline generation. Compilers for general-purpose processors can apply some advanced optimizations without much developer intervention. This is not the case for the new HLS compilers for FPGAs. To take advantage of the FPGA architecture, the designer must assist the HLS compiler by providing FPGA-specific optimizations such as concurrent execution, pipelining, data representation, etc. Several basic optimizations are presented in programming guides [Intel Corporation 2019a, Intel Corporation 2019b]. We will look here at some optimizations relevant to FPGA design.

1.4.1.1 Floating-point optimizations

Intel's new FPGAs are equipped with DSPs to perform floating-point operations. However, the cost of these floating-point operations can be reduced for applications with a tolerance for arithmetic precision by relaxing or balancing these floating-point calculations.

The order of arithmetic floating-point operations can be relaxed using the *-fp-relaxed* option in the *aoc* command. The relaxing is done by implementing balanced tree hardware as described in [Langhammer 2008]. The *-fpc* option directs the compiler to remove intermediary floating-point rounding operations and conversions whenever possible and to carry additional bits to maintain precision. The rounding is performed only at the end of the chain of floating-point operations. Using these

floating-point optimizations may introduce numerical variations in the computation results.

1.4.1.2 Global memory interleaving

The offline compiler, by default, interleaves global memory (1024 bytes interleaved) across the external memory banks to efficiently share the bandwidth between all the memory objects. This configuration leads to poor memory bandwidth utilization for compute-intensive algorithms due to the problem size and the non-contiguous memory access pattern. Therefore, using interleaved memory may result in high stall performance. In order to avoid this automatic memory interleaving, the offline compiler may be invoked with *-no-interleaving* flag in the *aoc* command. This makes it possible to perform read and write operations in different memory banks for high throughput. In the case where the board support package includes different memory types (DDR, QDR, HBM), the memory which has to be prevented from the interleaving must be specified in the *aoc* command (*-no-interleaving=DDR*, for instance). In our case, we notice that non-interleaved global memory access can reduce the worst-case stall percentage by half.

1.4.2 NDR kernels

Several optimizations must be applied to an NDR kernel to achieve high throughput. This programming kernel must avoid multiple barriers when using local memory and reduce stalls due to data dependencies.

Listing 1.1 – NDR kernel

```
1 __kernel void NDR_kernel(//kernel arguments
2                ){
3     // get index of the work item
4     int index = get_global_id(0);
5     // Some computations
6 }
```

1.4.2.1 Specifying work-group size

Intel provides an attribute to set work-group size in order to minimize the BRAM consumption. If not specified, the default work-group size is set at 256, leading to a potential waste of BRAM resources. This optimization can save valuable resources on the FPGA that could be used to increase the overall design performance. Specifying work-group does not impact the performance in all cases. However, when the kernel contains barriers and if a value is not determined, the compiler sets the work-group size to one at runtime even though the default value was set to 256 at compile time. This behavior can significantly degrade the design performance. The impact of work-group size on the performance has been evaluated in [Shata 2019]. They presented cases for their benchmark where the performance can be improved by setting a specific or a maximum value.

1.4.2.2 Parallelism

To achieve data parallelism for NDR kernels, SIMD inference or compute unit replication can be used through adequate attributes. The *num_simd_work_items* attribute is also used to allow kernel vectorization in a SIMD fashion. This enables work-item-level parallelism and allows more computation per work-item without modifying the kernel's body. The use of the SIMD attribute requires to specify the work-group size. The *num_compute_units* attribute is used to generate multiple compute units to increase the kernel throughput. Each work-group will be mapped to a compute unit. Therefore, all the work-items in a work-group may share the same resources. Both attributes increase throughput by increasing the number of FPGA resources. However, the SIMD attribute only replicates the data path of the pipeline while the compute unit attribute implements each instance as a single sufficient pipeline leading to more resource usage. Hence, the SIMD attribute is more area efficient than the other. Nevertheless, in practice, combining these two attributes will potentially give a good trade-off between resource consumption and throughput.

1.4.3 Single Work-Item (SWI) kernels

FPGAs have proven a strong ability to express fine-grained parallelism through pipelining. Therefore, it is encouraged by the major vendors of FPGAs to choose a SWI kernel instead of NDR kernel to better leverage the full potential of their architecture. A given function is designed as a deep pipeline with multiple stages. A key point in pipeline parallelism is to avoid memory stalls due to data or memory dependencies. A stall will hold a computation when the required data is not available, making some stages of the pipeline idle at some clock cycles. Also, the II optimization is another key for this kind of kernel. It should be noted that the ideal value of II is 1, which means that the design produces a new result at every clock cycle once the pipeline is filled. For a SWI kernel, particular attention is given to loops for optimization purposes since loops are vastly present in compute-intensive applications.

Listing 1.2 – SWI kernel

```
1 __kernel void SWI_kernel(//kernel arguments
2                 ){
3     for(int index = 0; index<N; index++){
4           // Some computations
5     }
6 }
```

1.4.4 Loop optimizations

In compute-intensive applications, we have a nest of nested loops to perform processing on large data. For simplicity of illustration, the following optimizations are

often applied to a single loop but are equally applicable to nested loops. One of these loop optimizations consists of merging the nested loops into one.

1.4.4.1 Loop pipelining

Loop pipelining consists of subdividing a loop's body into several micro-operations (called pipeline stages) that can be executed concurrently, as shown in Fig. 1.7. The aim is to overlap the loop iterations so that one iteration can start before the completion of its predecessor. A new iteration has to start as soon as possible to reduce the overall clock cycles of the loop. The number of clock cycles between two consecutive iterations is the II value of the pipeline. The same concept is present in general-purpose processors, allowing one to use the available resources on the device at its full potential. This is even more interesting on FPGAs as the synthesis tools synthesize custom designs based on the user's input description. Instead of using a fixed architecture, as is the case with general-purpose processors, HLS tools can generate a custom pipeline for each loop of a program according to the operations performed on FPGA. Loop pipelining uses FPGA hardware efficiently and avoids resource under-utilization. Therefore, loop pipelining improves the area usage and the design performance simultaneously. The AOC compiler attempts to pipeline,

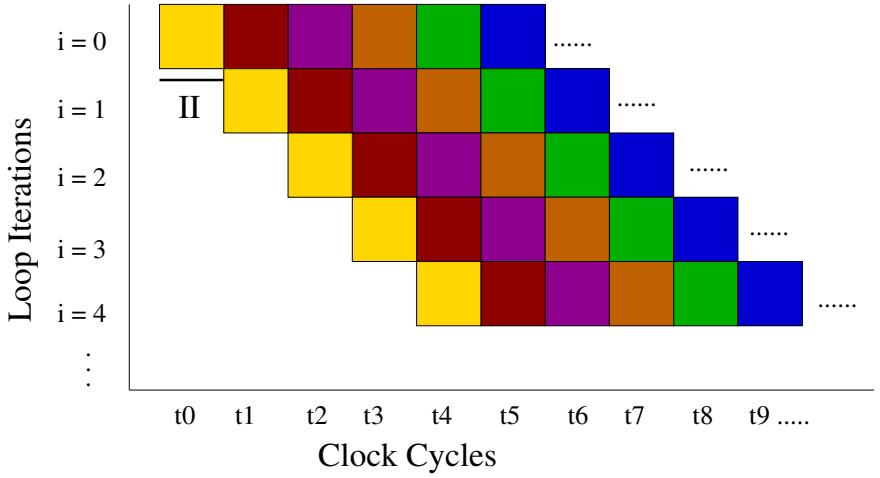


Figure 1.7 – Loop pipelining

by default, all the loops in the kernel for SWI with an II value of one (the optimal value). Loop-carried dependencies must be avoided to achieve the optimal II value. When the dependency is not avoidable, the designer should manage to handle it in one clock cycle by inferring shift register, for instance, or any other relevant optimizations. One important limitation of loop pipelining on FPGAs is the necessity to know the loop trip count at compile time to generate an efficient pipeline. When the loop trip count cannot be determined at compile-time, the compiler will not be able to analyze the loop and evaluate the loop exit condition, which will lead to the generation of a loop pipeline with high II value. Consequently, some recent

works have focused on source-level compiler-based optimization for loop pipelining using HLS compilers. The authors of [Liu 2018] have proposed a compiler-based loop optimization technique for uncertain and non-uniform memory dependencies. The pipeline benefits from the compile-time analysis and optimizes the runtime dependencies in order to improve the loop II and the overall performance. Also, a speculative loop pipelining technique has been proposed in [Derrien 2020] to handle dependencies in the pipeline and improve throughput. Their source-level approach is based on decoupling the data path and the control of the pipeline, and is meant to be included in HLS compilers without adding any complexity for the programmers.

1.4.4.2 Loop unrolling

Loop unrolling consists of partially or fully replicating the loop's body. The loop unrolling replicates the hardware resources within the loop scope to maximize the throughput. Fig. 1.8 shows an example of a loop where five iterations are running in parallel. Unrolling a loop maximizes the number of memory access per cycle leading to good exploitation of the memory bandwidth. Memory accesses for each parallel instance of the loop can be coalesced in burst mode when the access is contiguous in memory or if the compiler can statically analyze the access pattern. The loop trip also should be known at compile-time to effectively unroll the loop and ensure good throughput without wasting valuable hardware resources. Also, to apply a partial unrolling, the trip count should be divisible by the unrolling factor to reduce the resource overhead.

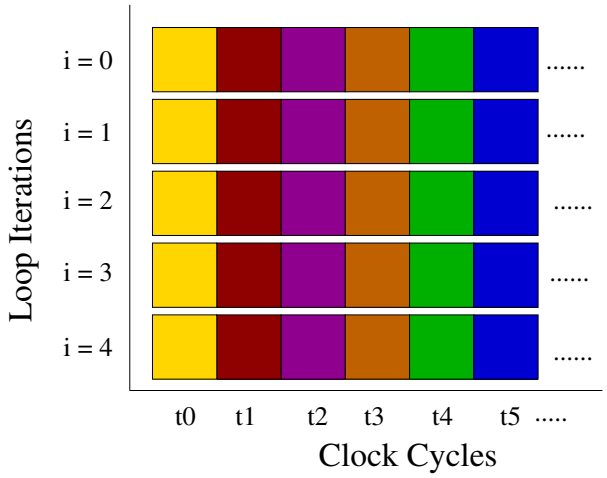


Figure 1.8 – Loop Unrolling

Unrolled loops still get pipelined and therefore benefit all the advantages of pipelining. Hence, a combination of these two techniques is widely used in SWI kernels on FPGAs to achieve good performance.

1.4.4.3 Nested loops fusion

Compute-intensive applications contain several nested loops. Loops with such configurations consume significant FPGA hardware resources with relatively long latencies. It is possible to merge these nested loops into a single loop in order to minimize pipeline latency and reduce the FPGA area. The *loop_coalesce* pragma is used to do this as in the code below:

Listing 1.3 – Loop interleaving

```
#pragma loop_coalesce
for(size_t j = 0; j < M; j++){
    for(size_t i = 0; i < N; i++){
        //Computation on (i,j)
    }
}
```

The compiler then converts the code above into the following one:

Listing 1.4 – Loop interleaving

```
size_t i = 0, j = 0;
while(j < M){
  //Computation on (i,j)
  i++;
  if(i == N){
    i = 0;
    j++;
  }
}
```

In case of multiple loops, one can specify a value (e.g., *#pragmaloop_coalesce* 2) to assert to the compiler how many loops are being coalesced (no value means all loops are merged). This pragma must be used with caution for some kernels because it might result in longer latencies and II values.

1.4.4.4 Initiation Interval (II)

The II value represents the number of the cycle between the launch of two consecutive loop iterations applied to SWI kernels. This parameter is a valuable key for optimization when dealing with pipeline execution on FPGAs. The compiler's main objective is to achieve an II value of one whenever possible, even at the cost of lower operating frequency. The performance model in (1.1) determines the number of clock cycles a pipeline might take to complete the execution.

$$cycles = P + II(N - 1). \tag{1.1}$$

Where P represents the pipeline depth, N the loop trip count, and II the initiation interval of the loop (or the function). We can notice that a high II value may result in a longer execution time for the design. The optimization of this parameter goes through the relaxing of the loop-carried dependencies, the reduction of the

accesses to the global memory, and ensuring stall-free memory accesses without any conflict. The compiler implements several basic optimizations through static analysis to optimize this parameter.

The ii pragma is available to tune the loop initiation interval value manually. Since the first priority of the compiler is to achieve the optimal II value, using this pragma to reduce the compiler-specified value may drastically degrade the operating frequency or cause synthesis failure due to unmet user constraints. Therefore, the pragma is only used to increase the compiler-specified II value in order to prioritize the operating frequency over non-crucial loops in the design. Indeed, the design may contain loops that are not in the critical path. These loops can have an II value higher than the optimal value without impacting the design performance. The next code shows the use of this pragma:

Listing 1.5 – ii pragma

```
1 #pragma ii <desired_value>
2 for(size_t i = 0; i < N; i++){
3   //Do something on vect
4 }
```

1.4.4.5 Other loop pragmas

There are many other loop optimizations detailed in Intel documents. Some of them are automatically applied by the compiler, while others request user intervention. However, the user can control them through *pragmas* to enable or disable them. The OpenCL compiler generates an HTML report for each kernel in order to allow the designer to go through the design information.

The ivdep pragma:

The ivdep pragma is used to assert to the compiler that there will be no load/store dependency between two particular instructions, even though his static analysis is unable to prove so. In the case of actual load/store dependencies, using this pragma can result in functional incorrectness. This pragma can also be used with the **safelen** parameter to specify the number of iterations that can execute before the dependencies appear.

The max_concurrency pragma:

The *max_concurrency* pragma enables concurrent loop iterations in parallel for significant throughput. Unlike loop unrolling, the loop body is not replicated when applying the *max_concurrency* pragma but only the memory objects that are private to the loop. This pragma creates several copies of local memory to store data for consecutive iterations required for concurrent computation. This pragma intends to keep a compute unit in the design occupied so all of them can perform a valid operation at each clock cycle. However, using this pragma on a loop that

contains a local memory object may result in extra BRAM usage due to the private copies created to support concurrent execution of the loop. Hence, one may use this pragma to control FPGA resource usage especially BRAM.

Listing 1.6 – Loop concurrency

```
#pragma max_concurrency M
for(size_t i = 0; i < N; i++){
  float vect[size];  //Do something on vect
}
```

The max_intereaving pragma:

The *max_intereaving* pragma is applied to nested loops and it is used to interleave the pipelined nested loop's iterations. This is applied when the compiler is unable to achieve an II of 1 for the inner loop. In order to increase the pipeline occupancy, the compiler interleaves the inner loop iterations with the outer loop to minimize the design II. For instance, for a given nested loops (j, i) as presented in the code below, the innermost loop having an II value greater than one (e.g., II=2) will lead to a pipeline inefficiency. A new iteration of i loop is launched every two clock cycles.

Listing 1.7 – Loop interleaving

```
for(size_t j = 0; j < M; j++){ // Loop with II=1
  #pragma max_interleaving 1
    for(size_t i = 0; i < N; i++){ // Loop with II=2
        //Do something
    }
}
```

Cycle	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
1	(0,0)	–	–	–	–
2	–	(0,0)	–	–	–
3	(0,1)	–	(0,0)	–	–
4	–	(0,1)	–	(0,0)	–
5	(0,2)	–	(0,1)	–	(0,0)
6	–	(0,2)	–	(0,1)	–
7	(0,3)	–	(0,2)	–	(0,1)
8	–	(0,3)	–	(0,2)	–

(a) Without interleaving

Cycle	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
1	(0,0)	–	–	–	–
2	(1,0)	(0,0)	–	–	–
3	(0,1)	(1,0)	(0,0)	–	–
4	(1,1)	(0,1)	(1,0)	(0,0)	–
5	(0,2)	(1,1)	(0,1)	(1,0)	(0,0)
6	(1,2)	(0,2)	(1,1)	(0,1)	(1,0)
7	(0,3)	(1,2)	(0,2)	(1,1)	(0,1)
8	(1,3)	(0,3)	(1,2)	(0,2)	(1,1)

(b) Interleaved

Figure 1.9 – Loop interleaving

In order to fully harness the pipeline, one could interleave the iterations of the two loops. Fig.1.9 shows the execution pattern of such (j,i) loop when applying interleaving or not. At the first clock cycle, the iteration $(j = 0, i = 0)$ will be initiated. Under normal pipeline execution (without the pragma) the second clock cycle is idle due to the loop II. However, thanks to the interleaving iteration $(j = 1, i = 0)$ can be launched. Then, at the third cycle the execution of iteration $(j = 0, i = 1)$ starts which is followed by $(j = 1, i = 1)$. The execution is performed this way until complete invocation of loop iterations. All stages of the pipeline are occupied by cycle 5 with no idle stages.

1.5 Conclusion

In this chapter, we have presented an overview of the computing architectures usually used in HPC as well as the development tools. Of particular interest are the FPGA architectures used in this work. FPGA technology used to be accessible only by a handful of hardware designers. Thanks to the HLS tools, this technology is now available to a large number of developers. Considering the advances in their internal architecture, FPGAs have become a potential candidate for HPC applications. However, to take advantage of their architecture using HLS tools, several FPGA-specific optimizations, some of which have been presented in this chapter, need to be applied. In the rest of this manuscript, we will present other more advanced optimizations and the application of these optimizations to real-world applications.

Chapter 2

SKA, an HPC use-case for FPGA-based HLS

Contents

Radioastronomical imaging aims to produce an image of the sky from the incoming signals delivered by radio-telescopes. Radio-telescopes produce high dimensional data requiring powerful computing systems in order to reconstruct the sky images under performance and energy efficiency constraints. Therefore, FPGAs are a potential alternative to GPUs to accelerate the complex algorithm chain. In this chapter, we present an overview of the imaging system used in radio interferometry and then focus on the acceleration of one of the steps of the imaging system, deconvolution. Square Kilometre Array (SKA) is the largest radio telescope in the world aimed at generating multidimensional images of the sky from a data stream at 7.2 Terabit/s without any storage capabilities. The data collected from the antennas will go directly into the SKA pipeline for real-time processing in several stages. In this pipeline, we are interested in the Science Data Processor (SDP) stage for the reconstruction of the sky image. Indeed, the data from each pair of antennas are correlated and calibrated before being processed by the imaging system. The input to the imaging system is the calibrated visibilities. This chapter presents the imaging system of the SKA pipeline and the iterative algorithms used to reconstruct the

sky image. The chapter uses the deconvolution use-case to illustrate the design of the algorithm on FPGA using OpenCL HLS tool.

2.1 Square Kilometre Array (SKA) imaging

This section deals with iterative optimization algorithms for solving inverse ill-posed problems and image reconstruction in radio astronomy. The sky image reconstruction problem is indeed an ill-posed inverse problem whose imaging system is presented here.

2.1.1 Iterative algorithms

Iterative methods are used to solve ill-posed inverse problems. The linear model of the reconstruction problem is modeled as:

$$g = Hf + \varepsilon \tag{2.1}$$

where g denotes measurements from an observation system, f is the unknown object, H is the linear model describing the response of the observation system, and ε represents the additive noise. The solving method can minimize the criterion without constraint, which does not guarantee the optimal solution due to the character of ill-conditioning of the system matrix H. Iterative methods with regularization are used to correct this ill-conditioning of the matrix by introducing regularization terms to impose constraints in order to penalize undesired solutions. The cost function J to be minimized is described as:

$$\begin{aligned} J(f) &= \frac{1}{2}||g - Hf||^2 + \lambda||Df||^2, \\ J(f) &= J_{LS} + J_{Reg}, \end{aligned} \tag{2.2}$$

where $\frac{1}{2}||g - Hf||^2$ is the data fidelity term, and $\lambda||Df||^2$ is the regularization term with λ as the regularization parameter. The first term intends to minimize the distance between the observation and estimation. The second introduces a prior knowledge in order to penalize certain solutions to the problem.

$$\hat{f} = \arg\min_f J(f). \tag{2.3}$$

Several least-square regularization approaches have been proposed in the literature such as the Total Variation regularization [Rudin 1992] or Tikhonov regularization [Tikhonov 1995].

The optimization algorithm used to minimize the cost function is the gradient descent:

$$f^{(n+1)} = f^{(n)} - \alpha.\nabla J(f^{(n)}), \tag{2.4}$$

with

$$\nabla J(f^{(n)}) = -2H^t(g - Hf) + \lambda D^t Df. \tag{2.5}$$

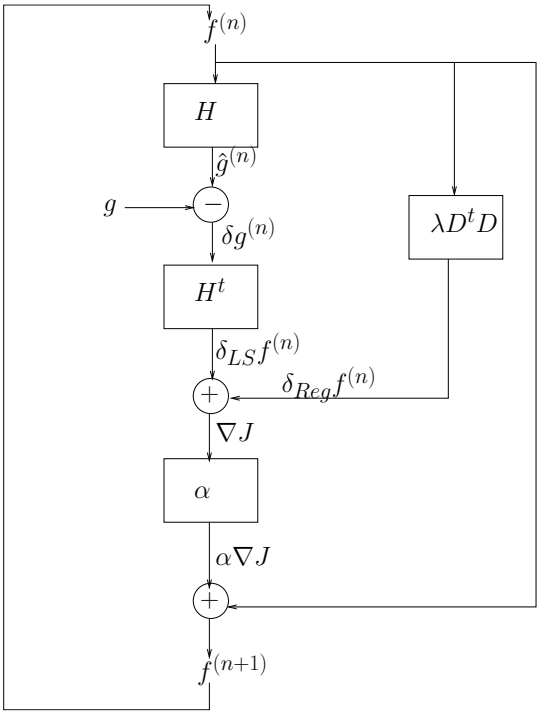


Figure 2.1 – Block diagram of iterative image reconstruction

Fig. 2.1 describes the bloc diagram of the iterative gradient descent optimization algorithm with regularization. The iterative nature of such optimization algorithms makes them costly in terms of time and computational resources. The method of solving this inverse problem without regularization does not guarantee the optimal solution, so it is more appropriate to solve the problem with regularization to impose constraints in order to penalize undesired solutions. The implementation of the iterative algorithm on hardware accelerators is also a major concern. An algorithm-architecture co-design approach is required to efficiently implement the algorithm to accelerators and respect the application's constraints. We discuss in this section the imaging system of the SKA pipeline. The different strategies for porting the iterative algorithm to the FPGA board will be presented in Chapter 5 for the tomography use-case.

2.1.2 SKA imaging

The purpose of the imaging system is to produce the sky image from the visibilities. Gridding and degridding operations are therefore used in this step. Indeed, the visibilities are in the Fourier space. However, these visibilities are rarely distributed on regular grids, which makes it impossible to use the inverse FFT (iFFT) to

produce a dirty image. Thus the purpose of the gridding operator is to place the calibrated visibilities onto regular grids followed by the iFFT to produce the dirty image. Since the process is iterative, the degridding allows restoring visibilities from the dirty image preceded by the FFT.

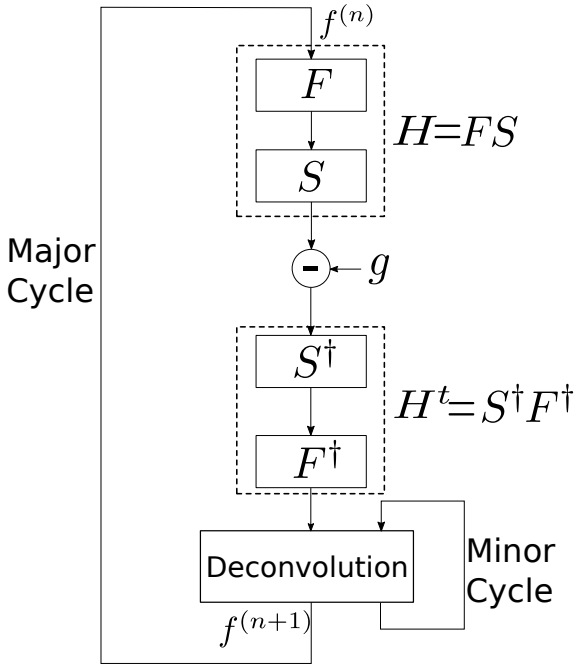


Figure 2.2 – Bloc diagram of radio astronomical imaging

Fig. 2.2 presents the bloc diagram of the imaging step, which is also an inverse problem $g = Hf + \varepsilon$ solved by iterative algorithms. The diagram has two levels of iteration: the *major iteration* and *minor iteration*. Indeed, the deconvolution step in the iterative algorithm is also an inverse problem that requires an iterative optimization algorithm to estimate the solution. The deconvolution which represents the *minor loop* is detailed in section 2.2. The forward model $H = FS$ consists of consecutive operations of 2D FFT and degridding. The operator F corresponds to the 2D FFT of the estimated cleaned image, which then passes through the degridding step S. The degridding step produces the predicted visibilities, which are then compared to the calibrated visibilities g. Then, the remaining visibilities are gridded by the gridding operator $S^\dagger$. The gridding is used to place the calibrated visibilities onto regular grids to allow the use of the 2D FFT algorithm instead of the expensive discrete Fourier transform. The iFFT operator $F^\dagger$ is applied to the calibrated visibilities in order to produce the dirty image. The dirty image is then deconvolved in the deconvolution step representing the *minor loop*.

The gridding and degridding steps are the most computationally intensive operators. These algorithms have a high parallelism potential on GPUs with substancial acceleration factors. Nevertheless, despite of the powerful computation offer by the GPUs, these devices are also very power-hungry. With this in mind, FPGAs seem

to be a strong potential candidate due to their energy efficiency. For this reason, the Image Domain Gridding (IDG) algorithm [Van der Tol 2018] has been widely evaluated on FPGAs using HLS tools. For instance, an FPGA implementation of the IDG algorithm using OpenCL has been presented in [Veenboer 2019]. The authors showed that FPGAs perform better than optimized CPU implementation. However, GPUs perform better than FPGAs for the gridding and degridding algorithms regarding throughput and energy consumption. The same IDG algorithm have been designed using Xilinx FPGAs by exploring different precisions in data representation [Corda 2022]. Indeed, FPGAs can support several levels of precision other than floating-point data type, and the authors have investigated the use of fixed-point for sky image reconstruction. They proceeded to the evaluation of the minimal precision requirement for data representation in radio astronomy in order to perform the design with a customized precision on FPGAs. Their work on precision evaluation has improved the performance of the IDG on FPGAs, particularly those from Xilinx. However, these performances still do not reach those obtained on GPUs for gridding and degridding algorithms. In this manuscript, we will only consider deconvolution, which is also an ill-posed inverse problem representing the minor loop. The deconvolution is an inverse problem, which is used to illustrate the FPGA design through OpenCL tool.

2.2 Deconvolution

The dirty image produced from the visibilities suffers from high artefacts because the sky image is convolved with a Point Spread Function (PSF). Image deconvolution intends to remove this convolution effect and produces an accurate image of the sky. Fig. 2.3 shows a PSF, a dirty image, and the corresponding deconvolved image. The deconvolved image has a smaller noise level than the dirty image. Several deconvolution algorithms are reported in the literature, mostly based on least square or CLEAN variant methods. In this manuscript, as the gridding and degridding steps have been actively explored on FPGA, we are interested in the step of dirty image deconvolution to restore the sky image. We will present in this section the deconvolution based on the least-square and CLEAN methods. We will highlight our interest in the CLEAN algorithm due to its simplicity compared to the least-square method.

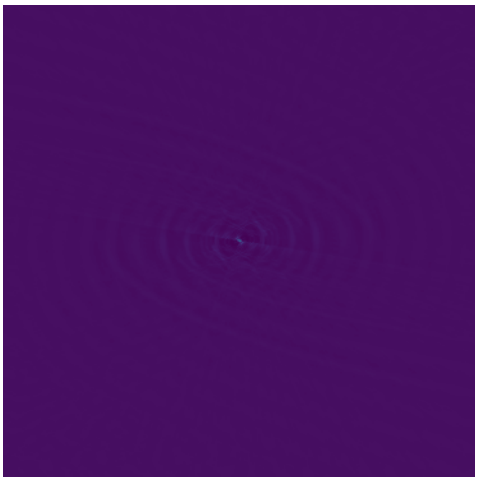
2.2.1 Least square deconvolution

The image deconvolution is an ill-posed inverse problem which can be solved by regularized least-square approach. The cost function to be minimized is given by:

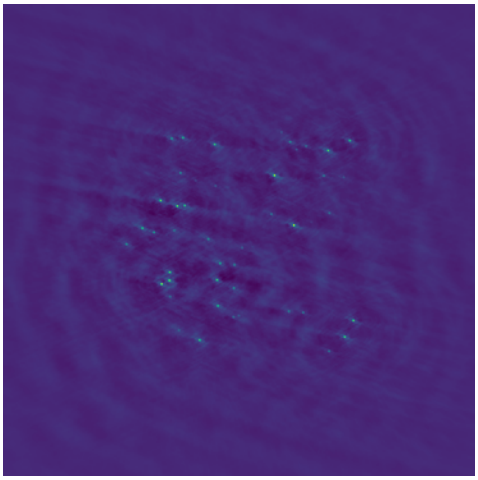
$$J(f) = \frac{1}{2}||g - Hf||^2 + \lambda||Df||^2, \tag{2.6}$$

The solution is then:

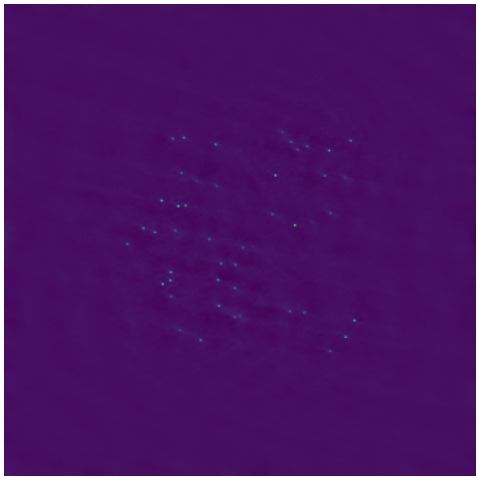
$$\hat{f} = (H^t H + \lambda D^t D)^{-1} H^t g. \tag{2.7}$$



(a) PSF



(b) Dirty image



(c) CLEAN image

Figure 2.3 – Image deconvolution

The gradient descent optimization is used to estimate the next image from the current estimation iteratively as shown in the diagram presented in Fig. 2.1.

The least-square deconvolution using the gradient descent method, as shown in Algorithm 1, can be parallelized on FPGA. Indeed, the deconvolution invokes several convolution operations iteratively. The computation of δ_{LS} (in Algorithm 1 line 4) requires convolution where the image and the mask sizes are large. This convolution is performed in the Fourier domain by element-wise multiplication using the convolution theorem. However, the convolutions required in the δ_{Reg} computation involve small kernels which can be efficiently performed in the image domain. The rest of the computation (∇J, $f^{(n+1)}$) can also be implemented on FPGA, which reduces the data transfer rate between the host and the kernel considerably. The complete gradient descent optimization can thus be performed on the FPGA in order to deconvolve the dirty image. However, due to its computation overhead,

Algorithm 1 Least square deconvolution

1: Set $f^{(0)}$
2: $n \leftarrow 0$
3: **repeat**
4: $\quad \delta_{LS} \leftarrow -H^t(g - Hf^{(n)})$
5: $\quad \delta_{Reg} \leftarrow \lambda D^t D f^{(n)}$
6: $\quad \nabla J \leftarrow \delta_{LS} + \delta_{Reg}$
7: $\quad f^{(n+1)} \leftarrow f^{(n)} - \alpha \nabla J$
8: $\quad n \leftarrow n + 1$
9: **until** Convergence is reached
10: **return** $f^{(n)}$

the least square deconvolution is not widely used in sky image reconstruction. The implementation of the gradient descent algorithm will be discussed in Section 5.6 in the context of tomography reconstruction. Alternatively, other deconvolution algorithms are used in radioastronomy for their efficiency. These algorithms are the so-called CLEAN variants, and we consider the original CLEAN algorithm due to its simplicity.

2.2.2 CLEAN deconvolution

The simplest and most commonly used deconvolution algorithm is the Högbom CLEAN [Högbom 1974]. The CLEAN algorithm intends to find iteratively the brightest peaks in a dirty image and subtract its contribution from the dirty image. The algorithm converges when a certain user-specified threshold is reached. The CLEAN algorithm is not a computational burden compared to gridding and degridding in the imaging pipeline. However, the CLEAN algorithm is memory-bound and the iterations of the procedure are sequential which cannot be parallelized. Within each iteration, the operation can be performed independently on each pixel of the image presenting a parallelism potential. The procedure of this algorithm is described in Algorithm 2.

The candidate image $\hat{I}$ (or model image) is empty and the dirty image I^D is copied in the residual image I^{res} as initialization. The intensity ($intensity_{p,q}$) and position (p,q) of the peak in the residual image (I^{res}) is searched in the first place (line 5). The procedure consists of building a candidate image $\hat{I}$ with the peak intensity scaled by the factor γ (line 6). Then, subtract from the residual image the PSF convolved (or multiplied in the Fourier domain) by $\hat{I}$ (line 7). The peak intensity of the updated residual image is searched again. This procedure is performed iteratively until the intensity is below a user-specified threshold or when a maximum number of iterations is reached. After convergence, the image $\hat{I}$ is convolved with an ideal clean beam, which is usually a 2D elliptical Gaussian fitted to the central lobe of the dirty beam.

Several variants of the CLEAN algorithm have been proposed in the littera-

Algorithm 2 Högbom CLEAN procedure

Require: I^D, PSF, γ, threshold, $iter_{max}$
Ensure: $\hat{I}$, I^{res}
1: $n \leftarrow 0$
2: $\hat{I}_{p,q} \leftarrow 0$
3: $I^{res} \leftarrow I^D$
4: **while** $intensity_{p,q} > threshold$ *and* $n < iter_{max}$ **do**
5: $\quad intensity_{p,q} \leftarrow$ find_ peak(I^{res})
6: $\quad \hat{I}(p,q) \leftarrow \hat{I}(p,q) + \gamma.intensity_{p,q}$
7: $\quad I^{res} \leftarrow I^{res} - PSF * \hat{I}$
8: $\quad n \leftarrow n + 1$
9: **end while**
10: $\hat{I}_{p,q} \leftarrow$ convolve($\hat{I}$, clean_beam)

ture in order to accelerate the deconvolution. For instance, the Clark CLEAN [Clark 1980] is an improvement of the original Högbom CLEAN for large image. The algorithm performs the deconvolution by searching the peak as in the original algorithm but working on a sub-region of the image. This peak search is called the minor iteration. The major iteration in this case consist of Fourier transforming the CLEAN components using FFT and sustracting it from the dirty image. Another variant of CLEAN is presented by [Schwab 1984] which includes the degridding step in the major iteration. The subtraction is performed at the visibility level which is more accurate than the Clark CLEAN. A multi-scale CLEAN algorithm, which is an extension the original Högbom CLEAN, has been proposed by [Cornwell 2008]. The multi-scale CLEAN also includes the degridding step in the major iteration. It uses different scale sizes to convolve the dirty image. The multi-scale algorithm operates simultaneously on all scales under consideration. Indeed, the dirty beam is convolved with each scale size which is precomputed before the procedure. Therefore, the peak search is performed on a set of images for different scales. An extension of the multi-scale CLEAN has been developped as multi-scale multi-frequency CLEAN variant [Rau 2011]. Nevertheless, we only consider the original Högbom CLEAN in this manuscript in order to avoid the use of gridding and deggriding operators in the variants that include the major iteration.

2.2.3 FPGA-based CLEAN using OpenCL

We implement the Högbom CLEAN presented in Algorithm 2 on Intel FPGA using OpenCL. The iterative loop of the algorithm is entirely carried out on FPGA instead of CPU. The crucial steps of this algorithm are the search for the peak intensity, the construction of the CLEAN map, and the update of the dirty image. This algorithm does not use many DSP blocks on the FPGA board. The algorithm has been described using the SWI and NDR model in OpenCL.

2.2.3.1 SWI design

The most crucial parts of the algorithm, which are the peak search (Algorithm 2, line 5), and the residual image update (line 7), can be synthesized in the same bitstream into different kernels. However, we merge these parts into one deeply pipelined kernel. The full iterative algorithm runs on FPGA. The host CPU is in charge of transferring data into the device memory, launching the kernel, and then transferring the results back. The peak search is a memory intensive operation which can be easily parallelized on multiple data. Also the access to the external memory is performed contiguously which is better to take advantage of memory coalescence and maximize the throughput. The reference version corresponds to the sequential version without any advanced OpenCL optimization. This version is an implementation designed for FPGA, considering the specificity of its architecture to have pipelined loops. Indeed, all loops are pipelined with a II of 1 in this implementation which is crucial for the SWI model. The peak search and the dirty image update are pipelined blocks with no data parallelism expression. The optimized version exploits OpenCL basic optimizations presented in Section 1.4. This pipelined version is then optimized by expressing parallelism to maximize the global memory bandwidth. The peak search has the potential for parallelism, where each pixel can be considered separately to evaluate the maximum value of the image. We therefore apply loop unrolling within this block to maximize bandwidth. Similarly, the image update, which requires some arithmetic operations (subtraction and multiplication), is also parallelized with loop unrolling. Each image pixel is updated independently of the others because there is no data dependency. The peak search does not require several computations and therefore has no impact on the DSPs consumption, but the update of the dirty image may slightly affect the number of DSPs used which is still very low for this algorithm.

The iterative nature of the algorithm, whose exit condition depends on a value calculated within the loop ($intensity_{p,q}$ calculated at line 5), means that the *while loop* loop is not pipelined. The iterations of the *while loop* in Algorithm 2 are executed sequentially. This is a limitation for the SWI execution model, which may well achieve better performance in the case of full pipelining. However, all other pipeline blocks, such as the peak search, are pipelined deeply to achieve better throughput. However, this is still better than launching the iterative loop from the host processor regarding the design throughput.

2.2.3.2 NDR design

The NDR version of the algorithm has also been described in order to compare it with the SWI version. The reference version is the parallel implementation of the sequential version of the algorithm where multiple work-items deal with different image pixels. The reference version only contains one compute unit and one work-group. Each work-item passes through this compute unit to process the algorithm. The pipelining is then at work-item level where the aim is to launch a new work-item every clock cycle. The optimized version is then designed to express parallelism

and use local memory to speed up the algorithm. In this version, parallelism is expressed using replication of the compute unit and loop unrolling to maximize bandwidth utilization. Multiple compute unit allows to execute different work-group on different compute unit which contributes in the reduction of memory stall. As the algorithm is memory-bound, it is essential to ensure that the memory is accessed stall-free. We use multiple work-groups to perform the peak search by reduction. The peak search is performed partially within each work-group and then we select highest peak among all the partial maximum intensity values from all work-groups. The work-items from the same work-group have access to the same data in the local memory. This NDR version does not suffer from the *while loop* problem because none of the loops are pipelined for this kernel model.

Parallelism is expressed at the thread or work-item level. The peak intensity search must be well managed because there is a need for communication between the work-items. In order to express parallelism, several work-items work simultaneously to search for the peak intensity, and it is necessary to synchronize them at a given moment. To better parallelize this task, we proceed to a search by reduction. This means that a partial peak search is performed within each work-group using local memory. This way, all work-items share the same data in local memory. Then a reduction between each work-group must be performed to keep only the maximum intensity. It is then necessary to have synchronization barriers to ensure that all work-items have finished before proceeding to the reduction among different work-groups. In this case, it is essential to specify the work-group size to allow the compiler to better manage the sharing of FPGA resources and ensure good performance. Another synchronization barrier is necessary between the update of the dirty image and the peak intensity search to avoid computational errors. We also apply loop unrolling to the non-pipelined loops in the design in order to increase the number of memory requests per clock cycle and maximize the bandwidth. Indeed, this could have been achieved by SIMD vectorization but due to the reduction, some conditional branching are thread ID-dependent. Therefore, the SIMD vectorization cannot be applied. The work-group size is then fixed with caution to reduce the BRAM consumption while delivering good performance.

2.2.3.3 Results

We have evaluated these designs on the DE10-Pro board equipped with the Intel Stratix 10 device (1SG280HU2F50E2VG). The parameters of the CLEAN algorithm are shown in Table 2.1. The sizes of the input dirty image and the PSF used in this experiment are 1280×1280 and 2560×2560, respectively. The PSF should be at least twice as large as the dirty image (in both directions). The dirty image is cleaned after 436 iterations when the use-specified threshold is reached. The computations are performed using single-precision floating-point data. We discuss the performance of the CLEAN deconvolution as presented in Table 2.2.

We compared the reference and the optimized versions regarding performance and design efficiency. Compared to the reference versions, the optimized versions

Table 2.1 – The CLEAN algorithm parameters

Parameter	γ	Threshold	$iter_{max}$	Dirty image	PSF
Value	0.2	0.1	1280	1280×1280	2560×2560

show an acceleration of the CLEAN algorithm. The resource usage shows that the designs do not use a high number of DSPs (less than 1%) because the computational intensity is very low. The CLEAN algorithm performs much more memory operations than computations, which explains the slightly high usage of BRAM memory. The SWI version achieves better performance than the NDR version for several reasons. This design's loops are fully pipelined with an initiation interval of one except for the *while loop*, which is not pipelined. However, updating the dirty image step contains loops whose exit condition could not be statically determined by the compiler resulting in high II values. Those loops have been modified with

Table 2.2 – Performances of the Högbom CLEAN deconvolution on Intel Stratix 10 device.

Design	Kernel	BRAM (%)	DSP (%)	Stall (%)	Occ (%)	Freq. (Mhz)	Time (s)	Energy (mWh)
Reference	SWI	4	< 1	11.3	28.8	361	22	140.5
	NDR	8	< 1	10.75	24.7	162.5	52.3	334
Optimized	SWI	9	1	7.2	3.2	256.2	1.37	8.8
	NDR	16	< 1	30.57	11.7	173.8	13	83

fixed exit conditions to help the compiler to achieve a good II value. This design is efficient in terms of execution time as shown in Table 2.2, with a low stall percentage. The occupancy of the design is not very high due to the use of multiple computation units and the non-pipelining of the outer loop (*while loop*). However, with more aggressive pipelining the design occupancy can be increased for more performance.

Loops are not pipelined in NDR kernels. Thus this design does not suffer from the non-pipelining of the *while loop*. The NDR design should have achieved better performance than the SWI version. Nevertheless, this is not the case for this use case. The NDR design suffers from a high stall percentage due to the usage of several synchronization barriers. Also, the compute unit replication in the design is very resource-consuming because of the local memory overhead. Loop unrolling is used to maximize memory requests. We use loop unrolling instead of SIMD vectorization because the design contains thread ID-dependent branches due to the

peak search by reduction.

The optimizations have allowed us to significantly improve the CLEAN algorithm's overall performance. The NDR version suffers from the use of multiple synchronization barriers. The SWI kernel provides the best performance in terms of execution time and energy efficiency despite the non-pipelining of the outer loop. Also the SWI versions achieve better operating frequency than the NDR version. The stall percentage of the SWI version is relatively low. However, the occupancy rate of our compute units remains low, which shows that there is room for further optimization. With an adequate adaptation of the algorithm to the FPGA architecture, this occupancy rate could be increased and thus improve the pipeline performance.

The hardware resource usage is not very high for these implementations. There is no point in duplicating these pipelines further, as the bandwidth limitations mean there will be no performance improvement. Indeed, bandwidth congestion will increase the percentage of stalls, degrading the design throughput. Nevertheless, the remaining FPGA resources can be used for other operations like porting the whole sky image reconstruction algorithm with major and minor loops.

2.3 Conclusion

This chapter deals with the acceleration of the Högbom CLEAN deconvolution algorithm on FPGA using OpenCL. The OpenCL language significantly reduces the development time on FPGAs compared to HDL languages. OpenCL optimizations are applied to assist the compiler to generate an efficient pipeline. The results of the CLEAN for radioastronomical imaging are discussed. The generated pipeline of the SWI version is more deep and efficient than the NDR. Therefore, The SWI achieves better performance than NDR for this algorithm. The main bottleneck of this algorithm is the memory accesses which impacts the performance. We take advantage of memory coalescence to leverage the bandwidth and maximize the design throughput. The OpenCL optimizations presented in Section 1.4 allows to increase the design performance. However, more aggressive FPGA optimizations, in the concept of algorithm-architecture co-design, are required to exploit the full potential of FPGA devices. The CLEAN algorithm is a simple use case. In the next chapter, we discuss more complex algorithms for illustrating OpenCL-based FPGA design in the case of tomography reconstruction.

Chapter 3

X-ray Computed Tomography Acceleration

Contents

Image reconstruction in tomography is an inverse problem solved by analytical or iterative methods. Iterative methods use several operators to reconstruct the 3D image. This chapter presents the different methods used in the literature to solve these problems. The approaches used to model the forward and backward-projection operators are discussed in this chapter. We present an overview of the acceleration of tomographic algorithms on hardware accelerators. The chapter also presents some first results on the acceleration of projection and back-projection operators on FPGA using the OpenCL HLS tool.

3.1 Tomography reconstruction algorithms

3.1.1 Image reconstruction

X-ray Computed Tomography (CT) is an imaging technique that initially found its application in the medical field. It has been extended to industrial applications such as non-invasive human body investigation and non-destructive testing of industrial materials. 3D CT aims to acquire the internal density f of 3D objects from external measurements g called sinogram or projection data. The 3D object is placed between an X-ray source and a detector plane, as illustrated in Fig. 3.1. The source and the detector move around the object along the φ axis to collect the sinogram. The local density of the object attenuates the radiations emitted from the source. Therefore, the reconstruction problem is to restore the volume object f from the measured projection data. Initially, analytical methods were used to reconstruct the image based on the Radon transform [Radon 1917]. One such algorithm is the well-established Filtered Back-Projection (FBP) [Feldkamp 1984], which can provide acceptable image quality and reconstruction time for clinical routine. However, the FBP algorithm has a considerable drawback regarding the image quality due to its sensitivity to noise producing images with artefacts [Ziegler 2007, Pan 2009, Baek 2010, Nelson 2011]. Iterative algorithms have emerged since the early 1970s [Gordon 1970] in order to improve image quality and reduce the radiation dose [Willemink 2013]. The idea of iterative reconstruction is not new, though its use was unfeasible because of the computational cost before GPU arrival [Geyer 2015]. Model-Based Iterative Reconstruction (MBIR) algorithms [Fessler 2000, Thibault 2007] are proved to produce better image quality at the cost of expensive computational time. These methods have become the most widely used methods for image reconstruction nowadays. Their mathematical foundations based on the resolution of inverse problems offer better modeling of the acquisition and geometry system. As a result, these MBIR methods overcome the limitations induced by analytical methods with the improvement of image quality, the removal of artefacts, the reduction of radiation dose in the patient's body, etc.

The direct model of the problem can be expressed as:

$$g = Hf + \varepsilon \tag{3.1}$$

where g represents the projection data, f the volume to be reconstructed (object of interest), H the system matrix, and ε the additive noise. Due to the additional noise, the inverse of the H matrix cannot be calculated. In addition, there is not enough projection data to compute the exact solution f. Therefore, the reconstruction problem is characterized as ill-posed in Hadamard's sense.

The 3D image f is estimated by minimizing a quadratic criterion J:

$$\hat{f} = \arg\min_f J(f) = \arg\min_f (\frac{1}{2}||g - Hf||^2 + \lambda||Df||^2). \tag{3.2}$$

Iterative methods of reconstruction are used to solve this linear problem by

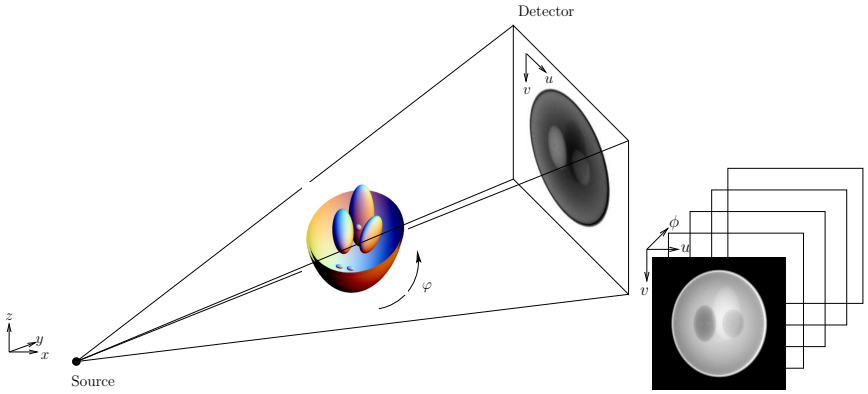


Figure 3.1 – X-RAY CT Projection.

performing several computations of the forward and backward projection using $J(f)$ the quadratic error and performing a gradient descent:

$$f^{(n+1)} = f^{(n)} - \alpha.\nabla J(f^{(n)}) \tag{3.3}$$

where $\nabla J(f^{(n)})$ represents the gradient of $J(f^{(n)})$:

$$\nabla J(f^{(n)}) = -2H^t(g - Hf) + \lambda D^t Df. \tag{3.4}$$

The solution f is approximated using the iterative algorithm until convergence is reached. Starting from an initial image $f^{(0)}$ computed from the measured sinogram, a new volume $f^{(n+1)}$ is calculated from the current estimation $f^{(n)}$, the gradient of the cost function and the function decreasing step size α.

The large size of the system matrix makes the storage of the matrix H and H^t impossible. Consequently, the coefficients of these matrices are computed on the fly by the forward and the backward projections, which are the most time-consuming steps in MBIR CT.

3.1.2 Projection/back-projection pairs

3.1.2.1 Unmatched pairs

As mentioned in the previous section, the matrix multiplication operations are not performed as presented in the equations due to the difficulty of storing the system matrix. Instead, the elements H_{ij} and H^t_{ij} of the system matrix are estimated on the fly by the projection and back-projection operators, respectively. These two operators are constantly invoked in MBIR methods. Several approaches have been proposed for the forward and back-projection, providing a trade-off between accuracy and computational cost. The voxel-driven and ray-driven approaches [Herman 1980, Zhuang 1994] have been first proposed to model the forward and back-projection for CT algorithms and are widely used in most applications. The voxel-driven approach traces the ray from the X-ray source through the center of the

voxel. The voxel-driven projection consists of accumulating each voxel's contribution in the corresponding detector cells. Its adjoint back-projection aims to update each voxel by the detector's cells that have contributed to it. A parallel implementation of a voxel-driven projection and back-projection pair will be problematic due to the access conflict in the projector. Indeed, multiple voxels might try to write in the same detector cell at a time, causing access conflict and leading to poor performance. However, its adjoint back-projector does not have such a limitation making it more convenient. Alternatively, the ray-driven approach is based on the ray-tracing technique from the source through the center of the detector. This approach is based on calculating the integral line along the ray with a specific weighting. Similarly, the ray-driven back-projector also follows the ray path and updates all the intersected voxels by the value of the corresponding detector cells. The ray-driven projection and back-projection present some limitations for parallel implementations. Like the voxel-driven projector, the ray-driven back-projector suffers from access conflict when several rays try to write in the same voxel at a time. Additionally, voxel-driven projection and ray-driven back-projection can also induce high-frequency solid artefacts in the reconstructed image, degrading the image quality [De Man 2002]. To address these issues, it has become common to use an unmatched ray-driven/voxel-driven pair of projector and back-projector in order to have a better between trade-off speed and accuracy. However, the reconstruction using unmatched projection and back-projection is subject to an additional problem because the matrix H^t is not the exact transpose of the forward matrix H. This approximation may lead to sub-optimal reconstruction and cause convergence concerns and, therefore, constitutes a drawback for their utilization in MBIR algorithms.

3.1.2.2 Matched pairs

Several methods have been proposed to overcome the issues mentioned above, using a matched projection and back-projection pair. The state of the art distance-driven approach [De Man 2002, De Man 2004] is one of such method. The distance-driven method allows a matched pair to perform the reconstruction while improving the image quality and avoiding artefacts. With acceptable computational complexity, the distance-driven approach combines the advantage of voxel-driven and ray-driven methods without their limitations. Another approach called separable footprint [Long 2010] has been proposed to enable the use of matched pairs. The separable footprint methods deal with separable functions to approximate the voxel footprint. The separable footprint methods are more computationally expensive due to the high complexity while ensuring better image accuracy than the distance-driven approach. The voxel footprint is approximated in the case of distance-driven by a rectangle covering the delimited region in the axial and the transaxial directions. However, the separable footprint performs this approximation using a rectangle in the axial direction and a trapeze in the transaxial direction for the Separable Footprint Trapezoidal-Rectangular (SFTR) projector. A trapeze is used in both

directions for the Separable Footprint Trapezoidal-Trapezoidal (SFTT) projector.

Other approaches based on splines have been recently proposed, such as spline-driven [Momey 2015] and magnification-driven [Savanier 2021] using cubic B-spline as the basis function to represent the object instead of the cubic voxel. These approaches use a more accurate mathematical model depending on the beam geometry to reconstruct the object with minimum projections number. These methods provide better image quality than the separable footprint methods at the cost of high computation complexity for a higher degree of B-splines.

Despite all these proposals for matched pairs, the unmatched pair (ray-driven projectors/voxel-driven back-projector) is still the most widely used in industrial CT reconstruction systems due to the trade-off between accuracy and computational cost offered by this pair. Therefore, in this work, we only focus on the unmatched pair by considering the ray-driven forward and voxel-driven backward projectors.

3.1.2.3 3D Projection algorithms

This section deals with ray-driven projectors used in iterative image reconstruction methods. The projection algorithms considered are those proposed by Siddon [Siddon 1985] and Joseph [Joseph 1982], and we will present them in this section.

Siddon projector

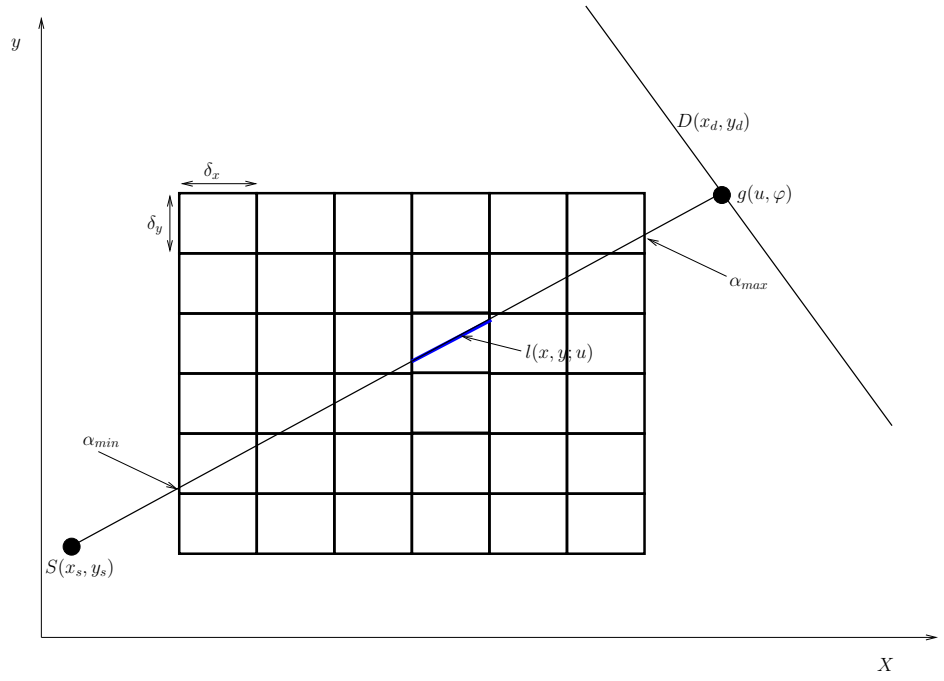


Figure 3.2 – 2D representation of Siddon ray-tracing method

Algorithm 3 Siddon projector SWI

Require: $\alpha[dim_\varphi], \beta[dim_\varphi], volume$
Ensure: $3D$ $sinogram$
1: **for all** φ, un, vn **do**
2: $Compute$ ray length L
3: $Compute$ $delta_\lambda$
4: $sino_{ray} \leftarrow 0$
5: **if** $delta_\lambda \geq 0$ **then**
6: $Compute$ λ_{min}
7: $Compute$ λ_{max}
8: $\lambda \leftarrow \lambda_{min}$
9: **while** $\lambda \leq \lambda_{max}$ **do**
10: $Compute$ coordinates of the intersection
11: $Compute$ intersection length l
12: $Compute$ Voxels coordinates (xn, yn, zn)
13: $sino_{ray} += l * volume[xn, yn, zn]$
14: $Update$ λ
15: **end while**
16: **end if**
17: $sinogram[un, vn, \varphi] = sino_{ray}$
18: **end for**

In order to compute the radiological path of a ray, the forward projection is expressed as follow:

$$g_{u,v,\varphi} = \sum_{x,y,z} l(x, y, z; u, v, \varphi) f(x, y, z) \tag{3.5}$$

where $l(x, y, z; u, v, \varphi)$ is the intersection length between the ray (u, v, φ) and the voxel (x, y, z), and $f(x, y, z)$ the attenuation coefficient at voxel (x, y, z).

Siddon pointed out most voxels of the system matrix are zero [Siddon 1985], so instead of summing over all voxels, it is much more efficient to follow the path of each ray, from the X-ray source $S(x_s, y_s, z_s)$ to the detector $D(x_d, y_d, z_d)$, through the volume and summing over only the non-zero values. Siddon's approach is burdensome on the computation of intersection coordinates for each voxel. To reduce the computational cost, more efficient methods for this algorithm have been developed by Jacob [Jacobs 1998] and Han [Han 1999]. Therefore, The adopted fast ray-tracing method is a combination of these two approaches [Jacobs 1998,Han 1999] based on Siddon's initial algorithm [Siddon 1985].

Algorithms 3 and 4 corresponds to the OpenCL SWI and NDR kernels for Siddon projector respectively. The SWI version is the fully pipelined version with sequential execution pattern. The parallelism is extrated in this programming model at the finest granularity with loop pipelining and concurrent execution. The NDR version extract thread-level parallelism to express data parallelism. As explained

Algorithm 4 Siddon projector NDR

Require: $\alpha[dim_\varphi], \beta[dim_\varphi], volume$
Ensure: $3D$ $sinogram$

```
1: ___attribute___((num_simd_work_items(M)))
2: ___attribute___((num_compute_units(N)))
3: ___attribute___((reqd_work_group_size(X,Y,Z)))
4: un = get_global_id(0)
5: vn = get_global_id(1)
6: φ = get_global_id(2)
7: Compute ray length L
8: Compute delta_λ
9: sino_ray ← 0
10: if delta_λ ≥ 0 then
11:    Compute λ_min
12:    Compute λ_max
13:    λ ← λ_min
14:    while λ ≤ λ_max do
15:       Compute coordinates of the intersection
16:       Compute intersection length l
17:       Compute Voxels coordinates (xn, yn, zn)
18:       sino_ray+ = l * volume[xn, yn, zn]
19:       Update λ
20:    end while
21: end if
22: sinogram[un, vn, φ] = sino_ray
```

in Chapter 1, the NDR execution model on FPGAs is different from the GPU one. This is because the number of compute unit on FPGAs is decided by the designers while GPUs have a fixed number of compute cores. Therefore, the designer must use OpenCL attributes to fix the number of compute units and vectorisation level in order to express data parallelism. The details of these kernels are given in Section 3.3.1.

Joseph projector

The principle of Joseph's algorithm [Joseph 1982] is to compute the line integral following the radiological ray along the axis x or y, which is the closest to the ray. The forward projector, depending on both axis, can be expressed as follows:

$$\begin{aligned} g_{u,v,\varphi} &= \frac{\delta_x}{sin(\varphi)} \sum_x f(x, y(u,\varphi;x), z(u,v,\varphi;x)) \quad \text{for} \quad \varphi < \frac{\pi}{4} \\ &= \frac{\delta_y}{cos(\varphi)} \sum_y f(x(u,\varphi;y), y, z(u,v,\varphi;y)) \quad \text{otherwise.} \end{aligned} \tag{3.6}$$

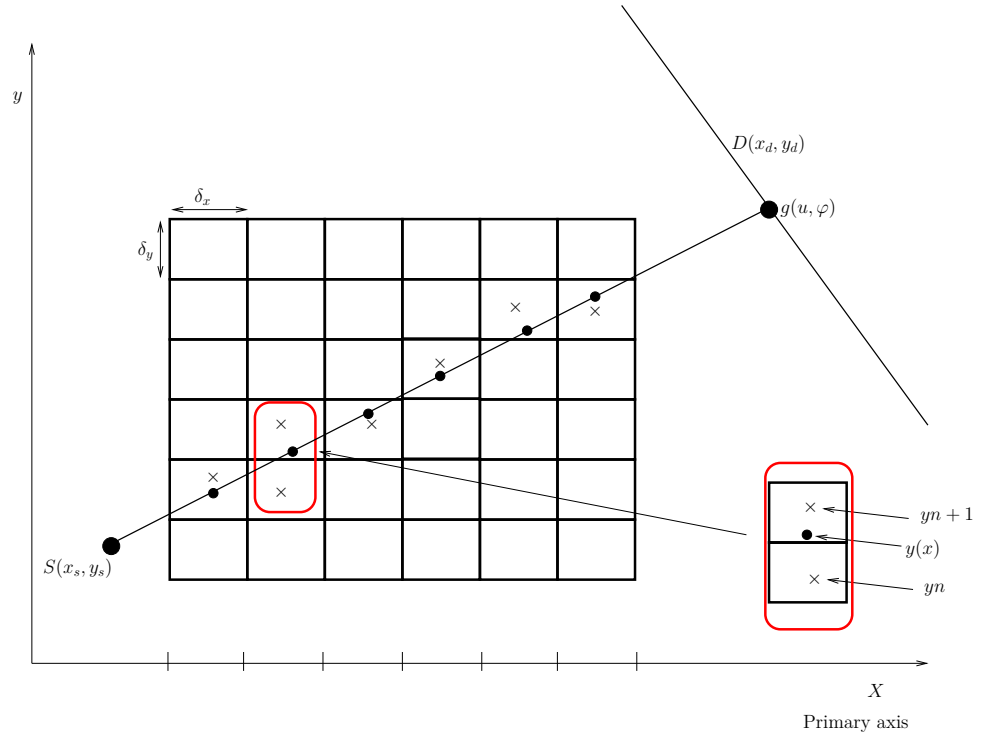


Figure 3.3 – 2D representation of Joseph ray-tracing method

The closest axis to the ray is considered as the primary (or major) axis and interpolation is used for non-primary axis (the orthogonal plan to the primary axis in 3D). The image is considered as a smooth function and bi-linear interpolation between voxels is used to approximate the image voxels.

We present the pseudo code of Joseph's projector in algorithms 5 and 6 representing the OpenCL SWI and NDR kernels respectively. The implementation details are discussed in Section 3.3.2.

3.1.2.4 3D back-projection algorithm

Backward projection is one of the most time-consuming steps in MBIR CT. The back-projection operator is also used in analytical reconstruction algorithm such as FBP [Feldkamp 1984]. The back-projection operator considered is the voxel-driven one illustrated in Fig.3.4. The 3D back-projection algorithm used in iterative reconstruction algorithm is given by:

$$f(c) = \int g(u(\varphi, c), v(\varphi, c), \varphi).w(\varphi, c)^2 d\varphi \tag{3.7}$$

where $c = (x, y, z)$ are the voxel coordinates, (u, v) are the cone beam coordinates, φ is the angular trajectory of the detector and w is the distance weight.

Algorithm 5 Joseph projector SWI

Require: $\alpha[dim_\varphi], \beta[dim_\varphi], volume$
Ensure: $3D$ $sinogram$
1: **for all** φ, un, vn **do**
2: $sino_{ray} \leftarrow 0$
3: **for all** xn **do**
4: // Constant required for yn_e and zn_e computations
5: *Compute constant values* (A, B, C, D)
6: *Compute* (yn_e, zn_e)
7: *Compute ray* length L
8: *Compute interpolation coefficients* $(C_{00}, C_{01}, C_{10}, C_{11})$
9: $sino_{ray} += C_{00} * volume[xn, yn_e, zn_e]$
10: $sino_{ray} += C_{01} * volume[xn, yn_e + 1, zn_e]$
11: $sino_{ray} += C_{10} * volume[xn, yn_e, zn_e + 1]$
12: $sino_{ray} += C_{11} * volume[xn, yn_e + 1, zn_e + 1]$
13: **end for**
14: $sinogram[un, vn, \varphi] = sino_{ray}$
15: **end for**

Algorithm 6 Joseph projector NDR

Require: $\alpha[dim_\varphi], \beta[dim_\varphi], volume$
Ensure: $3D$ $sinogram$
1: $\_\_attribute\_\_((num\_simd\_work\_items(M)))$
2: $\_\_attribute\_\_((num\_compute\_units(N)))$
3: $\_\_attribute\_\_((reqd\_work\_group\_size(X, Y, Z)))$
4: $un = get\_global\_id(0)$
5: $vn = get\_global\_id(1)$
6: $\varphi = get\_global\_id(2)$
7: $sino_{ray} \leftarrow 0$
8: **for all** xn **do**
9: // Constant required for yn_e and zn_e computations
10: *Compute constant values* (A, B, C, D)
11: *Compute* (yn_e, zn_e)
12: *Compute ray* length L
13: *Compute interpolation coefficients* $(C_{00}, C_{01}, C_{10}, C_{11})$
14: $sino_{ray} += C_{00} * volume[xn, yn_e, zn_e]$
15: $sino_{ray} += C_{01} * volume[xn, yn_e + 1, zn_e]$
16: $sino_{ray} += C_{10} * volume[xn, yn_e, zn_e + 1]$
17: $sino_{ray} += C_{11} * volume[xn, yn_e + 1, zn_e + 1]$
18: **end for**
19: $sinogram[un, vn, \varphi] = sino_{ray}$

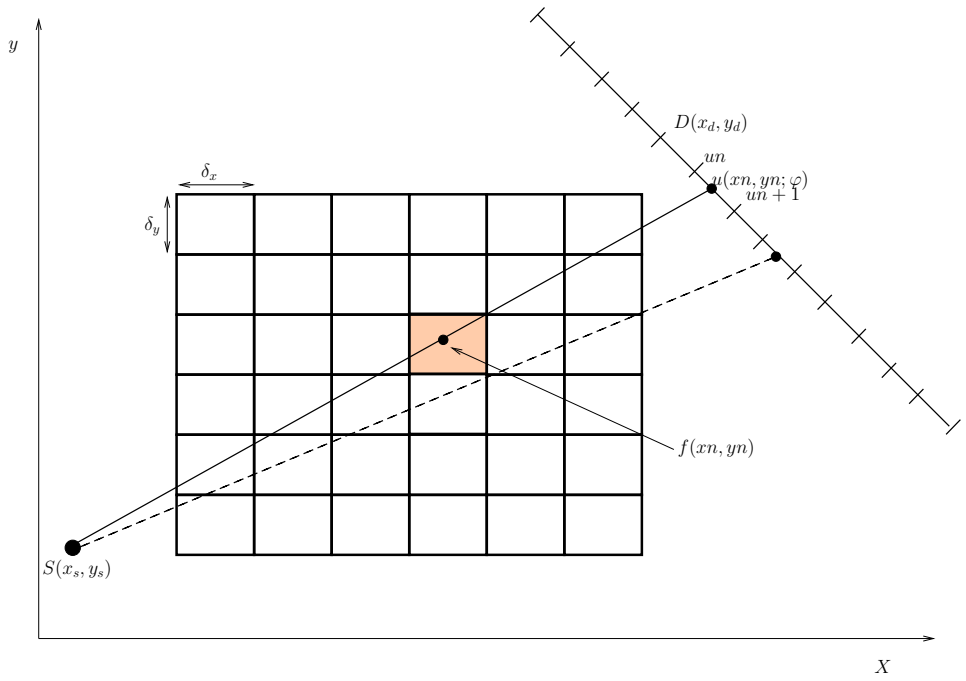


Figure 3.4 – Voxel-driven back-projector

$$u(\varphi, c) = x * cos(\varphi) + y * sin(\varphi) \tag{3.8}$$

$$v(\varphi, c) = x * sin(sin\varphi) - y * cos(\varphi) + z \tag{3.9}$$

For each voxel (x, y, z), the projection of its contribution is located at a position $(u(x, y, \varphi), v(x, y, z, \varphi))$ on the detector. The contribution on the detector is calculated using the bi-linear interpolation.

This voxel-driven back-projector is expressed in OpenCL for acceleration purpose. The SWI and NDR kernels are presented in Algorithms 7 and 8 respectively. OpenCL optimizations will be applied to these kernel in order to speed up the computation. The implementation details are discussed in Section 3.3.3.

3.2 Acceleration of CT algorithms

According to the literature, iterative reconstruction algorithms take several minutes to several hours. For instance, the computation time of the EM algorithm for 500 projections of 736 × 64 pixels of projection data in the CPU is 1.52 hours [Chen 2012a]. Hardware accelerators are required to reduce the reconstruction time of these routines. GPUs have been the preferred architecture for the past decade due to their parallel computing pattern.

Algorithm 7 Voxel-driven back-projector SWI

Require: $\alpha[dim_\varphi], \beta[dim_\varphi], sinogram$
Ensure: $3D$ $volume$
1: **for all** zn, yn, xn **do**
2: $voxel_{sum} \leftarrow 0$
3: #pragma unroll *factor*
4: **for all** φ **do**
5: $Compute(un, vn)$
6: $voxel_{sum} += sinogram[un, vn, \varphi]$
7: **end for**
8: $volume[xn, yn, zn] = voxel_{sum}$
9: **end for**

Algorithm 8 Voxel-driven back-projector NDR

Require: $\alpha[dim_\varphi], \beta[dim_\varphi], sinogram$
Ensure: $3D$ $volume$
1: $\_\_attribute\_\_((num\_simd\_work\_items(M)))$
2: $\_\_attribute\_\_((num\_compute\_units(N)))$
3: $\_\_attribute\_\_((reqd\_work\_group\_size(X, Y, Z)))$
4: $xn = get\_global\_id(0)$
5: $yn = get\_global\_id(1)$
6: $zn = get\_global\_id(2)$
7: $voxel_{sum} \leftarrow 0$
8: #pragma unroll *factor*
9: **for all** φ **do**
10: $Compute(un, vn)$
11: $voxel_{sum} += sinogram[un, vn, \varphi]$
12: **end for**
13: $volume[xn, yn, zn] = voxel_{sum}$

3.2.1 Acceleration on parallel architectures

Several works have focused on the acceleration of CT algorithms using parallel architectures. The unmatched ray-driven projector and voxel-driven back-projector have been largely explored in order to reduce the execution time. The Cell processor has been used in [Scherl 2007b] to optimize the FBP focusing on the most time-consuming filtering and back-projection steps. Also, the voxel-driven back-projection algorithm based on the Cell processor has been proposed by the authors of [Kachelriess 2006]. Before the apparition CUDA, GPU implementations of tomography algorithms were done through graphic languages [Mueller 2007]. However, the release of CUDA changed the situation by making the GPU much more attractive in the community with the proposition of some early work using CUDA language [Scherl 2007a, Knaup 2008]. Hence, in [Scherl 2007a], they proposed CUDA-

based FBP implementation on GPU where the back-projection was highly optimized due to the hardware support for bilinear interpolation on Nvidia GPU. Several other works have addressed the acceleration of the back-projector on the GPU architecture over time [Xu 2007, Rohkohl 2009, Zinsser 2013, Okitsu 2010, Jia 2014]. The Siddon projector combined with the voxel-driven back-projector were accelerated on the GPU using different iterative algorithms and offered a good trade-off between performance and image quality [Xu 2010a]. A branchless variant of the ray-driven Joseph projector is implemented on GPU taking advantage of their texture memories in [Dittmann 2016, Dittmann 2017]. Their approach was GPU-friendly and delivered good reconstruction time. Despite the high-frequency artefacts concern for matched ray-driven or voxel-driven pair of projector and back-projector, several GPU implementations of these matched pairs have been proposed in the literature. The authors of [Nguyen 2015] proposed a GPU acceleration of the ray-tracing projection and back-projection operators to increase the iterative algorithm throughput significantly. Also, the ray-tracing projector and back-projector have been widely optimized for GPU computation in order to avoid thread divergences due to multiple conditional branching [Xiao 2011, Xiao 2012, Gao 2012, Thompson 2014]. Hao Gao proposed a highly parallelizable ray-driven projector and back-projector with reduced computational complexity in [Gao 2012]. The 3D Siddon (Jacob's version) forward and back-projection has been adapted to the GPU architecture avoiding threads divergence within a warp in [Thompson 2014]. The authors exploit the cone-beam geometry and manage to have all the threads in the same execution context in a given warp.

The distance-driven and the separable footprint pairs have been evaluated on GPU architectures. The original distance-driven method presents an irregularity in the inner loop for pixel and detector boundaries computations causing conditional branching and cannot be efficiently parallelized on GPU. In [Basu 2006], the authors proposed a branchless version of the Distance-Driven (DD) which is suitable for GPU computation. This led to efficient GPU implementations of the DD pairs with significant improvement of the throughput [Schlifske 2016, Liu 2017] and also multi-GPU version [Mitra 2017]. Similarly to DD pairs, the matched Separable Footprint (SF) forward and back-projector have been accelerated on GPU. The first implementation of SF pairs have been proposed by Wu *et al.* [Wu 2011] followed by a faster version in [Xie 2017]. These implementations suffer from high data transfer amounts between the CPU and the GPU, making the PCIe bus the main bottleneck. This memory transfer bottleneck have been alleviated with newer implementations for the SF pair [Chapdelaine 2018, Georgin 2019].

3.2.2 Acceleration on specific circuits

Specific circuits such as ASICs and FPGAs are also considered in tomography reconstruction systems [Wu 1991]. These circuits allow to perform custom design based on the application and provide flexibility in the design. One ASIC-based multiprocessor implementation has been proposed in [Agi 1993] for parallel and fan beam

back-projection. However, FPGAs have more flexibility than ASICs, making them candidates for rapid prototyping and reducing time-to-market. In the past, these various parallelisms on FPGAs were extracted for tomography through the HDL requiring a strong hardware skills [Gac 2008, Pfanner 2011, Coric 2002, Kim 2012]. Gac *et al.* designed a 3D back-projector accelerator for Positron Emission Tomography (PET) based on an adaptive cache to improve the memory accesses. Their HDL design was efficient and performed a voxel update per clock cycle. This level of abstraction can be a heavy and time-consuming development based on the complexity of specific algorithms. A highly pipelined version of the SF forward projection algorithm with a two-memory level has been proposed in [Kim 2012]. The two-level memory allows to reduce the off-chip memory access and consequently increases the design throughput. Unfortunately, using FPGAs through HDLs is not viable as these languages are only accessible by traditional hardware designers. Furthermore, the development flow of these languages is highly time-consuming, as presented in Chapter 1, creating a barrier for complex applications. Hence, the emergence of tools with a high level of abstraction allows a broader audience to use FPGAs through HLL languages and make FPGAs attractive for computed tomography acceleration.

FPGAs with HLS have recently been subject of evaluation in CT reconstruction for many algorithms such as Maximum Likelihood Expectation Maximization [Cilardo 2020, Ravi 2019], 3D back-projection [Martelli 2018] or CT data alignment in memory [Wen 2020]. The industrial HLS tools that were the most mature began to be used for faster development. The authors of [Xu 2010b] provided an implementation of the 3D back-projection algorithm and compared it to the hand-written VHDL design regarding throughput and productivity. It is also worth mentioning the implementation of the matched ray-driven pair on FPGAs [Chen 2012b] using the AutoESL tool before the tool's company was acquired by Xilinx and became Vivado HLS. Forward and back-projection operators are the most expensive in tomography reconstruction. The memory access pattern of these algorithms is not regular, which makes them difficult for HLS compilers to analyze. Therefore these operators are characterized as memory-bound as memory access constitutes the main bottleneck. Due to the low bandwidth available on FPGAs, an irregular memory access pattern will strongly affect the application's performance. For this reason, special care must be paid to memory accesses to make them regular enough and thus take advantage of the FPGA's bandwidth. Choi *et al.* [Choi 2016] proposed a Ray-driven voxel-tile parallel approach that maximizes the data reuse rate to take advantage of FPGA BRAM. Their approach uses ray-tracing pair for helical CT reconstruction. With their new parallelism strategy, they avoid the bank conflict in the parallel scheme when multiple rays may try to update the same voxel [Huaxia Zhao 2003]. Zhang *et al.* [Zhang 2020] also proposed a parallel beam-based hardware implementation on FPGA to exploit on-chip BRAM intensively. The works of Choi and Zhang use Vivado HLS to synthesize the FPGA kernel. The authors of [Qiao 2021] proposed an FPGA design of iterative reconstruction for Transmission Electron Tomography (TET) by improving data locality to optimize

memory accesses.

The HLS tools have made it possible for FPGAs to be considered again as potential accelerators alongside GPUs. An algorithm-architecture co-design approach is necessary to take better advantage of FPGA architectures. This adaptation effort is even more substantial for HLS tools because HLS compilers are not as efficient as those used for general-purpose processors. In [Martelli 2019], the author explored the potential of high-level implementation on FPGAs based on OpenCL SDK applied to image processing applications such as computed tomography. The acceleration of forward and back-projection operators is also at the heart of this thesis. We have been interested in the unmatched ray-driven projector and the voxel-driven back-projector for hardware acceleration.

3.3 FPGA implementations with OpenCL HLS

This section presents the results of the first implementations of our operators to identify the main bottlenecks of FPGAs using OpenCL HLS tool. These implementations focus on the exploration of the SWI and NDR kernels by applying OpenCL relevant optimizations. The operators considered in this study are the Siddon and Joseph ray-driven projectors and the voxel-driven back-projector.

3.3.1 Siddon

The 3D Siddon projector has been described in OpenCL. Algorithm 3 corresponds to the SWI execution model and Algorithm 4 the NDR version. The critical path of the algorithm lies in the ray-tracing computation (*the while loop*).

The SWI version contains 4 nested loops following the order $\varphi \rightarrow vn \rightarrow un \rightarrow$ *while loop* as presented in the algorithm 3. These loops are all fully pipelined on FPGA in order to achieve maximum throughput. The *while loop* is the innermost loop in the design. The trip count of this inner loop[1] is not known at compile-time and differs from one iteration to the other of the outer loops. Therefore, achieving an II value of one is challenging for this loop. This is problematic because a high II value of an inner loop affects all the outer loops resulting in high latency for the design. The number of iterations of the innermost loop of this algorithm (*the while loop*) is different for different iterations of the outer loop $'un'$. This is because the number of iterations of the *while loop* depends on the length of the radiological ray. The loop exit condition is evaluated by comparing the value λ to λ_{max} which represents the end of the ray. The II value of this loop depends on the latency of updating λ which involves the computation of λ in x, y *and* z directions followed by some conditional branching to select the right direction. The offline compiler successfully pipelined all the loop in the design with an II value of one except for the *while loop*. This loop has an II value of 3 due to data dependency on λ. This is the best of what the compiler can do after all the optimizations we have applied

[1] This kind of loop is called an out-of-order loop in OpenCL terminology.

to the loop. Therefore, due to the high II value, loop unrolling cannot be applied to this loop in order to increase the design throughput because the unrolling may degrade the II by further increasing its value. The unrolling is then applied to the *un* loop for more parallelism.

The NDR implementation of this projector has also been evaluated as presented in Algorithm 4. The φ, vn *and* un loops are handled by work items in this execution model. The remaining loop is only the ray-tracing loop which is computed by each single work item. The exit condition issue present in SWI version is not present here because loops in NDR kernel are not pipelined. In order to achieve data parallelism, optimizations such as compute unit replication and SIMD vectorization are applied to this kernel.

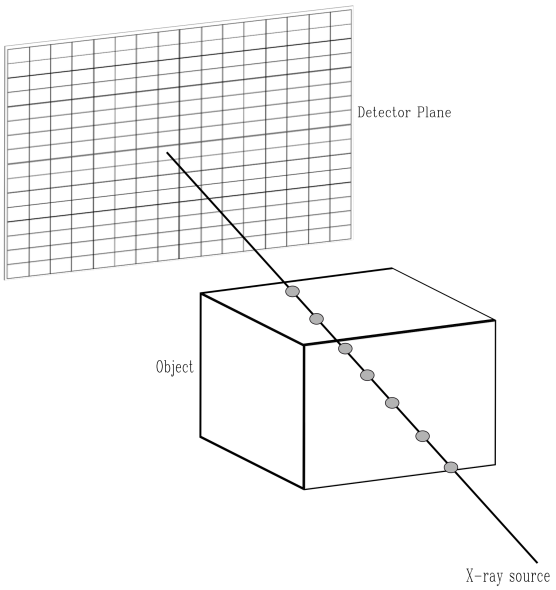


Figure 3.5 – 3D projection memory access pattern

In addition to the computation issues, the memory access pattern of this algorithm remains a critical concern. As illustrated in Fig.3.5, the ray passes through the 3D volume. The projection algorithm need to compute the sinogram pixel for each ray. For each traversed voxel, the intersection length is multiplied by the voxel intensity and the result is accumulated in the sinogram. The (xn, yn, zn) coordinates change for each voxel crossed by the ray resulting in non-consecutive memory access pattern with low data locality. The implementations take advantage of Intel automatic cache mechanism to optimize the memory access of the algorithm and reduce global access latency. However, due to irregular access pattern, memory coalescence is not possible which leads to high memory footprint for implementing the caches. Moreover, the SIMD vectorization cannot be applied to the NDR kernel due to the non-consecutive memory access pattern, and the parallelism is achieved

by compute unit replication.

The loop unrolling factor in the SWI and the compute unit replication depend on the memory usage of each design because the resources available on the FPGA must not be exceeded. Therefore, an unrolling factor of eight is applied to the un loop in the SWI kernel which represents a design with a single compute unit with eight processing elements. Each processing element is responsible of one ray-tracing at the time. The NDR kernel contains eight compute units in the design. The work group size has been specified to $(64, 1, 1)$ which means that each work group has 64 work items. The resource usage of the NDR version may be high because each compute unit performs several computations before the ray-tracing step.

3.3.2 Joseph

The Joseph projector has been described in OpenCL for x axis as the primary one. The Joseph projector performs the sampling following the primary axis (x or y depending on φ). Then, the bilinear interpolation is used for the remaining axis. The memory access pattern is irregular where the coordinates yn_e, zn_e change constantly following the radiological because they are computed on the fly. This irregularity makes the access prediction hard for the compiler as explained for the Siddon projector. Therefore, global memory access cannot be coalesced, and the compiler will infer multiple private caches for each memory access leading to the waste of valuable BRAM resources. The memory footprint is even more critical, with the bi-linear interpolation requiring more memory accesses.

In order to compute a sinogram pixel, this algorithm performs several operations at each sampling point including voxels coordinates, constant values required for the length of the ray and interpolation coefficients computations followed by the interpolation. These floating-point operations are expensive and required access to the volume data in global memory making the algorithm highly computationally-intensive. The projector is evaluated by applying the two execution models of OpenCL and their specific optimizations.

The SWI version of the Joseph projector is presented in Algorithm 5. We have four loops in this kernel in the order $\varphi \rightarrow vn \rightarrow un \rightarrow xn$ which are all fully pipelined. The sampling axis x representing the ray-tracing axis corresponds to the loop xn with a known number of iterations. Therefore, the exit condition evaluation is handled in one clock cycle. The trip count issue noticed with the kernel of Siddon projector is not problematic in the case of Joseph because the sampling axis remains constant. The II of all the loop in the design is one. In this condition the static design of the pipeline is efficient. However, the performance is highly affected by the memory access latency. The global accesses cannot be coalesced due to non-consecutive access pattern for the ray-tracing as mentioned above. Furthermore, the Joseph projector uses bilinear interpolation which has much higher memory footprint and more computationally expensive. In addition to basic optimizations applied to this kernel such as floating point optimization, caching memory accesses, etc, we applied loop unrolling to the xn loop in order to maximize the throughput.

The unrolling factor applied is eight, putting higher unrolling factor results in extra BRAM utilization.

Algorithm 6 represents the NDR version of Joseph projector. The φ, *vn and un* loops are handled by work items in this version. The xn loop which corresponds to the ray-tracing is not pipelined in the NDR kernel. The number of compute unit inferred in the design is four and SIMD vectorization cannot be applied due the memory access irregularity. The NDR version extracts the parallelism at data-level. However, due to the gap between the memory bandwidth on GPUs and FPGAs, this execution model is very limited on FPGAs in term of performance. Furthermore, the compute unit on FPGAs are instantiated by the designer depending on the application complexity and the hardware resources because they do not have several computing cores as is the case of GPUs.

3.3.3 Back-projection

The voxel-driven back-projector has been described in OpenCL from [Martelli 2019]. The SWI version is described in Algorithm 7 and the NDR version in Algorithm 8. The principle is to accumulate the elementary contribution of each detector pixel in line with the voxel under consideration for all angular projections. It is a massively parallelizable algorithm because each voxel can be considered independently of the others making it suitable for GPU architectures. The coordinates of the detector pixels are rarely aligned with the voxels, so the voxel contributions are calculated by interpolation. Bi-linear interpolation is often used to perform this approximation, but in our case, we use nearest-neighbor interpolation to reduce the computational complexity and save FPGA hardware resources.

We have evaluated this operator with the NDR and SWI execution models on FPGA. In the SWI version, there are four nested loops in the order $zn \rightarrow yn \rightarrow xn \rightarrow \varphi$. We applied several loop optimizations to this kernel, such as pipelining, unrolling, and coalescence, using Intel automatic cache to optimize global memory access. The loops in the design are all pipelined with an II value of one. Then, loop unrolling is applied to the innermost loop to achieve higher throughput. The accumulation of voxel intensity is performed by following the angular variations φ, which is the innermost loop. Loop unrolling is applied to this loop to achieve higher throughput. However, unrolling the φ loop does not mean that several voxels are computed simultaneously, but the accumulation of a single voxel is performed in parallel. Loop unrolling here does not imply that all the computations are performed simultaneously due to the necessity of reduction for this operation. The unrolling only means that all the data required for the computation is available at the same time. Therefore, the unrolling concerns mainly the memory accesses in this case. Once the data is available, a tree-based approach is used to perform the accumulation. The unrolling factor applied to this loop is 32 on Intel Arria 10 device. The NDR version uses the compute unit replication to express parallelism for the back-projector like the projectors. The zn, *yn and xn* are handled by work items in the design. This algorithm with a high parallelism potential is also affected

by the low bandwidth available on the FPGA for massive data parallelism.

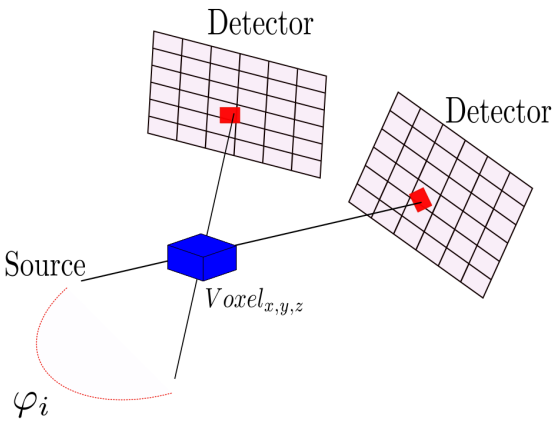


Figure 3.6 – 3D back-projector memory access pattern for reconstruction a single voxel

Similarly to the projector algorithms, the voxel-driven back-projector also has a strong memory access issue. As illustrated in Fig. 3.6, a single voxel is crossed by different rays from different projection angles corresponding to different detector planes. During back-projection of a voxel, we look for the contributions in the detector planes for each projection angle in order to reconstruct this voxel. Since the detector plane are spatially spaced in memory, the lack of data locality may strongly affect the design performance. This irregular memory access pattern is difficult to analyze and optimize by the offline compiler, even when using the Intel automatic cache. In addition, the coordinates of the projection pixels are computed on the fly, making the static prediction of the memory access further challenging.

3.4 Performance evaluation

3.4.1 Experiment setup

The experiment setup of this work is presented in this section. This setup will remain the same for the entire manuscript.

3.4.1.1 Dataset

We used in this experiment the 3D Shepp-Logan phantom in a cone-beam X-ray CT system with 256×256 detector cells. The detector pixel size is 14.2 mm in each dimension, and the acquisition produced 256 projections distributed between 0 and 2π. The size of the considered volume is a 256^3 voxels, and the voxel size is ($x = 6.05\ mm, y = 6.05\ mm, z = 5.86\ mm$). The Focus-Object Distance (FOD) is 98 mm, and the Focus-Detector Distance (FDD) is 230 mm.

3.4.1.2 Software

The Arria 10 device was part of the FLIK platform, which is a compact, all-in-one, and portable accelerator platform for laptops. The OpenCL release was the Intel FPGA SDK for OpenCL version 18.1.2.227 and the BSP 18.1.

The OpenCL release of the Stratix 10 device was the Intel FPGA SDK for OpenCL version 19.1.0.240 with the BSP 19.1. For both devices, their host machine were based on Linux OS.

Each kernel execution is monitored through the Intel FPGA dynamic Profiler for OpenCL. For each kernel, this tool provides, amongst other things, operating frequency, execution time, logic utilization, latency, bandwidth, and stall of most memory access.

3.4.1.3 Hardware devices

Table 3.1 – Platforms used in the experiment

Board	Compute resources	Freq (MHz)	External Memory	Peak float (TFLOPS)
CPU i7-3820	8 cores	3600	DDR3	0.88
GPU Jetson TX2	256 Cuda cores	1465	DDR4	0.75
GPU A100	6912 Cuda cores	1410	HBM2	19.5
FLIK Arria 10	1518 DSPs	480	DDR4	1.45
DE10-Pro Stratix 10	5760 DSPs	800	DDR4	9.2

We used the FLIK and the DE10-Pro boards for this experiment as shown in Table 3.1. The FLIK Arria 10 GX FPGA (10AX115N2F45E1SG), with 1150K logic elements, comes with 8 GB of DDR4-2133 memory, with a maximum frequency of 480 MHz. The FPGA is connected in PCIe connection (via Thunderbolt 3) to the host system. The FPGA was connected to the host via Thunderbolt 3 because the FLIK card is a compact, all-in-one, and portable accelerator platform for laptops. The DE10-Pro board is based on Intel Stratix 10 GX (1SG280HU2F50E2VG) with 32GB arranged in 2 banks DDR memory. The Stratix 10 device is the high-end FPGA with 5760 DSP slices. The performance presented in Table 3.1 corresponds to the theoretical peak performance in single-precision floating-point operations.

In this experiment we do not consider the data transfer between the host and the device; therefore the considered runtimes do not include memory transfer. However, to speed up the memory transfer, the allocated data must be aligned at least 64 bytes to allow for Direct Memory Access (DMA) transfer. To allocate an aligned memory, the posix function *posix_memalign* can be used by the host. In our experience the aligned memory achieves better transfer rate than the non-aligned memory in all cases. The AOC compiler allows designers to specify a seed value to relax the

routing constraints or solve the timing violations due to the design complexity. After validating the architecture on Stratix 10, we perform a seed sweep to choose the best design configuration that reaches the maximum frequency. The seed sweep permits determining an optimal seed value for a design without any change in the initial pipeline characteristic.

3.4.2 Results

The results of our implementations are presented in Table 3.2. For each operator, we have evaluated both NDR and SWI kernels on Intel Arria 10 device using OpenCL language. To fairly compare different implementations with different problem sizes, we use the Giga Updates Per Second (GUPS) indicator, which is unaffected by the size of the problem, using the formula given in [Chou 2011]:

$$GUPS = \frac{GU}{Time_{kernel}} \quad with \quad GU = \frac{N_{voxel} * N_{acc/voxel}}{1024^3} \tag{3.10}$$

with N_{voxel} the size of volume and $N_{acc/voxel}$ the number of accumulations per voxel.

3.4.2.1 Designs evaluation

Table 3.2 – Performances of forward and back-projection operators on Arria 10

Operator	Kernel	BRAM (%)	DSP (%)	Stall (%)	Occ (%)	Freq. (Mhz)	Time (s)	GUPS	GUPS/Watt
Siddon	SWI	60	56	87.7	29	188	10.8	0.37	0.025
	NDR	91	71	90	42	225	3.8	1.08	0.072
Joseph	SWI	29	9	86.2	12.6	216	22	0.18	0.012
	NDR	65	22	80.7	4.1	210	77.8	0.05	0.003
VD-BP	SWI	72	27	71	24.6	150	3.5	1.14	0.077
	NDR	94	31	79	18.7	168	28	0.14	0.009

In the case of Siddon, the SWI version is less optimal due mainly to the memory access pattern and the inefficient pipeline. The inefficiency of the pipeline is caused by the innermost loop with a non-fixed trip count, which penalizes the generated pipeline with a high II value. However, an NDR kernel does not suffer from this pipelining problem because the loops are not pipelined in this programming model. Instead, the pipeline is at the global function level, and the work items are executed through this pipeline one after the other. Therefore, the NDR version achieves the highest performance with eight compute units and the required work-group (64,1,1). Specifying the work-group size at compile-time helps the compiler build the hardware without extra resource utilization and may significantly improve the speedup. The NDR version achieves better performance than SWI version. Since

the sampling is done along one axis or the other depending on the projection angle for the Joseph projector, the trip count concern is avoided. However, the irregularity of the memory access persists due to the ray-tracing. For this reason, the compiler has no problem generating an efficient pipeline with all loops having an optimal II value. With such a pipeline, loop unrolling is applied to the innermost loop in order to enhance the design throughput up to the limit of the global memory bandwidth. We also evaluated the NDR version of the Joseph projector. Due to the non-contiguous memory access, we cannot apply SIMD vectorization to the NDR kernel to save hardware resources. Instead, we used compute unit replication to achieve parallelism. However, the performance of the NDR kernel is 3-fold slower than the SWI kernel because the pipelining in the latter is effective. Thus the SWI version is much more efficient and effective than the NDR version for Joseph's projector. However, these results are still far from the state-of-the-art implementations for these projectors reported in the literature.

The observation is the same for the voxel-driven back-projector, which is also strongly affected by irregular memory accesses. This voxel-driven operator has a strong potential for parallelism, especially on architectures with high computing units (e.g., GPU). As the NDR model on GPU is different from the one on FPGA, this operator does not exploit this parallelism well on FPGA using an NDR execution model because the memory bandwidth capacity is not sufficient enough to provide data to the computational units in order to reach the highest performance. Thus, the SWI kernel is more efficient than the NDR kernel for the voxel-driven back-projector. In the SWI kernel, all loops are fully pipelined with an optimal II. Then, we apply loop unrolling to achieve parallelism at the finest granularity possible.

The most important question when using FPGAs through the OpenCL tool is the choice of the programming model (NDR or SWI). Since FPGAs are more efficient in pipeline processing, the SWI model is recommended by all major vendors of this technology. This SWI execution model has many advantages in managing control and data flow execution model and is suitable for many applications. On the other hand, an NDR implementation can be convenient, especially in cases where it is impossible to have a pipeline whose inner loops cannot be efficiently pipelined, as with the Siddon projector.

3.4.2.2 Comparison

Table 3.3 shows the comparison of our work to other works on GPU and FPGA. Several works are presented in this table using GPU and FPGA architectures to accelerate CT algorithms. We notice that our implementations using OpenCL HLS are far behind other FPGA implementations regarding the design throughput when considering the same algorithm. For example, the Siddon projection accelerated in [Choi 2016] is 17.5× faster than our OpenCL design in terms of GUPS. Similarly for the voxel-driven back-projector, our implementation is outperformed by the ray-driven back-projector accelerated in [Choi 2016, Wen 2020]. Therefore, as

Table 3.3 – Performance comparison with other works

Accelerator	Kernel	Reference	Volume	Number of projections	Platform	Time (s)	GUPS
FPGA	Siddon	This work	256^3	256	Arria 10	3.8	1.08
	Joseph	This work	256^3	256	Arria 10	22	0.18
	VD-BP	This work	256^3	256	Arria 10	3.5	1.14
Other FPGAs	Siddon	[Choi 2016]	$512^2 \times 372$	831	Virtex-6	4.0	18.9
	VD-BP	[Gac 2008]	$128^2 \times 63$	96	Virtex-4	0.526	0.18
	RD-BP	[Choi 2016]	$512^2 \times 372$	831	Virtex-6	3.7	20.4
	RD-BP	[Wen 2020]	$1024^2 \times 128$	502	ZCU102	2.10	29.9
GPU	Siddon	[Thompson 2014]	512^3	720	Quadro 6000	5.9	15.25
	BP	[Chou 2011]	512^3	360	Tesla C1060	2.47	18.2
	BP	Ours	256^3	256	V100	0.011	364

mentioned above, the results of our early work are not comparable to the current state-of-the-art works on CT reconstruction using FPGAs. All operators suffer from their memory access pattern preventing the designs to achieve the best possible performance. By using the automatic caching mechanism offered by Intel OpenCL, we are able to improve performance slightly at the cost of using memory resources. Despite using these caches, memory remains the main bottleneck of these algorithms resulting in high stall percentages accompanied by an under-utilization of computational resources (e.g., DSPs).

We have also compared our results with those of GPUs and the performance gap is even greater than that observed on other FPGAs. All our operators are outperformed by the equivalent GPU implementation. The Siddon projector accelerated on GPU in [Thompson 2014] is 14.1× faster than our Siddon design on FPGA. The work of [Chou 2011] on GPU for the voxel-driven back-projector outperforms our implementation on FPGA by a factor of 16. It should be noted that these GPU are old generation devices and are not the latest powerful GPU. By comparing with more recent GPU device like Nvidia V100, the GPU outperforms the FPGA design by two order of magnitude. This performance gap is very large between FPGAs and GPUs especially since this is not what was noticed about 15 years ago. Indeed, a performance gap of one order of magnitude was noticed by [Gac 2008] when comparing GPU with FPGAs using HDLs. The major concern is now to enhance the performance of our OpenCL designs to fill this performance gap. We should at least reach the same efficiency as for HDL designs on FPGAs to reduce the gap as it was 15 years ago. Also, our OpenCL designs should be competitive with the state-of-the-art works on FPGAs as well. In order to achieve these goals, an algorithm-architecture co-design approach must be developed to better take advantage of FPGAs through the OpenCL tool.This approach must consider the specificities of FPGAs to apply advanced optimizations and an in-depth analysis of the algorithm, particularly on memory accesses. The projection and back-projection operators used in the IR methods have a strong potential for

data reuse that will have to be vigorously exploited to reduce their reconstruction time. In the rest of this manuscript, we will present how to take advantage of this data locality to implement a custom architecture for tomography algorithms using HLS tools. We will also use profiling tools to better depict our algorithm better, and the roofline model will also be used in this approach's analysis and optimization steps.

3.5 Conclusion

This chapter presents the tomographic reconstruction problem and its acceleration on hardware architectures. We also offer the first results of our attempts to accelerate forward and back-projection operators on FPGAs using the HLL OpenCL language. The results showed that directly implementing these algorithms on FPGAs does not guarantee a significant speedup even by applying OpenCL optimizations. The operators used in iterative image reconstruction methods are memory-bound. Since FPGAs' memory bandwidth is very low, there is a need to pay particular attention to optimizing memory accesses to take advantage of FPGAs. These algorithms' extensive memory footprint prevents using as many DSPs as possible to express more parallelism, as shown by the resource consumption. In order to improve the performance, We need to avoid using automatic caches and use the on-chip memory of the FPGA, which may reduce the memory access latency. To this end, we will develop a methodology for accelerating compute-intensive applications on FPGAs using HLS tools. This methodology aims to overcome the abovementioned concerns, such as memory bottleneck, high stall percentage, and inefficient use of hardware resources.

Chapter 4

Reaching FPGA Computational Peak Performance

Contents

The previous chapter highlighted the need for an FPGA acceleration methodology to take advantage of this architecture through HLS tools. This chapter proposes an architectural study based on an algorithm-architecture co-design approach in order to use FPGAs to accelerate compute-intensive applications. The objective is to propose a custom architecture for a given algorithm on FPGA. Such architecture

will result from a methodology to better characterize the algorithm by adapting it to the architecture as well as possible. This methodology will guide the design of the application on the hardware using standard profiling tools. In this chapter, we present the Berkeley roofline model used in this study as a profiling tool. Then the adopted methodology for custom architecture on FPGA is detailed with all the steps. After presenting advanced OpenCL optimizations to fully harness the FPGA device, the methodology is applied to a simple use-case of 2D convolution in order to illustrate the prefetching impact on the design performance.

4.1 Roofline model

The roofline model has been used in this work as a profiling tools to guide the design steps. The section presents the principle of the roofline model for general-purpose multicore CPU, manycore GPU and FPGA architectures.

4.1.1 Basic roofline model

The roofline model [Williams 2009] is a tool for visually and quickly observing the possible limitations of an algorithm relative to theoretical maximum performance on a target architecture. Thus the purpose of the model is to visually represent an application on a graph and identify potential bottlenecks. The model is characterized by two key parameters which defines two roofs: the device peak Computational Performance (CP) and the attainable bandwidth (BW). The CP is expressed in FLOPs and represents the maximum floating-point operations performed per second by the underlying device.

The basic roofline model is presented in Fig. 4.1 representing the attainable performance (in FLOP/s) as function of the Computational Intensity (CI) (in FLOP/Byte). The attainable performance for an application is expressed as follows:

$$\text{Attainable Performance } (FLOP/s) = \min(CP, CI \times BW). \tag{4.1}$$

The CI indicates the algorithm's complexity and represents the number of operations divided by the number of memory accesses.

$$CI = \frac{\#\text{Operations}}{\#\text{Memory access}} \tag{4.2}$$

The CI entitles to position a given algorithm either in the *memory-bound* or in the *compute-bound* area. This will identify the possible bottlenecks of the application and highlight the different ways to improve performance.

We illustrate two algorithms representing two cases in Fig. 4.1. The first algorithm with a low CI is located in the *memory-bound* area, i.e., the main bottleneck for this application is memory. Therefore, for this application, an increase in the CI would allow to bypass this memory lock and improve the performance. The second algorithm is in the *compute-bound* area, and the main bottleneck is the computational capacity. In this case, to optimize, it will be necessary to express different

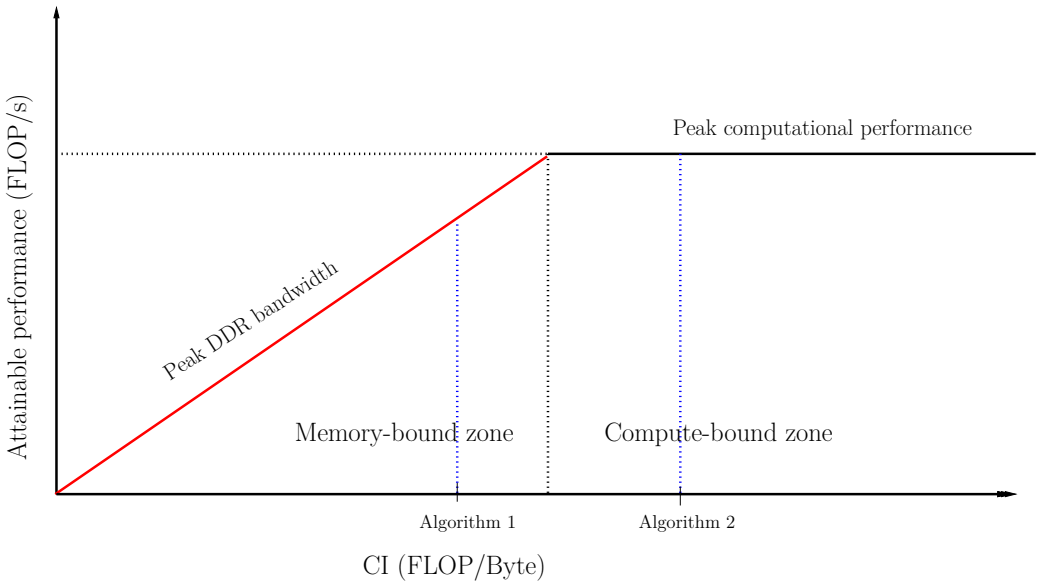


Figure 4.1 – Basic roofline representation with two applications: memory-bound and compute-bound examples

levels of parallelism both at the software and hardware level through instruction, data, or task parallelisms, etc. An important thing to note about roofline is that an application can be in the *compute-bound* area and still suffer from memory accesses, especially latencies. Indeed, one of the limitations of the roofline model is that it does not take into account memory latencies. However, applications that suffer from memory latency can still take advantage of the different levels of cache memory on CPUs.

The original roofline model was developed for multicore processors. These processors have a fixed architecture, making it trivial to determine their effective bandwidth and peak performance. Like multicore processors, GPUs with a fixed architecture have benefited from an extension of the roofline model to better exploit their full potential. Other works have focused on extending the roofline model to reconfigurable architectures like FPGAs. In the following section we present the roofline for FPGAs and the benefit of using them to optimize FPGA-based designs.

4.1.2 Roofline model for FPGAs

The work of Williams *et al.* [Williams 2009], focused on CPU multicore architectures, was extended to the FPGA architectures in [da Silva 2013] through HLS tools and taking into account the resource utilization of the device. For FPGAs, the model is both architecture- and application-dependent, so they introduced the scalability parameter determined by the available resources on the target FPGA for pipeline replication. The scalability gives us an indication of the replication potential of the elementary pipeline. Therefore, the computational roof is influenced by

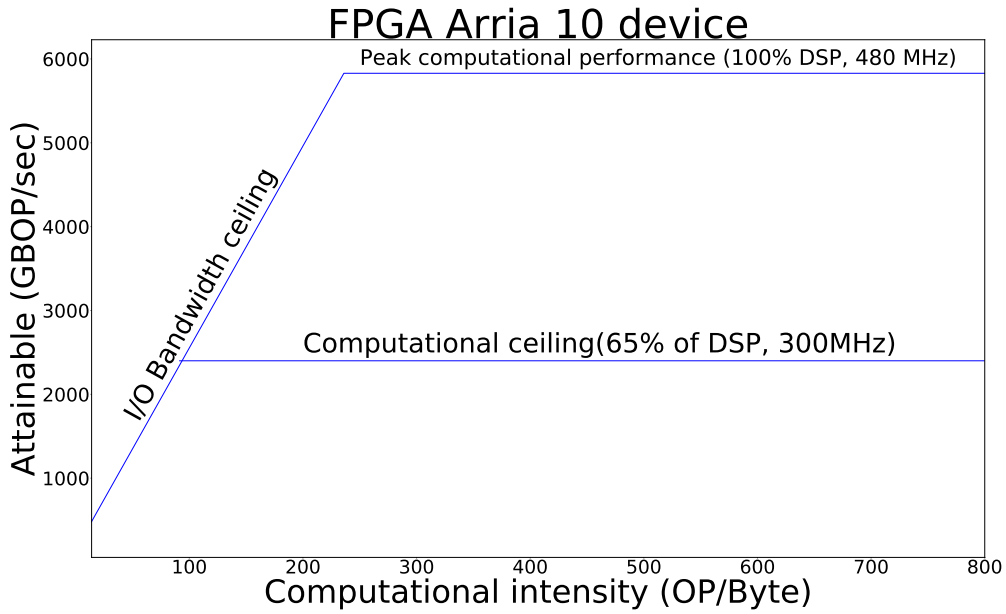


Figure 4.2 – Roofline representation on Intel Arria 10. The higher roof represents the FPGA capacity and the second roof is the ceiling for a design using 65% of DSP running at 300 MHz

the scalability factor and is given by the available resources on the target FPGA. The paradigm of the FPGA roofline is more complex regarding the bandwidth and the computational ceiling. The theoretical limits provided by vendors are far from those obtained after synthesis. We need to use the profiling tools available to collect data, which allows us to have more design information such as effective bandwidth, operating frequency, and runtime.

An example of roofline for a design occupying 65% of available DSPs running at 300 MHz on an Arria 10 device is presented in Fig. 4.2. The higher roof corresponds to the theoretical computational ceiling when the design uses the full DSP capability running at the maximum frequency. The second roof is the computational ceiling of the design with respect to the actual DSP usage and frequency. This second roof is the maximum attainable performance for the given pipeline. The applications will be plotted on that roofline regarding their CI. Depending on the CI, an application could be bounded by the memory ceiling or the computational ceiling. The goal is to have an application in the compute-bound area and get close to the roof.

In the original roofline model, among the key parameters, the CP and the bandwidth depend on the target architecture, and the CI depends on the algorithm. These parameters are all influenced by the architecture and the application for FPGAs. For example as illustrated in Fig. 4.2 the computational ceiling varies according to the amount of DSP consumed by the design. For this reason, the extended model for FPGAs has to take into consideration the specificity of their architectures to better characterize the application. For instance, one should not only consider

floating-point operations in the FPGA roofline because there are other alternatives for data representation on FPGA such as fixed-point representation. Determining peak performance on FPGAs can be complex because many types of resources such as LUTs, FFs, or DSPs can be used to perform arithmetic operations.

4.2 Adopted Methodology

We take advantage of a specific methodology for our algorithm-architecture co-design purpose. The methodology described in Fig. 4.3 has been adopted to take advantage of FPGAs in the acceleration of compute-intensive applications. This methodology consists of three main steps, from a thorough analysis of the application to its implementation on the target device. We first analyze the algorithm to have a clear vision of its memory access pattern, followed by an iterative evaluation of the roofline model. The methodology's last step concerns the design's scaling up, especially for high-end devices. In this section, we detail the main steps of this methodology in order to obtain a custom architecture for a specific algorithm.

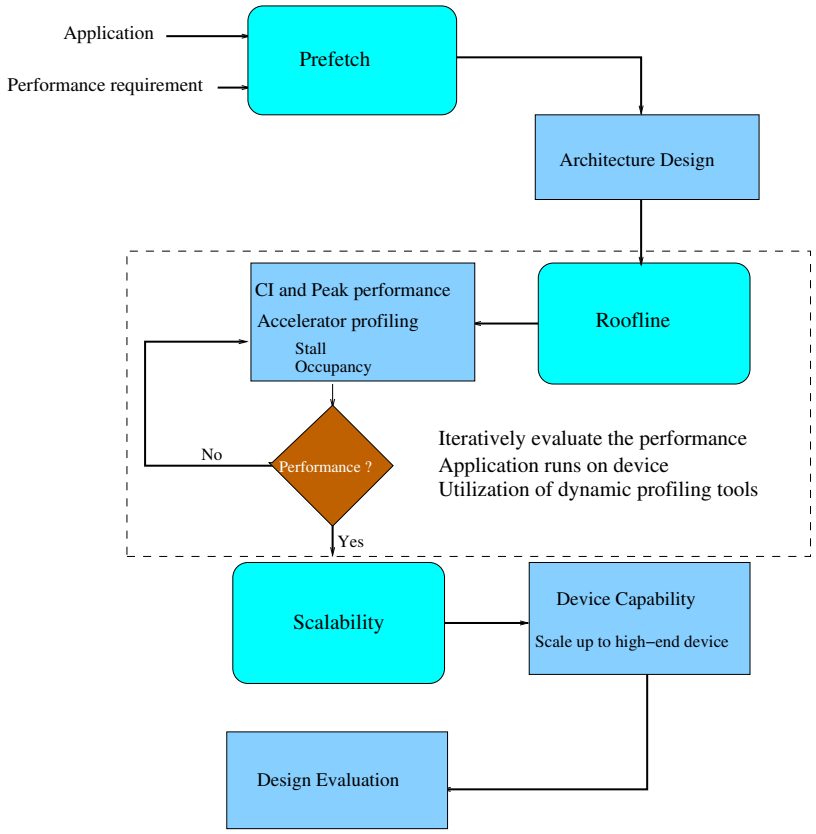


Figure 4.3 – Adopted methodology of acceleration on FPGA devices using HLS tools

4.2.1 Prefetch

Our methodology's first step consists of analyzing the algorithm with particular attention to memory accesses. Indeed, as indicated by the roofline model, HPC applications have two main bottlenecks: they are either limited by memory accesses or limited by computation. Therefore, we first focus on memory accesses in order to develop a memory access strategy. The memory access strategy makes accesses regular enough to allow the compiler to take advantage of the global memory bandwidth available on the target board. The regularity of memory accesses makes it possible to coalesce in burst mode and load data from external memory. Next, an offline analysis is performed based on the algorithm under consideration to maximize the data reuse rate by reducing the invocation of external memory as much as possible. This is done by limiting memory accesses to other types of memory with lower latency (e.g., local memory) or by taking better advantage of the caching mechanisms on the processors used. For FPGA devices, the purpose will consist of extensively harnessing the on-chip BRAM, and the offline access analysis enables to manage the resources accordingly.

From this study, one can start elaborating the algorithm's pipeline architecture on FPGA. Once again, the main idea is to optimize memory accesses to reduce global memory stalls. One way to do this is to separate the global memory accesses from the computations and use the on-chip BRAMs between the computational units and the global memory. This makes the data closer to the calculations with low and optimized access latencies. Nevertheless, inefficient use of local memory may result in worse performance than without using it. It is essential to ensure stall-free access to FPGA local memory in order to avoid arbitration. Otherwise, the compiler may have to arbitrate accesses, increasing the II of the pipeline. It is the responsibility of the designer to better elaborate the architecture in order to avoid all these concerns. In our methodology, a thorough knowledge of the internal architecture of FPGAs and the HLS development tools are required to adapt the algorithm better. Thus, applying advanced optimizations for pipeline and data parallelism, while exploiting the temporal and spatial localities made possible by the offline study should ensure performance enhancement of the FPGA design.

The pipeline architecture is thus developed in this step. This pipeline is therefore made up of one or several Processing Elements (PEs) that are in charge of performing specific operations. The function of the PE can vary from elementary arithmetic operations to more complex functions depending on the algorithm considered. The designer should efficiently design the PE within the compute unit to ensure the initiation interval and the memory access policy. The number of PEs in a compute unit is discussed in section 4.2.3.

4.2.2 Roofline

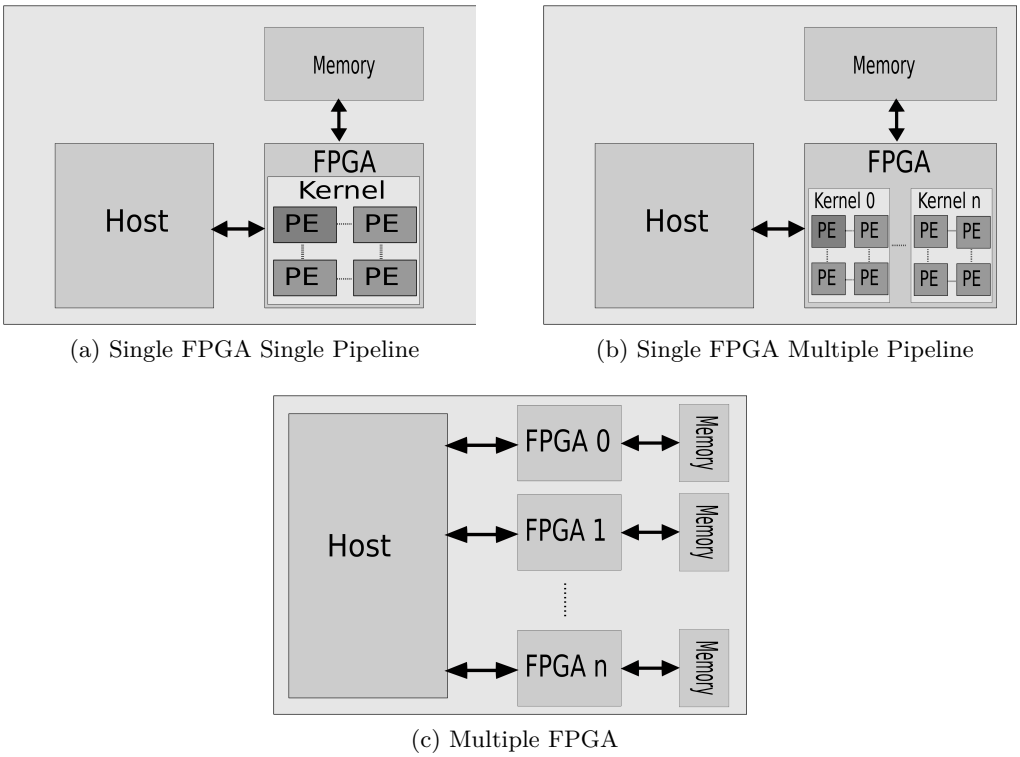
The roofline model is essential in this approach. The model guides the optimization steps to ensure good performance and better utilization of the underlying architec-

ture. At the end of the first step, we can determine the different parameters of the roofline model, namely the computational intensity, peak performance, and memory bandwidth. The determination of the computational intensity is not a trivial task for complex algorithm. Several profiling tools exist to determine its value depending on the target architecture such as CPU and GPU. For FPGAs, some HLS tools offer these information as well, while other HLS tools are not mature enough to provide basic information about the algorithm or even take into account multiple compiler-assisted optimizations. Nevertheless, the offline analysis performed in the first step of our methodology help to analyze the algorithm and compute the computational intensity accordingly. This information is then used to represent the algorithm on the roofline model. In the case of a memory-bound algorithm, increasing the data reuse rate of the algorithm increases the arithmetic intensity. Therefore the same algorithm must move from memory-bound zone to compute-bound area. Once in the compute-bound zone, the challenge is to exploit the computational units extensively to get close to the computational roof. This is done by expressing data, task, or pipeline parallelism. Hence, we analyze the performance iteratively using the roofline model to notice the impact of the different optimizations on the occupancy of our computational units and the stall percentage of memory accesses.

4.2.3 Scalability

Once the pipeline is efficient and meets all the performance and energy consumption constraints, we need to evaluate how scalable the pipeline is on the FPGA board used and deal with scaling on high-end boards. This step is necessary, especially on FPGA architectures because the hardware resources must be handled with caution. Furthermore, with various modes of expression of parallelism on FPGAs (i.e., coarse-grained and fine-grained parallelism), the use of one or the other may give different results concerning circuit synthesis, operating frequency, and computational throughput. Therefore, the designer must consider the capacity of the target FPGA as well as the synthesis tools used in order to express more parallelism with a deep pipeline without sacrificing the operating frequency of the pipeline. It is difficult for the HLS compiler to generate a deep pipeline with massive parallelism at the finest possible granularity and at the same time ensure adequate frequency. In order to limit routing congestion and have an efficient design, it is essential to make a trade-off in the choice of parallelism, especially with the use of HLS tools.

FPGA generations do not have the same computational resources and memory bandwidth. Therefore, the scaling up to higher-end devices must be under considerations by taking into account the device related characteristics for better deployment. Indeed, to scale a given design onto the FPGA die, one can proceed by several approach depending on the compute kernel and also the parallelism potential. An optimized pipeline designed for a low-end device can be scaled up to a higher-end FPGA using various strategies as illustrated in Fig. 4.4. An OpenCL kernel represents a compute unit that contains one or more PEs to perform a specific task. The PEs are fully pipelined on FPGA, and the replication allows for



(a) Single FPGA Single Pipeline

(b) Single FPGA Multiple Pipeline

(c) Multiple FPGA

Figure 4.4 – Design scalability on FPGA

higher parallelism. The replication factor depends on the hardware resources available on the FPGA and the global memory bandwidth. The goal of deep pipelining of the compute unit and parallelism with multiple PEs is to have an optimized, high-performance design on FPGA boards. However, high-end FPGA boards offering more computing power are continuously made available by the main vendors. Porting an optimized design from a low-end FPGA to a higher-end FPGA by gaining more performance due to the additional resources of the higher-end FPGA is a significant concern.

In order to overcome this issue, we discuss the scalability approach of an optimized design in our methodology. The low-end FPGA's optimized design can be deepened to express more parallelism as a single pipeline, as shown in Fig. 4.4a. The extra resources made available by the high-end device are used in this case to scale more PEs in the pipeline. Another way to express parallelism is to keep the initial optimized pipeline with its configuration and duplicate this kernel to have multiple instances running in parallel, as presented in Fig. 4.4b. This multikernel solution uses more logic resources than the first because all kernel objects are replicated instead of the PEs. Fig. 4.4c illustrates the multi-FPGA configuration for more computation power. A cluster with several FPGAs is used to perform a compute-intensive task. In this scenario, the communication topology of the FPGA cluster must be chosen with caution so that the communication between the FPGA

devices and the host devices is performed accordingly to achieve the best performance, depending on the underlying application.

4.3 Advanced optimizations

Our adopted methodology requires getting the best out of both the algorithm and the architecture to ensure efficiency. Therefore, the exploitation of advanced optimizations on FPGAs is essential to ensure performance and overcome the various bottlenecks on the FPGA architecture. It is even more critical for HLS compilers to benefit from these optimizations. These compilers are not as mature in static algorithm analysis as the standard compilers used for general-purpose processors. In this section, we will present advanced optimizations for FPGAs to guaranty an efficient pipeline, reduce the stall percentage and make the best use of the available computation units.

4.3.1 Reduce access to local memory

Table 4.1 – Memory access latencies on Intel Arria 10

Memory type	Memory implementation	Latency (clock cycles)	Throughput (GB/s)	Capacity (MB)
Global	DDR	240	34	8000
Local	BRAM	4/1	8000	66
Private	BRAM	4/1	8000	66
Private	Registers	1	240	0.2

We presented in Chapter 1 the four memory types available in OpenCL. Table 4.1 general information about OpenCL main memory types. A good application should make good use of these types of memory. The performance of a design is strongly impacted by its memory access policy. Global memory is the one that offers the most storage space but is the one with the longest access latency. It is essential to reduce access to this memory space as much as possible. There are several cache mechanisms on general-purpose processors to exploit data reuse. However, on FPGAs, it is up to the programmers to utilize dedicated on-chip memory blocks to store data on the chip. Local memory is the ideal candidate for this as it has low latency and very high bandwidth compared to global memory. The FPGA local memory is made of several M20K block to store data and each block memory contains two ports of access as shown in Fig. 4.5.

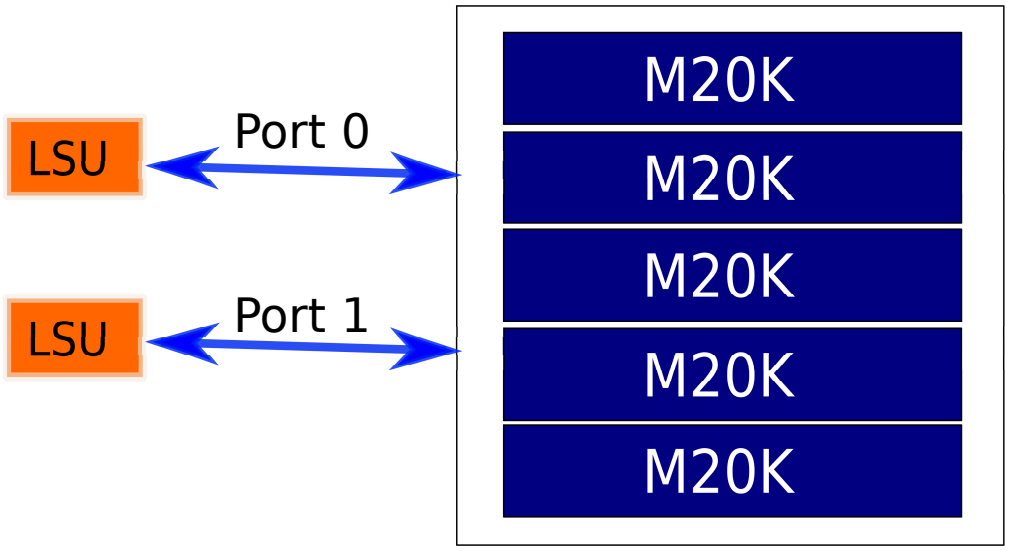


Figure 4.5 – The internal structure of Intel FPGA local memory

In order to lighten the global memory bandwidth, we are going to take advantage of the on-chip BRAMs. FPGA does not support cache hierarchy, as we can observe on general-purpose or specific-purpose processors such as CPU or GPU. Nevertheless, one can implement dedicated on-chip memory blocks based on the application memory access pattern. The accesses to local memory are mapped to a physical port. Each local memory has only two read/write ports. It is possible to use double pumping to have twice as many ports. The double pumping allows the compiler to run the memory clock twice the maximum frequency of the kernel. Thus, each local memory object can have at most four ports for simultaneous accesses. Intel recommends having four or fewer read/write accesses to local memory for stall-free access without arbitration [Intel Corporation 2019a].

In order to perform more than four concurrent accesses, the offline compiler performs memory banking or replication to avoid access conflict. The memory replication consists of replicating the memory object as many as possible to have enough access port. Each replicate memory contains exactly the same data.

Once the local memory is well configured with the right number of ports with no arbitration, one could create private copies for this local memory object. The goal of these private copies is to allow concurrent execution across different iterations of a given loop. Unlike replication, each memory copy contains different data for different loop iterations. However, the increase of number of copies will consume more BRAM resources.

4.3.2 Shift register

The shift register is one of the most essential optimization techniques used for FPGA. Indeed, FPGAs have several registers available for temporal data storage.

The advantage of register storage over BRAM is that their access latency is one clock cycle. However, storing high amount of data in register will lead to high FPGA resources utilization while BRAM resources are suitable for this kind of storage. Many HPC applications are suitable for shift register optimization such as image convolution, stencil computation, and the optimization can also be used to handle floating point computation. For instance, due to the latency of floating-point multiplication or reduction, loop-carried dependency may occur and lead to high II value (II will be the operation latency). Shift register can be used to store intermediate results of computation relaxing this dependency and therefore reduce the loop II to the ideal value. Nevertheless, the shift register optimization can only be applied to the SWI kernels on FPGAs.

4.3.3 Conditional branches balancing

FPGAs are known to efficiently handle conditional branching because they excel in concurrent execution. However, one can balance the conditional branches in the design to enhance the overall performance. To do this, care must be taken not to have global memory accesses or loops inside conditional branches. It makes more sense to put conditional branches inside loops and not the other way around to better manage the design initiation interval. As for global memory accesses, putting them outside the branching scope would allow the compiler to analyze them efficiently and to have a valid access address no matter the outcome of the conditional branch. Nevertheless, local memory accesses are not penalized by conditional branches as long as the accesses are stall-free and, therefore, will have no impact on the II.

4.3.4 Initiation Interval and operating frequency

One of the most significant limitations of OpenCL on FPGAs is the absence of multiple clock levels, as is the case with hardware description languages. Therefore, the entire OpenCL design is subject to a single global clock. However, there is a substantial trade-off between the operating frequency and the value of II for SWI kernels. The OpenCL compiler's highest priority is to ensure an II value of 1 for each loop without worrying about the frequency of the design. Depending on the target FPGA device and the BSP, the OpenCL compiler has a default f_{max} that one can change by using f_{max} attribute or the compiler option to manually fixed the target frequency. Yet, increasing this frequency beyond the default value will increase the initiation interval, which may degrade performance. Similarly, achieving an II of 1 for some of the non-critical loops in the design could reduce the overall frequency, which would also degrade performance. We can tune with the different OpenCL pragmas and attributes to handle these different parameters manually. We can increase the II of some loops that are not in the critical path of the design by the ii pragma for higher f_{max} or decrease the target frequency to have an optimal II for the critical loops of the design. However, it is not ingenious to specify a frequency above the default value or decrease the II value manually through the pragma at

the risk of routing troubles and synthesis failure.

4.3.5 Arithmetic operations overhead

The reconfigurability of FPGAs implies good management of their hardware resources. The arithmetic operations require thorough analysis and consideration to save logic [Uguen 2019]. Most HPC applications require intensive floating-point computations, hence reducing the number of complex and heavy operations (such as division, square root, and trigonometric calculations) is crucial. For instance, on the Arria 10 device, while a floating-point multiply-accumulation costs one DSP, a floating-point divide must cost 3.5 DSPs. So for a given algorithm, we should try to have fewer division as much as possible. For example, performing three divisions by the same value will cost 10.5 DSP slices. However, it will be more advantageous to compute the inverse of that value and then achieve three multiplications instead (3.5 + 3 DSPs). One division and three multiplication are better than three divisions in terms of both resource usage and latency. This particular attention is required for all the optimization steps for an FPGA design. Another important practice is to avoid *modulo* operation because the *modulo* computation requires a division and, therefore, is resource-consuming. One could replace the *modulo* with a bit-wise *"and"* operation, which is much more efficient on FPGA regarding resources and latency. Indeed, a *modulo* operation $N\%M$ can be replaced by $N\&(M-1)$ if M is a power of two.

4.4 Prefetch benefits illustrated for 2D Convolution

The methodology presented in Section 4.2 is applied to real-world applications. We applied this methodology to a simple use-case in order to illustrate the prefetching impact on the design performance. Indeed, the first step of the adopted methodology consists of an memory access analysis and prefetching to generate a pipeline that takes advantage of FPGA local memory. The considered use-case is the 2D convolution operator.

Convolution is one of the most used operators in signal and image processing applications such as edge detection, image blurring, noise reduction [Burger 2016]. The operator is very compute-intensive, mainly used in iterative algorithms for inverse problems such as deconvolution [Idier 2013]. It is essential to have significant computing power and a consequent bandwidth to meet the high computational demand. This section introduces an FPGA-based implementation of the 2D convolution operator. This operator is optimized for small convolution kernel sizes.

4.4.1 2D Convolution

The 2D convolution represents a multiply-accumulation operation between a mask (convolution kernel) and the pixels of a given image as presented in Fig. 4.6.

Let f be a 2D image of size (H, W) and h a mask of size (K, K), the convolution of f by h noted F is given by:

$$F(x,y) = \sum_{j=0}^{K-1} \sum_{i=0}^{K-1} f(x-(i-\lfloor\frac{K}{2}\rfloor), y-(j-\lfloor\frac{K}{2}\rfloor))h(i,j) \tag{4.3}$$

For $K = 3$, this equation can be written as:

$$\begin{aligned} F(x,y) = {} & f(x+1,y+1)h(0,0) + f(x,y+1)h(1,0) + f(x-1,y+1)h(2,0) \\ & + f(x+1,y)h(0,1) + f(x,y)h(1,1) + f(x-1,y)h(2,1) \\ & + f(x+1,y-1)h(0,2) + f(x,y-1)h(0,2) + f(x-1,y-1)h(2,2). \end{aligned} \tag{4.4}$$

To deal with the problem of image edge, we use the zero-padding method, which allows us to maintain the size of the input image. The pixel padding technique is often the acceptable solution to handle image edges, and zero-padding is the simplest to implement among the different types of padding used [Ström 2016].

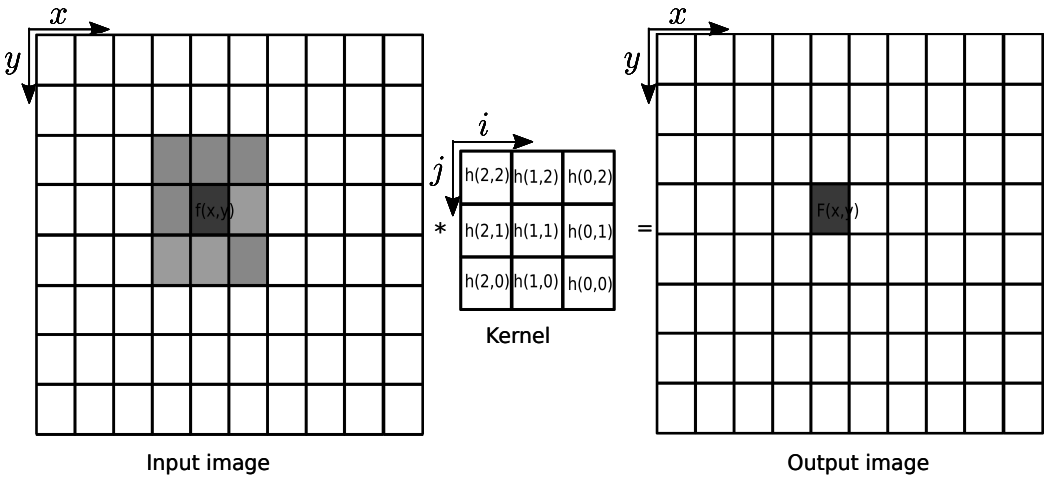


Figure 4.6 – 2D Convolution with a 3×3 mask. The '∗' is the convolution operator.

4.4.2 OpenCL-based convolution on FPGA

The convolution operator (4.3) has been described in a pipeline (SWI). The parallelism is expressed at each elementary output pixel computation level, where the multiplications between image pixels and the mask are performed in parallel, followed by the reduction. We can then express more parallelism by computing several output image pixels in parallel. This parallelism is implemented through loop unrolling, which replicates a loop's body for concurrent execution. Several output pixels are computed in parallel to increase the pipeline throughput if the number of DSPs allows it. The only drawback, in this case, is the FPGA global memory bandwidth, which will lead to high pipeline stalls during the execution due to the high number of access. The unrolling is applied to the loops over the pixels of the

image in order to calculate several output pixels simultaneously. In order to reduce the pressure on the memory bus with the parallelism degree, we use the on-chip local memory of the FPGA, whose access latency is very low compared to the global memory, to store the input data. The input data will be held in a local buffer using FPGA BRAM resources. Therefore, we reduce the number of external memory accesses because we load, at once, in local memory the required data to compute a line of pixels of the output image. Buffer data is replaced with new data after calculating each line of pixels for the output image. The loading of all these data in global memory will then be done in burst aligned mode because the required data are contiguous in memory. The global memory access rate can be further reduced by keeping the prefetched data into BRAM as long as possible. To do so, we only add a new line of data at the time and not replace all the data in the buffer. The incoming data should be stored at the bottom line of the on-chip buffer, which will require shifting the buffer's content. The buffer size is large, and shifting its contents is expensive and may slow the computations. In order to avoid this costly shift cost, we insert the new data at the least recently used line of the buffer and apply a circular shift register to the convolution mask buffer as illustrated in Fig. 4.7.

Traditionally, the shift register method is used on FPGA to perform the convolution using Hardware Description Languages (HDL). In that case, the shift is applied to the local buffer for the input data to add new incoming data at the bottom of the buffer. Therefore, the buffer size cannot be too large to avoid expensive shift costs. In our implementation, the shift is only applied to the mask buffer, which is implemented into FPGA registers. The data from the input image is inserted line by line into the local buffer and at a line that will not be used for the next computations. Moreover, our implementation is performed using high-level languages, leading to a faster development time than HDL.

The circular shift intends to switch the lines of the convolution mask so that the image and the mask coefficients may correspond. As shown in Fig. 4.7, at a given iteration n, the circular buffer contains all the input data to compute a line of the output image without access to global memory. However, for the next iteration $n+1$, the line at the top of the buffer is no longer required to convolve the following line of the output image. Therefore this line is replaced by new data from the input image, and then the bottom line of the mask buffer becomes the top to match coefficients. Then, for iterations $n+2$, the new input data is inserted at the second line of the local buffer and at the same time the mask buffer is shifted. The convolution is performed this way until the output image is completely calculated. The mask buffer is small compared to the prefetched data buffer and is implemented using registers instead of BRAM resources. Therefore, this circular shift register is less expensive and energy-efficient for the overall design. In order to make the better use of the FPGA device, we then replicate the compute unit to improve the overall throughput. By applying these different optimizations in the OpenCL kernel, we are able to evaluate the efficiency of our convolution design on Intel Stratix 10 device.

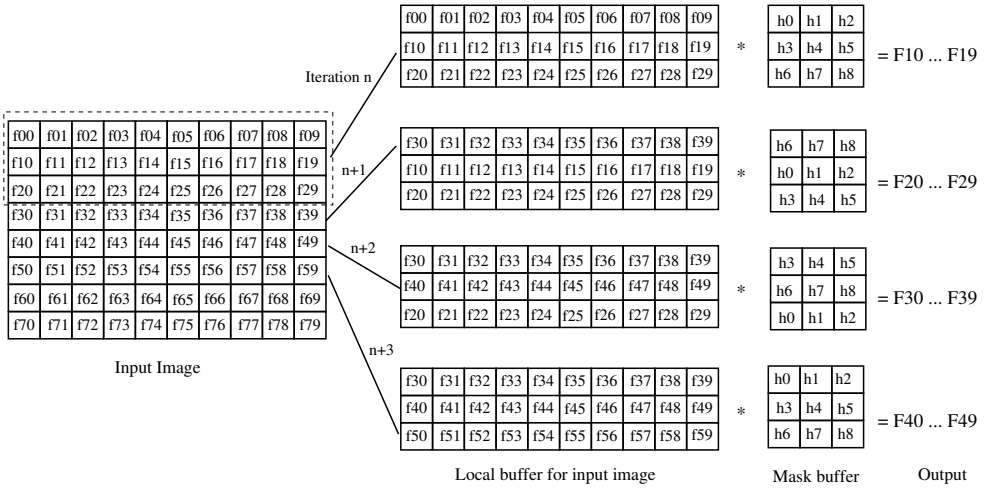


Figure 4.7 – Memory access strategy with circular shift applied to the mask. The '$*$' is the convolution operator

4.4.3 Performance evaluation

The execution times of OpenCL design for five different convolution mask sizes is presented in Fig. 4.8. We evaluate the the performance of four different versions

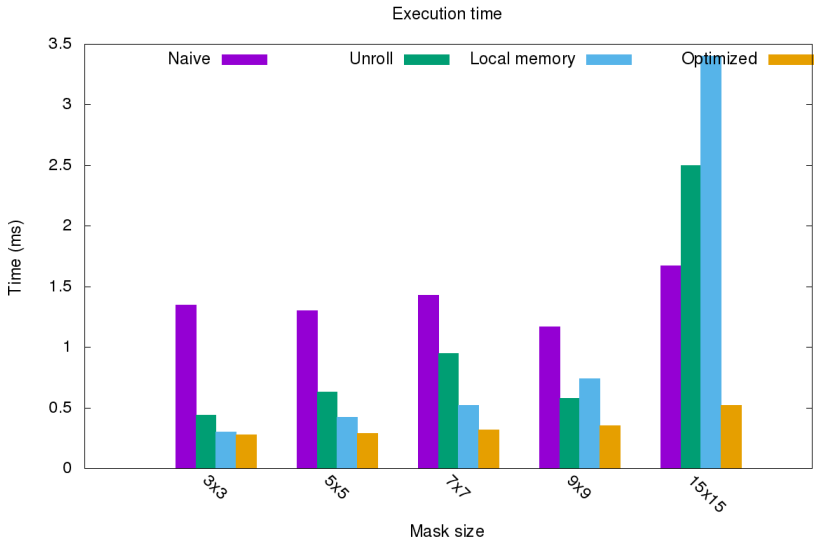


Figure 4.8 – OpenCL execution time of 2D convolution on Intel Stratix 10 FPGA

of the convolution operator from the naive version to our optimized version. These versions are as follows:

- **Naive**: This version describes the direct implementation of the operator in a pipeline execution model. The parallelism is expressed at each elementary output pixel computation level, where the multiplications between image pix-

els and the mask are performed in parallel, followed by the reduction. The computation of a new pixel starts at each clock cycle for an efficient pipeline.

- **Unroll**: This version uses loop unrolling to achieve higher throughput. Several output pixels are computed simultaneously in order to maximize the bandwidth.

- **Local memory**: This version uses on-chip BRAM to prefetch the input data in order to reduce the number of global memory access. The required data to compute a line of pixels of the output image are stored into a local buffer. Buffer data is replaced with new data to compute the new line of the output image.

- **Optimized**: This represents our optimized version presented in section 4.4.2 which combines loop unrolling, BRAM usage and a shift register mechanism. A shift register mechanism is applied to further reduce the global memory access and speed up the computation.

The optimizations have led to higher computational performance compared to the naive version. The naive version suffered from high pipeline stalls due to the high data demand from global memory for non-aligned memory accesses. The global bandwidth remains the main bottleneck by applying the loop unrolling on several pixels, but the performances are better than the naive version due to the parallelism degree and the cache mechanism implemented by the high-level compilers. Thus, the algorithm is characterized as *memory-bound*. Using FPGA on-chip memory to store data relieved the pressure on the memory bus by overcoming the global bandwidth bottleneck.

This bandwidth relief is made possible by prefetching into on-chip memory and accessing data contiguously. By analyzing Intel's profiling tool, we notice that the percentage of pipeline stall due to memory accesses has been significantly reduced. The occupancy rate of our compute units remains relatively low, which shows that there is still room for further optimization. Indeed, for the *Local memory* version, the data prefetched into local memory is only reused by the pixels of the same line, whereas the following lines should also use a part of this data. We make more memory accesses than necessary, which penalizes the performance of some versions using local memory, especially for large mask sizes (7×7, 9×9, 15×15). Therefore, we applied more advanced optimizations to the *Local memory* versions in order to improve the arithmetic intensity. The optimized versions presented above have allowed to enhance the arithmetic intensity of the design by reducing the number of memory access. Thus, we express more parallelism in the design by compute unit replication and achieving better throughput regardless of the convolution mask size.

We also performed a performance comparison of our convolution design using the OpenCL and oneAPI tools. The results showed a substantial performance loss for a large number of kernel sizes when using the oneAPI tool. Indeed, we noticed a significant difference in these two tools' global memory access latency.

This memory latency impacts the number of pipeline stages, especially in blocks requiring access to global memory. This behavior impacts the design performance slightly for an ideal pipeline as long as the pipeline can produce a result at each clock cycle. Regardless of the depth of the pipeline, once it is filled, it can perform an update at every clock cycle (or at regular intervals). However, all the pipeline stages will be impacted if the computation is stalled due to memory accesses (high stall percentage). The difference in the memory access latency for the two tools is strange because their compilers are both based on the same back-end. We have discussed the issue with Intel, and more investigations are required to elucidate the issue.

4.5 Conclusion

This chapter presents an acceleration methodology on FPGA intending to synthesize a custom architecture based on a given algorithm. The necessity of an algorithm architecture co-design approach is highlighted in order to get the best out of FPGAs using HLS tools. We present the different steps of this adopted methodology and show how we used the Berkeley roofline to guide our optimization steps. As highlighted in the previous chapter, this custom architecture makes it possible to overcome the various concerns of accelerating computationally intensive algorithms. We also presented some advanced optimizations to consider in order to better exploit the full potential of FPGAs in the compute-intensive world. The 2D convolution is accelerated on FPGA using the OpenCL language to illustrate the value of prefetching and its impact on the design throughput. However, since convolution is a simple case study, in the next chapter, the methodology is applied to the 3D back-projection algorithm used in MBIR algorithms for CT reconstruction.

Chapter 5

BP-Prefetch Architecture for CT Reconstruction

Contents

In this chapter, we will apply our acceleration methodology to the case study of tomographic reconstruction. The voxel-driven back-projection operator presented in Chapter 3 is considered to evaluate this methodology. This operator is very time-consuming and is frequently invoked in the iterative reconstruction algorithm. The algorithm is accelerated and evaluated on Intel FPGA using HLLs. The results are compared to our embedded GPU and workstation GPU implementations to highlight the FPGA efficiency. We also performed a full comparison with the state-of-the-art GPU and FPGA designs reported in the literature[1].

5.1 BP-Prefetch design

3D back-projection is one of the most time-consuming steps in iterative reconstruction. In this section, we explain the memory access strategy to avoid the main bottleneck of this algorithm. Then, we describe our BP-Prefetch architecture on FPGA along with all its main stages.

5.1.1 Back-projection algorithm

The 3D back-projector described in Section 3.1.2.4 is considered to apply our methodology of acceleration. The principle of the voxel-driven back-projector is to accumulate over the φ angle the contribution of all detectors (u, v) in line with the voxel (x, y, z).

$$f(c) = \int g(u(\varphi, c), v(\varphi, c), \varphi).w(\varphi, c)^2 d\varphi \tag{5.1}$$

$$u(\varphi, c) = x * cos(\varphi) + y * sin(\varphi) \tag{5.2}$$

$$v(\varphi, c) = x * sin(sin\varphi) - y * cos(\varphi) + z \tag{5.3}$$

where $c = (x, y, z)$ are the voxel coordinates, (u, v) are the cone beam coordinates, φ is the angular trajectory of the detector and w is the distance weight. The detector coordinates are computed using nearest neighbor interpolation. We use lookup table implementation for the trigonometric computation in order to save FPGA resources. This is done by saving into BRAM memory pre-computed values for all the sine and cosine required for our reconstruction.

[1]The contents of this chapter have been partially published in [Diakite 2021] and submitted in a journal[*Under review*].

5.1.2 Memory access strategy

The detector planes (u, v) are stored in memory following the projection angles (φ). The voxel-driven back-projector accumulates, for each voxel, the elementary contribution of each projection pixel in line with the given voxel under consideration for all angular projections. The reconstruction of a single voxel thus requires sinogram pixels located in different detector planes ($\varphi = 0$, $\varphi = 1$, ..., $\varphi = N_\varphi$). These projection data, also called sinogram, are stored in the long latency global memory of the FPGA board and cannot be fully transferred in FPGA on-chip memory because of their tremendous size. During each voxel back-projection, sinogram pixels accessed in memory are deterministic but discontinuous with high and irregular strides, as illustrated in Fig. 5.1. The coordinates of these pixels cannot be precomputed in practice because of the unsustainable storage cost it would require. Hence each voxel back-projection implies on-fly projection coordinate computations with irregular jumps in memory, making standard cache mechanisms inefficient in predicting the memory accesses for the next φ angles.

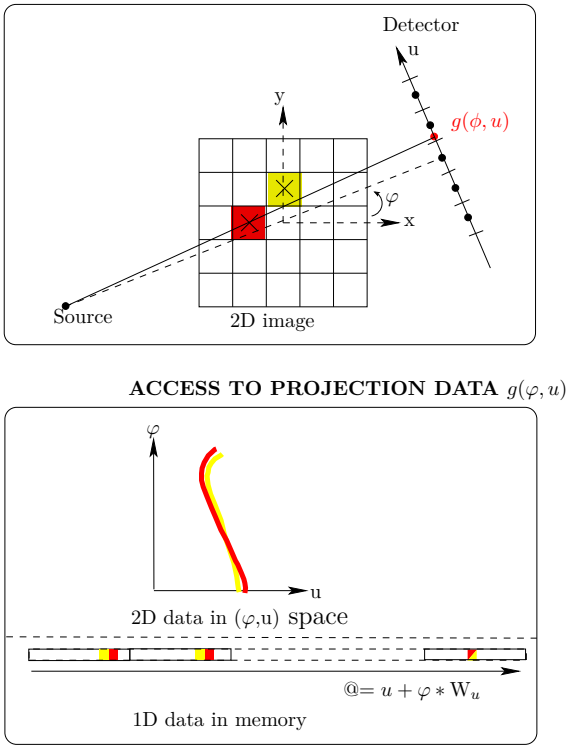


Figure 5.1 – Memory accesses for the voxel-driven back-projection in the 2D case. The red and yellow neighboring pixels draw close sinusoids in the 2D sinogram. This spatial locality is exploited during a voxel block reconstruction. W_u corresponds to the number of projection pixels

The Intel FPGA automatic on-chip cache implementations are, above all, efficient for contiguous and repetitive global memory access. Regarding non-sequential and random accesses, these automatic caches are inferred by the Intel compiler and are much less relevant for speeding up the application on FPGAs efficiently. Their inference is counterproductive and wastes valuable BRAM resources with a high risk of memory stalling. The acceleration of CT algorithms suffers from this memory stalling and consumption, which has been pointed out by [Diakite 2020] for the 3D back-projection algorithm, thus remaining a significant concern. An efficient prefetching memory strategy must be found facing this memory wall.

5.1.3 Reconstruction by voxel blocks

Performing a reconstruction by voxel blocks, i.e., having an inner loop on neighbored voxels on algorithm 9, increases the spatial and temporal locality compared to voxel by voxel reconstruction. The projection of a block (B_x, B_y, B_z) corresponds to a rectangle shape $(local_u, local_v)$ in the detector plane for a given projection angle φ_i as represented in Fig. 5.2. Inside this sinogram tile, a high data re-utilization exists during a voxel block reconstruction. Indeed, as illustrated in Fig. 5.1, a single ray may pass through several neighboring voxels in the 3D volume. Therefore, the re-utilization of projection data stored in BRAM can be exploited because all these voxels will read this single projection during the back-projection. Voxels in the same block will access the same sinogram tile for each projection angle. The main concern is to capture the projection data footprint without loss of information and calculate the coordinates of its boundary. For each voxel (x, y, z), its reconstruction depends on its N_φ angular projections. These projections are spatially distant due to their storage in the projection data following the order (u, v, φ).

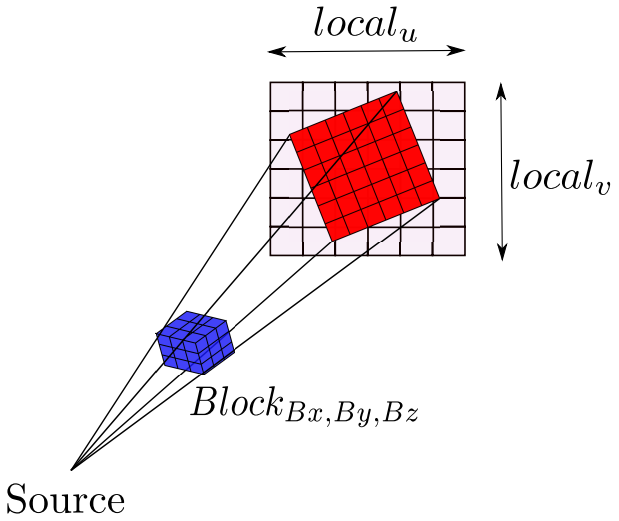


Figure 5.2 – Voxel block (in blue) and its projection (in red)

During a block reconstruction, prefetching in BRAM the sinogram tile associ-

ated with the voxel block allows a high reuse data rate for moderate costs in local memory for the storage of the sinogram tile required at the current φ iteration and the voxel block, but also in computation with the simple sinogram tile shape calculation. In order to identify the projection data footprint, we perform the following operations. The edge voxel in the bloc is projected in the detector plane, and its coordinates (u, v) are computed. Then the dimension of the projection data footprint is determined by the computation of $local_u$ and $local_v$ depending on block size (B_x, B_y, B_z) : $local_u = \sqrt{B_x^2 + B_y^2}$ and $local_v = \sqrt{2} * B_z$. The projection pixels in the located footprint (red rectangular shape in Fig. 5.2) are accessed with a regular memory access pattern. With this strategy, we gain in the temporal and spatial locality for memory accesses and consequently make these accesses contiguous for coalescence.

The use of FPGA BRAM provides higher bandwidth than the global memory bandwidth. Moreover, the access latency will be considerably reduced with a low stall percentage. This approach's advantage remains in reducing the memory access to local memory. Because the block size and shape decision is crucial for the data reuse rate and contributes to the DRAM transfer reduction, an offline memory access analysis is presented in Section 5.2.1.1.

5.1.4 BP-Prefetch architecture

The BP-cache design is the baseline version of the 3D back-projection using burst-coalesced cached Load Store Units (LSU) (algorithm 7) presented in [Diakite 2020]. OpenCL optimizations such as loop pipelining and unrolling were applied to this version to leverage the FPGA. The use of Intel automatic cache became a bottleneck when loop unrolling was applied to algorithm 7. Loop unrolling consists of fully or partially replicating the loop body, and consequently increases the BRAM usage, thus preventing DSPs' maximum use. Usually, loop unrolling does not increase BRAM usage. However, when using Intel's automatic cache with irregular access pattern, the offline compiler will have trouble coalescing memory accesses. This leads to the generation of a private cache for every global memory access. The cache is implemented using FPGA BRAM, hence several privates copies may result in a waste of valuable on-chip memory blocks. In our new BP-Prefetch design, data locality is manually managed. Therefore, unrolling does not affect the BRAM usage in a critical way.

The architecture of the BP-Prefetch design (algorithm 9) is presented in Fig. 5.3. The main stages of the architecture are made of a prefetching module and a compute unit. The compute unit is local-memory related and is placed after the prefetching module. The compute unit is based on multiple Processing Element (PE) for parallelism. In the proposed architecture, the critical path consists of reconstructing the block of voxels with the innermost loop over the voxels. The loop body, considered as PE, can be replicated for parallel voxel intensity computation by loop unrolling with a factor N. This number of PEs depends on the target FPGA available resources. To express fine-grained parallelism, a PE is designed as

Algorithm 9 BP-Prefetch algorithm

```
1: for all B_k do
2:   #pragma max_concurrency M
3:   for all φ_j do
4:     Pretching projection data
5:     #pragma unroll N
6:     for all Voxel_i ∈ B_k do
7:       Compute(u, v)
8:       block_{B_k}[Voxel_i] += projections[u, v]
9:     end for
10:  end for
11:  volume ← block_{B_k}
12: end for
```

a pipeline for good efficiency. Each PE is responsible for the calculation of u, v, and voxel accumulation. The occupancy rate of the compute unit depends heavily on the ability of the prefetcher to provide the data. We inferred multiple copies of the prefetched data for concurrent computation and therefore increased the pipeline occupancy.

The N PEs must have stall-free access to local memory when reading the projection data. We use memory replication as explained in Section 4.3.1 to ensure stall-free access. The memory replication amount depends on the number of PEs N in the design. Depending on the resource available, we may also infer M private copies to lunch multiple iterations of loop φ.

The architecture's input is the projection data in the detector plane as shown in Fig. 5.3. For each projection angle φ_i, we prefetch all projection data required (red rectangle) for the voxels accumulation in the block. We load more data, from global memory, than required to ensure correct reconstruction and take advantage of memory coalescence. After the accumulation over all projections angle φ_i, the reconstructed block of voxels (blue cube) is written back to the volume stored into the global memory.

5.1.4.1 Memory prefetcher module

The PEs in the design are fed in data by the prefetcher. The global access is a long latency operation, so we leverage the global bandwidth by making accesses in burst mode thanks to the offline memory analysis. When the kernel and the memory controller run at the same frequency, FPGAs external memory can perform 512-bit access per cycle per memory bank to saturate the bandwidth. With this in mind, the prefetcher module uses memory coalescence by performing multiple accesses simultaneously. Indeed, the projection pixels to prefetch are located in the same tile, which allows multiple pixels to be merged into a wider access from global memory. The number of accesses per cycle is chosen with caution because overusing

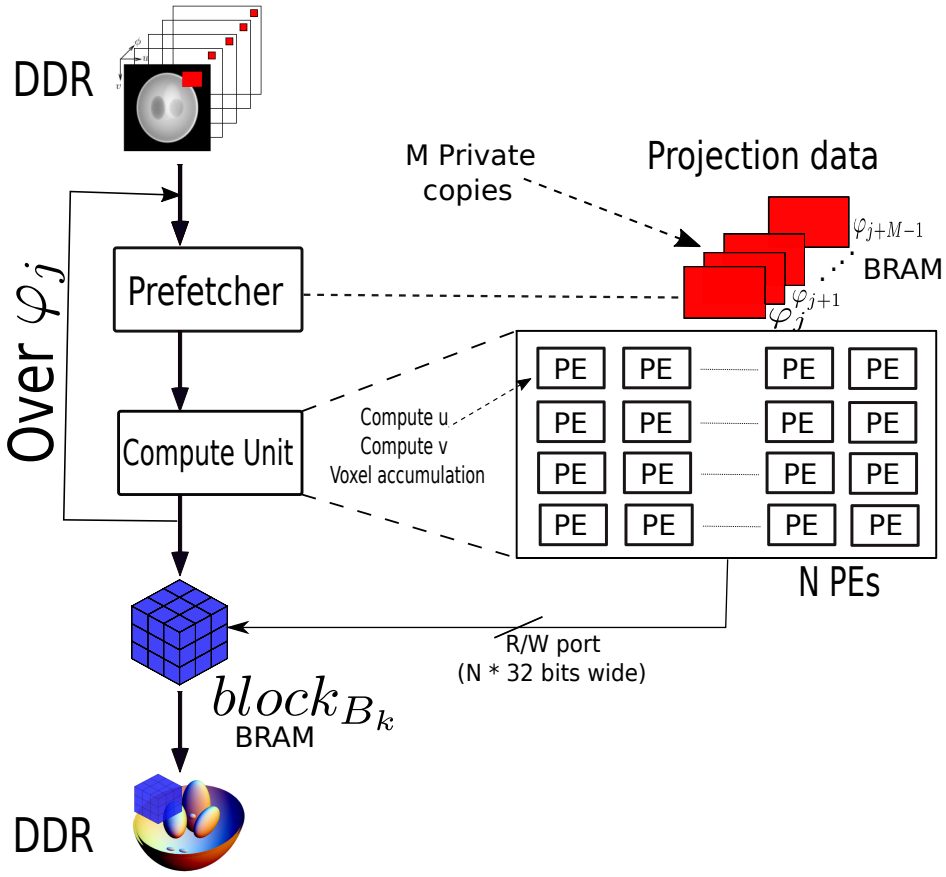


Figure 5.3 – BP-Prefetch pipeline

the bandwidth could lead to worse f_{max} for the kernel. In order to preserve the kernel operating frequency, we perform the memory accesses with respect to the available bandwidth on the target FPGA device.

The FPGA boards used in this work are equipped with two DDR memory banks offering a memory bus 1024-bit wide. The memory access pattern is regular, thanks to our access strategy. The prefetcher module computes the edge coordinates of the projection data footprint based on our offline analysis to identify the data to be fetched, as presented in Section 5.1.3. We use memory coalescence in order to fill the local memory with the projection data and maximize the bandwidth. Therefore, the prefetcher module performs a wide access of 16 floating-point values per clock cycle. The module does not exploit the full bus capacity because of the additional memory operations, such as the global volume updates.

The private copies as mentioned above in subsection 5.1.4 inferred for concurrent execution helped to hide memory accesses latency in the pipeline, hence maximizing the design throughput.

5.1.4.2 Pipeline Initiation Interval (II)

A compute-intensive application optimization requires a specific focus on loops. The FPGAs excel in the loop pipelining for greater cyclic efficiency, and this is combined with loop unrolling to maximize throughput. The 3D back-projection algorithm has four nested loops zn, yn, xn ($Voxel_i$ coordinates) and φ (see Algorithm 7). The new method (Algorithm 9) allows us to have a specific tiling in our design thanks to the reconstruction by block. The computation of detector coordinates u and v for the image edge could prevent the compiler from achieving the best II for the innermost. The compiler assumes false dependency due to conditional branching scope. We move this computation out of the branching scope and use a temporal register to avoid this false dependency to address this problem. Therefore, the compiler can achieve an II value of one for all the loops in the design. Each PE has to compute the coordinates of the detector (u, v) and the voxel accumulation.

5.2 Architecture tuning

This section discusses the design scalability on Intel Arria 10 and Stratix 10 devices. In order to take advantage of our design, the offline memory access analysis is performed to choose the block size and shape with optimal data reuse rate. The study of the CI is also presented for the roofline model.

5.2.1 Offline Analysis

5.2.1.1 Offline Memory Access Analysis

The prefetched projection data depend on the shape of the block of voxels as described in Section 5.1.3. The data reuse rate is computed by the following formula and illustrated in Fig. 5.4:

$$Data\ reuse = \frac{B_x * B_y * B_z}{\#Memory\ access\ I/O} \tag{5.4}$$

where the $\#Memory\ access\ I/O$ represents the number of actual projection pixels used and is obtained by a static analysis of the CPU code.

Memory accesses are no longer irregular thanks to our access strategy presented in Section 5.1.2. The objective is to further reduce access to the external memory of the FPGA by exploiting the data reuse potential. Indeed, a single pixel in projection data is used by several voxels during the back-projection. The offline analysis first locates the footprint of the projection data required for a given voxel block and a given projection angle. Then, the algorithm is run on the CPU, and a counter is assigned to the projection data to identify the pixels that are used during the back-projection of the block for a projection angle. These pixel numbers are used in the calculation of the data reuse rate. This allows us to ignore some of the pixels present in the projection data footprint at the time of the memory accesses, especially those at the edges of the footprint, so as not to fall back into memory

accesses with irregular strides. In order to evaluate the data reuse rate, the size of the voxel block and the number of pixels used in the projection data are required, as presented in (5.4).

The block size and shape decision depends on the BRAM consumption, the $\#Memory\ access\ I/O$ and the data reuse rate. A larger block may require a larger sinogram tile during back-projection. This large-size sinogram tile may consume several BRAM resources. Therefore, a trade-off between the data reuse rate and resource consumption must be done in order to avoid synthesis failure.

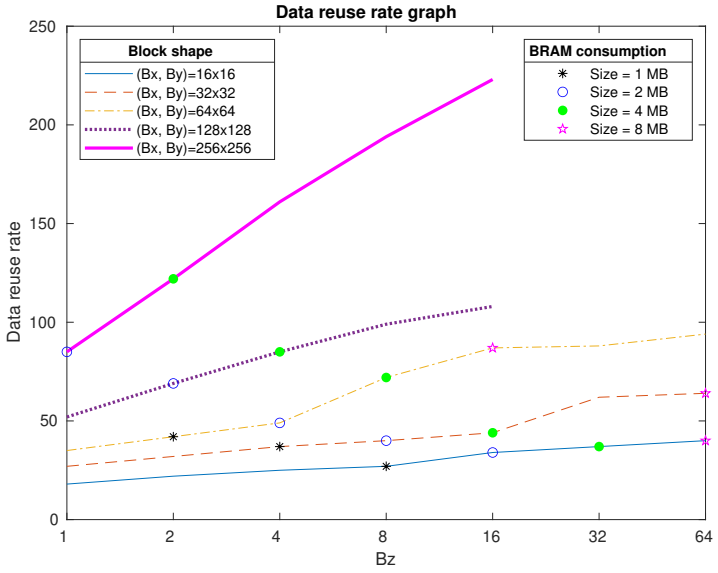


Figure 5.4 – Data reuse rate with 3 different shapes with B_z variation

The data reuse rate regarding different block shape is illustrated on Fig. 5.4 . The size of the BRAM needed to store the prefetched projection data is mentioned in order to choose to best (B_x, B_y, B_z) configuration. For a fixed number of voxels reconstructed, this requested memory size varies greatly : a (32,32,8) block requires twice as much as BRAM than a (64,64,2) block to store their projection data. This offline memory access analysis helps to manage the memory resource which is critical for this design. Indeed, with the replication and private copies overhead allowing concurrent execution, the BRAM size indicated in Fig. 5.4 have to be multiplied by the replication factor and the number of private copies. Assuming that we have three read ports per replicated memory and M private copies for concurrent execution, the required memory size will be multiplied by $\frac{N}{3}$ (the replication factor) and M (number of copies). For instance for a $\frac{N}{3} * M * 4$ MB available on BRAM, this offline study points that the highest data reuse rate is obtained for (256,256,2) blocks.

5.2.1.2 Offline Computational Intensity analysis

We use the roofline model to iteratively analyze our algorithm and guide the optimizations. Each algorithm results in a specific roofline for FPGAs. The performance roof is determined by the number of resources consumed and the effective operating frequency. The dynamic profiler gives the effective (measured) DRAM bandwidth achieved. Table 5.1 lists the number of operations Giga Bytes Operations Per Second (GBOPS) and the number of global memory accesses of the 3D back-projection in GB. We determine the CI for different versions of the algorithm. The BP-Cache design is memory-bound with low CI (see Table. 5.1) due to the lack of spatial and temporal data locality in the projection data. The CI of this version is very low, elaborating another strategy to access off-chip memory might substantially increase the CI and allow the use of more DSP slices to improve the performance.

Table 5.1 – Static analysis of CI for different blocks

	Block size	#Operations (GBOPS)	#Memory accesses (GB)	CI
BP-cache	N/A	236	17.2	13.7
BP-Prefetch	$64 \times 64 \times 1$	253	1.1	236
	$32 \times 32 \times 16$	253	0.67	377
	$64 \times 64 \times 8$	253	0.6	419
	$128 \times 128 \times 4$	253	0.57	443
	$256 \times 256 \times 2$	253	0.55	456

We can see that CI increases as long as the block size grows until it reaches the maximum data reuse rate, and at the same time, the DRAM transactions are decreased. This variation is because of the data reuse potential of our memory access strategy. The total number of operations remains the same. In contrast, the global memory invocations are considerably reduced. Therefore we perform the same number of computations for fewer access to the global memory. The CI of our design is improved by the new strategy and it is no longer in the memory bound area on the roofline.

The memory operations presented in Table 5.1 take into account memory coalescence allowed by loop unrolling. The bytes operations in Table 5.1 include arithmetic operations (multiply-accumulate, divide...), and logic comparison performed in our design. Some computations are floating-point related, and other operations involve integer operations.

5.2.2 Design on Arria 10

The possible number of PEs on Arria 10 device is $N = 64$ PEs in our architecture without exceeding available resources. Each PE must have free access to local

memory, which requires a physical port for each memory read. In the case of insufficient ports, the memory requests are made with arbitration which causes a severe performance problem for the pipeline by increasing the II [Intel Corporation 2019a]. We perform memory replication for multiple accesses in order to avoid arbitration. Our architecture requires N read-ports (32 bits wide) for all the PEs to read projection data in local memory. Thanks to the memory access strategy, global memory accesses are contiguous and are coalesced when loading projection data. Hence, only one write-port is needed to write the projection data from global into BRAM memory. The total number of memory replications will depend on the number N of PEs in the architecture. Each replicate has four ports due to double pumping (three reads and one write) and contains the same projection data for the block reconstruction. In this configuration, the total replication will be at least $\lfloor \frac{N}{3} = 22 \rfloor$

In addition, we place $M = 8$ private copies of the local memory for concurrent execution of loop iterations (φ loop in algorithm 2). This reduces the overall latency and increases the pipeline efficiency at the cost of additional BRAM consumption.

5.2.3 Replication potential on Stratix 10

We also use the Intel Stratix 10 device for our design in order to express more parallelism. The Stratix 10 is a high-end FPGA with thousands of DPS slices and more BRAM resources than the Arria 10 device. Consequently, our pipeline could benefit from more PEs and achieves better throughput. The replication factor N of our architecture is fixed to 256 for the Stratix 10 devices. We have two ways of synthesizing this design: as a single kernel pipeline with all the PEs or multikernel model with several compute units running in parallel.

The single kernel represents the architecture presented in Fig. 5.3 with N=256 PEs. The throughput of this version will depend heavily on the ability of the prefetcher module to provide the data to the PEs. By having 256 PEs running in parallel, the number of physical read ports must increase, which leads to local memory replication overhead. In this scenario, all the PEs compute voxels in the same block at a time. The challenge to best leverage the pipeline is to feed it with data and maximize the throughput. The issue is that all 256 PEs are fed with data from the same prefetcher module. The higher number of PEs in the design makes the compute unit more efficient than the prefetcher module. This imbalance between the compute unit and the prefetcher module will result in degrading the pipeline efficiency. In order to balance between the compute unit and the prefetcher module, a multikernel approach has been proposed. We split the 256 PEs into several kernels, where each kernel has its own prefetcher module for more balance. The prefetcher module designed in Section 5.1.4.1 is able to keep 64 PEs sufficiently occupied to ensure high efficiency. However, increasing the number of PEs requires an increase in the capacity of the prefetcher module. Although, as explained in Section 5.1.4.1, increasing the prefetcher module capacity strongly affects the kernel's operating frequency. We can have multiple prefetcher modules in the design without sacrificing the kernel frequency. Indeed, several prefetcher modules do not affect the kernel

frequency because the compiler considers them separately and schedules their use of the memory bus in various clock cycles during the pipeline execution.

We replicate our SWI kernel four times to have multiple kernels in concurrent execution in order to further improve the design efficiency as illustrated in Fig. 5.5. Therefore we have the same pipeline architecture with four instances, and each ker-

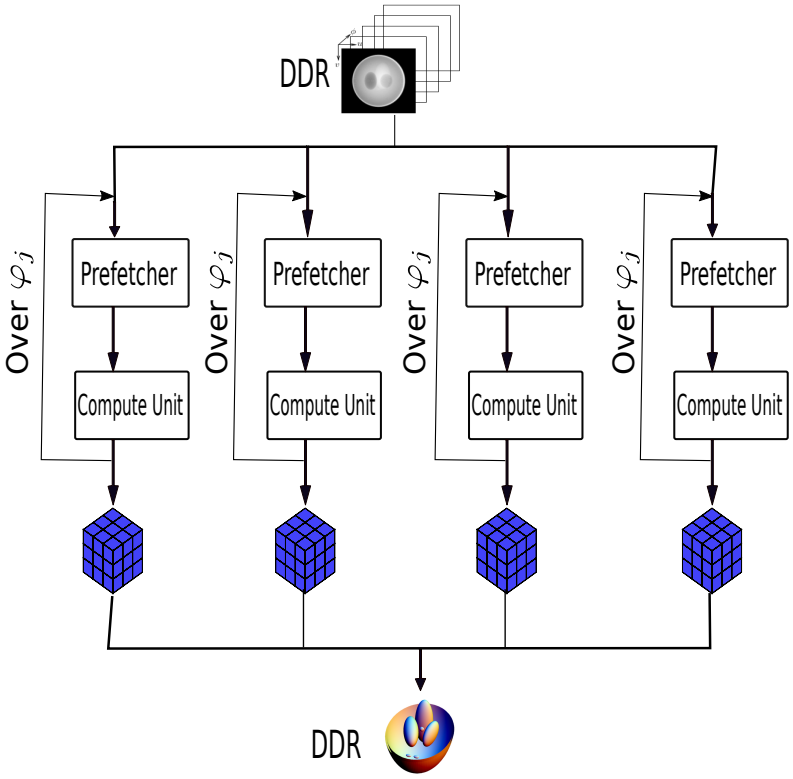


Figure 5.5 – The multikernel architecture of the 3D back-projection algorithm

nel has 64 PEs. The kernel replication is quite identical to compute unit replication available in the NDR kernel. However, the replication of the kernel consumes additional hardware resources because the whole kernel object is replicated. On Stratix 10, the multikernel model is not recommended as optimization, especially if there is a data exchange between the kernels. Nevertheless, if there are no dependencies between the kernels, the only concern is the resource consumption overhead. The available resources on the Stratix 10 device are enough to support this replication overhead. This is good for our design because the four kernels run concurrently on different portions of the 3D volume without any communication.

5.3 Results

We evaluated in this section the performance of our design on Intel FPGA devices guided by the roofline analysis. The performance metrics such as the execution time, the pipeline stall, and the occupancy are discussed. The dataset used in this

work and the software and hardware platforms have been described in Section 3.4.1. For the sake of reproducibility, the source code of this project is publicly available[2].

5.3.1 Design evaluation

5.3.1.1 Arria 10 results and roofline

Table 5.2 shows the results of our implementation using the Arria 10 device. The performance metrics such as operating frequency, execution time, stall percentage, and occupancy are presented in this table. The stall percentage represents the amount of time memory access causes pipeline stall. In contrast, occupancy represents the percentage of time when a work item (thread) performs a valid memory instruction. The more the occupancy is, the more the compute units are active during the execution. The BP-cache version suffers from a high pipeline stall percentage because of the memory access pattern, making the global bandwidth the main bottleneck. The execution time of this version is not acceptable for CT reconstruction routine. The DSP usage was very low because of high BRAM usage to implement the cache. Therefore, extra compute unit replication was impossible, and the BP-cache design could not achieve higher throughput.

Table 5.2 – Block size variation effect on the performance on Arria 10

Version		BRAM (%)	DSP %	Stall (%)	Occ (%)	Freq (Mhz)	Time (s)	GUPS
BP-Cache		72	27	71.27	24.6	150	3.5	1.14
Prefetch	$32^2 \times 16$	73	58	0.2	74.7	179.2	0.502	7.97
	$64^2 \times 8$	72	63	0.06	84.1	189	0.425	9.4
	$128^2 \times 4$	68	63	0.56	90.2	176	0.423	9.45
	$256^2 \times 2$	73	62	0.06	94	180	0.396	10.1

The BP-Prefetch design (Algorithm 9) achieved better performance compared to the BP-Cache version. It overcomes the issue mentioned above by providing a good design with acceptable memory access pattern. The results show the effect of block shape and size variation. We saw in Fig. 5.4 that the reuse rate varies with the number of voxels in the block, i.e., high block size results in a high reuse rate. However, the BRAM resource consumption should be considered concerning the available resources on the target FPGA. A block size that consumes at most 4 MB has been used to store the projection data for our architecture configuration, and the data reuse rate is high enough for this amount of memory. The results of our architecture on Arria 10 are presented in Table 5.2 for different block sizes and shapes. The stall percentage is very low for all the chosen blocks, and the

[2]The source code is available at https://github.com/nicolasgac/tomoGPI

occupancy is high to ensure good execution time. The change in the data reuse rate is noticed in the increase of the occupancy of the architecture.

In this design, all loops are successfully pipelined with an II value of 1. There is no memory access conflict, and the data dependencies are handled in one clock cycle. We tune the block sizes and shapes to have the best performance possible for our architecture. We have obtained a better execution time with the $256 \times 256 \times 2$ block, which corroborates the static study performed on the data reuse. Compared to the previous BP-cache design, we achieved a speedup of 9.2 at 180 MHz.

The roofline plots are presented in Fig. 5.6 for four different blocks using the Arria 10 device. The chosen blocks are the ones that provided a high reuse rate, therefore the shapes and sizes are different. These high reuse rates will result in high CI moving the algorithm out of the memory-bound area. Since the algorithm is no longer in the memory-bound zone, the challenge is to get as close as possible to peak performance, i.e., the computational ceiling. This can be problematic because even if the application is not limited by memory on the roofline, it can be prevented from having the maximum performance because of the impact of the memory access latency. Unfortunately, the effect of these memory latencies is not directly visible on the roofline. For this different optimization techniques are used such as the access latencies hiding or the maximization of concurrent executions to occupy the compute units. As shown in roofline, by increasing the size of our blocks, the CI also increases, allowing us to have a better occupancy rate and get close to the computational ceiling. The computational ceiling of our architecture is not affected by the algorithm's CI because our block size does not affect our design's compute configurations. The BRAM usage is concerned by the block size variation while the DSP consumption is slightly impacted.

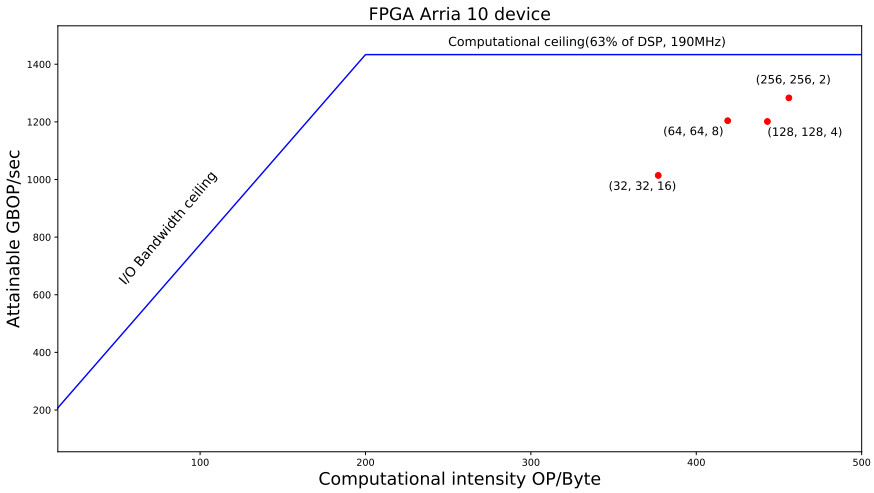


Figure 5.6 – Roofline of BP-Prefetch with different B_z on Arria 10

A 100% occupancy rate would be on the horizontal line of our computational ceiling. However, our best design on Arria 10 has an occupancy of 94% which is close to computational ceiling.

5.3.1.2 Scalability to Stratix 10

Our design is then implemented and evaluated on Intel Stratix 10 device as a single kernel and multikernel. In the multikernel version, we replicated this design with the same block size on our Stratix 10. Therefore we have four compute units working on four blocks of voxels concurrently. The BSP ensures the OpenCL device bandwidth. In practice, the effective design bandwidth is low compared to the BSP's maximum

Table 5.3 – Single kernel versus multikernel on Stratix 10

Design	Block	BRAM (%)	DSP (%)	Stall (%)	Occ (%)	Freq (MHz)	Time (s)
Single kernel	$64^2 \times 8$	50	49	8.28	61.5	127	0.32
	$128^2 \times 4$	31	50	4.01	60	87	0.47
	$256^2 \times 2$	40	57	3.89	60.2	117.2	0.24
Multikernel	$64^2 \times 8$	33	57	13.91	82.3	172.5	0.12
	$128^2 \times 4$	36	57	1.45	83.1	133	0.84
	$256^2 \times 2$	40	57	2.28	85.1	127	0.51

capability. The bandwidth follow-up is primordial for an acceleration equivalent to the parallelism factor (the ideal speedup). We use the bandwidth to its full capacity without sacrificing operating frequency.

The roofline model of the single kernel and multikernel designs is presented in Fig. 5.7 on the Stratix 10 device. The single pipeline version contains 256 PEs working on the same block of voxels at the time. Despite the concurrent execution of the loops, the pipeline occupancy rate was not optimal. The prefetcher module could not efficiently feed the pipeline for better throughput. Furthermore, the operating frequency of this version was slightly low, as shown in Table 5.3 due to the complexity of the single compute unit. Compared to the Arria 10 implementation, the Stratix 10 results were not as good as expected, even though the design used a high number of PEs for a single kernel. However, the block $256 \times 256 \times 2$ provided the best execution time with a 1.6× speedup using 4× as much PEs as the Arria 10 design.

The multikernel configuration allows the design to achieve a better operating frequency. This improved frequency enables further saturation of the global memory bandwidth and the occupancy rate of each kernel. The routing constraints are relaxed with this strategy of splitting the compute unit. Another main difference between the two designs is that the multikernel version runs on four blocks in parallel. The volume was divided into four subvolumes reconstructed by the four

compute units. The reconstruction is performed block by block within each sub-volume. Therefore the multikernel version is preferred over the single kernel, which achieves the best operating frequency and the optimal design occupancy. Despite

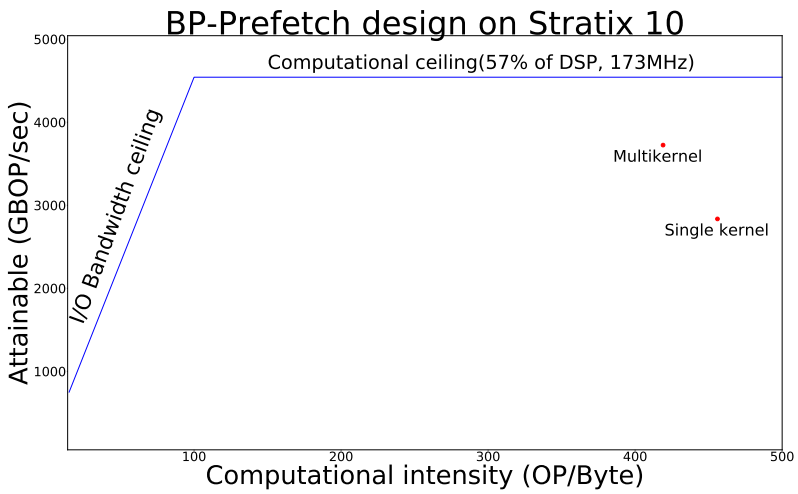
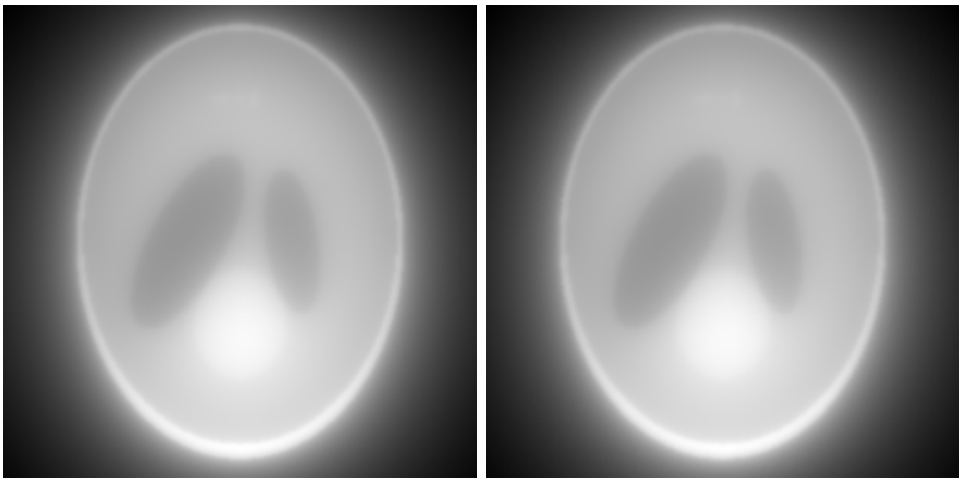


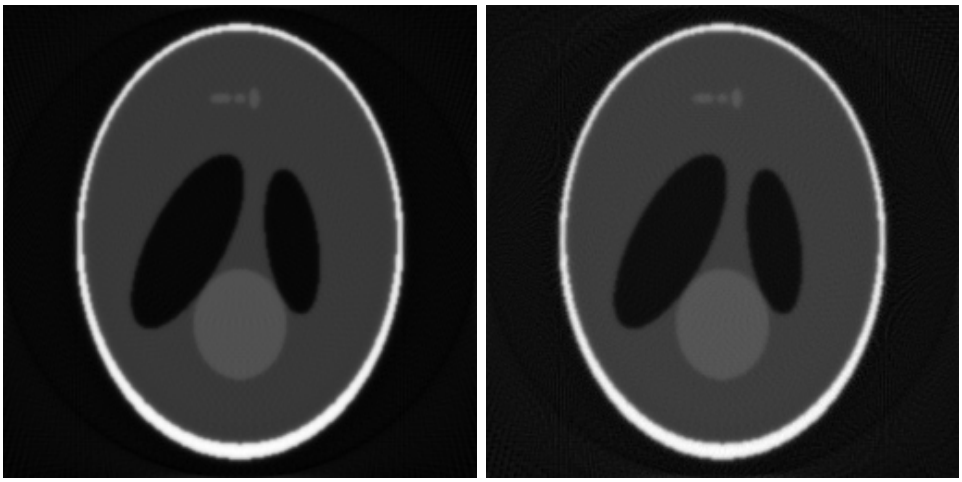
Figure 5.7 – Roofline of BP-Prefetch on Stratix 10

the high occupancy for blocks $256 \times 256 \times 2$ and $128 \times 128 \times 4$, their execution times are worse than the single kernel version. This is due to a prediction mechanism implemented by the AOC compiler on FPGAs. The computations of the projection data coordinates are performed on the fly, and the hardware tries to predict these computations to optimize the design. By analyzing the dynamic profiler, we notice that the percentage of prediction hit was very low for these two blocks, which means most of the computations performed were invalid. The success of prediction rate was given by the *activity* metric in the dynamic profiler. The best execution time was obtained for the block $64 \times 64 \times 8$ with high occupancy and a good prediction rate. It should be noted that the prediction failed when the number of PEs is lower than the number of voxels per line in the block. This is the reason we do not have the failed prediction for a single kernel because the number of PEs (256) is equal to or greater than the voxels per line of the block (64, 128, or 256). For multikernel we had 64 PEs, and for blocks with more than 64 voxels per line, the predictions failed several times. Therefore, a good consistency is required between the chosen block and the number of PEs.

Thus, the multikernel has achieved an execution time of 0.12 s at 172.5 MHz and preserved the pipeline occupancy for each compute unit with a block size of $64 \times 64 \times 8$. However, the stall percentage of the multikernel version is slightly high because we have multiple kernels with multiple prefetcher modules soliciting the global bandwidth. Nevertheless, this increase in memory stall does not impact the overall performance that much, and yet the design achieved significant throughput.



(a) Reconstructed by GPU. (b) Reconstructed by FPGA.

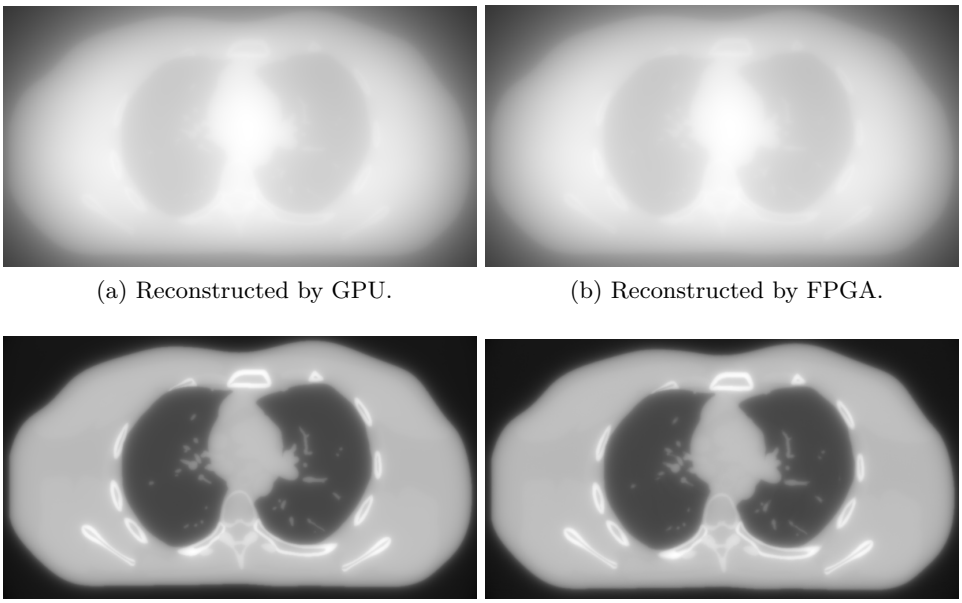


(c) Reconstructed by GPU. (d) Reconstructed by FPGA.

Figure 5.8 – Shepp-Logan phantom reconstructed image slice 128. Fig. 5.8a-5.8b: The reconstructed image using single back-projection. Fig. 5.8c-5.8d: The reconstructed image after the full MBIR algorithm.

5.3.2 Image Accuracy

We use Shepp-Logan phantom and XCAT phantom [Segars 2008] test cases to evaluate our design. Slices of the 256^3 volume of Shepp-Logan phantom and $1024^2 \times 256$ on XCAT phantom are illustrated in Fig. 5.8 and Fig. 5.9, respectively. The image slices are reconstructed by GPU and FPGA using single-precision floating-point representation. We used the nearest neighbor method for interpolation on FPGA, while the GPU version used the bi-linear interpolation method. Table 5.4 shows the evaluation of the FPGA reconstructed images with respect to GPU ones.The Universal Quality Index (UQI) [Wang 2002], the Correlation Coefficient



(a) Reconstructed by GPU. (b) Reconstructed by FPGA.

(c) Reconstructed by GPU. (d) Reconstructed by FPGA.

Figure 5.9 – XCAT phantoms reconstructed image slice 128. Fig. 5.9a-5.9b: The reconstructed image using single back-projection. Fig. 5.9c-5.9d: The reconstructed image after the full MBIR algorithm.

(CC), the Normalized Root Mean Squared Error (NRMSE), and the Signal to Noise Ratio (SNR) metrics have been used in order to evaluate our design accuracy compared to the GPU reconstructed image. According to these metrics, the FPGA reconstructed images are accurate enough for CT application compared to the GPU reconstructed images. The slight loss of accuracy is due to the lack of bilinear interpolation.

Table 5.4 – Image quality evaluation between GPU and FPGA reconstructed images on different datasets

Dataset	Algorithm	UQI	CC	NRMSE	SNR
Shepp-Logan	Back-projection	0.999	0.999	0.0162	37.5
	MBIR (100 iterations)	0.996	0.996	0.099	25.9
XCAT	Back-projection	0.999	0.999	0.0659	35.4
	MBIR (100 iterations)	0.999	0.999	0.0457	37.9

5.4 Comparison and discussion

We compare the results of our design with the state-of-the-art tomography implementations on GPU and FPGA. In this comparison, the overall throughput, design efficiency, power consumption, and resource consumption are assessed.

5.4.1 Implementation on general-purpose processors

In this section, we will talk about the CPU and GPU implementations of the 3D the back-projection algorithm. Indeed, the CPU and GPU architectures are used as the reference platforms for comparison in our work.

5.4.1.1 CPU implementation

The CPU implementation is the reference version of the 3D back-projector. It is the sequential execution mode of the algorithm with three major loops over voxels within the 3D volume as illustrated in Algorithm 10. The loops order will have an impact on the overall performance on CPU because the memory access pattern will not be same. Indeed, the CPU processors are recipient of advanced cache mechanism that extensively exploit the spatial and temporal data locality.As mentioned above, this algorithm has an irregular memory access pattern which strongly impact the performance. However, the multiple cache level on CPU processors can optimize these memory accesses with the right loop ordering (Algorithm 10).

Algorithm 10 Voxel-driven back-projector on CPU

Require: $\alpha[dim_{\varphi}], \beta[dim_{\varphi}], sinogram$
Ensure: $3D\ volume$
1: **for all** zn, yn, xn **do**
2: $voxel_{sum} \leftarrow 0$
3: **for all** φ **do**
4: $Compute(un, vn)$
5: $voxel_{sum} += sinogram[un, vn, \varphi]$
6: **end for**
7: $volume[xn, yn, zn] = voxel_{sum}$
8: **end for**

5.4.1.2 GPU implementation

An optimized CUDA implementation from [Gac 2014] is used for our embedded and HPC GPUs. The voxel-driven 3D back-projection algorithm can be massively parallelized on GPU architecture since the computation of each voxel is independent of the others. GPUs with their multiple CUDA cores are therefore used to accelerate this algorithm. The bi-linear interpolation is used in this case thanks to the texture

units presented on the GPU. Indeed, the GPU textures are read-only cache including hardware that can perform filtering and interpolation.

5.4.2 Performance comparison

Table 5.5 – Performance comparison of our work and other works.

Ref.	Back-projector[a] (geometry, volume)	Arch.	Platform (Year, Freq., Process size, Cores)	Time (s)	GUPS	Update /cycle /op[b]	PE	Cycle /update /PE
This work	VD-float, 256 updates/voxel (CT circular, 256^3)	**FPGA**	Arria 10 (2014, 189 Mhz, 20 nm, 1518)	0.396	10.1	0.030	64	1.12
			Stratix 10 (2016, 172 Mhz, 14 nm, 5760)	0.12	33.3	0.032	256	1.23
		GPU	C2050 (2011, 1.15 Ghz, 40 nm, 448)	0.129	31.1	0.032	448	15.4
			P100 (2016, 1.46 Ghz, 16 nm, 3584)	0.017	237	0.027	3584	18.7
			V100 (2017, 1.38 Ghz, 12 nm, 5120)	0.011	364	0.028	5120	18.1
			A100 (2020, 1.41 Ghz, 7 nm, 6912)	0.009	424	0.023	6912	21.4
			TX2 (2016, 1.46 Ghz, 16 nm, 256)	0.25	16.0	0.023	256	21.8
[Chou 2011]	VD-float, 360 updates/voxel (CT circular, 512^3)	**GPU**	C1060 (2008, 1.30 Ghz, 55 nm, 240)	2.47	18.2	0.031	240	15.9
[Gac 2008]	VD-fix, 480 updates/voxel (PET, $128^2 \times 63$)	**FPGA**	Virtex-4 (2004, 200 Mhz, 90 nm, 160)	0.526	0.88	0.025	8	1.70
[Choi 2016]	RD-fix, 831 updates/voxel (CT helical, $512^2 \times 372$)		Virtex-6 (2009, 100 Mhz, 40 nm, 3456)	14.8	5.1	0.052	64	1.17
[Wen 2020]	RD-fix, 502 updates/voxel (CT helical, $1024^2 \times 128$)		ZCU102 (2018, 300 Mhz, 16 nm, 2520)	2.10	29.9	0.024	-	-

[a] *The back-projector studied in our work is Voxel-Driven with floating point computation (VD-float). Back-projectors studied by Choi [Choi 2016] and Wen [Wen 2020] are Ray-Driven with fixed-point computation (RD-fix).*

[b] *op. : hardware operators (adders and multipliers). GPU cores have two FP32 (Floating Point on 32 bits) op. with one multiplier accumulator; DSP Intel FPGA have two FP32 op. with one multiplier accumulator; DSP Xilinx have three fixed-point op. with a pre-adder and a multiplier accumulator.*

Several works on CT algorithm acceleration on hardware architectures have been reported. These different works use different approaches such as ray-driven, voxel-driven or separable footprint. Choi *et al.* [Choi 2016] has proposed a ray-driven voxel-tile parallel approach to accelerate the EM algorithm on FPGA. They exploit the data reuse to leverage the BRAM memory by minimizing global memory access and expressing a high parallelism. Gac *et al.* [Gac 2008] has proposed a pipelined architecture exploiting the spatial and temporal locality of the back-projection algorithm. Their method overcomes the memory bottleneck by loop reordering to take advantage of cache memory using HDL language. FPGA designs with HDL, as in [Gac 2008], are known to provide efficient pipeline architecture, and this same efficiency can also be seen when using HLS tools. An asynchronous beam-based parallelism to accelerate the Mumford-Shah (MS) algorithm with a high data reuse rate and low external memory transactions has been proposed by Zhang *et al.* [Zhang 2020]. Their approach reduces the computational cost of the backward projection to a lightweight operation. Wen *et al.* [Wen 2020] has proposed a data management strategy to achieve a best reuse rate and exploit the FPGA on-chip

memory to accelerate the forward and backward projections. A forward projection parallel architecture on FPGA has been proposed by Kim *et al.* [Kim 2012]. Their projector architecture was fully pipelined, and exploited loop-level parallelism for high performance. In [Chou 2011], the authors proposed a GPU-based reconstruction algorithm to accelerate the forward and backward projections. Their approach takes advantage of the GPU architectures to perform multiple rays computation with multiple threads while avoiding thread divergence.

Table 5.5 shows that our embedded GPU implementation achieved slightly better performance than the Arria 10 design in terms of GUPS. Vivado HLS is commonly used as HLS tool for FPGA accelerations as in [Choi 2016, Wen 2020], although we use Intel FPGA SDK for OpenCL in this work. OpenCL SDK is at a somewhat higher level of abstraction than Vivado HLS, giving the designer more control over the pipeline by using HLS compilers such as Vivado HLS or Intel HLS compiler. Choi *et al.* [Choi 2016] used the Convey HC-1ex platform based on helical geometry. This platform runs at 100 Mhz of operating frequency with four FPGAs, and the authors' design consumes 1408 DSP slices. Wen *et al.* [Wen 2020] targeted the Xilinx ZCU102 platform based on an UltraScale FPGA and with an overall DSP utilization of 1476 at 299.97 Mhz.

Our Stratix 10 design with four compute units and a total of 256 PEs achieved an overall throughput of 33.3 GUPS at 172.5 Mhz. We compared the results to our GPU implementation using an embedded GPU and a powerful desktop GPU. The Stratix 10 design outperforms our embedded GPU implementation on the same dataset. However, the GPU A100 was the most efficient platform regarding throughput. The DSP usage of the four kernels is 3282 slices. The results show better performance compared to all the implementations in Table 5.5. Our design achieves a 6.5× speedup of throughput in terms of GUPS compared to the back-projector of Choi *et al.*. The Convey HC-1ex platform of Choi *et al.* [Choi 2016] contains four FPGAs. However, The results reported here are for one FPGA in order to make a fair comparison since we use a single FPGA platform. However, their back-projector is a ray-driven approach, and their PE is responsible for ray tracing. Our PE performs a voxel update since the back-projector used in this work is a voxel-driven approach. Moreover, each ray traverses an average of 168 voxels in our dataset, while it traverses 1004 voxels in their dataset. They have a higher potential for data re-utilization in their dataset than ours. Nevertheless, our design exploits more parallelism and concurrent computations to achieve high throughput.

We then evaluated the design efficiency of all the FPGA implementations by comparing the number of updates performed per cycle by each hardware operator (adder and multiplier). Our OpenCL implementation on Arria 10 and Stratix has approximately the same design efficiency as the HLS ones (Table 5.5). It should be noted that the work on FPGA related in the literature used fixed-point representation to perform the reconstruction while we use floating-point single-precision for our design. Potentially, performances could be improved by a factor of two if Intel DSP is used as two fixed-point Multiplier ACcumulator (MAC) instead of one floating point MAC.

Finally, we have evaluated the pipeline efficiency of each PE. Our FPGA architectures on Arria 10 and Stratix 10, Gac *et al.* [Gac 2008] HDL design and Choi *et al.* [Choi 2016] have an efficiency close to the optimal of one cycle per update per PE. Conversely, GPU cores have an efficiency of around 20 cycles per update. It highlights the strength of FPGA technology which allows the design of customized architectures with a high computation efficiency even if its lower frequency clock and hardware resources density make it, at the end, slower than GPU.

5.4.3 Resource and power analysis

5.4.3.1 Resource consumption

Table 5.6 shows the resource usage of the $64 \times 64 \times 8$ block version . As mentioned above, our design on Arria 10 contains 64 PEs on Arria 10 device and 256 PEs on Stratix 10. The BRAM usage also includes the memory replication overhead in order to support concurrent access within the pipeline. The resources that are consumed by the design on Stratix 10 are shown in Table 5.6 for our four compute units. The table reveals that the extra logic consumption is due to the kernel replication overhead in terms of LUT and DSP. It should be noted that the LUTs are larger in Xilinx devices (6-input) than in Intel devices (4-input). Therefore it is the percentage of usage that matters.

Table 5.6 – FPGA resources consumption

Reference	FF	LUT	BRAM	DSP
Ours Arria 10	407183 (25%)	184616 (23%)	1967 (73%)	949 (63%)
Ours Stratix 10	1338708 (36%)	604963 (32%)	3898 (33%)	3282 (57%)
Choi *et al.*	1263716 (33%)	1142380 (60%)	3680 (64%)	1408 (40%)
Wen *et al.*	200062 (36%)	235928 (86%)	1352 (74%)	1476 (58%)

Our design consumed more DSP slices than other FPGA designs due the high level of parallelism exploited by our method. The complexity of our pipeline is not identical with the ray-tracing PE for other works because our approach is based on voxel-driven. Moreover, this work uses single-precision floating-point numbers for the volume image and the projection data, while other works [Choi 2016, Wen 2020] use fixed-point values. Floating-point computations are more expensive than fixed-point regarding latency and hardware resources.

5.4.3.2 Power consumption

Table 5.7 shows the power consumption of our design on different architectures. Intel Arria 10 and Stratix 10 boards contain a power sensor that reads the actual power consumed by the board using a software API provided by the vendor. We used the Intel power monitor to read the sensor and measure the power consumption of the FLIK Arria 10 board. The *fpgainfo* tool from the Open Programmable Acceleration Engine (OPAE) C toolkit was used to measure the power sensor on Intel Stratix 10 board. Therefore, The power values reported in Table 5.7 are the actual power consumption of the FPGA board. The reported values correspond to the maximum Thermal Design Power (TDP) of 250W for the GPU boards.

Table 5.7 – Design energy efficiency

Device	Process (nm)	Power(W)	Time (s)	GUPS/Watt
GPU P100	16	250	0.017	0.95
GPU V100	12	250	0.011	1.45
GPU A100	7	250	0.009	1.70
Jetson TX2	16	12.9	0.25	1.22
Arria 10	20	14.9	0.396	0.68
Stratix 10	14	23.5	0.12	1.42

The embedded Jetson is more energy-efficient than our architecture on Arria 10 FPGA, which was designed using OpenCL for the back-projection algorithm. Conversely, the Stratix device is more efficient than Arria 10 and the embedded GPU. We can say that the Stratix 10 is as energy-efficient as the GPUs except for the A100 device. However, the difference in the process size between Stratix 10 and A100 devices (14 nm versus 7 nm) is quite significant. The FPGA can potentially become advantageous than GPUs because there is still room to reduce the process size. By exploiting this technological gap, FPGAs can benefit from more computing resources and be more energy efficient.

5.5 Tomography oneAPI

We built our BP-prefetch architecture using the oneAPI HLS tool. As oneAPI is Intel's new higher abstraction framework based on the SYCL standard, using it would allow us to evaluate the design and compare the results with OpenCL design.

5.5.1 Design

The oneAPI design took advantage of the methodology applied to the 3D back-projector and the underlying FPGA devices. The difference relies in the change of the design tool. The same BP-Prefetch architecture has been designed on Intel

Arria 10 and Stratix 10 devices. As explained in Section 1.3.3, the oneAPI framework inherited from the OpenCL specification with additional features enabling the full design on FPGA through DPC++ language. The OneAPI and OpenCL tools are very similar in terms of kernel design. The optimizations are the same, with some differences in the names of attributes and pragmas. In addition, the OpenCL compiler is encapsulated in the DPC++ compiler, meaning there should not be much difference in the architecture synthesis. Nonetheless, the host code of oneAPI is much shorter in terms of lines of code than OpenCL. The reduction of these lines of code is interesting because OpenCL is much more verbose on the host side. Initializing the OpenCL platform to run a kernel on an FPGA is a tedious task that requires the developer to handle everything by hand from start to finish. OneAPI automates a good part of these tasks, which the compiler handles with little developer intervention. Furthermore, data transfer between host and kernel, which can be implicit or explicit, is also simplified with the oneAPI tool. By ensuring the same level of performance as OpenCL, oneAPI seems more user-friendly and capable of attracting most software developers in order to use FPGAs to speed up their applications.

5.5.1.1 Arria 10 design

The algorithm is implemented on Arria 10 as a multi-stage deep pipeline similarly to the OpenCL design with 64 PEs running in parallel. Each PE has stall-free access to BRAM memory for reading and writing into the projection data. The local memory is double-pumped to reduce the replication overhead. Multiple private copies of local memory are also used to express maximum concurrency and keep up the PEs during the design execution. Buffers are used to contain data from the host to the device. Then, the compiler creates the appropriate LSU depending on the kernel access pattern to perform the global memory accesses. The same pipeline optimizations that have been applied to the OpenCL kernel are applied to the oneAPI kernel as well. In the oneAPI kernel, all loops are fully pipelined efficiently with an II value of one. The local memory accesses are all stall-free without arbitration because no access ports are shared. The global memory access remains a long latency operation, but thanks to our memory access strategy and the exploitation of the data reuse potential, these accesses are made consecutive and reduced in number. In addition, the private copies of local memory for the concurrent execution of the different loop iterations make it possible to hide the latency of the global memory accesses.

5.5.2 Performance analysis and comparison

Table 5.8 presents the comparison of oneAPI results with OpenCL in terms of design efficiency, throughput and resource consumption. Both tools were used to implement the architecture of the back-projector on FPGA. An architecture based on the non-optimized version (BP-Cache) and one for the optimized version (BP-Prefetch)

Table 5.8 – Comparison between oneAPI and OpenCL on Arria 10

Design	Tool	BRAM	DSP	$\#PEs$	Freq (MHz)	Time (s)	GUPS
BP-Cache	OpenCL	1958 (72%)	406 (27%)	32	150	3.5	1.14
	OneAPI	2088 (77%)	374 (25%)	32	163	6.43	0.62
BP-Prefetch	OpenCL	1952 (71%)	949 (63%)	64	189	0.396	10.1
	OneAPI	1041 (38%)	878 (58%)	64	229	0.51	7.84

with our algorithm-architecture co-design strategy. Analyzing the performance of both designs, we see a performance loss of the oneAPI tool compared to OpenCL. The difference lies in the efficiency of the pipeline generated for each tool. By analyzing the pre-compilation report files, the pipeline generated by oneAPI is slightly deeper than that of OpenCL due to their memory management. For a very deep pipeline with a large percentage of stalls, the number of idle stages is very high per clock cycle, which makes the pipeline inefficient. However, in terms of operating frequencies, the oneAPI tool manages to reach higher frequencies. Moreover, oneAPI seems to better handle the hardware resources of the FPGA by using fewer DSPs for the same number of PEs and sometimes much less BRAM compared to OpenCL. The development time of oneAPI have been slightly reduced because it requires fewer lines of code compared to OpenCL. A trade-off between performance, resource consumption and development time is necessary to choose one tool over another.

However, oneAPI and OpenCL offer a more valuable reduction of development time and simplicity of use than traditional HDLs. This substantially increases design productivity when using HLS tools. Nevertheless, the full synthesis for the HLS tools is still time-consuming (several hours depending on the design complexity) due to the long placement and routing steps, unlike CPU and GPU architectures. Both tools allow code portability between different architectures such as CPU, GPU, and FPGA. Similar to OpenCL, the portability of oneAPI does not ensure performance, and the algorithm might be rethought according to the underlying architecture. The number of lines of codes of oneAPI is reduced compared to OpenCL on the host side. However, the number is approximately the same on the kernel side because oneAPI is encapsulated in OpenCL.

5.6 Iterative reconstruction algorithm

In this section, we present the TomoGPI reconstruction toolkit developed at the L2S laboratory and the additions to the software made during this thesis. We

present the different strategies for parallelizing the iterative loop of the optimization algorithm on FPGA for tomography. Indeed, so far we have only considered the two computational burdens of the algorithm, which are the projector and back-projector. We consider two implementation strategies for the algorithm: CPU-FPGA and full FPGA implementations.

5.6.1 TomoGPI

TomoGPI is the iterative reconstruction software developed in L2S laboratory at partnership with Safran for 3D computed tomography used in medical imaging and non-destructive testing. This software makes it possible to read raw data coming from the scanners and to reconstruct them by the methods conventionally used (analytical and iterative methods). The TomoGPI software allows an acceleration of the reconstruction algorithms thanks to its use of single and multi-GPU servers. Indeed, in its initial version, TomoGPI only supports the mono-GPU acceleration of iterative algorithms. Then multi-GPU parallelization was added to the software. The work of this thesis has allowed TomoGPI to have support for FPGA boards with HLS tools. Several operators have been accelerated on FPGA and added to the software.

Several other reconstruction toolkits exist in the literature, such as Astra [Palenstijn 2017], RTK [Rit 2014], or TIGRE [Biguri 2016]. These toolkits use non-dual pairs for reconstruction, whereas TomoGPI has matched and unmatched pairs. Furthermore, unlike TomoGPI, these toolkits do not support the FPGA-based reconstruction of iterative algorithms.

5.6.2 CPU-FPGA strategy

The first implementation strategy is to run the iterative loop on the host processor and delegate the computation of the projection (Hf) and back-projection ($H^t\delta g$) operators to the accelerator card, often GPUs or FPGAs. In this approach, all data resides on the host processor, and memory transfers are made at each iteration to perform the calculations and retrieve the results via the PCIe bus. For large-scale problems such as tomography, the bandwidth of the PCIe bus is a bottleneck for this algorithm. Fig. 5.10 highlights the block diagram of the iterative reconstruction algorithm with regularization for a CPU-FPGA platform. The strategy employed here is to run the iterative algorithm on the CPU host processor. The FPGA device handles the most computationally expensive operators. As illustrated in the figure, there is a need to transfer data to and from the FPGA device to execute the algorithm correctly. Each step on the block diagram is highlighted, whether it is a computation or a memory transfer. We can see that only the two most expensive operations are accelerated on FPGAs, and the CPU handles everything else. CPU operations can also be accelerated on FPGA, even if they are not the most crucial ones, such as regularization or gradient calculation. In addition to these two bottlenecks (memory transfer and iterative algorithms run on the CPU), there

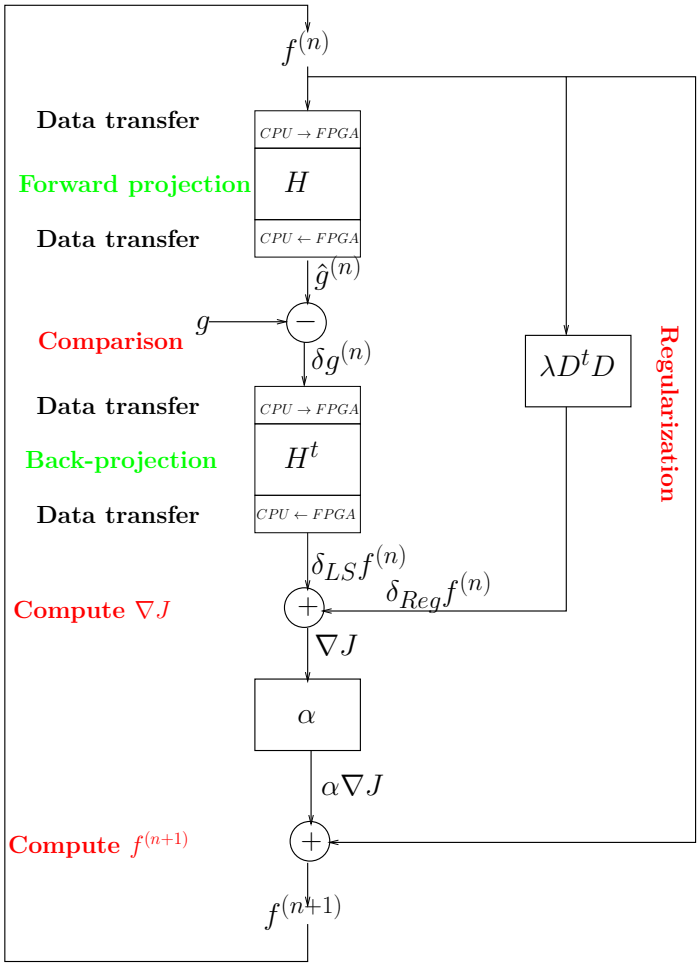


Figure 5.10 – Block diagram of iterative image reconstruction using CPU and FPGA. Data transfers (black), blocks run on CPU (red) and FPGA (green) are highlighted.

is another issue related to FPGAs partial reconfiguration time during execution.

Indeed, the FPGA being a reconfigurable circuit, it is necessary to reconfigure it with the bitstream during the runtime of the host processor program. There are two possible scenarios for projection and back-projection: the two operators are synthesized in a single bitstream, or each operator is synthesized in a separate bitstream. Each case has its advantages and disadvantages.

5.6.2.1 Separate bitstream scenario

The projector and the back-projector kernels are compiled and synthesized separately in this case. Therefore, during the iterative algorithm, it will be necessary at each iteration to configure the FPGA for forward projection and do the same for back-projection until the algorithm's convergence. The FPGA reconfiguration time is of the order of a few seconds as we observed on Intel Arria 10 and Stratix 10

devices. Although OpenCL allows partial reconfiguration (about 5 seconds) using the PCIe bus instead of JTAG, the reconfiguration time is still significant compared to general-purpose processors. However, this iterative reconfiguration strongly impacts the overall reconstruction time and will be challenging to accept in the clinical routine. This makes FPGAs less competitive with GPUs for computationally intensive applications as the latter do not suffer from this reconfiguration delay nor longer compilation times.

It is possible to use two FPGAs in the system so that each operator resides on a separate FPGA. Each FPGA will be configured once, and the application is no longer penalized by reconfiguration time. However, even with two FPGAs, it is only possible to use one FPGA at a time because the sequential aspect of the algorithm. Therefore, both FPGA boards are not fully utilized, resulting in low efficiency.

5.6.2.2 Single bitstream scenario

The other scenario consists of synthesizing the kernels in the same bitstream for the FPGA. In this case, we have to reconfigure the FPGA only once at the beginning of the iterative algorithm. As long as the bitstream on the board does not change, there is no need to reconfigure it. The disadvantage of this approach is that the resources of the FPGA are shared between the two operators, which could cause a loss of performance due to the low level of parallelism for each operator. Furthermore, only half of the hardware resources will be active at any time due to the sequential launch of the operators in the iterative algorithm. This dramatically reduces the efficiency of the FPGA and results in poor performance. The scenario is similar to the case of separate bitstreams with two FPGAs. The efficiency of the pipeline is strongly impacted.

5.6.3 Full FPGA strategy

In order to overcome the aforementioned issues, it is possible to launch the full iterative algorithm on FPGA instead of the host processor. Therefore, all the steps at each iteration are performed on FPGA device. The reconfiguration is performed only one time in this case whether using a single or multiple FPGAs device for the reconstruction. This allows to avoid the multiple data transfer between the host and the device through PCIe as is the case when the gradient descent is performed on CPU. This also means that all data reside on the FPGA global memory and not the host memory. The least square regularization $\delta_{reg} f^{(n)}$, the difference $\delta g^{(n)}$, the scaling by α and the $f^{(n)}$ computations are now performed on the FPGA. All these computations can benefit from pipelining and parallelism on FPGA. Furthermore, we take advantage of our convolution operator optimized on FPGA to perform the regularization step because this convolution is performed with small mask sizes. Our FPGA-based convolution presented in Section 4.4.2 performs well for small mask sizes thanks to the memory optimization.

However, having the whole iterative loop on the FPGA device does not nec-

essarily solve the problem of sub-optimal use of the FPGA. Indeed, despite the projection and back-projection operators being both on the same bitstream, their execution is purely sequential because of the iterative nature of the optimization algorithm. This iterative nature makes it impossible to take advantage of pipelining on FPGA, which is a strong potential of this architecture. Nevertheless, the pipelining is strongly exploited within each step of the optimization algorithm.

5.7 Conclusion

This chapter presents a hardware acceleration of the back-projection algorithm for CT reconstruction using FPGA local memory efficiently. A fully OpenCL-based custom pipeline architecture using memory prefetching to reduce global memory transactions is designed to accelerate the algorithm. The prefetching of the projection data into the local memory, allowed by the offline memory access analysis, permits us to leverage the global memory bandwidth. Our architecture performs better throughput based on an efficient pipeline with no stall on Intel FPGA devices for the back-projection algorithm. Furthermore, we present a multikernel approach to maximize the Stratix 10 design throughput and operating frequency. We have ensured the bandwidth follow-up by the prefetcher module to keep up occupying the compute units as much as possible. The systematic use of the Berkeley roofline model highlighted the optimization steps in our development. We achieved 0.6× and 2.1× throughput against our embedded GPU implementation on Arria 10 and Stratix 10, respectively. FPGAs could be more competitive against GPUs by reducing the process size and gaining computational power and energy efficiency. The OpenCL designs have been compared to the new oneAPI. We notice that the OpenCL tool offers better performance than oneAPI. Our design with OpenCL outperforms the related CT reconstructions using Vivado HLS on FPGA.

Chapter 6

Conclusion and Perspectives

6.1 Conclusion

This thesis deals with the exploration of new high-level synthesis tools on FPGA boards for tomographic reconstruction and radioastronomy. The enormous resurgence in the internal technology of FPGAs has positioned them as a potential candidate for resolving large-scale problems more quickly. Added to this architectural improvement is the emergence of mature HLS tools, which facilitate custom architecture synthesis over HDL languages. Although HLS tools allow development time to be considerably reduced while offering higher productivity, significant effort is still needed to exploit FPGAs to their full potential. Indeed, a thorough knowledge of the FPGA architecture and programming paradigm is required to assist HLS compilers in producing efficient designs. In this thesis, we present several advanced optimizations to leverage FPGA devices. Such FPGA-aware optimizations are essential for compilers to synthesize an optimal design. Although the technical designs of HLS tools are improving, the compilers still struggle to exploit some optimizations due to their inability to integrate the underlying architecture's specificities, as is the case with standard compilers. The designer must therefore assist the compiler in analyzing and synthesizing the algorithm.

Chapter 2 and 3 showed that a direct implementation of compute-intensive algorithms on FPGAs does not guarantee a significant speedup even by applying OpenCL optimizations. The OpenCL results do not reach the performance level of HDL designs in terms of throughput and efficiency. The hardware resources are not efficiently exploited. The algorithms should extensively exploit the FPGA BRAM to overcome the memory bottleneck and take advantage of multiple DSP slices in order to achieve coarse grained parallelism. The need of an algorithm-architecture co-design approach for FPGAs in order to accelerate memory-bound algorithms has been highlighted. Therefore, we proposed a custom architecture of the 3D back-projector based on a customized memory access strategy to reach a full computational throughput.

The proposed architecture derives from a specific methodology in order to ex-

tensively exploit the FPGA's BRAM memory. This methodology is based on elaborating a suitable memory access strategy for a memory-bound algorithm, a roofline analysis to guide the design steps, and the design scalability. The proposed architecture significantly improved the overall performance of the 3D back-projection algorithm. We notice that the scaling up of the design to a higher-end FPGA device is not a trivial task in order to preserve pipeline efficiency. The scaling up from Arria 10 device to Stratix 10 has been studied to balance the design's computational units and the prefetcher modules. The scalability study allows to preserve the pipeline efficiency and improves the design's overall performance.

The performance gap between FPGAs through HLS tools and GPUs is still wide. GPUs are now one order of magnitude faster than FPGA with HLS tools, as was the case 15 years ago [Gac 2008]. Nevertheless, thanks to HLS tools, the development time of FPGAs has been considerably reduced compared to HDL design time. There is hope for FPGAs regarding the design power consumption. In our study, our architecture on FPGA consumes as much energy as the GPU implementation. However, FPGAs are still better than GPUs in terms of worst-case power consumption. Furthermore, considering the technology gap between the two architectures, there is a large room for improvement in the transistor process for FPGA.

HLS tools like OpenCL and oneAPI are addressed to software developers, according to Intel. However, these tools are still not easy-to-use compared to other high-level languages. The use of these tools also requires a deep knowledge of FPGA architectures and their programming models in order to guarantee performance. FPGA vendors are expected to make these tools more user-friendly and improve their static analysis of algorithms to generate more efficient pipelines. These tools may include several research insights, such as the work of [Derrien 2020] to enhance their efficiency. In order to substitute OpenCL with oneAPI, as Intel wants, the oneAPI tool must be further enhanced and offer the same level of performance as the OpenCL tool.

6.2 Perspectives

Several research perspectives can be followed from work presented in this manuscript in order to better position FPGAs for acceleration in HPC applications. Firstly, the main vendors can make improvements to the cards for process size and the tools for their ease of use and efficiency in generating designs. Secondly, developers must properly select the algorithms used for FPGA acceleration and be aware of the specificities of FPGA architecture in their design approach.

There is much hope for FPGA technology because there is still room for improvement in its architecture. In order for FPGAs to catch up with GPUs, improvements are needed in both internal architecture and synthesis tools. For example, the global memory bandwidth of FPGAs is much lower than that of graphics processors, which constitutes a colossal design gap. However, the latest FPGAs are equipped with

High-Bandwidth Memory 2 (HBM2) memory (already available on GPU), which could give this technology another boost in the acceleration of HPC applications. Intel Stratix 10 MX and Xilinx Alveo U50 both contain an FPGA with two HBM2 memories that reach a theoretical memory bandwidth of 410GB/s and 460GB/s, respectively. The achievable bandwidth of these boards is close to Nvidia GPU with HBM2, which is about 650GB/s. Hence, FPGA-based HBM2 has been evaluated for several memory-bound applications using HLS tools in [Choi 2020]. Regarding computational resources, FPGAs are also struggling to compete despite the ever-increasing number of DSPs. This is due to CUDA cores' high operating frequency and more specific units on GPUs, such as tensor cores. Regardless, the new upcoming Intel Agilex device based on HBM2 is supposed to offer up to 40% higher performance and 40% lower power than its predecessor Stratix 10 family.

FPGAs struggle to compete with GPUs for regular algorithms with fewer controls and conditional branches. However, algorithms with several control and branching could benefit from the high flexibility of the FPGA to design a custom pipeline that best fits the algorithm. It is then necessary to identify the algorithms that are not suitable for GPUs in order to implement them on FPGAs by designing a custom architecture. The methodology adopted in this algorithm-architecture co-design approach can be applied to ray-driven projectors allowing them to benefit from the same acceleration as the voxel-driven back-projector. Indeed, several adjacent rays traverse the same voxels in the 3D volume. By identifying a set of adjacent rays, their computation is performed at the local memory level by storing the voxels traversed along the rays in BRAM memory. The matched separable footprint pair could also be accelerated on FPGA. This SF pair contains several conditional branching making it non-suitable for GPU acceleration. In addition, using a matched pair allows to share the same FPGA architecture for the projector and back-projector since they require the same computations but on different data. Also, the parallelization of the iterative loop to overcome the various concerns has been highlighted in Chapter 5. An important axis of research would be to explore this parallelization further to alleviate the concern of partial reconfiguration time of the FPGA or its sub-optimal use. This issue is avoided for a matched pair because the same architecture is used for forward and back-projection resulting in full pipeline efficiency. However, a pipelining mechanism can be elaborated for the optimization algorithm to efficiently use the FPGA for the acceleration of unmatched pair. In this case, the pipelining can be appropriate for both single bitstream or two FPGAs cases for forward and back-projection.

Exploring a multi-FPGA design, especially in the radioastronomy use case, is also essential. FPGAs are well established in the correlator in the SKA pipeline. However, the imaging system requires more computational power and is dominated by the GPU. Multi-FPGA platforms can show a considerable advantage in the SKA context for dataflow processing for several reasons. FPGAs can receive data through multiple peripherals other than the PCIe bus, such as Ethernet. Also, the inter-FPGA communication using channels or pipes results in more efficiency because there is no host processor intervention. Finally, the SKA project's energy constraints

could be met by the FPGAs due to their low worst-case power consumption.

The interest of using profiling tools such as the roofline model has been highlighted in this study. The roofline model is generated automatically for general-purpose processors. The dynamic profiler of the OpenCL compiler is not mature enough to produce the roofline on FPGA. Hence, this work performs the roofline analysis manually thanks to our offline algorithm analysis. The automatic generation of the roofline model on FPGA using HLS tools could be more appropriate and save valuable time in the design process. This model could also be further extended to better suit FPGAs by considering different memory models (global, constant, local), data representation (float-, fixed-point, custom precision), or a peak performance evaluation that includes DSPs, LUTs, etc.

Scientific contributions

Journal papers

- **D. Diakite**, N. Gac, **X-ray Tomography Reconstruction Accelerated on FPGA through High-Level Synthesis Tools**, *IEEE Transactions on Biomedical Circuits and Systems (Under review)*

International conferences

- **D. Diakite**, N. Gac, M. Martelli, **OpenCL FPGA Optimization guided by memory accesses and roofline model analysis applied to tomography acceleration**, *31st International Conference on Field- Programmable Logic and Applications (FPL), Aug 2021, Dresden, Germany*

- **D. Diakite**, M. Martelli, N. Gac, **An OpenCL pipeline implementation on Intel FPGA for 3D backprojection**, *6th International Conference on Image Formation in X-Ray Computed Tomography, Aug 2020, Regensburg, Germany*

National conferences

- **D. Diakite**, N. Gac, **FPGA implementation of 2D Convolution using OneAPI and OpenCL**, *16ème Colloque National du GDR SOC2 2022, Strasbourg, France*

- **D. Diakite**, N. Gac, **Premiers résultats de comparaison des outils oneAPI et OpenCL pour la convolution 2D sur FPGA**, *28ème Colloque Francophone de Traitement du Signal et des Images GRETSI 2022, Nancy, France*

- **D. Diakite**, N. Gac, **Memory prefetching for tomography acceleration on FPGAs through HLS tools**, *15ème Colloque National du GDR SOC2 2021, Rennes, France*

Software

- N. Gac, **D. Diakite**, A. Djafari, TomoGPI - Logiciel de reconstruction en tomographie CT: https://github.com/nicolasgac/tomoGPI

Résumé en Français

Les architectures FPGA ont été pendant longtemps utilisé à travers les langages de description matérielle (HDL) comme le VHDL ou le Verilog. L'utilisation de ces langages HDL a fait que les FPGAs étaient accessible uniquement par les développeurs matérielles ce qui a contribué à la non-consideration des FPGAs dans diverses applications de calcul intensif. L'apparition des nouveaux outils de synthèse de haut niveau (HLS) pour FPGA a été un facteur majeur pour leur prise en compte dans des applications complexes. Ces outils HLS permettent de réduire considérablement le temps de développement des FPGAs par rapport à l'utilisation des langages HDL. Cependant, l'exploitation du plein potentiel de ces architectures à travers les outils HLS a toujours été une préoccupation majeure. Par conséquent, une approche d'adéquation algorithme-architecture est nécessaire pour mieux tirer parti de ces architectures reconfigurables. Les outils HLS fournissent les directives nécessaires pour implémenter efficacement les applications et ainsi produire un pipeline efficace. Toutefois l'efficacité de ces pipelines n'atteint pas encore celle obtenue par les langages HDL, mais leur temps de développement est drastiquement réduit. Cette thèse vise à explorer les méthodologies d'accélération des algorithmes de problèmes inverses mal posés sur les architectures FPGA grâce à de nouveaux outils HLS appliqués à la reconstruction tomographique et à la radioastronomie.

La tomographie à rayon X vise à acquérir la densité interne f d'objets 3D de mesures externes g appelé sinogramme. Un objet (volume 3D) est placé entre une source de rayons X et un plan de détecteur. La reconstruction du volume est un problème inverse souvent mal posé au sens de Hadamard. La matrice système étant de taille très grande dont le stockage est non envisageable, les coefficients de cette matrice sont approchées à la volée par le projecteur et le rétroprojecteur. Les méthodes itératives de reconstruction sont utilisées pour résoudre ce problème linéaire en effectuant plusieurs calculs des opérateurs de projection et de rétroprojection par la minimisation de l'erreur quadratique en utilisant l'algorithme d'optimisation de descente de gradient. La reconstruction par des méthodes itératives prend plusieurs minutes à plusieurs heures, selon la littérature. Pour réduire le temps de calcul de ces routines, des accélérateurs matériels sont requis. Dans la communauté de calcul haute performance, les GPUs ont été l'architecture préférée de la dernière décennie

en raison de leur architecture massivement parallèle et leur facilité d'utilisation via le langage de programmation parallèle CUDA. Notre objectif est de mettre en évidence l'utilisation des FPGA en utilisant les outils de synthèse de haut niveau. Les architectures de FPGA peuvent être une alternative au GPU grâce à leur dernière technologie de DSP, leur efficacité cyclique du pipeline et de leur faible consommation énergétique.

Dans le passé, les FPGAs pour la tomographie à rayon X étaient configurés en utilisant les langages HDL, l'objectif étant de construire un pipeline et d'effectuer une mise à jour des voxels à chaque cycle d'horloge. Le principal problème de l'implémentation HDL est le temps de développement très long, même si les résultats étaient significatifs en termes de temps d'exécution. Plus récemment, de nouvelles approches basées sur des outils HLS pour FPGA ont été développées, afin d'obtenir de meilleurs temps de reconstruction et une meilleure efficacité énergétique tout en réduisant le temps de développement. Cette thèse présente tout d'abord une vue d'ensemble de ces outils HLS avant de se focaliser sur les outils d'Intel. Nous présentons des optimisations basiques et avancées dont le compilateur peut bénéficier du développeur pour assurer une efficacité du pipeline généré.

Nous présentons la méthodologie d'adéquation algorithme-architecture que nous avons adoptée pour la conception d'une architecture sur-mesure sur FPGA. Cette méthodologie est basée sur l'élaboration d'une stratégie d'accès à la mémoire appropriée pour un algorithme limité par la mémoire (memory-bound). Une analyse hors ligne de l'algorithme permet d'augmenter son intensité arithmétique resultant à un allègement substantiel de la bande passante de la mémoire. Nous exploitons le potentiel des FPGAs à exprimer le pipelining et le parallelisme de données pour atteindre des performances meilleures. Cette étude tire également parti de la mémoire sur puce des FPGAs pour stocker des données temporairement et réduire le nombre d'accès à la mémoire externe. Les accès à la mémoire locale sont complètement stall-free afin de ne pas pénaliser l'efficacité du pipeline. Cette thèse traite également la mise à l'échelle de l'algorithme sur des FPGAs haut de gamme dans le but d'assurer la portabilité des performances et préserver l'efficacité du pipeline.

Cette méthodologie est appliquée à l'opérateur de rétroprojecteur 3D utilisé en reconstruction tomographique. L'architecture sur-mesure de ce rétroprojecteur tire profit d'une stratégie d'accès à la mémoire bien adaptée. En effet, cette stratégie d'accès mémoire permet d'exploiter le potentiel de réutilisation des données de cet algorithme, augmentant ainsi l'intensité arithmétique de l'algorithme. Les résultats expérimentaux montrent que les FPGAs atteignent des performances élevées en terme de temps d'exécution et d'efficacité du pipeline. Une etude comparative a été effectué pour évaluer nos designs avec d'autres travaux sur FPGA basé sur les outils HLS. Les résultats sont également comparé aux implementations sur des GPUs HPC et embarqué. Nous discutons également des différentes stratégies de parallélisation de l'algorithme d'optimisation itératif avec régularisation des moindres carrés sur FPGA. Les cas d'études de resolution de problèmes inverses sont présentés notamment en reconstruction tomographique et en radioastronomie (e.g., déconvolution). En effet l'algorithme d'optimisation peut être lancer complètement

sur FPGA au lieu d'avoir des étapes de calcul sur le processeur hôte. Une telle stratégie permet d'éviter les transferts mémoire coûteux entre l'hôte et la carte accélératrice. En revanche ceci peut soulever d'autres problématiques de parallélisation qui sont discutées dans ce manuscrit.

Bibliography

[Agi 1993] I. Agi, P.J. Hurst and K.W. Current. *An image processing IC for back-projection and spatial histogramming in a pipelined array.* IEEE Journal of Solid-State Circuits, vol. 28, no. 3, pages 210–221, Mar 1993.

[Baek 2010] Jongduk Baek and Norbert J. Pelc. *The noise power spectrum in CT with direct fan beam reconstruction.* Medical Physics, vol. 37, no. 5, pages 2074–2081, 2010.

[Barbacci 1973] Mario R Barbacci and Daniel P Siewiorek. *Automated exploration of the design space for register transfer (RT) systems.* In Proceedings of the 1st annual symposium on Computer architecture, pages 101–106, 1973.

[Basu 2006] Samit Basu and Bruno De Man. *Branchless distance driven projection and backprojection.* In Computational Imaging IV, volume 6065, pages 274–281. SPIE, Feb 2006.

[Biguri 2016] Ander Biguri, Manjit Dosanjh, Steven Hancock and Manuchehr Soleimani. *TIGRE: a MATLAB-GPU toolbox for CBCT image reconstruction.* Biomedical Physics & Engineering Express, vol. 2, no. 5, sep 2016.

[Burger 2016] Wilhelm Burger and Mark J Burge. Digital image processing: an algorithmic introduction using java. Springer, 2016.

[Calypto 2004] Calypto. *Calypto Design Systems. Catapult: Product Family Overview.* 2004.

[Canis 2011] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown and Tomasz Czajkowski. *LegUp: high-level synthesis for FPGA-based processor/accelerator systems.* In Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '11. ACM Press, 2011.

[Chapdelaine 2018] Camille Chapdelaine, Nicolas Gac, Ali-Mohammad Djafari and Estelle Parra-Denis. *New GPU implementation of Separable Footprint (SF)*

Projector and Backprojector : first results. In The Fifth International Conference on Image Formation in X-Ray Computed Tomography, pages 314–317, Salt Lake City, United States, May 2018.

[Chen 2012a] J. Chen, J. Cong, L. A. Vese, J. Villasenor, M. Yan and Y. Zou. *A Hybrid Architecture for Compressive Sensing 3-D CT Reconstruction.* IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 2, no. 3, pages 616–625, Sep 2012.

[Chen 2012b] Jianwen Chen, Jason Cong, Ming Yan and Yi Zou. *FPGA-accelerated 3D reconstruction using compressive sensing.* In Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays, FPGA '12, pages 163–166, New York, NY, USA, Feb 2012. Association for Computing Machinery.

[Choi 2016] Young-kyu Choi and Jason Cong. *Acceleration of EM-Based 3D CT Reconstruction Using FPGA.* IEEE Transactions on Biomedical Circuits and Systems, vol. 10, no. 3, pages 754–767, Jun 2016.

[Choi 2020] Young-kyu Choi, Yuze Chi, Jie Wang, Licheng Guo and Jason Cong. *When HLS Meets FPGA HBM: Benchmarking and Bandwidth Optimization.* arXiv:2010.06075 [cs], Oct 2020.

[Chou 2011] Cheng-Ying Chou and Yi-Yen Chuo. *A fast forward projection using multithreads for multirays on GPUs in medical image reconstruction.* Journal of Medical Physics Research and Practice, 2011.

[Cilardo 2020] Alessandro Cilardo. *Evaluating Reconfigurable Hardware for Accelerating Industrial CT.* In 2020 IEEE 7th International Conference on Industrial Engineering and Applications (ICIEA), pages 93–97, 2020.

[Clark 1980] BG Clark. *An efficient implementation of the algorithm'CLEAN'.* Astronomy and Astrophysics, vol. 89, 1980.

[Corda 2022] Stefano Corda, Bram Veenboer, Ahsan Javed Awan, John W. Romein, Roel Jordans, Akash Kumar, Albert-Jan Boonstra and Henk Corporaal. *Reduced-Precision Acceleration of Radio-Astronomical Imaging on Reconfigurable Hardware.* IEEE Access, vol. 10, pages 22819–22843, 2022.

[Coric 2002] Srdjan Coric, Miriam Leeser, Eric Miller and Marc Trepanier. *Parallel-beam backprojection: an FPGA implementation optimized for medical imaging.* In Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays, FPGA '02, pages 217–226, New York, NY, USA, Feb 2002. Association for Computing Machinery.

[Cornwell 2008] Tim J Cornwell. *Multiscale CLEAN deconvolution of radio synthesis images.* IEEE Journal of selected topics in signal processing, vol. 2, no. 5, pages 793–801, 2008.

[Coussy 2008] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn and Eric Martin. *GAUT: A high-level synthesis tool for DSP applications.* In High-Level Synthesis, pages 147–169. Springer, 2008.

[da Silva 2013] Bruno da Silva, An Braeken, Erik H. D'Hollander and Abdellah Touhafi. *Performance Modeling for FPGAs: Extending the Roofline Model with High-Level Synthesis Tools*, Dec 2013.

[De Man 2002] B. De Man and S. Basu. *Distance-driven projection and backprojection.* In 2002 IEEE Nuclear Science Symposium Conference Record, volume 3, pages 1477–1480, Nov 2002.

[De Man 2004] Bruno De Man and Samit Basu. *Distance-driven projection and backprojection in three dimensions.* Physics in Medicine and Biology, vol. 49, no. 11, pages 2463–2475, Jun 2004.

[Dennard 1974] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous and Andre R LeBlanc. *Design of ion-implanted MOSFET's with very small physical dimensions.* IEEE Journal of solid-state circuits, vol. 9, no. 5, pages 256–268, 1974.

[Derrien 2020] Steven Derrien, Thibaut Marty, Simon Rokicki and Tomofumi Yuki. *Toward Speculative Loop Pipelining for High-Level Synthesis.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 11, pages 4229 – 4239, 2020.

[Diakite 2020] Daouda Diakite, Maxime Martelli and Nicolas Gac. *An OpenCL pipeline implementation on Intel FPGA for 3D backprojection.* In 6th International Conference on Image Formation in X-Ray Computed Tomography, 2020.

[Diakite 2021] Daouda Diakite, Nicolas Gac and Maxime Martelli. *OpenCL FPGA Optimization guided by memory accesses and roofline model analysis applied to tomography acceleration.* In 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), pages 109–114, 2021. ISSN: 1946-1488.

[Dittmann 2016] Jonas Dittmann. *Efficient ray tracing on 3D regular grids for fast generation of digitally reconstructed radiographs in iterative tomographic reconstruction techniques.* arXiv:1609.00958 [physics], Sep 2016.

[Dittmann 2017] Jonas Dittmann, Randolf Hanke and EZRT Fraunhofer. *Simple and efficient raycasting on modern GPU's read-and-write memory for fast forward projections in iterative CBCT reconstruction.* In The 14th International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine, 2017.

[Feldkamp 1984] L. A. Feldkamp, L. C. Davis and J. W. Kress. *Practical cone-beam algorithm.* JOSA A, vol. 1, no. 6, pages 612–619, Jun 1984.

[Fessler 2000] Jeffrey A Fessler, M Sonka and J Michael Fitzpatrick. *Statistical image reconstruction methods for transmission tomography.* Handbook of medical imaging, vol. 2, pages 1–70, 2000.

[Gac 2008] Nicolas Gac, Stéphane Mancini, Michel Desvignes and Dominique Houzet. *High Speed 3D Tomography on CPU, GPU, and FPGA.* EURASIP Emb Sys, 2008.

[Gac 2014] Nicolas Gac and Ali-Mohammad Djafari. *opgpuTomoGPI - Ref CNRS du pré-dépôt APP 11562-01 (num IDDN prochainement disponible)*, Aug 2014.

[Gao 2012] Hao Gao. *Fast parallel algorithms for the x-ray transform and its adjoint.* Medical Physics, vol. 39, no. 11, pages 7110–7120, 2012.

[Georgin 2019] Nicolas Georgin, Camille Chapdelaine, Nicolas Gac, Ali Mohammad-Djafari and Estelle Parra. *Multi-streaming and multi-GPU optimization for a matched pair of Projector and Backprojector.* In 15th International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine, volume 11072. International Society for Optics and Photonics, May 2019.

[Geyer 2015] Lucas L. Geyer, U. Joseph Schoepf, Felix G. Meinel, John W. Nance, Gorka Bastarrika, Jonathon A. Leipsic, Narinder S. Paul, Marco Rengo, Andrea Laghi and Carlo N. De Cecco. *State of the Art: Iterative CT Reconstruction Techniques.* Radiology, vol. 276, no. 2, pages 339–357, Jul 2015.

[Gordon 1970] Richard Gordon, Robert Bender and Gabor T Herman. *Algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and X-ray photography.* Journal of theoretical Biology, vol. 29, no. 3, pages 471–481, 1970.

[Gupta 2008] Rajesh Gupta and Forrest Brewer. *High-level synthesis: A retrospective.* In High-level synthesis, pages 13–28. Springer, 2008.

[Han 1999] G. Han, Z. Liang and J. You. *A fast ray-tracing technique for TCT and ECT studies.* In 1999 IEEE Nuclear Science Symposium. Conference Record. 1999 Nuclear Science Symposium and Medical Imaging Conference (Cat. No.99CH37019), volume 3, pages 1515–1518 vol.3, Oct 1999. ISSN: 1082-3654.

[Herman 1980] Gabor T Herman. *Image reconstruction from projections.* Academic Press, 1980.

[Högbom 1974] JA Högbom. *Aperture synthesis with a non-regular distribution of interferometer baselines*. Astronomy and Astrophysics Supplement Series, vol. 15, 1974.

[Huaxia Zhao 2003] Huaxia Zhao and A. J. Reader. *Fast ray-tracing technique to calculate line integral paths in voxel arrays*. In 2003 IEEE Nuclear Science Symposium. Conference Record (IEEE Cat. No.03CH37515), volume 4, pages 2808–2812, Oct 2003.

[Idier 2013] Jérôme Idier. Bayesian approach to inverse problems. John Wiley & Sons, 2013.

[Intel Corporation 2019a] Intel Corporation. *Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide*. https://www.intel.com/content/www/us/en/docs/programmable/683521/19-1/introduction.html, 2019.

[Intel Corporation 2019b] Intel Corporation. *Intel FPGA SDK for OpenCL Pro Edition: Programming Guide*. https://www.intel.com/content/www/us/en/docs/programmable/683846/19-1/introduction.html, 2019.

[Intel Corporation 2021a] Intel Corporation. *Intel Hyperflex Architecture High-Performance Design Handbook*. https://www.intel.com/content/www/us/en/docs/programmable/683353/21-3/fpga-architecture-introduction.html, 2021.

[Intel Corporation 2021b] Intel Corporation. *Intel oneAPI Programming Guide*. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html, 2021.

[Intel Corporation 2021c] Intel Corporation. *Intel Stratix 10 GX/SX Device Overview*. https://www.intel.com/content/www/us/en/docs/programmable/683729/current/gx-sx-device-overview.html, 2021.

[Jacobs 1998] Filip Jacobs, Erik Sundermann, Bjorn De Sutter, Mark Christiaens and Ignace Lemahieu. *A Fast Algorithm to Calculate the Exact Radiological Path through a Pixel or Voxel Space*. Journal of computing and information technology, vol. 6, no. 1, pages 89–94, Mar 1998.

[Jia 2014] Xun Jia, Peter Ziegenhein and Steve B. Jiang. *GPU-based high-performance computing for radiation therapy*. Physics in Medicine and Biology, vol. 59, no. 4, pages R151–182, Feb 2014.

[Joseph 1982] Peter M Joseph. *An improved algorithm for reprojecting rays through pixel images*. IEEE transactions on medical imaging, vol. 1, no. 3, pages 192–196, 1982.

[Kachelriess 2006] Marc Kachelriess, Michael Knaup and Olivier Bockenbach. *Hyperfast Perspective Cone–Beam Backprojection.* In 2006 IEEE Nuclear Science Symposium Conference Record, volume 3, pages 1679–1683, Oct 2006. ISSN: 1082-3654.

[Keryell 2020] Ronan Keryell, Maria Rovatsou and Lee Howes. *SYCL specification generic heterogeneous computing for modern C++.* page 274, 2020.

[Khronos 2009] OpenCL Working Group Khronos. *The OpenCL Specification: Version 1.2.* https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf., 2009.

[Kim 2012] Jung Kuk Kim, Jeffrey A. Fessler and Zhengya Zhang. *Forward-Projection Architecture for Fast Iterative Image Reconstruction in CT.* IEEE Trans Sign Process, 2012.

[Knaup 2008] Michael Knaup, Sven Steckmann and Marc Kachelriess. *GPU-based parallel-beam and cone-beam forward- and backprojection using CUDA.* In 2008 IEEE Nuclear Science Symposium Conference Record, pages 5153–5157, Oct 2008. ISSN: 1082-3654.

[Koch 2016] Dirk Koch, Frank Hannig and Daniel Ziener. Fpgas for software programmers. Springer, 2016.

[Langhammer 2008] Martin Langhammer. *Floating point datapath synthesis for FPGAs.* In 2008 International Conference on Field Programmable Logic and Applications, pages 355–360, Sep 2008. ISSN: 1946-1488.

[Liu 2017] Rui Liu, Lin Fu, Bruno De Man and Hengyong Yu. *GPU-Based Branchless Distance-Driven Projection and Backprojection.* IEEE Transactions on Computational Imaging, vol. 3, no. 4, pages 617–632, Dec 2017.

[Liu 2018] Junyi Liu, John Wickerson, Samuel Bayliss and George A. Constantinides. *Polyhedral-Based Dynamic Loop Pipelining for High-Level Synthesis.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 37, no. 9, pages 1802–1815, Sep 2018.

[Long 2010] Y. Long, J. A. Fessler and J. M. Balter. *3D Forward and Back-Projection for X-Ray CT Using Separable Footprints.* IEEE Transactions on Medical Imaging, vol. 29, no. 11, pages 1839–1850, Nov 2010.

[Martelli 2018] M. Martelli, N. Gac, A. Mérigot and C. Enderli. *3D Tomography back-projection parallelization on Intel FPGAs using OpenCL.* Journal of Signal Processing Systems, 2018.

[Martelli 2019] Maxime Martelli. *Approche haut niveau pour l'accélération d'algorithmes sur des architectures hétérogènes CPU/GPU/FPGA. Application à la qualification des radars et des systèmes d'écoute électromagnétique.* phdthesis, Université Paris Saclay (COmUE), Dec 2019.

[Martin 2009] Grant Martin and Gary Smith. *High-level synthesis: Past, present, and future*. IEEE Design & Test of Computers, vol. 26, no. 4, pages 18–25, 2009.

[Mitra 2017] Ayan Mitra, David G. Politte, Bruce R. Whiting, Jeffrey F. Williamson and Joseph A. O'Sullivan. *Multi-GPU Acceleration of Branchless Distance Driven Projection and Backprojection for Clinical Helical CT*. The Journal of imaging science and technology, vol. 61, no. 1, 2017.

[Momey 2015] Fabien Momey, Loïc Denis, Catherine Burnier, Eric Thiébaut, Jean-Marie Becker and Laurent Desbat. *Spline driven: high accuracy projectors for tomographic reconstruction from few projections*. IEEE Transactions on Image Processing, vol. 24, no. 12, pages 4715–4725, 2015.

[Moore 1965] Gordon E Moore*et al. Cramming more components onto integrated circuits*, 1965.

[Mueller 2007] Klaus Mueller, Fang Xu and Neophytos Neophytou. *Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography?* In Computational Imaging V, volume 6498, pages 183–194. SPIE, Feb 2007.

[Nane 2012] Razvan Nane, Vlad-Mihai Sima, Bryan Olivier, Roel Meeuws, Yana Yankova and Koen Bertels. *DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler*. In 22nd International Conference on Field Programmable Logic and Applications (FPL), pages 619–622. IEEE, 2012.

[Nane 2015] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi*et al. A survey and evaluation of FPGA high-level synthesis tools*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 35, no. 10, pages 1591–1604, 2015.

[Nelson 2011] Rendon C. Nelson, Sebastian Feuerlein and Daniel T. Boll. *New iterative reconstruction techniques for cardiovascular computed tomography: How do they work, and what are the advantages and disadvantages?* Journal of Cardiovascular Computed Tomography, vol. 5, no. 5, pages 286–292, Sep 2011.

[Nguyen 2015] Van-Giang Nguyen and Soo-Jin Lee. *Parallelizing a Matched Pair of Ray-Tracing Projector and Backprojector for Iterative Cone-Beam CT Reconstruction*. IEEE Transactions on Nuclear Science, vol. 62, no. 1, pages 171–181, Feb 2015.

[Numan 2020] Mostafa W Numan, Braden J Phillips, Gavin S Puddy and Katrina Falkner. *Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains*. IEEE Access, vol. 8, pages 174692–174722, 2020.

[Okitsu 2010] Yusuke Okitsu, Fumihiko Ino and Kenichi Hagihara. *High-performance cone beam reconstruction using CUDA compatible GPUs.* Parallel Computing, vol. 36, no. 2, pages 129–141, Feb 2010.

[Palenstijn 2017] Willem Palenstijn, Jeroen Bédorf, Jan Sijbers and Kees Batenburg. *ASTRA toolbox.* ASCI, 2017. open access.

[Pan 2009] Xiaochuan Pan, Emil Y Sidky and Michael Vannier. *Why do commercial CT scanners still employ traditional, filtered back-projection for image reconstruction?* Inverse Problems, vol. 25, no. 12, Dec 2009.

[Pelcat 2016] Maxime Pelcat, Cédric Bourrasset, Luca Maggiani and François Berry. *Design productivity of a high level synthesis compiler versus HDL.* In 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), pages 140–147. IEEE, 2016.

[Pfanner 2011] Florian Pfanner, Michael Knaup and Marc Kachelriess. *High performance parallel backprojection on FPGA.* Jul 2011.

[Pilato 2013] Christian Pilato and Fabrizio Ferrandi. *Bambu: A modular framework for the high level synthesis of memory-intensive applications.* In 2013 23rd International Conference on Field programmable Logic and Applications, pages 1–4. IEEE, 2013.

[Putnam 2008] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, S. Eggers and Andrew Putnam. *CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures.* In Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on, pages 173–178, September 2008.

[Qiao 2021] Linjun Qiao, Guojie Luo, Wentai Zhang and Ming Jiang. *FPGA-accelerated Iterative Reconstruction for Transmission Electron Tomography.* In 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 152–156, May 2021. ISSN: 2576-2621.

[Radon 1917] Johann Radon. *1.1 über die bestimmung von funktionen durch ihre integralwerte längs gewisser mannigfaltigkeiten.* Classic papers in modern diagnostic radiology, vol. 5, 1917.

[Rau 2011] Urvashi Rau and Tim J Cornwell. *A multi-scale multi-frequency deconvolution algorithm for synthesis imaging in radio interferometry.* Astronomy & Astrophysics, vol. 532, 2011.

[Ravi 2019] Murali Ravi, Angu Sewa, Shashidhar T. G. and Siva Sankara Sai Sanagapati. *FPGA as a Hardware Accelerator for Computation Intensive Maximum Likelihood Expectation Maximization Medical Image Reconstruction Algorithm.* IEEE Access, vol. 7, pages 111727–111735, 2019.

[Rit 2014] S Rit, M Vila Oliva, S Brousmiche, R Labarbe, D Sarrut and G C Sharp. *The Reconstruction Toolkit (RTK), an open-source cone-beam CT reconstruction toolkit based on the Insight Toolkit (ITK).* Journal of Physics: Conference Series, vol. 489, no. 1, 2014.

[Rohkohl 2009] C. Rohkohl, B. Keck, H. G. Hofmann and J. Hornegger. *Technical Note: RabbitCT—an open platform for benchmarking 3D cone-beam reconstruction algorithmsa).* Medical Physics, vol. 36, no. 9Part1, pages 3940–3944, 2009.

[Rudin 1992] Leonid I Rudin, Stanley Osher and Emad Fatemi. *Nonlinear total variation based noise removal algorithms.* Physica D: nonlinear phenomena, vol. 60, no. 1-4, pages 259–268, 1992.

[Savanier 2021] Marion Savanier, Cyril Riddell, Yves Trousset, Emilie Chouzenoux and Jean-Christophe Pesquet. *Magnification-driven B-spline interpolation for cone-beam projection and backprojection.* Medical Physics, vol. 48, no. 10, pages 6339–6361, 2021.

[Scherl 2007a] Holger Scherl, Benjamin Keck, Markus Kowarschik and Joachim Hornegger. *Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA).* In 2007 IEEE Nuclear Science Symposium Conference Record, volume 6, pages 4464–4466, Oct 2007. ISSN: 1082-3654.

[Scherl 2007b] Holger Scherl, Mario Koerner, Hannes Hofmann, Wieland Eckert, Markus Kowarschik and Joachim Hornegger. *Implementation of the FDK algorithm for cone-beam CT on the cell broadband engine architecture.* In Medical Imaging 2007: Physics of Medical Imaging, volume 6510, pages 1666–1675. SPIE, 2007.

[Schlifske 2016] Daniel Schlifske and Henry Medeiros. *A fast GPU-based approach to branchless distance-driven projection and back-projection in cone beam CT.* In Medical Imaging 2016: Physics of Medical Imaging, volume 9783, pages 742–749. SPIE, Mar 2016.

[Schwab 1984] FR Schwab. *Relaxing the isoplanatism assumption in self-calibration; applications to low-frequency radio interferometry.* The Astronomical Journal, vol. 89, pages 1076–1081, 1984.

[Segars 2008] William Paul Segars, Mahadevappa Mahesh, Thomas J Beck, Eric C Frey and Benjamin MW Tsui. *Realistic CT simulation using the 4D XCAT phantom.* Medical physics, vol. 35, no. 8, pages 3800–3808, 2008.

[Shata 2019] Kholoud Shata, Marwa K Elteir and Adel A El-Zoghabi. *Optimized implementation of OpenCL kernels on FPGAs.* Journal of Systems Architecture, vol. 97, pages 491–505, 2019.

[Siddon 1985] Robert L Siddon. *Fast calculation of the exact radiological path for a three-dimensional CT array.* Medical physics, 12(2):252–255, Mar 1985.

[Singh 2008] Satnam Singh and David J. Greaves. *Kiwi: Synthesis of FPGA Circuits from Parallel Programs.* In 2008 16th International Symposium on Field-Programmable Custom Computing Machines, pages 3–12, 2008.

[Ström 2016] Henrik Ström. *A Parallel FPGA Implementation of Image Convolution*, 2016.

[Thibault 2007] Jean-Baptiste Thibault, Ken D. Sauer, Charles A. Bouman and Jiang Hsieh. *A three-dimensional statistical approach to improved image quality for multislice helical CT.* Medical Physics, vol. 34, no. 11, pages 4526–4544, 2007.

[Thompson 2014] William Moreau Thompson and William R. B. Lionheart. *GPU Accelerated Structure-Exploiting Matched Forward and Back Projection for Algebraic Iterative Cone Beam CT Reconstruction.* In The Third International Conference on Image Formation in X-Ray Computed Tomography, 22-25 June 2014, Salt Lake City, Utah, USA., 2014.

[Tikhonov 1995] Andrei Nikolaevich Tikhonov, AV Goncharsky, VV Stepanov and Anatoly G Yagola. Numerical methods for the solution of ill-posed problems, volume 328. Springer Science & Business Media, 1995.

[Uguen 2019] Yohann Uguen, Florent De Dinechin, Victor Lezaud and Steven Derrien. *Application-specific arithmetic in high-level synthesis tools.* In ACM Transactions on Architecture and Code Optimization, 2019.

[Van der Tol 2018] Sebastiaan Van der Tol, Bram Veenboer and André R Offringa. *Image Domain Gridding: a fast method for convolutional resampling of visibilities.* Astronomy & Astrophysics, vol. 616, 2018.

[Veenboer 2019] Bram Veenboer and John W. Romein. *Radio-Astronomical Imaging: FPGAs vs GPUs.* In Ramin Yahyapour, editeur, Euro-Par 2019: Parallel Processing, volume 11725, pages 509–521. Springer International Publishing, Cham, 2019.

[Villarreal 2010] Jason Villarreal, Adrian Park, Walid Najjar and Robert Halstead. *Designing Modular Hardware Accelerators in C with ROCCC 2.0.* In 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, pages 127–134, 2010.

[Wang 2002] Zhou Wang and Alan C Bovik. *A universal image quality index.* IEEE signal processing letters, vol. 9, no. 3, pages 81–84, 2002.

[Wen 2020] Shuang Wen and Guojie Luo. *FPGA-accelerated Automatic Alignment for Three-dimensional Tomography.* In 2020 IEEE 28th Annual International

Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 172–176, May 2020. ISSN: 2576-2621.

[Willemink 2013] Martin J. Willemink, Pim A. de Jong, Tim Leiner, Linda M. de Heer, Rutger A. J. Nievelstein, Ricardo P. J. Budde and Arnold M. R. Schilham. *Iterative reconstruction techniques for computed tomography Part 1: Technical principles.* European Radiology, vol. 23, no. 6, pages 1623–1631, Jun 2013.

[Williams 2009] Samuel Williams, Andrew Waterman and David Patterson. *Roofline: an insightful visual performance model for multicore architectures.* Communications of the ACM, vol. 52, no. 4, pages 65–76, Apr 2009.

[Wu 1991] M.A. Wu. *ASIC applications in computed tomography systems.* In [1991] Proceedings Fourth Annual IEEE International ASIC Conference and Exhibit, Sep 1991.

[Wu 2011] Meng Wu and Jeffrey A Fessler. *GPU acceleration of 3D forward and backward projection using separable footprints for X-ray CT image reconstruction.* In Proc. Intl. Mtg. on Fully 3D Image Recon. in Rad. and Nuc. Med, volume 6, 2011.

[Xiao 2011] Kai Xiao, Bo Zhou, Xiaobo Sharon Hu and Danny Ziyi Chen. *Shell : Accelerating Ray Tracing on GPU.* 2011.

[Xiao 2012] Kai Xiao, Danny Z. Chen, X. Sharon Hu and Bo Zhou. *Efficient implementation of the 3D-DDA ray traversal algorithm on GPU and its application in radiation dose calculation.* Medical Physics, vol. 39, no. 12, pages 7619–7625, 2012.

[Xie 2017] Xiaobin Xie, Madison G. McGaffin, Yong Long, Jeffrey A. Fessler, Minhua Wen and James Lin. *Accelerating separable footprint (SF) forward and back projection on GPU.* In Medical Imaging 2017: Physics of Medical Imaging, volume 10132. International Society for Optics and Photonics, Mar 2017.

[Xu 2007] Fang Xu and Klaus Mueller. *Real-time 3D computed tomographic reconstruction using commodity graphics hardware.* Physics in Medicine and Biology, vol. 52, no. 12, pages 3405–3419, May 2007.

[Xu 2010a] Fang Xu. *Fast implementation of iterative reconstruction with exact ray-driven projector on GPUs.* Tsinghua Science and Technology, vol. 15, no. 1, pages 30–35, Feb 2010.

[Xu 2010b] Jimmy Xu, Nikhil Subramanian, Adam Alessio and Scott Hauck. *Impulse C vs. VHDL for Accelerating Tomographic Reconstruction.* In 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, pages 171–174, May 2010.

[Zhang 2020] Wentai Zhang, Linjun Qiao, William Hsu, Yong Cui, Ming Jiang and Guojie Luo. *FPGA Acceleration for 3D Low-Dose Tomographic Reconstruction.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pages 1–1, 2020.

[Zhuang 1994] Weihua Zhuang, Sanjay S Gopal and TJ Hebert. *Numerical evaluation of methods for computing tomographic projections.* IEEE transactions on nuclear science, vol. 41, no. 4, pages 1660–1665, 1994.

[Ziegler 2007] A. Ziegler, Th. Köhler and R. Proksa. *Noise and resolution in images reconstructed with FBP and OSC algorithms for CT.* Medical Physics, vol. 34, no. 2, pages 585–598, 2007.

[Zinsser 2013] Timo Zinsser and Benjamin Keck. *Systematic performance optimization of cone-beam back-projection on the Kepler architecture.* Proceedings of the 12th Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine, pages 225–228, 2013.

[Zohouri 2019] Hamid Reza Zohouri. *High Performance Computing with FPGAs and OpenCL.* arXiv:1810.09773 [cs], Sep 2019.