Automatic Parallelization for Heterogeneous Embedded Systems

Rokiatou Diarra

université PARIS-SACLAY

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

NNT : 2019SACLS485

Thèse de doctorat

Automatic Parallelization for Heterogeneous Embedded Systems

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud

Ecole doctorale n°580 Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : Traitement du Signal et des Images

Thèse présentée et soutenue à Digiteo Labs, Gif-sur-Yvette, le 25/11/2019, par

MME ROKIATOU DIARRA

Composition du Jury :

Dominique HOUZET Professeur, Université Grenoble Alpes (GIPSA-Lab)	Président
Mohamed SHAWKY Professeur, Université de Technologie de Compiègne (COSTECH)	Rapporteur
Bertrand GRANADO Professeur, Sorbonne Université (LIP6)	Rapporteur
Michèle GOUIFFES Maître de Conférences, Université Paris-Sud (LIMSI - CNRS)	Examinateur
Marc DURANTON Chercheur, Commissariat à l'énergie atomique (LIST)	Invité
Alain MÉRIGOT Professeur, Université Paris-Sud (SATIE)	Directeur de thèse
Bastien VINCKE Maître de Conférences, Université Paris-Sud (SATIE)	Co-encadrant de thèse

Abstract

Recent years have seen an increase of heterogeneous architectures combining multi-core CPUs with accelerators such as GPU, FPGA and Intel Xeon Phi. GPU can achieve significant performance for certain categories of application. Nevertheless, achieving this performance with low-level APIs (e.g. CUDA, OpenCL) requires to rewrite the sequential code, to have a good knowledge of GPU architecture, and to apply complex optimizations that are sometimes not portable. On the other hand, directive-based programming models (e.g. OpenACC, OpenMP) offer a high-level abstraction of the underlying hardware, thus simplifying the code maintenance and improving productivity. They allow users to accelerate their sequential codes on GPU by simply inserting directives. OpenACC/OpenMP compilers have the daunting task of applying the necessary optimizations from the user-provided directives and generating efficient codes that take advantage of the GPU architecture. Although the OpenACC / OpenMP compilers are mature and able to apply some optimizations automatically, the generated code may not achieve the expected speedup as the compilers do not have full view of the whole application. Thus, there is generally a significant performance gap between the codes accelerated with OpenACC/OpenMP and those hand-optimized with CUDA/OpenCL.

To help programmers for speeding up efficiently their legacy sequential codes on GPU with directive-based models and broaden OpenMP/OpenACC impact in both academia and industry, several research issues are discussed in this dissertation. We investigated OpenACC and OpenMP programming models and proposed an effective application parallelization methodology with directive-based programming approaches. Our application porting experience revealed that it is insufficient to simply insert OpenMP/OpenACC offloading directives to inform the compiler that a particular code region must be compiled for GPU execution. It is highly essential to combine offloading directives with loop parallelization constructs. Although current compilers are mature and perform several optimizations, the user may provide them more information through loop parallelization constructs clauses in order to get an optimized code. We have also revealed the challenge of choosing good loop schedules. The default loop schedule chosen by the compiler may not produce the best performance, so the user has to manually try different loop schedules to improve the performance. We demonstrate that OpenMP and OpenACC programming models can achieve best performance with lesser programming effort, but OpenMP/OpenACC compilers quickly reach their limit when the offloaded region code is computed/memory bound and contain several nested loops. In such cases, low-level languages may be used. We also discuss pointers aliasing problem in GPU codes and propose two static analysis tools that perform automatically at source level type qualifier insertion and scalar promotion to solve aliasing issues.

Résumé

L'utilisation d'architectures hétérogènes, combinant des processeurs multicœurs avec des accélérateurs tels que les GPU, FPGA et Intel Xeon Phi, a augmenté ces dernières années. Les GPUs peuvent atteindre des performances significatives pour certaines catégories d'applications. Néanmoins, pour atteindre ces performances avec des API de bas niveau comme CUDA et OpenCL, il est nécessaire de réécrire le code séquentiel, de bien connaître l'architecture des GPUs et d'appliquer des optimisations complexes, parfois non portables. D'autre part, les modèles de programmation basés sur des directives (par exemple, OpenACC, OpenMP) offrent une abstraction de haut niveau du matériel sous-jacent, simplifiant ainsi la maintenance du code et améliorant la productivité. Ils permettent aux utilisateurs d'accélérer leurs codes séquentiels sur les GPUs en insérant simplement des directives. Les compilateurs d'OpenACC/OpenMP ont la lourde tâche d'appliquer les optimisations nécessaires à partir des directives fournies par l'utilisateur et de générer des codes exploitant efficacement l'architecture sous-djacente. Bien que les compilateurs d'OpenACC/OpenMP soient matures et puissent appliquer certaines optimisations automatiquement, le code généré peut ne pas atteindre l'accélération prévue, car les compilateurs ne disposent pas d'une vue complète de l'ensemble de l'application. Ainsi, il existe généralement un écart de performance important entre les codes accélérés avec OpenACC/OpenMP et ceux optimisés manuellement avec CUDA/OpenCL.

Afin d'aider les programmeurs à accélérer efficacement leurs codes séquentiels sur GPU avec les modèles basés sur des directives et à élargir l'impact d'OpenMP/OpenACC dans le monde universitaire et industrielle, cette thèse aborde plusieurs problématiques de recherche. Nous avons étudié les modèles de programmation OpenACC et OpenMP et proposé une méthodologie efficace de parallélisation d'applications avec les approches de programmation basées sur des directives. Notre expérience de portage d'applications a révélé qu'il était insuffisant d'insérer simplement des directives de déchargement OpenMP/OpenACC pour informer le compilateur qu'une région de code particulière devait être compilée pour être exécutéé sur la GPU. Il est essentiel de combiner les directives de déchargement avec celles de parallélisation de boucle. Bien que les compilateurs actuels soient matures et effectuent plusieurs optimisations, l'utilisateur peut leur fournir davantage d'informations par le biais des clauses des directives de parallélisation de boucle afin d'obtenir un code mieux optimisé. Nous avons également révélé le défi consistant à choisir le bon nombre de threads devant exécuter une boucle. Le nombre de threads choisi par défaut par le compilateur peut ne pas produire les meilleures performances. L'utilisateur doit donc essayer manuellement différents nombres de threads pour améliorer les performances. Nous démontrons que les modèles de programmation OpenMP et OpenACC peuvent atteindre de meilleures performances avec un

effort de programmation moindre, mais les compilateurs OpenMP/OpenACC atteignent rapidement leur limite lorsque le code de région déchargée a une forte intensité arithmétique, nécessite un nombre très élevé d'accès à la mémoire globale et contient plusieurs boucles imbriquées. Dans de tels cas, des langages de bas niveau doivent être utilisés. Nous discutons également du problème d'alias des pointeurs dans les codes GPU et proposons deux outils d'analyse statiques qui permettent d'insérer automatiquement les qualificateurs de type et le remplacement par scalaire dans le code source.

Dedication

To my late father Oumar DIARRA

To my dear mother Korotimi DIARRA

To my dear brothers and sisters

To Minkoro FOMBA, my best half

This work would not have been possible without your love and your unconditional support.

Acknowledgements

I thank God for giving me the courage, the patience, and the strength to face all the difficulties and obstacles I have encountered in the last three years.

The successful completion of this thesis would not have been possible without the guidance, infallible support and encouragement of many people. I take this opportunity to express my sincere gratitude and appreciation to all those who contributed to the success of my thesis.

I am deeply indebted to both my advisor Alain MÉRIGOT and my co-supervisor Bastien VINCKE for their persistent encouragement and motivation, for their continual and creative feedback, for their stimulating suggestions in all time of work, and for their constructive criticism. I also thank them for their patience and for believing in my capacities. Their ingenious suggestions and thought provoking propositions have helped me widen my perspective on the subject matter of this dissertation.

I sincerely thank Mr Mohamed SHAWKY and Mr GRANADO Bertrand for the honor they have shown me for having accepted to be rapporteurs of this thesis and for their interest in my research topic. My sincere thanks also go to Mr Dominique HOUZET and Mrs Michèle GOUIFFES for agreeing to evaluate this work as reviewers.

I would like to sincerely express my gratitude to the members of the SATIE - ENS Saclay laboratory in general and those of the MOSS team in particular.

I would like to thank my family. In particular my late father Oumar DIARRA, my dear mother Korotimi DIARRA, my sisters Satara and Nana, and my brothers. I thank them for their encouragement and support throughout my years of study. I take this opportunity to tell them how grateful I am for everything they have done for me.

Last, but not least, I would like to thank my dear love Minkoro FOMBA. I thank him for his patience, his infallible support, his advice and his encouragement in everything I undertake.

Publications

1. "Exploring Performance Improvement Opportunities in Directive-Based GPU Programming", Rokiatou Diarra, Alain Merigot, Bastien Vincke, Conference on Design and Architectures for Signal and Image Processing (DASIP), October 2018, Porto, Portugal

2. "RECKA and RPromF: Two Frama-C Plug-ins for Optimizing Registers Usage in CUDA, OpenACC and OpenMP Programs", Rokiatou Diarra, Alain Merigot, Bastien Vincke, IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), September 2018, Madrid

3. "Towards automatic restrictification of CUDA kernel arguments", Rokiatou Diarra, Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, September 2018

Contents

List of Figures

List of Tables

Glossary

Krn Parallelize a code region with the OpenACC' kernels construct. xiv, 81–83

KrnLp Combine the OpenACC' directives kernels and loop. xiv, 81–83

KrnLpInd Add the OpenACC' independent clause to KrnLp. xiv, 81–83, 87, 88, 90, 91, 156

KrnLpIndCol Collapsing the loops associated to a KrnLpInd construct. xiv, 81–83, 87, 88, 157

KrnLpIndTil Tiling the loops associated to a KrnLpInd construct. xiv, 81–83, 87, 88, 157

KrnLpUnc Use a #pragma acc loop inside a #pragma acc kernels directive. xiv, 81–83

Par parallelize a code region with the OpenACC' parallel construct. xiv, 81–83, 88

ParLp Combine the OpenACC' directives parallel and loop. xiv, 81, 82

ParLpInd Add the OpenACC' independent clause to ParLp. xiv, 81–83, 87, 88, 90, 91, 156

ParLpIndCol Collapsing the loops associated to a ParLpInd construct. xiv, 81–83, 87, 88, 157

ParLpIndTil Tiling the loops associated to a ParLpInd construct. xiv, 81–83, 87, 88, 157

ParLpUnc Use a #pragma acc loop inside a #pragma acc parallel directive. xiv, 81, 82

Tgt Parallelize a code region with the OpenMP' target construct. xiv, 85–87

TgtPr Parallelize a code region with the OpenMP' target parallel composite construct. xiv, 85, 86

TgtPrFr Parallelize only the outer loop with the OpenMP' target parallel for composite construct. xiv, 85–87, 92, 154

TgtPrFrCol Collapsing the loops associated to a TgtPrFr construct. xiv, 85–87, 92, 93

TgtPrFrCol_Set Setting the total number of threads executions the code region associated to a TgtPrFrCol. xiv, 155

TgtPrFrIO Parallelize the outer loop with the OpenMP' target parallel for directive and the inner loop with parallel for. xiv, 85–87

TgtPrFrIOSch Add the OpenMP' schedule(static, 1) clause to TgtPrFrIO. xiv, 85–87

TgtPrUnc Use a #pragma omp parallel inside a #pragma omp target directive. xiv, 85, 86

TgtTm Parallelize a code region with the OpenMP' target teams construct. xiv, 85–87

TgtTmDs Parallelize a code region with the OpenMP' target teams distribute composite construct. xiv, 85–87

TgtTmDsPrFr Parallelize the inner loop(s) of a TgtTmDs region with the OpenMP' parallel for construct. xiv, 85–87, 154

TgtTmDsPrFrCol Collapsing the loops associated to a TgtPrFr construct. xiv, 85–87

TgtTmDsPrFrCol_Set Setting the total number of threads executions the code region associated to a TgtTmDsPrFrCol. xiv, 155

TgtTmDsPrFrIO Parallelize the outer loop with the OpenMP' target teams direstribute parallel for construct and the inner loop with parallel for. xiv, 85–87

TgtTmDsPrFrSch Add the OpenMP' schedule(static, 1) clause to TgtTmDsPrFr. xiv, 85–87, 154

TgtTmUnc Use a #pragma omp teams inside a #pragma omp target directive. xiv, 85–87

Acronyms

ADAS Advanced Driver-Assistance Systems. xiv

API Application Programming Interface. xiv

CPU Central Processing Unit. xiv

CUDA Compute Unified Device Architecture. xiv

DSL Design Specific Language. xiv

DSP Digital Signal Processor. xiv

FPGA Field-Programmable Gate Array. xiv

GPU Graphics Processing Unit. xiv, 1

OpenACC Open Accelerators. xiv

OpenCL Open Computing Language. xiv

OpenMP Open Multi-Processing. xiv

SAD Sum of Absolute Difference. xiv

SIMD Single Instruction on Multiple Data. xiv

SSE Streaming SIMD Extensions. xiv

VO Visual Odometry. xiv

Introduction

In the last decade, there has been a continual transition from architectures relying on multi-core processors to those using many-core processors, often in heterogeneous architecture configurations. This transition has been most prominently realized in the increasing use of Graphics Processing Unit (GPU) as general-purpose computational accelerators. These accelerators offer massively parallel computing capabilities to users.

The first processors did not have a parallel architecture. With the upcoming of Intel's $x86$ architecture, it became the standard architecture for several years. Thus, softwares were mainly developed for this particular architecture. This worked for many years, as technological advances (the increasing of the count of transistors on integrated circuits) allowed to improve processors' performance without changing the architecture significantly. This effect had been predicted by Gordon Moore in 1965 which became too widely known as *"Moore's Law"*. However, these advances had slowed down and this had caused that scaling performance with the number of transistors became a critical objective [5, 6]. To further improve the performance of processors, other methods had been proposed. One of these is parallel computation. Parallel architectures can range from several complex processors to thousands of simpler processors operating in parallel. Currently, there are several manufacturers of parallel systems. Thus, the market is filled with all kinds of parallel architectures ranging from few, but fast compute cores (e.g CPUs) to processors with thousands, but rather slow cores (e.g GPUs). Also, processors, specifically tuned for a specific purpose, are available.

Further, hardware architectures undergo constant changes and improvements, introducing new, changed or removed functionality. NVIDIA is one of the big GPUs providers. NVIDIA's GPUs consist of a complex memory hierarchy with a series of different automatic and self-organized caches that need to be efficiently used (see 1.5.2). In the last fives NVIDIA's GPUs generations, many significant changes to this memory hierarchy have been applied so that code written for prior generations usually does not necessarily work as efficient as it could on newer generations. For example, NVIDIA's Fermi architectures [1] give users the ability to dynamically adjust the amount of the self-managed shared memory and the L1 cache for each function execution. The next generation, Kepler architecture, added more trade-off options to choose. In the Maxwell Generation, this feature was entirely removed. So in three consecutive generations of NVIDIA's GPUs architectures, the behavior has been constantly changed. Another example is vector processing units in CPUs. Advanced Vector Extensions (AVX) 1.0 were added in the Intel's $\mathrm{I7}-2700K$ CPUs while the $\mathrm{I7}-4765T$ CPUs have the AVX 2.0. As AVX is not backward compatible, AVX 2.0 instructions cannot be used on any older CPU, so that programmers have to explicitly check for

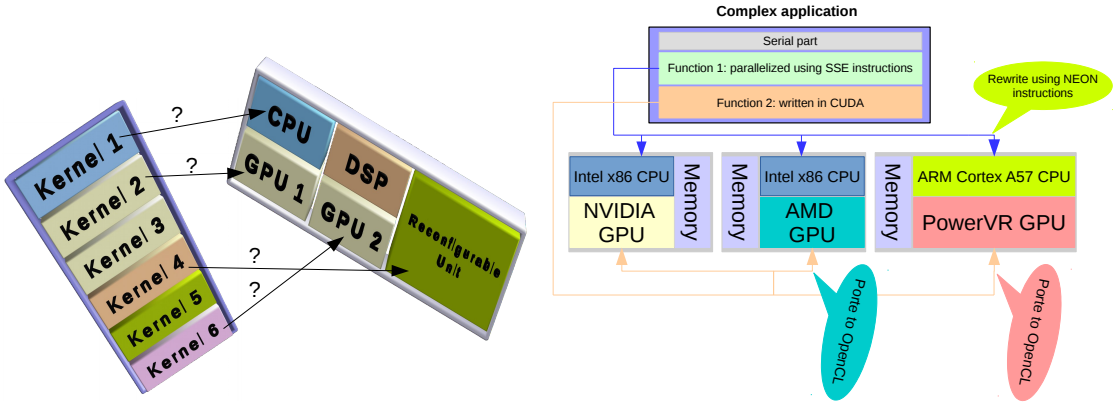
the capabilities of the CPU their software is running on.

Recent years have seen an increase of massively-parallel architectures that are based on heterogeneous architectures combining multi-core CPUs with accelerators such as GPUs, Intel Xeon Phi processors, Digital Signal Processing (DSPs), and Field Programmable Gate Array (FPGA). The current situation of embedded systems is marked by two important events: the explosion of parallelism and the rise of heterogeneous systems. For example, in the context of driver assistance systems, the need for complex processing of environment reconstruction make several manufacturers propose circuits integrating different computing components. Those generally contain from 4 to 16 cores ARM-type embedded processors and accelerators of various types: one or more GPU(s), DSPs, vector processors or specialized processors wiring image processing algorithms. For instance, there are the NVIDIA Tegra X1 embedded card (4 ARM Cortex A57, 4 ARM Cortex A53 and 1 Maxwell GPU), Texas OMAP TDA2× SoC (2 ARM Cortex A15, 2 ARM Cortex M4, 1 DSP, 1 GPU, 1 IVA HD, and 4 EVE Analytic Processors), the Renesas RCAR H2 platform (4 ARM Cortex A57, 4 ARM Cortex A53, 1 ARM Cortex R7, 1 GPU PowerVR, 1 image recognition processor, and 1 DSP), and the world leader MobileEye products. Thus all computing systems, from mobile to supercomputers, are becoming heterogeneous, massively parallel computers for higher power efficiency and computation throughput.

Although these heterogeneous architectures embedded or not provide impressive computing power with reasonable power consumption, their programming in an efficient manner remains an open problem [7]. The parallel processing power, provided by heterogeneous architectures, comes with the necessity of writing efficient code that enables all processors to solve a problem together. Indeed, the diversity of heterogeneous systems' components and different levels of parallelism make their programming based on tools and even paradigms totally different (SIMD intrinsics, OpenMP, OpenACC, CUDA, OpenCL, etc.). Figure 1b shows an example of application portability problem on heterogeneous architectures. Another challenge of heterogeneous computing is distributing the workload among the different cores to get the best performance with the lowest power consumption, as depicted in Figure 1a. Therefore, there is no automatic parallelization mechanism and porting of a complex application of such systems can represent months work.

Motivations

GPUs can achieve significant performance for certain categories of application (e.g computer vision algorithms, dense linear algebra). Nevertheless, achieving this performance not relies only on an important effort of programming and code tuning, but also good knowledge of GPUS architecture. In order to take advantage of GPUs computing potential, significant modifications of sequential applications are required. The existing low-level APIs such as CUDA and OpenCL usually require users to be expert programmers and restructure the code largely. Further, CUDA and OpenCL offer a number of features for performance optimization as the architecture is directly accessible to the user that makes possible to obtain excellent performance but adds complexities for application developers. Optimized CUDA and OpenCL codes are usually coupled with specific devices. This leads to a less productive and more error-prone software development process that



(a) Workload distribution challenge (b) Application portage challenge

Figure 1 – Illustration of two main challenges of heterogeneous computing

is challenging to be adopted by the rapidly growing HPC and applications for intelligent vehicle markets. With low-level APIs such as CUDA and OpenCL, programmers not only have to develop efficient compute kernels and manage data allocation/deallocation on GPUs memory, but also data transfers between CPUs and GPUs.

An alternative approach would be to use high-level directive-based programming models, such as OpenACC [8] and OpenMP [4], to parallelize applications on accelerators. Directive-based programming models allow the user to insert both directives and runtime calls into existing Fortran or C/C++ source code, enabling a portion of their code to execute on the accelerator. Using directives, programmers may give hints to compilers to perform certain transformations and optimizations on the annotated code regions. The user can insert directives incrementally to parallelize and optimize a program, enabling a productive migration path for legacy applications. The major advantage of the directive-based approach is that it offers a high-level programming abstraction thus simplifying the code maintenance and improving productivity. Further, directive-based programming models can offer a good trade-off between productivity, portability, and performance.

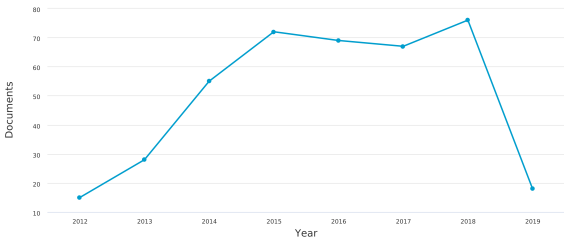
OpenMP is a parallel programming interface comprising a set of compiler directives, library routines, and environment variables. It has established itself as the de facto standard for writing parallel programs in C/C++ and Fortran on shared memory multi-core CPU systems. OpenMP added an initial extension to its feature set for making use of accelerators in version 4.0, and further extended its accelerator support in version 4.5. The target directive identifies the offload region running on the accelerator, wherein a massive number of threads may be organized into teams as prescribed by the programmer.

OpenACC, a directive-based parallel programming interface in many ways inspired by OpenMP, was the first standardized specification released to facilitate programming of accelerators. Unlike OpenMP which offers a primarily prescriptive interface for expressing parallelism, OpenACC provides both prescriptive and descriptive mechanisms for this purpose. In particular, much of OpenACC directives are intended for describing additional program information to compilers, so that it may more effectively generate code for execution on accelerators. OpenACC' descriptive in-

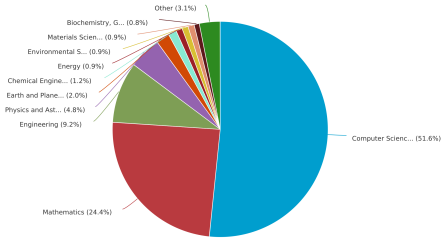
terface allows flexibility for the compiler to interpret how to map high-level language abstractions to the hardware layer.

The OpenMP and OpenACC accelerator models have not evolved independently from each other. For example, OpenMP adopted unstructured data directives and asynchronous execution of offload regions from OpenACC 2.0 into its 4.5 accelerator model specification. Conversely, OpenACC performed semantic changes to some of its data clauses in version 2.5, resulting in consistency with the corresponding behavior in the OpenMP 4.0 specification. In general, user experience with OpenACC has informed the direction taken by OpenMP in its specifications for accelerators programming.

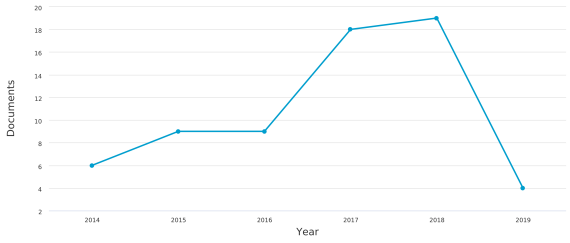
Since the first release of OpenACC in 2011 and that of OpenMP 4.0 in July 2013, many works have been done to evaluate their capabilities in term of performance, portability, and productivity. Many works have also proposed new directives and clauses for addressing some features not currently offered by OpenACC and OpenMP, but also tools to assist programmers for application acceleration with these two standards. For instance, Figure 2 shows the Scopus database search results for "OpenACC" and "OpenMP offloading".



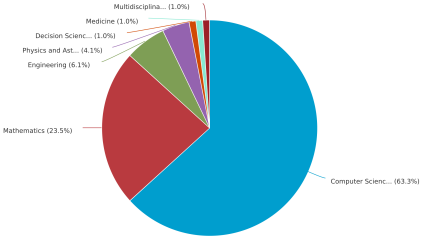
(a) OpenACC related publications per years



(b) OpenACC related published papers subject area



(c) OpenMP 4/4.5 related publications per years



(d) OpenMP related published papers subject area

Figure 2 – Scopus database search results for "OpenACC" and "OpenMP offloading"

The performance gap [9, 10, 11, 12] between programs accelerated with OpenACC and OpenMP using their relatively high-level abstractions compared to lower-level CUDA and OpenCL versions indicates that more optimization research is required.

Contributions and Dissertation Outline

Since OpenACC and OpenMP are high-level directive-based models, the compilers usually translate OpenMP/OpenACC' directives to a low-level language (e.g. assembly, CUDA, OpenCL). CUDA and OpenCL, two low-level languages, are flexible for the user to apply any optimization he wants. But this requires the user to throughly understand the underlying architecture so that the applied optimizations can utilize the architecture efficiently. OpenACC and OpenMP, therefore require the compiler to apply those optimizations automatically. However, without enough information, the compiler is not able to do the optimizations as well as the user who is an expert in both the ported application and the architecture. Even though the compiler can apply some optimizations automatically, it may not achieve the expected speedup as the compiler does not have a full view of the whole application. Because of these reasons, there is generally a significant performance gap between the codes accelerated with OpenACC/OpenMP and those hand-optimized with CUDA/OpenCL. In summary, to help programmers for speeding up efficiently their legacy sequential codes on GPU with directive-based models and broaden OpenMP/OpenACC impact in both academia and industry, several research issues are discussed in this dissertation. The contributions of this dissertation are as follows:

- The first half of Chapter 1 provides the background of the research work in this dissertation, while the second half gives an overview and a comparison of main parallel programming approaches for GPUs. We also discuss some of the main optimization strategies for GPU programming in this chapter.

- Chapter 2 starts with an in-depth analysis of OpenMP and OpenACC specifications major features. Next, we review OpenMP and OpenACC standards evolution, in the last ten years, in order to provide an overview of their pass and present status, as well as their future directions. Although OpenMP and OpenACC are similar, they present some important differences. Thus, an in-depth comparison of the two standards is also addressed in this chapter. Since OpenMP and OpenACC are standards, generic and not coupled with any specific architecture, we analyze their suitability for GPU programming and present their limitations in terms of features in comparison to low-level languages (e.g. CUDA).

- In Chapter 3, we address problems encountered in general when parallelizing an application and those specific to GPU programming. Next, we discuss optimization strategies and steps to parallelize legacy sequential codes with OpenMP and OpenACC. Next, we analyze two OpenMP and OpenACC compilers, namely LLVM/Clang and PGI compiler, in order

  - to understand how these compilers interpret OpenMP/OpenACC directives,
  - to see if these implementations comply with the OpenMP and OpenACC standards, and
  - to understand the behavior and the performance of OpenMP/OpenACC directives when the targeted accelerator is a GPU.

Next, to understand the root cause of the performance of OpenMP/OpenACC, we parallelize synthetics programs, applications, and kernels taken from the Rodinia and PolyBench benchmark suites with OpenMP, OpenACC, and CUDA and analyze the obtained results. We present a set of optimizations, most of which must be done manually in the source code level, to tune the application performance. We analyze a number of choices and combinations of optimization techniques and study their impact on application performance. We learn that poorly selected options or using system default options for optimizations may lead to performance degradation. We also compare the OpenMP and OpenACC versions performance of some applications to that of their well-tuned CUDA versions to present the reasoning behind the performance gap.

- GPUs have several levels of caches and caching read-only data in the L1 cache may improve application performance. Nevertheless, the compilers might not always be able to detect that the read-only condition is satisfied for some data because of pointers aliasing issues. We discuss, in Chapter 4, pointers aliasing problem in GPU codes and present available solutions (e.g. type qualifier, compiler flags). Next, we propose two static analysis tools that perform automatically at source level type qualifier insertion and scalar promotion to solve aliasing issues.

- Synthetics programs and benchmarks are convenient to explore performance improvement opportunities with OpenMP/OpenACC and to assess their performance. However, real-world applications are more complex than those found in benchmarks in general. Thus, in Chapter 5, we use lesson learned from Chapters 3 and 4 to parallelize a visual odometry application with OpenACC, OpenMP, and CUDA and analyze the performance results.

In this dissertation, the related work is not in a separate chapter, it is discussed in each chapter. The Chapter 6 concludes this dissertation.

Chapter 1

Background

Contents

This chapter introduces the main notions, technologies, and methods used in this thesis. We start with a general presentation of computer architecture and the definition of important concepts in section 1.1. Section 1.2 explains how and why the processor's architecture and performance have evolved from a single processor system to massively parallel multiprocessor system. This allows us to introduce the Graphics Processing Unit (GPU), which are the main targeted architecture in this thesis. In section 1.3, we define the embedded systems and specify the type of embedded system being used in this thesis. Since all computing systems, from mobile to supercomputers, tend to become heterogeneous, massively parallel computers for higher power efficiency and computation throughput, we introduce heterogeneous systems in section 1.4. Section 1.5 gives a more in-depth description of GPU architecture and presents the targeted platform in this thesis. Finally, section 1.6 gives a large overview of parallel programming approaches for accelerators.

1.1 Computer Architecture

Among the most complex systems ever created by humans, we find modern microprocessors. Indeed, a complete high-performance processor and the logic required to interface it to external devices can fit on a single silicon chip. In terms of performance, the processors implemented on a single chip today dwarf the room-sized supercomputers just 20 years ago. Even the embedded processors found in everyday appliances such as smartphones, tablets, personal digital assistants and handheld game systems are far more powerful than the early developers of computers ever envisioned. In this section, we give a brief description of computer architecture in general. We also describe the memory hierarchies found in modern processors.

The processor, memories, and bus system are the main components of a computer. Add to that the input-output devices, such as monitors, keyboards, mouses, etc., to allow interaction with the user. Although computer architecture has evolved over the years, all microprocessors designed until today are organized globally in the same way. A modern processor usually consists of several computing units or Processing Units including Arithmetic and Logical Units, Floating-Point Units, Branch Predictor Unit, etc. Processors have a number of registers to save intermediate results. They have also a Control Unit that reads and decodes the program's instructions and the PUs's operations. The processor can further contain input/output interfaces to communicate with other hardware components, such as the memory system. There are different types of memory in a computer. Volatile memory is a form of memory that retains data only if it is receiving power while nonvolatile memory retains data even in the absence of a power source. To distinguish between the volatile memory used to hold data and programs while they are running and this nonvolatile memory used to store data and programs between runs, the term main memory or primary memory is used for the former, and secondary memory for the latter [13].

Memory is characterized by three properties: capacity, bandwidth, and latency. Capacity is the amount of data that can be stored in a memory. Bandwidth or throughput is the rate at which data can be read from or stored into a semiconductor memory by a processor. Latency is the time between initiating a request for data at a particular memory address until it is retrieved by a processor. Capacity and latency evolve together, thus when the capacity of a memory increase,

its latency increases also. It has already been mentioned that processors have registers to store intermediate results. Registers are extremely fast, very limited in capacity but have a very low latency while the main memory has a very high latency and a large capacity. Chapter 4 will provide mode details on register usage for code optimization. To hide the latency between the main memory and processors that are faster, the caches are used. The Cache was the name chosen to represent the level of the memory hierarchy between the processor and main memory in the first commercial computer to have this extra level [13]. Cache memory consists of a small, fast memory that acts as a buffer for the main memory. Every general-purpose computer built today, from servers to low-power embedded processors, includes caches. Caches are queried first in order, from the nearest to the farthest when a processor requires data from memory. Currently, processors have several caches organized in levels. The closest level to a core named L1 is the smallest. The next levels have more space being further away from the hearts, so they are getting bigger and bigger. Figure 1.1 is a schematic illustration of the memory hierarchy of a computer.

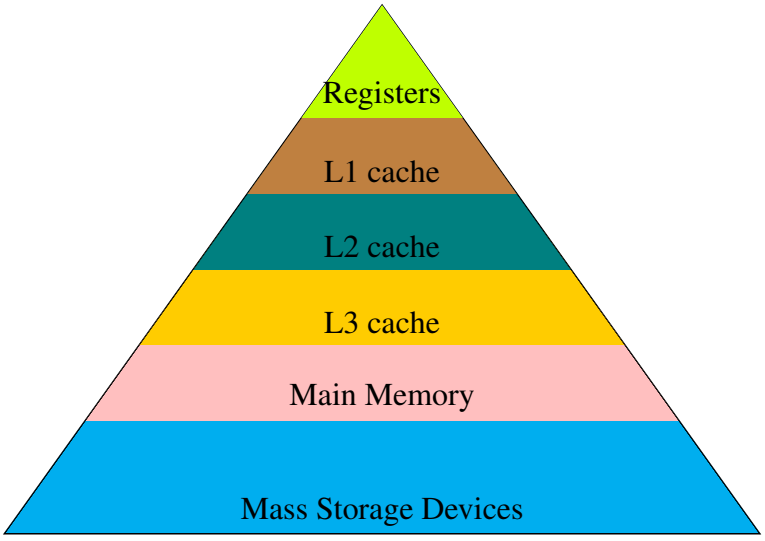


Figure 1.1 – Memory Hierarchy of a computer. With increasing bandwidth, the latency improves but the capacity decreases ([13])

1.2 Parallel Architecture

Massively parallel processors are an emerging class of hardware architectures primarily represented by modern Graphics Processing Units (GPUs) that have been developed to support massively data-parallel computations. In this section, we are going to see how processor performance scaled until the early 2000s and then halted and why massively parallel processors provide a viable solution to the performance scaling problem. We are also going to see parallel architectures currently available. Furthermore, we briefly introduce the main parallelization approaches that correspond to Flynn's taxonomy and some metrics frequently used in performance evaluation.

In 1965, Gordon Moore predicted the doubling of the number of transistors on integrated circuits every $18-24$ months [14]. This prediction thus announced the advent of the silicon revolution [15]. Although Moore's prediction was later adjusted to half the initial rate, the underlying linear trend was soon apparent and became widely known as Moore's law. An important observation pertaining to Moore's law is that increasing the transistor count of a microprocessor does not necessarily correlate with performance. Therefore, scaling performance with the number of transistors became a critical objective.

Initially, the first solutions were to increase clock speeds on one hand and to improve the internal architecture of individual processors (deeper pipeline, speculative execution, etc.) on the other hand. However, by 2000s, it had become obvious that processor performance growth was facing two major constraints: power dissipation problem and diminishing returns of the improvements in the internal architecture of individual processors [6]. Indeed, by 2004, the strategy of scaling down the size of CMOS circuits, reducing the supply voltage and increasing the clock rate was becoming infeasible. Since a chip's power consumption is proportional to the clock speed times the supply voltage squared, the inability to continue to lower the supply voltage halted the ability to increase the clock speed without increasing power dissipation [13, 6, 5].

The power limit has forced a dramatic change in the design of microprocessors. A way around the performance-scaling bottleneck due to the power dissipation problem was provided by Chip Multiprocessors also called Multi-Core Microprocessors (CMP) [16]. Early in the 21nd century, improvements in single-processor performance slowed. Furthermore, reductions in transistor size continue apace and then, so more transistors can still be packed onto chips albeit without the speedups seen in the past [6]. As a result, the computer-hardware industry has started building chips with multiple processors. Current chips range from several complex processors to thousands of simpler processors, and future generations will keep adding more [6]. Needless to say that this change in hardware requires a concomitant change in the software programming model. New parallel architectures can deliver better performance only if the software can efficiently use them. Therefore, to efficiently use current parallel architectures, applications must use a parallel programming model.

There are different parallel architectures. Machines with a number of processors that function independently and asynchronously are considered to be Multiple Instruction Multiple Data (MIMD) machines. The meaning of the term MIMD is very wide, as it describes not only multi-processor, multi-core but also any interconnected compute cluster. Simultaneous Multi-Threading (SMT) is the process of a CPU splitting each of its physical cores into virtual cores, which are known as threads. This is done in order to increase performance and to facilitate the change of context between two threads. Single Instruction Multiple Data (SIMD) architectures allow concurrent execution of the same operation on multiple data operands, what is usually called *data parallelism*. SIMD machines can be exploited either by programming explicit code with SIMD instructions (e.g SSE, AVX, NEON, etc.) or by using the auto-vectorization feature available in most compilers (e.g gcc, icc, ...).

GPUs have been introduced in the mid-90's. However, their design has been very rudimentary compared to today's GPUs. Initially, they have been specially designed for 3D rendering, but

today they are used to solve compute-intensive problems in various fields. Nevertheless, between their first appearances and today, their design has enormously evolved. Today, GPUs are massively parallel processors with up to several thousand cores in a single chip.

Intel Xeon Phi processors are another approach of designing a massively parallel architecture. It puts up to 72 cores onto a chip and is a mixture of a massively parallel GPU and a multi-core CPU, also called Many Integrated Core (MIC) architecture [17]. The Intel Xeon Phi are usually intended for use in supercomputers, servers, and high-end workstations. Unlike GPGPU like NVIDIA Tesla, Xeon Phi can, with less modification, run software that was originally targeted to a standard $\times 86$ CPU. Although Intel Xeon Phi are massively parallel, we did not use them in this thesis because their use in embedded systems, low-end workstations, etc. is less widespread than that of GPUs. However, works presented in this thesis can also be applied on Xeon Phi.

In addition to these generalist accelerators, there are specialized processors such as Digital Signal Processing (DSPs), the Epiphany-V (a $1024-$core processor) [18] or Google's TPU [19] that are specially designed to operate efficiently on the operations needed for certain applications. Specialized processors usually provide specialized hardware processing units. Besides general purpose processors and accelerators, there is Application Specific Integrated Circuits (ASIC) which are customized for a particular use or application, rather than intended for general-purpose use. In the case of ASIC, the algorithm itself is put into hardware without any means of altering after it has been manufactured. This results in the best possible performance per energy consumption ratio. However, the development of ASIC circuits is time consuming and expensive. Field Programmable Gate Array (FPGA) is more general. Indeed, they can be reconfigured but also translate the program they are executed directly into hardware. Although they usually have a very low clock frequency compared to CPUs, they can achieve much higher performance in specialized applications or when non-standard variable types are used. Table 1.1 summaries main features of CPUs, GPUs, ASICs and FPGAs.

Modern processors can have multiple level caches. In a multi-processor system, the L1 and L2 caches are generally private for each core while the L3 cache is shared between processors of a multiprocessors system. In this context, every cache level has to be kept synchronized with the next higher level to ensure that cores work on right data. The same cache hierarchy can be found in GPUs. Section 1.6 and Chapter 4 discuss about cache usage for application performance optimization on GPU.

1.2.1 Parallel Computing

Before the mandatory switch to parallel architecture, programmers could rely on innovations in hardware and compilers to improve the performance of their programs without having to change a line of code. Today, for programmers to get significant improvement in response time, they need to rewrite their programs to take advantage of parallel processors and accelerators. Moreover, to get the historic benefit of running faster on new microprocessors, programmers will have to continue to improve the performance of their code as the number of cores increases [13]. Parallelism has always been critical to performance in computing, but it was often hidden. Indeed, pipelining

Table 1.1 – Summary main features of CPUs, GPUs, ASICs and FPGAs

	CPU	GPU	ASIC	FPGA
Processing	Single / Multi-core	Thousands of identical processor cores	Application specific integrated circuit	Configurable integrated circuit
Programming	C, C++, Java, Pyton, Assembly, etc.	Kernel-based, directive-based paradigms	Application-specific (e.g: TensorFlow) API	HDL (Verilog, VHDL); newer systems include C/C++ through openCL
Strengths	Versatility, multitasking, ease of programming	Massive processing power	Custom-designed for application with optimum combination of performance and power consumption	Reconfigurable accommodates massively parallel operation, wide choice of features: DSPs, CPUs
Weaknesses	Optimized for sequential processing with limited parallelism	May have high power consumption, problems must sometime be reformulated to take advantage of parallelism	Longest development time, high cost, cannot be changed without redesigning the silicon	Second-longest development time, poor performance for sequential operations

(overlapping of multiple instructions execution) is one of the examples of parallelism hidden from the programmer.

There are several ways to increase parallel computations and application performance at the same time. Within the same application, it is possible to have multiple active separated calculations, called threads. Note that high performance can mean high throughput for independent tasks, called task-level parallelism or process-level parallelism [13]. These tasks are usually independent single-threaded applications. This approach is in contrast to running a single application on multiple processors. We will use the term parallel processing program to refer to a single program that runs on multiple processors simultaneously.

Flynn's taxonomy [20] classification was based on the number of instruction streams and the number of data streams. Thus, a conventional uniprocessor has a single instruction stream and a single data stream, and a conventional multiprocessor has multiple instruction streams and multiple data streams [13]. These two categories are abbreviated **SISD** and **MIMD**, respectively. Another category is the Single Instruction, Multiple Data (SIMD) architecture. The virtues of SIMD are that all the parallel execution units are synchronized and they all respond to a single instruction that emanates from a single program counter (PC) [13].

1.2.2 Processor Performance

We have already seen the evolution of computer design to gain even more performance. To be able to compare the performance of different processors, it is necessary to define performance. Computer performance can be defined in several ways. Hence, different performance metrics exist. Response time or execution time is a performance metric frequently used in performance

evaluation. It means the total time required for the computer to complete a task, including memory accesses, I/O activities, operating system overhead, ... [13]. Note that performance and execution time are reciprocals, increasing performance requires decreasing execution time. The throughput or bandwidth, another measure of performance, is the number of tasks completed per unit time. The CPI (Cycles Per Instruction), the average number of clock cycles each instruction takes to execute, is another performance metric. Other measures such as Instructions per Second (IPS) or Floating Point Operations Per Second (FLOPS) are used today, depending on the application and the architecture targeted. The main performance metrics used in this thesis is the execution time.

To compute the improvement (also called *speedup* (S)) of a specific application on different processors or to determine the improvement achieved through the parallelization of a code, the ratio between original execution time ($T_{original}$) and optimized or parallelized execution time ($T_{optimized}$) can be calculated.

$$S = \frac{T_{original}}{T_{optimized}} \tag{1.1}$$

In general, an application can not be entirely parallelized, hence the execution time consists always of a serial (T_{serial}) and a parallel ($T_{parallel}$) fraction as can be seen in the Equation 1.2. This equation, where *p* is the number of parallel processors, is also called Amdhal's law.

$$T_{total} = T_{serial} + \frac{T_{parallel}}{p} \tag{1.2}$$

Different applications are sensitive to different aspects of the performance of a computer system. Therefore, the performance of a program depends on the algorithm, the language, the compiler, and the targeted hardware. Overall we can see that the performance of an application can be limited by three factors which are the latency, the bandwidth, and the computation.

1.3 Embedded Systems

Embedded systems are the largest class of computers and span the widest range of applications and performance. Embedded computer systems can be found everywhere: car, smartphone, tablet, IoT devices, the computers in a television set, the networks of processors that control a modern airplane or cargo ship, etc. Embedded computing systems are designed to run one application or one set of related applications that are normally integrated with the hardware and delivered as a single system [13]. Embedded applications often have unique application requirements that combine a minimum performance with stringent limitations on cost or power. Although Intel processors, e.g $\times 86$, are the most used in the desktop, personal computers and server markets, ARM processors are one of the most popular CPUs for mobile and embedded design in IoT devices, due to its focus on low power consumption (for longer battery life). While $\times 86$ processors are Complex Instruction set Computer, ARM processors are Reduced Instruction Set Computer. Intel and ARM processors have many differences, including their processing power, power consumption, software, and applications. V. A. Rafael et al. [21] evaluated performance and power efficiency of x86 and

ARM architectures and found that ARM systems are more power efficient for SQL and static HTTP servers while x86 architecture is still more power efficient for floating point computation. In this thesis, by embedded systems, we mean embedded cards or processors such as NVIDIA Tegra K1/X1/X2 cards, OMAP systems of Texas, the RCAR platform of Renesas, etc.

1.4 Heterogeneous Systems

Today, the prevailing solution towards scalable performance comes from a class of hardware configurations commonly known as heterogeneous systems. The designation "heterogeneous parallel system" refers to parallel systems using more than one type of processor. Usually, heterogeneity in the context of computing refers to architectures with different Instruction Set Architecture (ISA). However, the Big.little architecture of ARM is for example an exception, because the instruction sets are the same and the heterogeneity refers to the speed of different microarchitectures of the same ISA [22]. The need to increase the computing power of systems without increasing energy consumption is one of the factors driving the manufacture of these heterogeneous circuits. Today, modern heterogeneous systems combine on the same integrated circuit different types of processors including a host CPU, GPU, FPGA, DSP, video/audio accelerators, Intel Xeon Phi, cryptographic accelerators, programmable network processors, etc. Parallel heterogeneous systems are present in all areas of computing, from powerful servers to consumer products such as mobile phones and tablets. Figure 1.2 shows a theoretical heterogeneous parallel system example. Heterogeneous systems offer an impressive theoretical computing power with low power consumption. Recent findings show that a heterogeneous architecture that exploits the diversity offered by multiple ISAs, can outperform the best ISA-like architecture with 21% to 23% energy savings [23]. These systems gain in performance and energy efficiency not only by adding the same type of processors, but also by adding dissimilar coprocessors incorporating specialized capabilities to handle particular tasks.

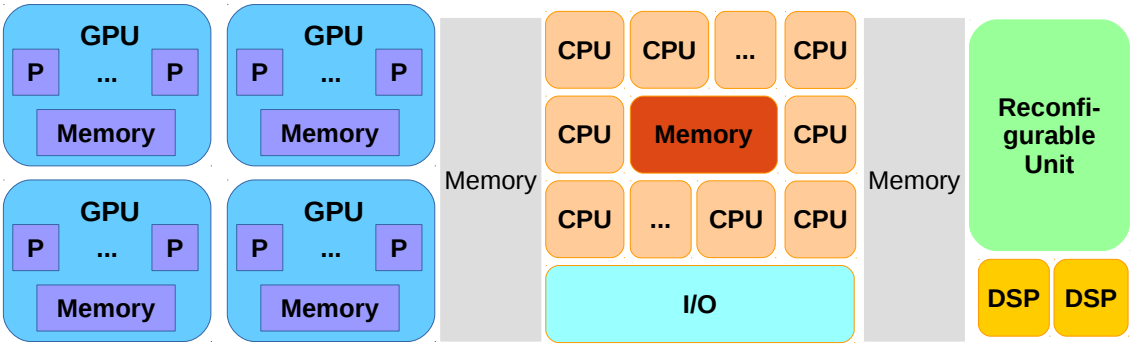


Figure 1.2 – Example of a heterogeneous computing system. Because of the cost of programmability, finding a system with the level of heterogeneity shown here is unlikely. A real system will have only a subset of these types.

Heterogeneous architectures provide impressive theoretical computing power, but their effective exploitation remains an open problem [7]. Yet another challenge is distributing the workload

among the different cores to get the best performance with the lowest power consumption. Therefore, heterogeneous systems present the major disadvantage of the extreme complexity of their programming. The heterogeneity of devices and the different levels of parallelization make the programming of these different entities based on tools, or even paradigms, completely different. In addition, while the computing community is racing to build tools and libraries to ease the use of these systems, effective and confident use of these systems will always require knowledge about low-level programming of devices consisting a heterogeneous system. If parallel programming is hard, heterogeneous programming is that hard, squared. Defining and building a productive, performance-portable heterogeneous programming system is hard. There are several programming strategies that attempt to solve this problem, including OpenMP and OpenACC which are the focus of this thesis. For the remainder of this thesis, the heterogeneous system will consist of CPUs and GPUs. We choose CPU+GPU like systems as targeted hardware platform because they are currently the most used configuration in embedded systems.

1.5 Target Architecture and Platform

As GPUs are the main target architecture used in this thesis, this section gives an introduction to GPU and an overview of NVIDIA GPU architectures. This covers five hardware generations, ranging from the *"Fermi"* up to the most recent *"Volta"* architecture. We terminate this section by presenting the two platforms used in this thesis.

1.5.1 Graphics Processing Units (GPU)

Massively parallel processors, GPUs are everywhere from mobile devices to supercomputers. They have been specifically designed to run thousands of calculations in parallel. GPUs cores are much simpler than those of CPUs, with fewer features and a lower clock frequency but their high number of processing cores compensates for this [13].

Although having several cores, GPUs are clearly distinguished from conventional SIMD machines as a single high-level instruction can span millions of data elements, each one representing an individual control path or thread. GPUs cores aim at hiding latency by having thousands of instructions in-flight to ensure that there will always be something executing on the available cores. GPUs memory is oriented toward bandwidth rather than latency. There are even special graphics DRAM chips for GPUs that are wider and have higher bandwidth than DRAM chips for CPUs [13]. GPU cores form Streaming Multiprocessors (SM) that perform the same operation in parallel, so every single core does not require its own controlling infrastructure.

There are typically two types of GPU: integrated and dedicated. Integrated GPUs share the same chip with the CPU, while dedicated (or discrete or attached) GPU's are a separate piece of hardware connected to a separate bus. Attached GPUs are equipped with their own memory. This requires to explicitly copy data between the system memory and the GPU. This copy can become a major bottleneck in many applications. Manufacturers such as Intel or AMD, therefore, provide CPUs

with directly integrated GPUs. This allows the GPUs to be directly attached to the system bus and access the main memory.

Different types of GPUs can be found in the market. Intel focuses mainly on low-end and multimedia GPUs that require only a small amount of energy and therefore can be especially used in low-power and mobile systems. The same applies to the Mali GPUs from ARM, which are specifically trimmed for smartphone applications. These GPUs provide only limited compute capabilities. However, Intel plans to produce high-end graphics card for consumers. Matrox mainly provides GPUs for multi-display setups in professional environments with advanced features, low-energy consumption, and high reliability. AMD is mainly established in low-end and gaming GPUs, but AMD recently released Radeon Instinct GPUs to expand onto the High-Performance Computing (HPC) market. NVIDIA tries to provide GPUs for the entire market from low-end, over gaming, professional up to HPC. Typically, the GPU has a two-level hierarchical architecture. It is made of vector processors at the top level, and each vector processor contains a large number of scalar processors at the lower level. This hierarchical architecture is found in almost all GPU regardless of the manufacturer (NVIDIA, AMD, etc.).

1.5.2 NVIDIA GPUs

NVIDIA GPUs has developed several generations of architecture. To distinguish between its GPUs supported functionality, NVIDIA introduced the Compute Capabilities (CC).

Fermi Generation Fermi[1] (CC $= 2.\times$) architecture, successor of the NVIDIA's Tesla microarchitecture, was introduced in 2009. It was used in GPUs of the 400, 500, low-end GPUs of the 600, some Quadro ($\times 000$ and NVS models) and Tesla C series. Fermi was the first GPU with unified compute cores and a true cache hierarchy. As in normal CPUs, the L1 cache serves as an additional layer between the L2 cache and the cores. Unified address space for local, shared and global memory was introduced with the new instruction set. Fermi architecture has 64KB of RAM with a configurable partitioning of shared memory and L1 cache. Hence, shared memory and L1 cache sizes can be dynamically adjusted for each kernel execution and the programmer can choose either to prefer shared memory or L1 cache. One of these is assigned 48KB and the other 16KB. The first Fermi-based GPU features up to 512 CUDA cores which are organized in 16 Streaming Multiprocessors (SM) of 32 cores each. Figure 1.3 shows the first Fermi-based GPU architecture. A CUDA core executes a floating point or integer instruction per clock for a thread. Each core has a fully pipelined integer Arithmetic Logic Unit and Floating Point Unit. Fermi supports concurrent kernel execution, where different kernels of the same application context can execute on the GPU at the same time. Concurrent kernels execution allows programs that execute a number of small kernels to utilize the whole GPU.

Kepler Generation The Kepler architecture (CC $=$ [3.0, 3.2, 3.5 and 3.7]), successor of Fermi, was released in 2012. Kepler was Nvidia's first microarchitecture to focus on energy efficiency. It was used in GPUs of the 600, 700, 800, low-end GPUs of the 900, Quadro K and Tesla K

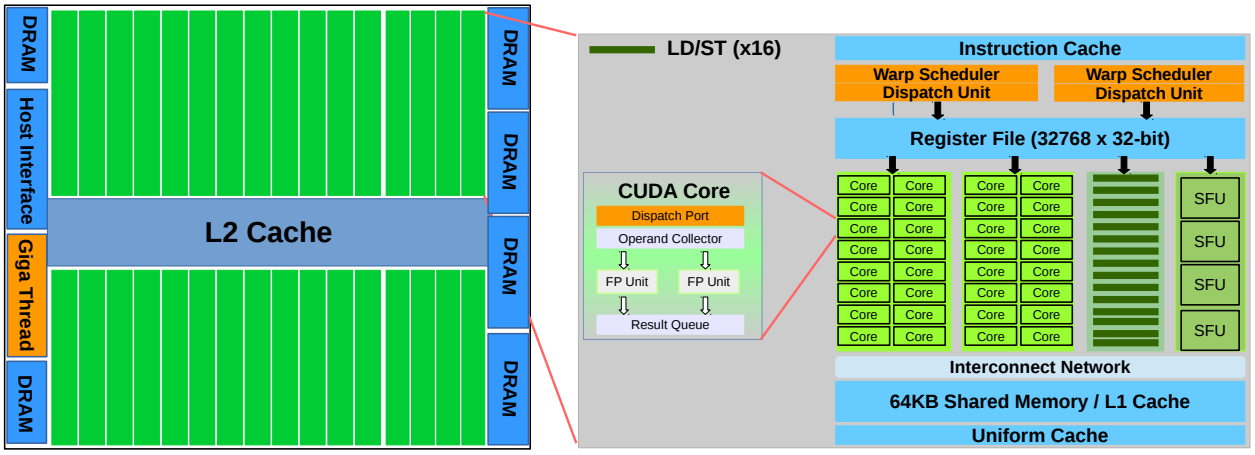


Figure 1.3 – First Fermi-based GPU architecture: 16 SM of 32 cores each [1]

series. There have been two major revisions of the Kepler architecture, starting with CC 3.0 and 3.5. GPUs with CC 3.0 introduced the shuffle functions that allow accessing memory from other threads within the same warp (a set of 32 threads) without additional hardware registers. Further, Kepler added a third mode to the L1 cache and shared memory, to equally distribute them to 32KB each. The shared memory supports to adjust the size of shared memory banks either to serve 32 or 64B banks, depending on the data that is supposed to be stored. Further, the operation mode of the L1 cache had been changed to only serve local memory accesses. Most likely this was done to remove any synchronization between the SMs for global memory, to reduce the communication between the SMs. With the second generation of Kepler (CC > 3.5) dynamic parallelism has been introduced, allowing that GPUs functions (also called *kernels*) start other kernels directly from the GPU.

Maxwell Generation The Maxwell architecture is the successor of Kepler. The first generation of Maxwell GPUs or GM107/GM108 graphics, with CC $= 5.0$, was released in 2014 as Geforce GTX 745, GTX 750/750 Ti, GTX 850M/860M, 830M/840M and Quadro Mxxx. The 900 series (CC $= 5.2$) and Jetson TX1/Tegra X1 (CC $= 5.3$) embedded processors were the second generation of Maxwell. Maxwell architecture introduced significant changes to the memory system. First of all, the shared memory no longer can be configured, neither the capacity nor the memory bank size. Further, the L1 cache has been merged with the non-coherent cache. The L2 cache size was increased from 256KB on Kepler to 2MB on Maxwell. Accordingly, the memory bus was reduced from 192 bit on Kepler to 128 bit, further saving power. The SM design from Kepler was also modified and renamed to SMM for Maxwell.

Pascal Generation The Pascal architecture (CC $= [6.0, 6.1$ and $6.2]$) is the successor of Maxwell. It was released in 2016 with the Geforce GTX 10XX and Tesla P series. One of the new features of Pascal is unified memory. Unified memory model allows both CPU and GPU to access main system memory and memory on the graphics card with the help of a technology

called "Page Migration Engine". Another new feature is NVLink, a high-bandwidth bus between the CPU and GPU, and between multiple GPUs. NVLink allows much higher transfer speeds than those achievable by using PCI Express. Further, the Tesla P100 features the new HBM (High Bandwidth Memory) memory. This allows much higher memory capacities (up to 16GB) and bus width (4096B).

Volta Generation The Volta architecture (CC = 7.0), the successor of Pascal, is the newest NVIDIA GPU architecture. The first graphics card to use it was the data center Tesla V100 [24]. It has also been used in the Quadro GV100 and Titan V. Architectural improvements of the Volta architecture include among others new SM architecture optimized for deep learning, the second generation of NVLink, HBM2, a new combined L1 data cache and shared memory unit and tensor cores. Table 1.2 summurize main features of NVIDIA GPUs architecture from Kepler to Volta.

Table 1.2 – Technical Specifications and Feature Support per Compute Capability [3]

	Kepler				Maxwell			Pascal			Volta	
	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.5
Dynamic Parallelism	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
16-bit float op	No	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Tensor Core	No	No	No	No	No	No	No	No	No	No	Yes	Yes
Max grids / device	16	4	32	32	32	32	16	128	32	16	128	128
Max blocks / SM	16	16	16	16	32	32	32	32	32	32	32	16
Max warps / SM	64	64	64	64	64	64	64	64	64	64	64	16
Max threads / SM	2^{10}		2^{10}	2^{10}	2^{10}	2^{10}	2^{10}	2^{10}	2^{10}	2^{10}	2^{10}	2^{11}
Registers/SM	64K	64K	64K	128K	64K	64K	64K	64K	64K	64K	64K	64K
Registers/block	64K	32K	64K	64K	64K	64K	32K	64K	64K	32K	64K	64K
Registers/thread	63	255	255	255	255	255	255	255	255	255	255	255
Max shared memory per/SM (in KB)	48	48	48	112	64	96	64	64	96	64	96	64

1.5.3 NVIDIA Quandro M2000M

Our first targeted platform is a laptop with an NVIDIA Quadro M2000M hosted in an Intel I7 CPU. Quadro M2000M GPU was released in 2015. It is part of Maxwell first generation and relies on the GM107 graphics processor. It features 640 CUDA cores, 40 texture mapping units and 16 ROPs (Render Output Unit). Figure 1.4 shows the bloc diagram of this GPU. The chip has 4GB of GDDR5 memory, which are connected using a $128-$bit memory interface. The GPU is operating at a frequency of 1.14GHz. Quadro M2000M GPU has no display connectivity, as it is not designed to have monitors connected to it. Table 1.3 summarizes Quadro M2000M and Tegra X1 GPU main features.

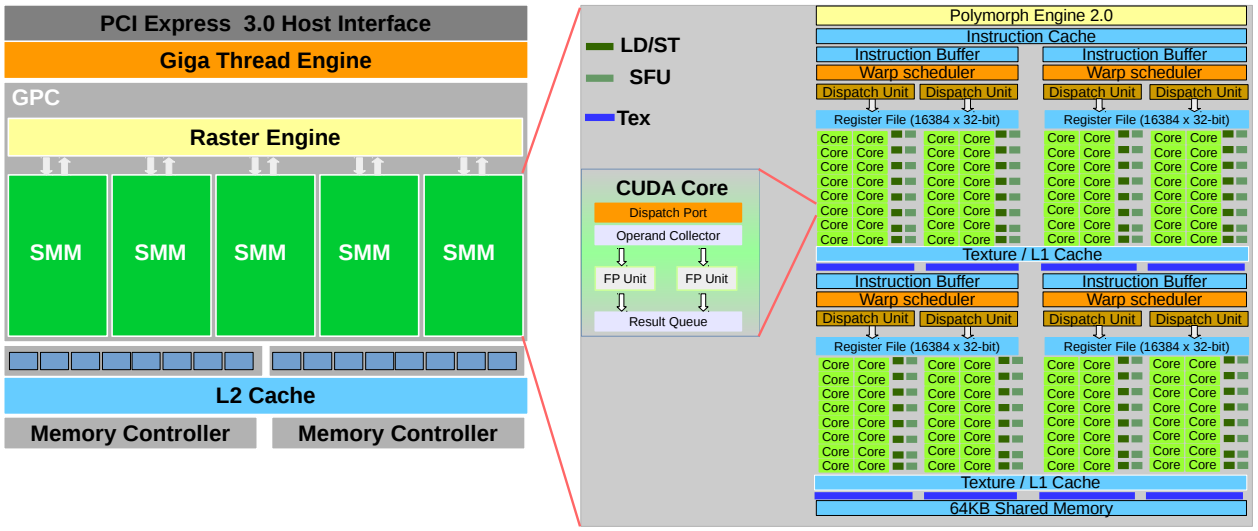


Figure 1.4 – NVIDIA Quadro M2000M block diagram

1.5.4 NVIDIA Tegra X1

NVIDIA Tegra X1 [2] SoC, an embedded card, is the second platform targeted in this thesis. NVIDIA Tegra X1 SoC includes an NVIDIA Maxwell GPU and ARM CPUs. It offers the same features as the laptop and desktop Maxwell based products. The GPU offers 256 CUDA cores. Tegra X1 delivers class-leading performance and incredible energy efficiency while supporting all the modern graphics and compute APIs. Figure 1.5 shows the block diagram of Tegra X1 SoC.

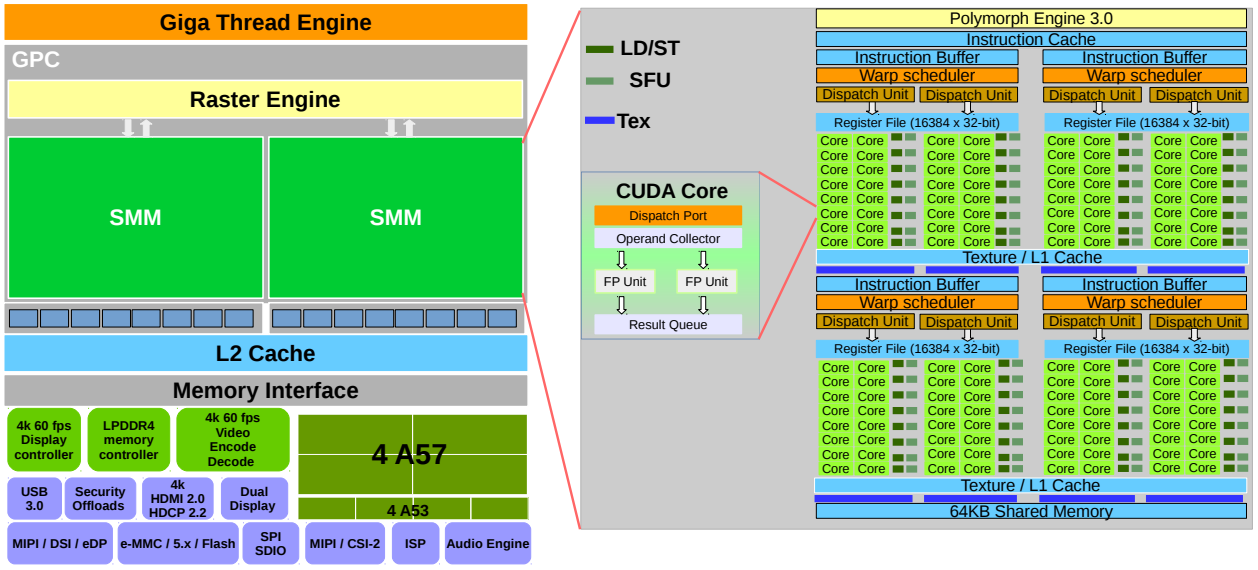


Figure 1.5 – NVIDIA Tegra X1 embedded card (printed from [2])

Table 1.3 – Summary of Quadro M2000M and Tegra X1 GPUs main features

	Quadro M2000M	Tegra X1
CC	5.0	5.3
SMM	5	2
CUDA cores per SMM	128	128
Texture mapping unit	40	16
ROPs	16	16
GPU clock rate	1.14GHz	0.07GHz
Memory clock rate	2.5GHz	13MHz
Memory Bus Width	128−bit	64−bit
Global memory	4GB	4GB
L2 Cache Size	2MB	256KB
Constant memory	65KB	65KB
Shared memory	48KB	48KB
Registers available per block	65536	32768

1.6 GPUs Programming Approaches

The diversity of massively parallel architectures leads to heterogeneous programming models. In this section, We are going to see that parallel programming models can be low or high level. Low-level or kernel-based languages, strongly related to architecture, are difficult to handle but can offer good performance since they were designed for these architectures. On the other hand, high-level languages are easier to use but can result in lower performance compared to kernel-based languages. The diversity of programming models for massively parallel architectures poses problems not only for development but also for portability. Thus, we present some automatic parallelization tools that aim to automatically generate parallel code for GPUs from a sequential program. We terminate this part with a brief presentation of some domain specific languages.

1.6.1 Kernel-Based Programming Languages

CUDA

First GPUs could only be programmed with low-level graphics-based languages (i.e. shading languages) which discouraged non-graphics experts from using them. In 2006, NVIDIA introduced a real change in GPUs architecture. Moreover, with the release of CUDA, GPUs became popular and usable for general purpose computing. In addition, the first release of the OpenCL (Open Computing Language) specification by KHRONOS GROUP in 2008 was another milestone as it enabled software portability across different heterogeneous devices and platforms [25]. CUDA and OpenCL are relatively similar. In order to facilitate the generality of the definitions presented and used in the rest of the thesis, CUDA's terminologies will be adopted.

CUDA consists of a compute model and a programming language [3]. CUDA's compute model is designed for massively parallel processors. In CUDA's compute model, threads are organized in SIMD groups, which are called warps. A warp consisted of 32 threads. Multiple warps form a block and blocks are grouped in grid. The current maximum is 1024 threads per block or 32 warps per block. During kernel execution, one block of threads is mapped onto one SM on the GPU. Figure 1.6 illustrates thread hierarchies in CUDA and how they are mapped to the GPU.

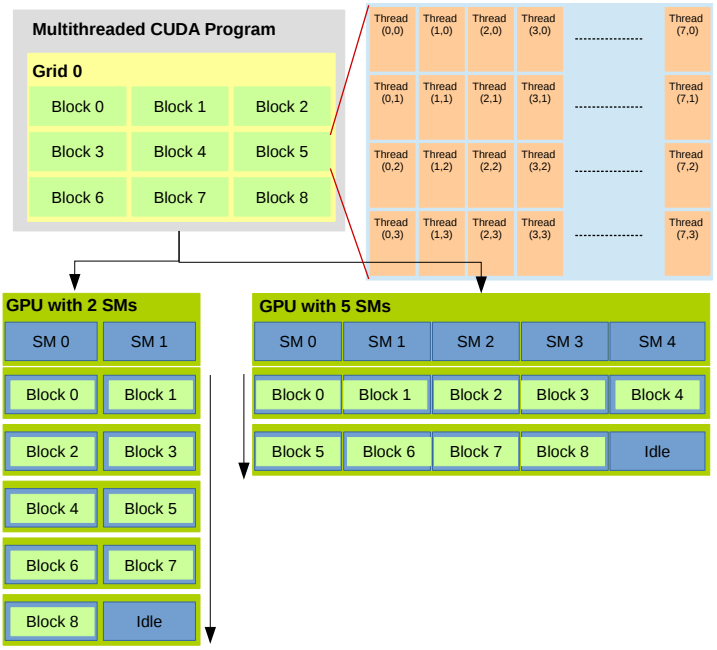


Figure 1.6 – Schematic illustration of a CUDA program with 9 blocks and how they are scheduled onto two different GPUs

Although hardware multithreading is used on GPU to hide latencies, if a condition (that depends on the thread ID) is not fulfilled by all threads in a warp, a thread divergence occurs. In this situation, only threads that fulfill the condition continue the execution until the end of the conditional block. An example for thread divergence in a same warp is illustrated in Figure 1.7.

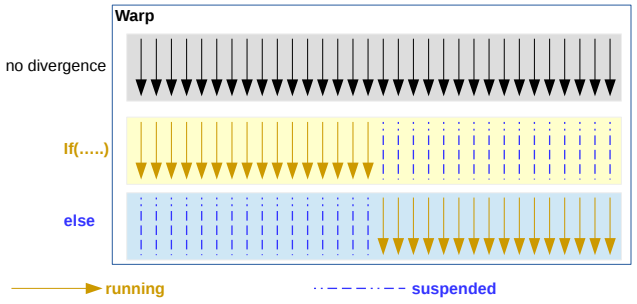


Figure 1.7 – Example for thread divergence in a same warp

The main memory of the GPU, called *global memory* or *device memory*, can be accessed from all

threads for read and write operations. To guarantee that all threads in a warp, a block or the entire GPU read the most recent data, special synchronization functions can be used if needed. Within a block, data can be shared between threads of this block. To do that, data must be put in the shared memory (visible on Figure 1.3 for example). The shared memory, a very fast on-chip memory, is organized in memory banks. Simultaneous access to the same bank results in conflicts, that cause a serialization of the memory access. The CUDA compute model provides another memory called *local memory*, which resides in the global memory but is private to a single thread. Local memory accesses have same high latency and low bandwidth as global memory accesses [3]. On GPUs of compute capability $5.x$, local memory accesses are always cached in L2 in the same way as global memory accesses. Due to the limited number of registers per thread, the local memory allows using more memory per thread. Additionally, each SM has a small but fast, read-only memory for constant values (called *constant memory*). Constant memory resides in device memory and can be written to by the host system prior to calling a kernel.

Since global memory has a high latency, inter-thread data locality must be carefully considered in order to optimize global memory access performance. Indeed, memory operations are issued per warp on GPU. When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more memory transactions. For example, if threads in a warp request data from consecutive memory addresses, the data can be ready in 4 or fewer transactions since all the data transferred is efficiently used. This is referred to as a *coalesced memory access*. If on the contrary, threads in a warp request data from scattered addresses, up to 32 memory transactions may be necessary to make the data ready for all the threads within this warp.

The CUDA programming language consists of two different parts, one for the host system that controls the execution and one for the device that performs the actual computation. The host system is responsible to allocate memory, copy data to and from the device and launch kernels. Two different host APIs are available. The first is the Runtime API that allows writing GPU and CPU code in the same file. The second API is the Driver API. It not only supports all features of the Runtime API but also comes with advanced features, e.g., to dynamically load kernel implementations during runtime. This allows choosing different implementations of a kernel according to the underlying hardware without recompiling the actual application. However, the Runtime API is tuned for usability, the Driver API is tuned for features so that it requires more programming effort than the Runtime API.

To write a kernel, certain requirements have to be fulfilled. First, it is necessary to annotate kernel with the *__global__* keyword, so that the compiler knows that this function has to be executed on the GPU. Every output needs to be stored into memory that is passed as a pointer to the kernel. The kernel itself has then to be written from the perspective of a single thread, whereas all threads execute the same code. The programmer specifies at least the number of threads per block at kernel's launch site. To distinguish between the threads, it is possible to acquire the ID of a thread inside a block, or the ID of a block, as well as the sizes and counts of blocks by the variables *threadIdx*.x, y, z, *blockIdx*.x, y, z, *blockDim*.x, y, z and *gridDim.x, y, z*.

NVIDIA also provides a pseudo-assembly language for CUDA capable devices, called PTX (Par-

allel Thread eXecution architecture). The PTX is an intermediate assembler. Once a CUDA code is written, *nvcc*, a compiler from NVIDIA, translates it into PTX, and the graphics driver contains a compiler which translates the PTX into a binary code which can be run on the GPU.

Since data transfer to and from device memory is costly, when a program uses a large array, data copies can be overlapped in some GPUs (CC $\geqslant 2.\times$). This is called concurrent data transfers. Concurrent operations are managed through *streams*. A stream is a sequence of commands that executes in order. Different streams may execute their commands out of order with respect to one another or concurrently.

Listing 1.1 shows a naive implementation of Sobel filter 3×3 in CUDA. Lines 2 to 10 shows a naive implementation of the Sobel filter in CUDA. On the CPU side, CUDA runtime functions are called to allocate device memory and transfer the input image onto the device (shown between lines 16 and 18). The numbers of thread that each block will contain is defined at the line 21 while the total number of blocks is defined at line 23 or 26. In this configuration, each thread works on one pixel. The Sobel filter kernel is launched on device with the defined number of threads at line 24 or 28. After the computation finishes, the data is copied back from the GPU.

```
1  //kernel definition
2  __global__ void sobelCUDA(const uint8_t *input, uint8_t *output_h,
3                            uint8_t *output_v, const int H, const int W) {
4    int x=blockDim.x*blockIdx.x+threadIdx.x; //get thread index in block
5    int y=blockDim.y*blockIdx.y+threadIdx.y;
6    if (x >= W || y >= H) return;
7    output_h[x+y*W]=(-input[(x-1)+(y-1)*W]-2*in[x+(y-1)*W]-in[(x+1)+(y-1)*W]
8                     +in[(x-1)+(y+1)*W]+2*in[x+(y+1)*W]+in[(x+1)+(y+1)*W])/9;
9    output_v[x+y*W]=(-input[(x-1)+(y-1)*W]+in[(x+1)+y*W]-2*in[(x-1)+y*W]
10                    +2*in[(x+1)+y*W]-in[(x-1)+(y+1)*W]+in[(x+1)+(y+1)*W])/9;
11 }
12
13 void sobelLauncher(const uint8_t *input, uint8_t *output_h,
14                    uint8_t *output_v, const int H, const int W) {
15   int mem_size = H * W * sizeof(uint8_t);
16   uint8_t *d_input, *d_output_h, *d_output_v;
17   cudaMalloc((void **) &d_input, mem_size); //allocate memory on device
18   cudaMalloc((void **) &d_output_h, mem_size);
19   cudaMalloc((void **) &d_output_v, mem_size);
20   //transfert input image from host to device memory
21   cudaMemcpy(d_input, input, mem_size, cudaMemcpyHostToDevice);
22   //set grid and block sizes
23   dim3 Db(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y, 1);
24   dim3 Dg((size_t)(ceil(((float)W)/((float)DIM_THREAD_BLOCK_X))),
25           (size_t)(ceil((float)H/((float)DIM_THREAD_BLOCK_Y))), 1);
26   //kernel invocation
27   sobelCUDA<<<Dg,Db>>>(d_input, d_output_h, d_output_v, H, W);
28   //copy back results
29   cudaMemcpy(output_h, d_output_h, mem_size, cudaMemcpyDeviceToHost);
30   cudaMemcpy(output_v, d_output_v, mem_size, cudaMemcpyDeviceToHost);
31   //free device memory
32   cudaFree(d_input);
33   cudaFree(d_output_h);
```

```
34    cudaFree(d_output_h);
35 }
```

Listing 1.1 – Example of Sobel filter naive implementation in CUDA

The CUDA version of Sobel filter seen in Listing 1.1 is not optimized. Indeed, each pixel of input image is read many times. One optimization can be to reduce the number of unnecessary data loads by dividing the Sobel filter processing into horizontal and vertical passes. Additionally, the shared memory can be used to store the portion of data that is used by threads in the same block and thus reduce global memory accesses. Another optimization can be the usage of textures instead of shared memory. Therefore, to obtain an optimized CUDA code, the programmer must understand well both GPUs architecture and CUDA optimization strategies like memory-coalescing access, efficient usage of shared memory, textures, kernel splitting and tiling technology. Additionally, grid and block configurations, computing behaviors of each thread, and synchronization problems need to carefully be tuned [26].

Performance optimization in CUDA

Generally speaking, performance optimization in CUDA may includes:

1. Maximize parallel execution by structuring the code in a way that it exposes as much parallelism as possible and efficiently maps this parallelism to the various components of the GPU to keep them busy most of the time.

2. Maximize parallel execution between the host, the devices, and the bus connecting the host to the devices, by using asynchronous functions calls and streams.

3. Minimize synchronization functions calls.

4. Avoid thread divergence.

5. Optimize memory usage to achieve maximum memory throughput by avoiding un-coalesced memory accesses or use shared memory to reduce redundant data loads for example.

6. Caching, in the unified L1/texture cache, data that is read-only for the entire lifetime of the kernel by marking pointers used for loading such data with both the *const* and *__restrict__* qualifiers (see Chapter 4).

7. Consider the capabilities (maximum size of grids, blocks, warps, register, etc.) of the targeted GPU. For example the number of registers used by a kernel can have a significant impact on the number of resident warps and therefore on the overall performance of this kernel. Considering the case of a kernel that uses 64 registers and each block has 512 threads and requires very little shared memory. If we execute this kernel on a device of CC $5.x$, then two blocks can reside on the multiprocessor since they require $2 \times 512 \times 64$ registers, which exactly matches the number of registers available on the SM (see Table 1.2). But as soon

as the kernel uses one more register, only one block can be resident since two blocks would require more registers than that are available on the SM.

8. The block size should be chosen as a multiple of the warp size (32) to avoid wasting computing resources with under-populated warps as much as possible. On the other hand, the grid size should be a multiple of the number of the targeted GPU's SMs to avoid idle SM during the kernel execution.

OpenCL

OpenCL (Open Computing Language) [27] is a language-based programming model for heterogeneous platforms including CPUs, GPUs, DSPs, FPGAs, etc. OpenCL provides an API to control the platform and program execution on the compute devices. Unlike CUDA, OpenCL has a cross-vendor and cross-platform software portability. OpenCL execution model is similar to that of CUDA but requires explicit use of operations such as: the creation of the device context, the command queue, the program objects, the memory objects, etc.

SYCL

SYCL [28] is a cross-platform abstraction, shared source, C++ programming layer for OpenCL. It is built on top of OpenCL 1.2 and based on standard C+ + 11. SYCL adds the ease of use and flexibility of single-source C++ to OpenCL underlying concepts: portability and efficiency. SYCL allows programmers to shared write an entire program in valid C+ + 11, without having to add additional non-standard keywords to their source. During the compilation phase, compiler separates accelerator kernels and generates code to execute them on a device. Runtime resolves data dependencies and schedules host-device data movement. Currently, the two most mature implementations of SYCL are *triSYCL* and *computecpp* and both have not support for NVIDIA GPUs.

1.6.2 Directives-based approaches

A directive is a code line, typically starting by *"#pragma"*, that tells the compiler something. Directives have been used for several years for the parallelization of code on CPU. They are easy to use because they may not require rewriting the source code. When a compiler encounters a directive he does not know, he simply ignores it. Thus, directives are portable. GPUs programming using directives-based approaches is an alternative to CUDA and OpenCL. Because of their ease of use, directive-based approaches can offer a good trade-off between productivity, portability, and performance. However, such programming strategies impose technical challenges on compiler optimizations, which could result in lower performance than with kernel-based languages. OpenMP and OpenACC are the two most widely used directives-based approach for accelerators programming. Since they are the main focus of this thesis, we will take a deeper look into their specifications in the next chapter. There are many directives-based approaches for GPU programming, besides

OpenMP and OpenACC, that have been proposed by the parallel programming community. Here, we give an overview of two of them.

HMPP

HMPP (Hybrid Multicore Parallel Programming) is a framework consisting of compiler directives, tools and software runtime that supports multi-core processor parallel programming in C and Fortran on Unix platforms [29, 30]. HMPP directives are used to annotate the original code with instructions to execute a routine, the *codelet*, on a accelerator. A codelet is a pure function whose arguments are alias-free and does not contain static/volatile variable declarations and no reference to global variables, except variables declared as *"resident"* by an OpenHMPP directive. HMPP provides several directives for device initialization, codelet execution specification, data transfer to and from device memory and synchronization mechanisms. If the targeted device is an NVIDIA GPU, HMPP generates CUDA code. HMPP can also generate OpenCL code for OpenCL devices. S. Grauer-Gray et al. [31] demonstrated that the performance of auto-tuned HMPP-generated implementations can approximate that of simple CUDA ans OpenCL codes. HMPP is compatible with OpenMP and MPI. Although HMPP did not have the same success as OpenMP and is not widely used, some works have been done around it. CAPS with some French academic partners (e.g. INRIA, CNRS, etc.) have launched the OpenHMMP project that aims to make HMPP a standard for accelerator programming.

```
1  #pragma hmpp sobelHMPP codelet, target=CUDA, \
2  args[output_h,output_v].io=inout //codelet definition
3  void sobelHMPP(const uint8_t *input, uint8_t *output_h,
4                 uint8_t *output_v, const int H, const int W) {
5    for(int y = 1; y < H; y++)
6      for(int x = 1; x < W; x++) {
7        //lines 7 to 10 of Listing 1.1
8      }
9  }
10 int main(int argc, char **argv){
11   //.....
12 #pragma hmpp sobelHMPP advancedload args[input] .size={W, H} //Preload data
13 #pragma hmpp sobelHMPP callsite //codelet invocation
14   sobelHMPP(input, output_h, output_v, H, W);
15 #pragma hmpp sgemm delegatedstore, args[output_h,output_v] //copy results
16   //.......
17 }
```

Listing 1.2 – Sobel filter implementation in HMPP

OmpSs

OmpSs [32] programming model, proposed by the Barcelona Supercomputing Center, is another directive-based approach for aiding application porting to heterogeneous architectures. OmpSs,

based on OpenMP directives, is implemented using the Mercurium compiler and the Nanos++ runtime. The programmer must annotate his code with compiler directives specifying task-based parallelism. OmpSs annotations are interpreted by the Mercurium source-to-source compiler, which supports Fortran, C, and C++ languages. For each call to the annotated functions the compiler generates a call to the Nanos++ runtime system to create a new task. If the target is a CUDA/OpenCL device, the result is compiled by a native CUDA/OpenCL compiler. In the case of the FPGAs, OmpSs uses the High-Level Synthesis tools from FPGA vendors to generate the IP configurations for the FPGA. Besides performing a task-based parallelization, the runtime system moves the data as needed between host and devices. Since the code generated by compiler can be less optimized than a hand-optimized CUDA code, OmpSs provides an interoperability mode with CUDA. Indeed, programmer can write the kernel code in CUDA and uses OmpSs directives to manage data. Listing 1.3 shows the example of Sobel filter parallelization with OmpSs.

```
1  #pragma omp target device(cuda) copy_deps
2  //specify data that must copied to and from GPU memory
3  #pragma omp taskinput(input[H;W]) output(output_h[H;W],output_v[H;W])
4  void sobelOmpSs(const uint8_t *input, uint8_t *output_h,
5                  uint8_t *output_v, const int H, const int W) {
6    for(int y = 1; y < H; y++)
7      for(int x = 1; x < W; x++) {
8        //lines 7 to 10 of Listing 1.1
9      }
10 } //copy back output_h and output_v
11 int main(int argc, char **argv){
12   //.....
13   sobelOmpSs(input, output_h, output_v, H, W);
14   //.....
15   //example of using a CUDA kernel with OmpSs
16   #pragma omp target device(cuda) copy_deps
17   #pragma omp taskinput(input[H;W]) output(output_h[H;W],output_v[H;W])
18   {
19     dim3 block(DIM_THREAD_BLOCK_X, DIM_THREAD_BLOCK_Y, 1);
20     dim3 Dg(W/DIM_THREAD_BLOCK_X, H/DIM_THREAD_BLOCK_Y, 1);
21     sobelCUDA<<<Dg,Db>>>(d_input, d_output_h, d_output_v, H, W);
22   }//copy back output_h and output_v
23   //.......
24 }
```

Listing 1.3 – Sobel filter implementation in OmpSs

1.6.3 Automatic parallel code generation tools

Unlike low-level languages that require from the programmer to write parallel code and directives-based approaches that allow the programmer to mark code portion to be parallelized with directives, automatic parallelization of sequential code has become increasingly relevant in accelerators programming. Many works have been done in this area. Here we give a brief description of some

of them.

Par4All

Par4All [33] is an automatic parallelizing and optimizing compiler for C and Fortran programs. It was developed by HPC Project with the purpose of migrating existing sequential applications to various hardware targets such as multicore systems, high-performance computers or GPUs. Par4All takes a sequential C or Fortran program as input and creates new OpenMP < 4.0, CUDA or OpenCL source codes. The original source codes remain mainly unchanged for well-formed programs. The automatic transformation process is based on the Parallelization Infrastructure for Parallel Systems, which is a framework for source-to-source program analysis, optimization, and parallelization. Par4all does array privatization, reduction variable recognition, and induction variable substitution. It also does memory data transfer optimization when the target code is CUDA or OpenCL. Par4All is no longer maintained and tends to be obsolete.

PPCG

PPCG (Polyhedral Parallel Code Generator) [34] is a source-to-source compiler that can generate CPU and GPU code with one or more CUDA kernels from any static affine loop nest. PPCG is based on polyhedral compilation techniques. It combines affine transformations to extract data-parallelism with a code generator to orchestrate it over multiple levels of parallelism and memory. To convert a fragment of a C program to CUDA, the programmer must precede this fragment by a line containing *#pragma scop* and *#pragma endscop* at the end of targeted code fragment, and pass the flag *–target=cuda* to the PPCG compiler.

KernelGen

KernelGen [35] is a parallelization framework. It supports the major scientific programming languages including C and Fortran, and has multiple backends that can generate target code for both X86 CPUs and NVIDIA GPUs. KernelGen aims to move onto GPU the maximum possible portion of code, including memory allocations, creating efficient data layout principally for GPU computations. In KernelGen, loops parallelism analysis is based on Polly (a high-level loop and data-locality optimizer and optimization infrastructure for LLVM) and CLooG (a free software and library to generate code for scanning Z-polyhedra). These analyses are complemented with GPU-specific LLVM IR code generation. And then, LLVM IR is further lowered into PTX assembler using NVPTX backend.

Cetus

Cetus [36] is a source-to-source compiler for ANSI C programs. Cetus uses static analyses such as scalar and array privatization, reduction variables recognition, symbolic data dependency testing,

and induction variable substitution. Cetus provides auto-parallelization of loops through private and shared variables analysis and automatic insertion of OpenMP directives. Cetus can also transform OpenMP code into GPGPU code [37], by identifying the regions that will be transformed into kernels, based on the inflection points that constitute the synchronization instructions: omp barrier, omp flush, and omp critical. Then, for work partitioning, the identified regions are replaced with CUDA kernel calls. The distribution of the data is carried out relying on appropriate memory transfers.

DawnCC

DawnCC [38] provides a suite compiler-related methods to insert automatically OpenACC or OpenMP 4.0 device directives in sequential C/C++ code that can then be interpreted by compatible compilers to generate machine code. DawnCC is built on top of the LLVM (version 3.7) compilation framework, whose intermediate representation is used as input for the static analyses. DawnCC can be used stand-alone or through an online interface. Listing 1.4 shows the outputs of DawnCC for the Sobel filter routine (lines 4 to 10 of Listing 1.3). DawnCC is easy to use since it does not require any intervention from the user and knowledge of OpenACC and OpenMP. However, DawnCC has several limitations. First, DawnCC add supplementary instructions in order to determine arrays that must be copied to and from the device (e.g. GPU, Intel Xeon Phi) memory as well as the size of data that will be transfered. These instructions up to 116 for the Sobel filter routine. These instructions can increase the execution time of the Sobel filter routine. Second, DawnCC can not optimize data transfer between host and device memory. For instance, it is not necessary to copy the arrays *out_h* and *out_v* to device memory since these arrays are not read. However, DawnCC copy these arrays to and from device memory, thus increasing the time spent in data transfer as well as the overall execution time of our Sobel filter routine. Third, DawnCC does not insert any optimization clauses. As we will see in Chapter 3, It is insufficient to simply insert only *kernels* (for OpenACC, line 8 of Listing 1.4) or *target* (OpenMP, line 19 of Listing 1.4). To be clear, insert only *kernels* or *target* severely degrades code performance.

```
1  //the output for OpenACC
2  void sobelDawnCCOpenACC(....) {
3    long long int AI1[111]; AI1[0] = 2 * w; AI1[1] = AI1[0] + 2;
4    AI1[2] = AI1[1] * 1; AI1[3] = w + 2; AI1[4] = AI1[3] * 1;
5    AI1[5] = AI1[0] * 1; AI1[6] = w * 1; //... and so on ....
6    #pragma acc data pcopyin(in[AI1[91]:AI1[92]]) \
7    pcopy(out_h[AI1[109]:AI1[110]],out_v[AI1[109]:AI1[110]]) if(!RST_AI1)
8    #pragma acc kernels if(!RST_AI1)
9    for(int y = 1; y < H; y++)
10     for(int x = 1; x < W; x++) { /*lines 7 to 10 of Listing 1.1*/ }
11 }
12 //the output for OpenMP
13 void sobelDawnCCOpenMP(....) {
14   long long int AI1[111]; AI1[0] = 2 * w; AI1[1] = AI1[0] + 2;
15   AI1[2] = AI1[1] * 1; AI1[3] = w + 2; AI1[4] = AI1[3] * 1;
16   AI1[5] = AI1[0] * 1; AI1[6] = w * 1; //... and so on ....
17   #pragma omp target data map(to: in[AI1[91]:AI1[92]]) map(tofrom: \
```

```
18    out_h[AI1[109]:AI1[110]],out_v[AI1[109]:AI1[110]]) if(!RST_AI1)
19    #pragma omp target if(!RST_AI1)
20    for(int y = 1; y < H; y++)
21      for(int x = 1; x < W; x++) { /*lines 7 to 10 of Listing 1.1*/ }
22  }
```

Listing 1.4 – Sobel filter implementation in OmpSs

1.6.4 Domain Specific Language

Domain Specific Languages (DSL) are languages specially designed for a specific field such as medical imaging, computer vision, HPC, etc. Generally, they require to rewrite the application to be parallelized in the DSL language. However, they are typically very portable since the code has to be written once and for each target architecture the compiler generates a suitable optimized code. Many DSLs have been proposed for efficient parallelization of different fields. Here, we give a brief description of some of them.

Halide

Halide [39], a DSL for image processing, uses a functional programming paradigm to express kernels in a compact and concise way. Images have no explicit storage but are pure functions that define the value of each pixel. Halide is capable of generating code for various target architectures (multi-core CPU, GPUs, DSP, etc.). The developer needs to specify a schedule that defines how the algorithm should be mapped onto the target architecture in order to obtain an efficient code. Halide's schedule can be manually specified and requires the developer to have a certain degree of architecture and domain knowledge. However, as the schedule is evaluated dynamically by the compiler, it can be altered or even entirely replaced at run time. Halide provides also an auto-scheduling function which allows the compiler to choose the best parallelization strategy according.

HIPAcc

HIPAcc (Heterogeneous Image Processing Acceleration) [40] is a framework for the automatic parallel code generation of image processing algorithms. It consists of a DSL that is embedded into C++ and a source-to-source compiler. Exploiting the compiler, image processing algorithms written in DSL code can be translated into multiple target languages such as CUDA, OpenCL or Renderscript. HIPAcc compiler is based on the Clang/LLVM 3.4 compiler infrastructure. HIPAcc parses C/C++ code and generates an internal Abstract Syntax Tree (AST) representation. Operating on this representation, HIPAcc will generate two kinds of code: host code for managing kernel launches and memory transfers, and device code containing the actual kernel description in the specified target language.

Vulkan

Vulkan [41] initially announced as "OpenGL next generation", is a cross-platform 3D graphics and computing API. Vulkan is specially designed to target real-time $3D$ graphics applications such as video games and interactive media across all platforms. Contrary to OpenGL, Vulkan is intended to offer higher performance, more balanced CPU/GPU usage and parallel tasking.

Numerical Template Toolbox (NT2)

NT2 [42] is an open source C++ library that attempts to simplify the development, debugging and optimization of high-performance computing applications by providing a Matlab like syntax that eases the transition between prototype and actual application. Like with others DSL seen above, the user must rewrite its program using NT2 syntax and data types. NT2 keeps a high level of expressiveness by exploiting the architecture-specific information as early as possible in the code generation process. It selects architectural features from either compiler-based options or user-defined preprocessor. Like in Halide, with NT2 the user can test different optimizations by setting different runtime supports for a given architecture until required performance is partially or completely satisfied.

1.7 Summary

In this chapter, we have remembered the definitions of some key notions of parallel computing. We talked about the two important inflection points that occurred in the computing world: the switch to parallel processors and the rise of GPUs. GPUs are throughput-oriented devices that use hundreds of cores to execute a massive number of threads and utilize hardware multithreading to hide latency. We reviewed GPU architectures and diverse approaches for their programming. Low-level languages, strongly related to the architecture, allow expert programmers to hand-tune their codes to exploit the full capabilities of the GPU. However, the programmer must have substantial knowledge of the hardware details of the GPU in order to achieve good performance. Directive-based programming models, that are relevant to our research, aim to provide interfaces for exploiting the power of GPUs in a more productive and portable manner. Table 1.4 presents a summary of parallel programming approaches discussed in this chapter. OpenMP and OpenACC, the main focus of this thesis, are widely used for accelerators programming. In the next chapter, we will discuss OpenMP and OpenACC specifications and their suitability for GPU programming.

Table 1.4 – Summary of parallel programming approaches

Category	Name	Target	Compiler	Input	Output	Alive	Complexity
Kernel-based languages	CUDA	NVIDIA GPU	nvcc, LLVM/Clang	CUDA code	–	yes	***
	OpenCL	CPU, GPUs, FPGA	clang, clcc ioc64	OpenCL code	–	yes	***
	SYCL	CPU, GPUs, FPGA	triSYCL computeCpp sycl-gtx	SYCL code	–	yes	***
Directive based approaches	HMPP	CPU, GPU, FPGA	CAPS	user can annotate C/C++ code with HMPP directives	Annotated code		**
	OmpSs	CPU, GPU, FPGA	Mercurium Nanos++ runtime	C/C++/Fortran annotated code with OmpSs directives		yes	
	OpenACC	CPU, GPU, FPGA	PGI, Cray cc, Intel icc, etc.	C/C++/Fortran code	Annotated code	yes	***
	OpenMP	CPU, GPU, FPGA	LLVM/Clang, Gcc, Intel icc, etc.	C/C++/Fortran code	Annotated code	yes	***
Automatic parallel code generation tools	Par4All	CPU, GPU	–	C/Fortran code	OpenMP CUDA OpenCL	No	–
	PLUTO	CPU	–	C code	OpenMP code	yes	–
	PPCG	CPU, GPU	–	Serial code	CUDA/OpenCL code	yes	–
	AutoPar	CPU	–	Serial C/C++ code	OpenMP code for CPUs	yes	–
	KernelGen						
	Cetus						
	Dawncc	GPU	–	Serial C/C++ code	C/C++ code annotated with OpenMP 4.0 or OpenACC directives	yes	–
Domain Specific Languages	Halide	CPU, GPUs FPGA, DSP	Halide compiler	Halide code	–	yes	***
	HIPAcc	CPU, GPUs, FPGA	–	HIPAcc code	CPU, CUDA, OpenCL code	yes	***
	Vulkan						
	NT2	CPU, GPUs, DSP		NT2			

Chapter 2

Analysis of OpenMP & OpenACC Specifications

Contents

Nowadays, GPUs are everywhere from mobile devices to supercomputers. As seen in Section 1.6, GPUs can be programmed with low-level models (e.g., CUDA, OpenCL), directives-based approaches (e.g., HMPP, OmpSs, OpenACC, OpenMP, etc.), automatic parallel code generation tools (e.g., Par4All, PPCG, DawnCC, ...), and domain-specific languages (e.g., Halide, HIPAcc, SYCL). Unlike low-level and domain-specific languages, and automatic parallel code generation tools, directive-based approaches allow programmers to provide simple hints, known as "directives", to the compiler, identifying which areas of code to accelerate, without requiring programmers to modify or adapt the underlying code itself. By exposing parallelism to the compiler, directives help the compiler perform the detailed transformation of mapping the computation onto the accelerator. Since OpenMP and OpenACC are the main focus of this dissertation, we describe the major features of their specifications in 2.1 and 2.2. They are categorized into four subjects: execution model, compute's offloading directives, memory model, and data management mechanism. Then, OpenMP and OpenACC specifications comparison is discussed in Section 2.3. As already said, porting of legacy CPU-base code using OpenMP/OpenACC, only requires programmers to add several lines of annotations before the sections where they need to be accelerated, without changing code structures. However, these over-simplified parallel programming models also bring to users limitations which may prevent full use of the available architectural resources, potentially resulting in greatly reduced performance when compared to highly manually tuned CUDA code. This last point is discussed in Section 2.4. The OpenMP and OpenACC specifications use generic terms to describe the heterogeneous programming environment. In general, the host is a CPU and the device is the accelerator (e.g., GPU).

2.1 OpenMP

OpenMP is undoubtedly the most used standard for several years for the parallel programming for shared memory CPUs. OpenMP 4.0 [43] extended the OpenMP shared memory programming model to support accelerators (e.x. GPUs, Intel Xeon Phi). The OpenMP fork-join model was extended with the introduction of device constructs for programming accelerators. Since the release of OpenMP 4.0 in July 2013, OpenMP continues to evolve with the addition of new directives. The current version is OpenMP 5.0 [4] which was released in November 2018. In this section, we will describe OpenMP accelerator execution model, its memory model as well as compute offload and data management directives.

2.1.1 Execution Model

OpenMP's execution model for accelerators assumes that the main program runs on the host, which can offload compute-intensive regions on one or more devices. The execution model chooses by OpenMP, for accelerators programming, is host-centric. In other terms, the host device offloads compute-intensive regions (identified using directives by the programmer) to target devices. Typically, each device has its own threads, which are distinct from host threads or other device threads. The target devices execute the offloaded code regions in parallel. Each offloaded region usually

becomes a kernel running on the device.

When an offloading (or a *target*) directive is encountered, a new target task is generated for execution on the target device. This new task encloses the target region code. Then, the host:

- allocates memory on the target device,
- initiates the data transfer,
- transfers the generated kernel code to the accelerator,
- passes arguments to the kernel function,
- places the kernel on the launching queue,
- waits for completion,
- transfers the results back to the host, and deallocates the memory on the target device. Figure 2.1 shows a schematic illustration of this execution model.

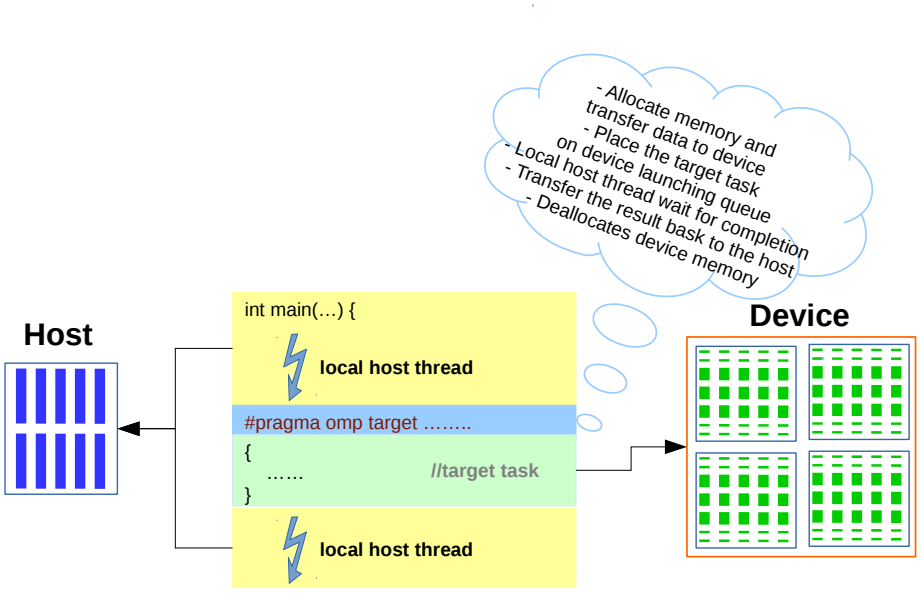


Figure 2.1 – Illustration of OpenMP's execution model for accelerators

Concerning the generated target task, it may be executed sequentially on a target device by an initial thread as if the target region is part of an initial task region that is generated by an implicit parallel region. Finally, OpenMP provides clauses for asynchronous execution of target regions.

2.1.2 Compute Offloading Directives

In OpenMP 4.0 and higher, the programmer can specify the region of code to be executed in an accelerator by using the *target* directive. Indeed, when the *target* directive is encountered a target task is generated for the structured-block following this directive. If data-mapping clauses are specified, a device data environment is created and data are mapped to the device's memory. Then the target task is executed on that device. Results data can be copy back to the host's memory after the target task completion. The *target* directive transfer control of device's code execution to a *single* device thread. The *target* directive accepts following clauses:

- *if (optional):* allows the conditional execution of the target task. To be clear, if the condition is evaluated true, then the target task is executed on the device, otherwise, it is executed on the host.
- *device:* allows choosing the device to be used if more than one accelerator is available.
- *private:* makes each thread has its own private copy of a variable, and modifications made by a thread to its copy are not visible to other threads.
- *firstprivate:* makes each thread has its own instance of a variable, and this instance should be initialized with the value of the variable.
- *map(to | from | tofrom | alloc | release | delete):* specifies how an original variable is mapped from the current host thread's data environment to a corresponding variable in the device data environment.
- *is_device_ptr:* indicates that a list item is a device pointer already in the device data environment and that it should be used directly.
- *defaultmap(to | from | tofrom | alloc | firstprivate | none | default):* explicitly determines the data-mapping attributes of variables that are referenced in a *target* construct.
- *allocate:* allows the user to specify the memory allocator to be used to obtain storage for private variables of the *target construct*.
- *in_reduction:* specifies that there is a reduction operation in the generated target task.
- *nowait:* allows an asynchronous execution of the target task.
- *depend:* to specify data dependency between different part of the program.

Listing 2.1 and Figure 2.2 show how the Sobel 3×3 filter seen in 1.6.1 can be offloaded on a target device by using the *target* directive. From the *target* directive at line 4, the compiler will generate instructions for the target device initialization, memory allocation for *in*, *out_h* and *out_v*, the copy of *in* in device's memory, the launch of target task on device and the copy back of *out_h* and *out_v* to host's memory. The generated target task corresponds to the block of code from line 5 to 13

and will be executed on the target device. The number of threads participating in the target task execution on device is implementation defined, but the target task is executed by a single device thread in general.

```
1  int main (....) {
2    //.....................
3    //ofload the computation of a Sobel 3x3 filter
4    #pragma omp target map(to:in[0:w*h]) map(from:out_h[0:w*h], out_v[0:w*h])
5    for(int x = 1; x < h - 1; x++)
6      for(int y = 1; y < w - 1; y++) {
7        out_v[y + w * x] = -in[(y-1)+w*(x-1)] + in[y+w*(x+1)] -
8        2*in[y+w*(x-1)] + 2*in[y+w*(x+1)] - in[(y+1)+w*(x-1)]
9        + in[(y+1)+w*(x+1)];
10       out_h[y + w * x] = -in[(y-1)+w*(x-1)] - 2*in[(y-1)+w*x]
11       - in[(y-1)+w*(x+1)] + in[(y+1)+w*(x-1)] +
12       2*in[(y+1)+w*x] + in[(y+1)+w*(x+1)];
13     }
14   //.....................
15 }
```

Listing 2.1 – Accelerate the Sobel 3x3 filter with the target directive

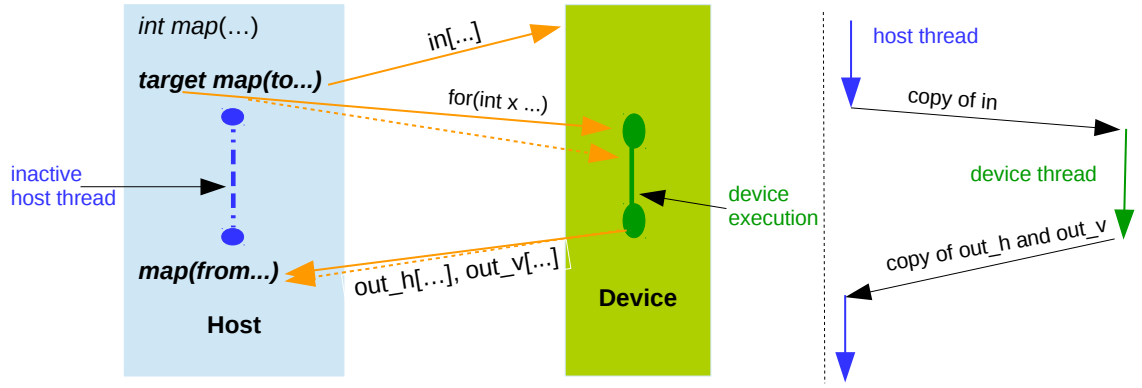


Figure 2.2 – Offloading of Sobel 3×3 filter computation with the *target* directive. The offloaded region is executed by a single device thread.

In addition to the *target* directive, OpenMP provides two other directives to describe more in detail how the code to be offloaded must be parallelized. These two additional directives are: *teams* and *distribute*.

The teams construct

When the device master thread encounters the *teams construct*, a league of thread teams is created and the master thread in each team executes the *teams region*. In addition to the clauses *private*, *firstprivate* and *allocate*, the *teams* construct accept the clauses:

- *shared:* makes variables in the list to be shared among all the threads in league of thread teams. All threads access the same storage area for shared variables.
- *default(shared | none):* allows the user to affect the data-sharing attribute of the variables appeared in the *teams construct*.
- *reduction:* specifies that a variable is used to perform some forms of recurrence calculations in parallel.
- *thread_limit:* allows the user to set the number of threads participating in the contention group that each team initiates. However, the implementation, i.e. the compiler, is free to choose another number that can be less than or equal to the value specified in the *thread_limit* clause.
- *num_teams:* can be used to set the number of teams that will be created, but the compiler is free to choose another number that can be less than or equal to the value specified by the user.

There is no implicit barrier at the end of a teams construct. Listing 2.2 shows the Sobel 3×3 (seen in Listing 2.1) offloading with the target and teams directives. Figure 2.3 illustrates performed operations when the device master thread encounters the *teams construct*.

```
1  int main (....) {
2    //.....................
3    //ofload the computation of a Sobel 3x3 filter
4    #pragma omp target map(to:in[0:w*h]) map(from:out_h[0:w*h], out_v[0:w*h])
5    #pragma omp teams num_teams(h) thread_limit(128)
6    for(int x = 1; x < h - 1; x++)
7      for(int y = 1; y < w - 1; y++) {
8        out_v[y + w * x] = -in[(y-1)+w*(x-1)] + in[y+w*(x+1)] -
9        2*in[y+w*(x-1)] + 2*in[y+w*(x+1)] - in[(y+1)+w*(x-1)]
10       + in[(y+1)+w*(x+1)];
11       out_h[y + w * x] = -in[(y-1)+w*(x-1)] - 2*in[(y-1)+w*x]
12       - in[(y-1)+w*(x+1)] + in[(y+1)+w*(x-1)] +
13       2*in[(y+1)+w*x] + in[(y+1)+w*(x+1)];
14     }
15   //.....................
16 }
```

Listing 2.2 – Accelerate the Sobel 3x3 filter with the target and teams directives

The distribute construct

The *distribute construct*, that must be strictly nested inside a *teams directive*, specifies that the iterations of one (or more loops if they are collapsed) will be executed by the league of teams (previously created from the *teams construct*). The iterations of the loop(s) associated to a *distribute directive* are distributed across the threads of teams that execute the *teams region*.

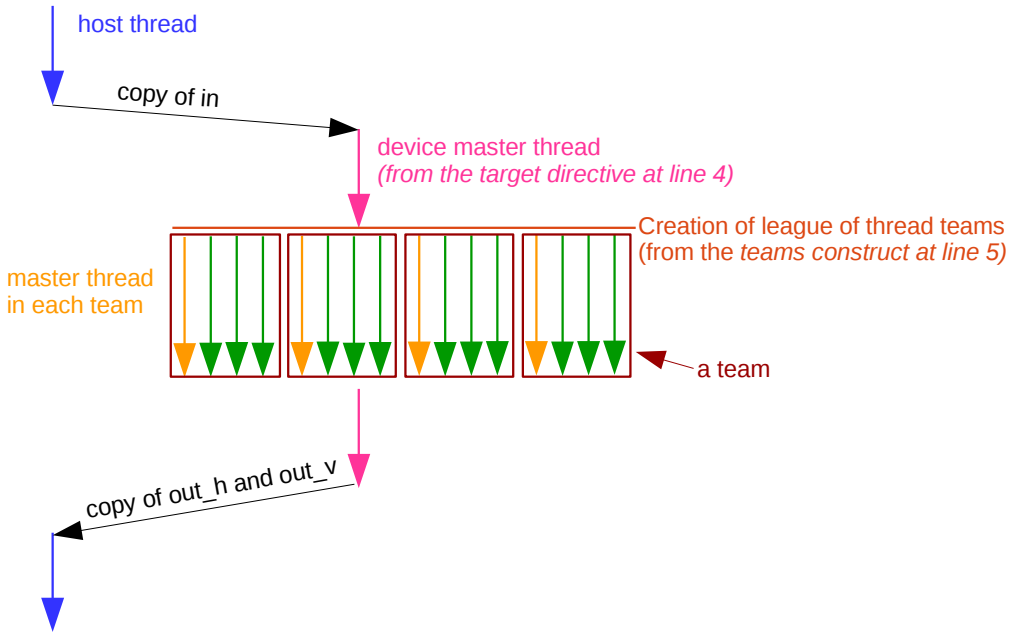


Figure 2.3 – Offloading of Sobel 3×3 filter computation with the *target* and *teams* directives. Although a league of thread teams is created, only the device master thread executes the target task. Iterations of loops x and y are not distributed among threads.

In addition to the clauses *private*, *firstprivate*, and *allocate*, the *distribute* construct accept the following clauses:

- *lastprivate:* makes a private variable is updated after the end of the *distribute construct*.
- *collapse:* can be used to specify how many loops are associated with the *distribute construct*.
- *dist_schedule(chunk_size):* may be used to specify that the iterations of associated loop(s) will be divided into chunks of size *chunk_size*. Chunks will be assigned to the teams in a round-robin fashion in the order of the team number.

As for the *teams directive*, there is no implicit barrier at the end of a *distribute construct*. Listing 2.3 shows the Sobel 3×3 filter, of Listing 2.2, parallelization with the *distribute construct*. By adding the *distribute directive* with the clause *collapse* at line 6, iterations of loops x and y will be collapsed and distributed among teams. However, only the master thread of each teams will executes the teams iterations. Since, we set the number of threads within each teams to 128, the *chunk_size* can also be set to 128.

```
1 int main (....) {
2   //.....................
3   //ofload the computation of a Sobel 3x3 filter
4   #pragma omp target map(to:in[0:w*h]) map(from:out_h[0:w*h], out_v[0:w*h])
5   #pragma omp teams num_teams(h) thread_limit(128)
6   #pragma omp distribute collapse(2) dist_schedule(128)
```

```
7   for(int x = 1; x < h - 1; x++)
8     for(int y = 1; y < w - 1; y++) {
9       out_v[y + w * x] = -in[(y-1)+w*(x-1)] + in[y+w*(x+1)] -
10      2*in[y+w*(x-1)] + 2*in[y+w*(x+1)] - in[(y+1)+w*(x-1)]
11      + in[(y+1)+w*(x+1)];
12      out_h[y + w * x] = -in[(y-1)+w*(x-1)] - 2*in[(y-1)+w*x]
13      - in[(y-1)+w*(x+1)] + in[(y+1)+w*(x-1)] +
14      2*in[(y+1)+w*x] + in[(y+1)+w*(x+1)];
15    }
16  //.....................
17 }
```

Listing 2.3 – Accelerate the Sobel 3x3 filter with the target + teams + distribute constructs

In order to allow all threads within a teams participate to teams's iterations execution, the traditional *parallel for* construct must be add to the *distribute* directive at line 6 of Listing 2.3. Lines 1 to 7 in Listing 2.4 illustrate this case.

```
1  #pragma omp target map(to:in[0:w*h]) map(from:out_h[0:w*h], out_v[0:w*h])
2  #pragma omp teams num_teams(h) thread_limit(128)
3  #pragma omp distribute parallel for collapse(2) dist_schedule(128)
4  for(int x = 1; x < h - 1; x++)
5    for(int y = 1; y < w - 1; y++) { //all threads within a teams participate
         to teams region execution
6      //.....................
7    }
8
9  #pragma omp target map(to:in[0:w*h]) map(from:out_h[0:w*h], out_v[0:w*h])
10 #pragma omp teams num_teams(h) thread_limit(128)
11 #pragma omp distribute dist_schedule(128)
12 for(int x = 1; x < h - 1; x++) {
13   //only master threads of teams execute teams's iterations
14   //.........
15   #pragma omp parallel for
16   for(int y = 1; y < w - 1; y++) { //all threads within a teams  execute
       this loop in parallel
17     //.........
18   }
19 }
```

Listing 2.4 – Fully parallelization with target + teams + distribute + parallel + for

Loops are not always ready to be collapsed, in such case, the *parallel for* construct can be added to nested loops. Lines 9 to 19 in Listing 2.4 shows an example of the parallelization of the nested loop. When loops instructions are regular, they can be parallelized in a SIMD fashion by using the *simd* construct instead of the *parallel for* directive.

OpenMP directives can be combined. Thus, any of the constructs *parallel*, worksharing-loop *for*, *simd* and *loop* can be combined with the *target*, *teams* and *distribute* directives to specify how

nested loops should be parallelized. Figure 2.4 shows different possible combination of directives, from left to right.

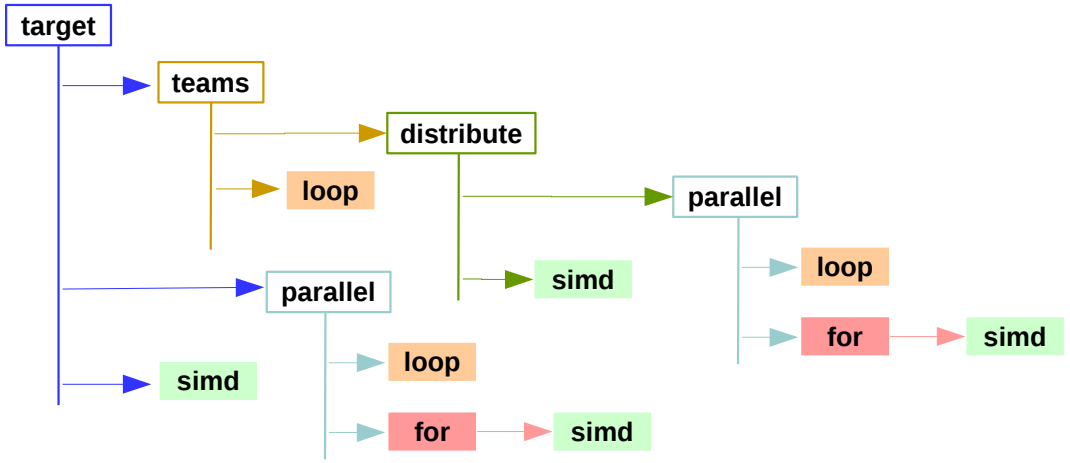


Figure 2.4 – Possible combinations of OpenMP construct for accelerators programming

On a combined construct that includes *target* and *teams* constructs, the values of *num_teams* and *thread_limit* clauses are evaluated on the host device on entry to the target construct.

2.1.3 Memory Model

OpenMP's memory model, for accelerators programming, assumes that the host and the device may have separate memories as illustrated in Figure 2.5. Therefore, the data movement between host and device memories may be explicitly managed. Since host and target devices are supposed to have separate memories, each device has a device data environment that is defined by its implicit target data region. When a target task is running on a target device, references to the host's original variable refer to the corresponding variable in the device data environment. In this context, if an original variable is not currently mapped and a corresponding variable does not exist in the device data environment then accesses to the original variable may result in an illegal access error and lead to a segmentation fault.

OpenMP provides a large set of data mapping directives, clauses, and runtime library routines as part of its standard. Data-mapping directives, clauses, and runtime library routines determine how an original variable in the host's data environment is mapped to a corresponding variable in the device's data environment.

2.1.4 Data-Mapping Directives, Clauses and Runtime Library Routines

Since OpenMP's memory model assumes that the host and the device may have separate memories, data used inside offloaded codes must be mapped into device memory. In addition to using the map

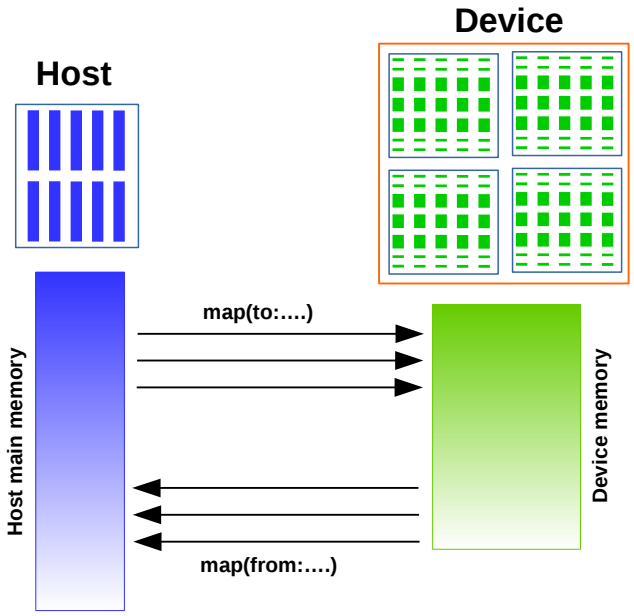


Figure 2.5 – OpenMP memory model for accelerators programming

clause with the *target* directive (as seen in 2.1.2), OpenMP provides three other ways to map the host's data to the device's memory.

The first way is to use the *target data* directive. This directive maps variables, for the extent of the offloaded region, to the device's data environment depending on the data-mapping clauses used. Listing 2.5 shows an example of using the *target data* directive. This directive accepts following data-mapping clauses:

- *map(to | from | tofrom | alloc)*,
- *use_device_ptr(ptr-list):* pointers listed in this clause are privatized and the device pointers to the corresponding list items in the device data environment are assigned into the private versions,
- and *use_device_addr(list):* variable listed in this clause have the address of the corresponding object in the device data environment inside the *target* construct.

```
1  int main (....) {
2    //.....................
3    #pragma omp target data map(to:in[0:w*h]) \
4    map(from:out_h[0:w*h], out_v[0:w*h])
5    { //in is copied from host to device
6      #pragma omp target teams distribute parallel for collapse(2)
7      for(int x = 1; x < h - 1; x++)
8        for(int y = 1; y < w - 1; y++)
9          //Sobel 3x3 filter computation
10
```

```
11      //other computation on device that uses out_h and out_v .......
12    } //out_h and out_v are copied from device to host
13  }
```

Listing 2.5 – Example of using the target data directive

The second way is to use the *target enter data* and *target exit data* directives as showed in Listing 2.6. The *target enter data* construct specifies that variables are mapped to a device data environment. This construct accepts the following clauses: *if*, *device*, *map(to | alloc)*, *depend*, and *nowait*.

The *target exit data* construct specifies that list items are unmapped from a device data environment according to the data-mapping clauses used. This construct accepts same clauses as the *target enter data* directive. However, the *map type* must be either *from*, *release*, or *delete*.

```
1  int main (....) {
2    //.....................
3    #pragma omp target enter data map(to:in[0:w*h]) //in is copied from host
         to device
4    #pragma omp target teams distribute parallel for collapse(2)
5    for(int x = 1; x < h - 1; x++)
6      for(int y = 1; y < w - 1; y++)
7        //Sobel 3x3 filter computation
8
9    //other computation on device that uses out_h and out_v .......
10   #pragma omp target exit data map(from:out_h[0:w*h], out_v[0:w*h])
11 }
```

Listing 2.6 – Example of using the target enter and exit data directives

Finally, the last way is to use OpenMP's device memory routines to allocate data, to copy them to or from device memory and to release allocated memory. A detailed description of all OpenMP's device memory routines available can be found in [4]. Listing 2.7 shows examples of using OpenMP's device memory routines.

```
1  int main (....) {
2    //.....................
3    int h = omp_get_initial_device(); //return the id of the current host
4    int t = omp_get_default_device(); // return the id of the default device
5    uint8_t *d_in    = (uint8_t*)omp_target_alloc(w*h*sizeof(uint8_t), t);
6    uint8_t *d_out_h = (uint8_t*)omp_target_alloc(w*h*sizeof(uint8_t), t);
7    uint8_t *d_out_v = (uint8_t*)omp_target_alloc(w*h*sizeof(uint8_t), t);
8    omp_target_memcpy(d_in, idn, w * h * sizeof(uint8_t), 0, 0, t, h);
9    #pragma omp target teams distribute parallel for collapse(2) \
10   is_device_ptr(d_in, d_out_h, d_out_v)
11   for(int x = 1; x < h - 1; x++)
12     for(int y = 1; y < w - 1; y++)
13       //Sobel 3x3 filter computation
14   //copy back data to host
15   omp_target_memcpy(d_out_h, out_h, w*h*sizeof(uint8_t), 0, 0, t, h);
16   omp_target_memcpy(d_out_v, out_v, w*h*sizeof(uint8_t), 0, 0, t, h);
```

```
17   omp_target_free(d_in, t); //release allocated memory
18   omp_target_free(d_out_h, t); omp_target_free(d_out_v, t);
19 }
```

Listing 2.7 – Example of using OpenMP's device memory routines

When the same data is alternately used in both the host's code and device's code, the *target update* construct may be used to make the corresponding list items in the device data environment consistent with their original list items, according to the specified motion clauses. The motion clause must be either *to* or *from*. The *target update* directive has the clauses: *if*, *device*, *depend*, and *nowait*. Listing 2.8 shows an example of using the target update directive.

```
1  //.....................
2  #pragma omp target data map(to:in[0:w*h]) \
3  map(from:out_h[0:w*h], out_v[0:w*h])
4  {
5    #pragma omp target teams distribute parallel for collapse(2)
6    for(int x = 1; x < h - 1; x++)
7      for(int y = 1; y < w - 1; y++)
8        //Sobel 3x3 filter computation
9    #pragma omp target update from(out_h[0:w*h])
10   //other computation on host that modifies out_h ......
11   #pragma omp target update to(out_h[0:w*h])
12   //other computation on device that modifies out_h ......
13 }
```

Listing 2.8 – Example of using the target update directive

OpenMP provides also the *declare target* directive that can be used to specify that variables or routines are mapped to a device for later use.

2.1.5 OpenMP's Specifications Evolution from CPU to Accelerators Programming

The OpenMP standard is constantly evolving, according to computer systems evolution, since its first releases. Each new version adds new directives, clauses, routines, etc. We summarize the main evolutions of the OpenMP standard.

OpenMP 4.0

OpenMP's releases before the 4.0 were designed for parallel processing on multi-processors host systems. The release 4.0 extended OpenMP traditional work-sharing model to accelerator programming, with new directives *target*, *target data*, *target update*, *declare target*, *teams* and *distribute*. Although multi-devices targeting was possible, this first release was very rudimentary. For

example, the only way to move data between the host and the device memory was the *map* clause. Therefore, data allocated with other low-level languages such as CUDA could not be used inside a target region without first copying them with the *map* clause. Furthermore, only the *target data* directive could create a device data environment for the extent of the offload region. In other terms, if several regions of code had to be offloaded and that these regions have some common data, they had to be grouped in a single structured-block following the *target data* directive (as seen in Listing 2.5) so that common data are copied only once.

OpenMP 4.5

The release of version 4.5 solved some of these limitations. First, the impossibility to use allocated data on the device with other API such as CUDA has been overcome with the addition of the clauses *use_device_ptr* and *is_device_ptr*. Moreover, OpenMP 4.5 adds device memory routines to allocate, copy and free data on device memory (as seen in Listing 2.7). OpenMP 4.5 has also added the *target enter data* and *target exit data* directives which allow to map (or unmap) data to (or from) a device data environment for several offloaded regions or the whole program (see example in Listing 2.6).

The clauses *nowait* and *depend* have been added to the *target* directive to allow respectively the asynchronous execution of targets regions and the creation of dependency between offloaded regions.

Finally, the composite constructs *target parallel*, *target parallel for*, *target parallel for simd* and *target simd* were added. These constructs are shortcut for specifying a *target* construct containing a *parallel*, a *parallel for*, a *parallel for simd*, or a *simd* construct and no other statements.

OpenMP 5.0

OpenMP 5.0 comes with several new features for both multi-processors CPU and accelerators. Some of the new directives added are specific to accelerators. Among these directives, there is the *requires* directive for specifying requirements for the execution of all code in a compilation unit.

It is well known that optimizing the cost of moving data between the host and the target device is a major challenge in accelerator programming. Some accelerators manufacturers (e.x. NVIDIA) provide an *unified memory access* programming model to simplify the complexities of memory management. This technology allows applications to allocate data that can be read or written from code running on either host or devices. The clauses *unified_address* and *unified_shared_memory* of the *requires* directive allow to exploit the *unified memory access* programming model.

Another interesting new directive is *declare mapper*. With this directive, it is possible to define a user-defined mapper for a given type for use in a map clause.

The combination of *teams* directive with the newly added *loop* directive allows to parallelize a teams region otherwise than with the *distribute* directive.

Accelerators may have complex memory hierarchies as shown in GPUs architecture (with registers,

shared memory, constant memory, texture memory, L1 and L2 caches, local memory, and global memory) in Sections 1.5.1 and 1.6.1. OpenMP 5.0 provide predefined memory spaces to use with the *allocate* directive or the two clauses *allocate* and *uses_allocators* to specify how a set of variables would be allocated. Since OpenMP 5.0 is relatively new, it will be necessary to wait for its first implementations to appear to know how the predefined memory spaces would be mapped to accelerators memory hierarchies by the implementers. Table 2.1 shows the predefined memory spaces added by the OpenMP 5.0. The third column is a proposition of the mapping of these predefined memory spaces to NVIDIA GPU memory hierarchies.

Table 2.1 – Predefined memory spaces added by the OpenMP 5.0 (Table 2.8 of [4])

Memory space	**Storage selection intent**	**Mapping to GPU**
omp_default_mem_space	system default storage	GPU global memory
omp_large_cap_mem_space	storage with large capacity	
omp_const_mem_space	storage optimized for constant variables	GPU constant memory
omp_high_bw_mem_space	storage with high bandwidth	
omp_low_lat_mem_space	storage with low latency	

Table 2.2 summarizes OpenMP directives evolutions (by adding new directives) from version 3.1 to 5.0.

Table 2.2 – Summary of OpenMP's concepts evolution from CPU to accelerators programming

Version	**New concepts**
OpenMP 4.0 (2013)	- SIMD vectorization (*simd* variant constructs), - Offloading a code region to an accelerator (*target* variant constructs), - Allowing tasks dependency creation (*depend* clause), - Makes deep task synchronization more flexible (*taskgroup* construct), - Allowing user-defined reduction (*declare reduction* construct), - Allowing cancellation points creation (*cancel* variants constructs)
OpenMP 4.5 (2015)	- Allowing nestable parallel loops that create OpenMP tasks (*taskloop* variant constructs), - Allowing asynchronous execution of *target regions* (*nowait* and *depend* clauses), - Makes all scalar variables in *target regions firstprivate*, - Allowing explicit data sharing attributes to *target* construct (*private*, *default-map*, ...), - Allowing mapping of global variables on device (*declare target* construct), - Adding support for unstructured data mapping for devices (*enter/exit data* constructs), - Allowing structures elements handling, - Allowing device explicit allocation, deallocation, and memory transfers
OpenMP 5.0 (2018)	- Adding support for applications that require implementation-specific features, - Allowing runtime control of the execution of device constructs, - Adding supports for inclusive and exclusive scan computations, - Adding support for tasks reduction, - Adding supports for different kinds of memories, - Allowing reverse offloading

2.1.6 OpenMP 4.0/4.5 Implementations

There are some implementations of OpenMP for accelerators programming.

- Among the commercial compilers, there is Cray cc compiler (version 8.7) that supports all features of OpenMP 4.5 and can target accelerators such as the NVIDIA K20X.
- IBM xl compiler (version $16.1.1$) fully support OpenMP 4.5 features including the target constructs, but can only be installed on systems whose host is IBM POWER CPU.
- Intel icc compiler (version ≥ 17.0) has support for OpenMP 4.5 and it can target for example Intel Xeon Phi devices.
- NVIDIA PGI pgcc/pgc++ supports OpenMP 4.5 but, at the time of writing this document, *target* regions are implemented with default support for the multi-core host as the target.
- Concerning non-commercial and academic compilers, there is GNU Compiler Collection (version ≥ 4.9) which official releases have support to target Intel Xeon Phi devices.
- The academic ROSE compiler supports some features of OpenMP 4.0 accelerator directives to target NVIDIA GPUs, but it is not yet a mature implementation.
- The LLVM/Clang compiler fully supports OpenMP 4.5 (official release of version 7.0 and a trunk branch of version 4.0) to target NVIDIA GPUs.

2.2 OpenACC

OpenACC [8] is a directive-based performance-portable programming model. Unlike OpenMP, OpenACC was specifically designed for accelerators programming since its first release 1.0 in 2011. In this section, we are going to describe the OpenACC execution model, its memory model as well as compute offload directives.

2.2.1 Execution Model

As OpenMP, OpenACC's execution model is a host-directed execution with an attached accelerator (e.x., GPU, Intel Xeon Phi). The compute-intensive regions are offloaded by a host thread on an attached accelerator or executed on the host (if the device is the host itself) under control of a host thread. The compute-intensive regions to be offloaded are either parallel regions (or work-sharing loops), or kernels regions (one or more distinct loops), or serial regions (blocks of sequential code). If a compute-intensive regions must be offloaded, the host thread is responsible for:

- allocating memory on the accelerator device
- initiating data transfer

- sending the code to the accelerator
- passing arguments to the compute region
- queuing the accelerator code
- waiting for completion
- transferring results back to the host
- deallocating memory

Figure 2.6 depicts OpenACC execution model.

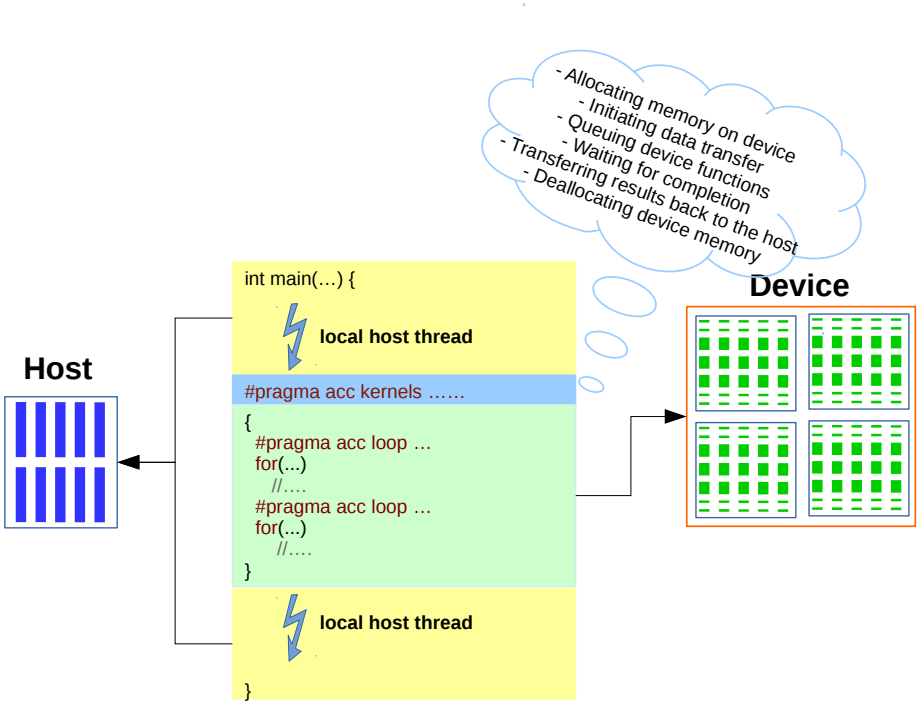


Figure 2.6 – Illustration of OpenACC execution model

Most current parallel systems support two or three levels of parallelism since most of them are constituted by multiple execution units, each of which can execute multiple threads. OpenACC provides these three levels of parallelism via *gang*, *worker* and *vector* parallelism.

- Gang level is coarse-grain parallelism which is fully parallel execution across execution units.
- Worker level is fine-grain parallelism which is usually implemented as multiple threads execution within each execution unit. Workers are created by gangs.
- Vector level is for SIMD or vector operations within a worker.

Thus, a *gang* may contain one or more worker which can also contain one or more vector. Therefore, on an accelerator device, a single vector lane of a single worker of a single gang is called a *device thread*.

Most accelerators support asynchronous execution with respect to the host device. Like OpenMP, OpenACC provides clauses and directives that allowing asynchronous execution on such devices.

2.2.2 Compute Offloading Directives

OpenACC provides three offloading directives which are *parallel*, *kernels* and *serial*. They are described below.

The parallel construct

The *parallel* construct provides a prescriptive language feature that allows the user to explicitly control compiler translation. The entire structured-block, following the *parallel* directive, becomes a single computational kernel running on the current accelerator. In fact, when a host thread encounters an parallel construct, a number of gangs of workers are created to execute the *parallel region* on the accelerator. In addition to *if*, *private*, *reduction*, *firstprivate*, *deviceptr* (equivalent of OpenMP's *is_device_ptr* and *use_device_ptr*), and *default(none | present)* clauses, the *parallel* construct accepts the following clauses:

- *async (integer):* allows the local thread continues with the code following the *parallel* construct while the offloaded code may be processed on device asynchronously. The argument of the *async* clause can be used to create different execution queues on device.
- *wait (integer):* in the absence of this clause, the kernel may be enqueued or launched or executed immediately on the device. However, if this clause appears (with or without an argument) in a *parallel* construct, the kernel may not be launched or executed until all operations enqueued up to this point by this thread on the associated asynchronous device activity queues have completed. To simplify, the *wait* clause allow to create synchronization points.
- *num_gangs:* can be used to define the number of parallel gangs that will execute the *parallel region*.
- *num_workers:* allows user to set the number of workers within each gang.
- *vector_length:* may be used to define the number of vector lanes.
- *device_type(*) or device_type(list-of-device):* allows user to specify which clauses or clause arguments must be applied to which device. Clauses that precede any *device_type* are *default clauses* (i.e., are applied on all devices available in the system), while clauses that follow a *device_type* are *device-specific clauses*.

- *copy:* allows to copy a variable to the device at entry to the *parallel region* and from the device at exit from the region. This clause is the equivalent of OpenMP's *map(tofrom:)* clause.

- *copyin:* can be used to inform the compiler that a variable must be copied to the device before starting the computation. This clause is the equivalent of OpenMP's *map(to:)* clause.

- *copyout:* allows to inform the compiler that a variable must be copied from the device before exiting from the *parallel region*. This clause is the equivalent of OpenMP's *map(from:)* clause.

- *create:* can be used to allocate memory on device for variables or arrays that are used only in the *parallel region*. This clause is the equivalent of OpenMP's *map(alloc:)* clause.

- *no_create:* allows the user to inform the compiler that it is not necessary to allocate memory for a variable.

- *present:* allows to specify that a variable is already present in the current device memory.

- *attach:* can be used to attach a pointer in device memory to the device copy of the data by initiating an update for the pointer in device memory to point to the device copy of the data.

There is an implicit barrier at the end of the *parallel* construct. Listing 2.9 shows an example of offloading the Sobel 3×3 filter (seen in Listing 2.1) with the *parallel* construct.

```
1  int main (....) {
2    //.....................
3    #pragma acc parallel copyin(in[0:w*h]) copyout(out_h[0:w*h],out_v[0:w*h])
4    {
5      for(int x = 1; x < h - 1; x++)
6        for(int y = 1; y < w - 1; y++) {
7          //Sobel 3x3 filter computation .......
8        }
9      //Other computation on device that may use in, out_h or out_v
10   } //out_h and out_v are copied to host
11   //.....................
12 }
```

Listing 2.9 – Accelerate the Sobel 3x3 filter with the OpenACC parallel construct

Figure 2.7 depicts a schematic execution of the code showed in Listing 2.9. If there is not any other directive within the structured-block (as it is the case in Listing 2.9) to specify how the offloaded code must be parallelized, then all gangs will execute all the code within the region redundantly. However, the implementation (i.e., compiler) will typically generate a kernel with a single gang of a single worker if it fails to parallelize the code within the region. Thus, inner loops (lines 5 and 6 in Listing 2.9) will be executed sequentially.

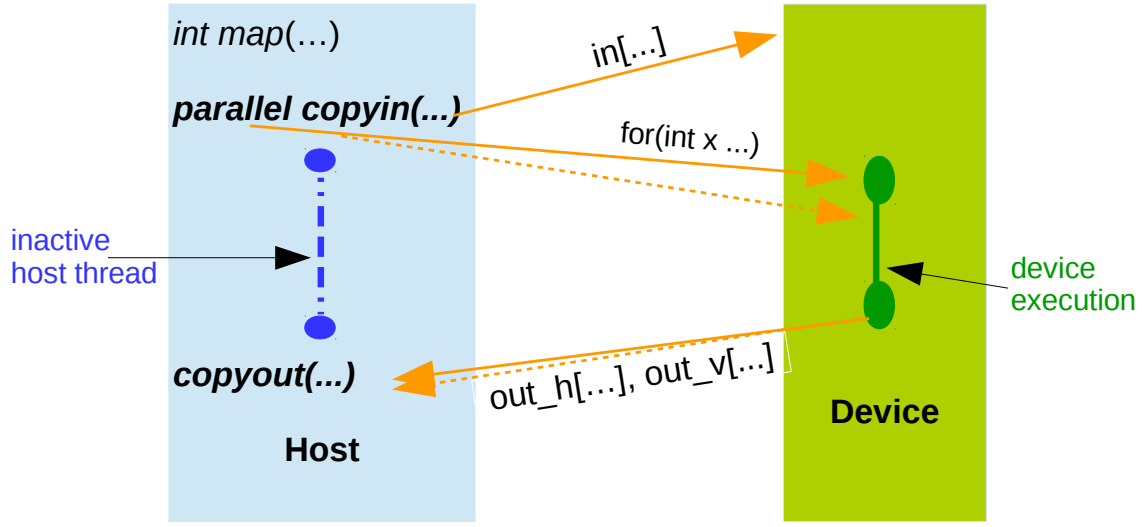


Figure 2.7 – Illustration of Sobel 3×3 filter computation offloading with the *parallel* construct.

The kernels construct

Unlike the *parallel* directive, *kernels* construct provides a descriptive capability that allows the compiler to take more control over code translation. The compiler may split the code in the *kernels region* into a sequence of accelerator kernels. Typically, each loop nest becomes a distinct kernel. When a local thread encounters a *kernels* construct, it launches the sequence of kernels in order on the device. As *parallel* construct, there is an implicit barrier at the end of a *kernel* construct. The *kernels* construct has the same clauses (with the same meaning) as the *parallel* construct, except the *private*, *reduction*, and *firstprivate* clauses. Listing 2.10 shows the *kernels* construct version of code seen in Listing 2.9.

```
1  int main (....) {
2    //.....................
3    #pragma acc kernels copyin(in[0:w*h]) copyout(out_h[0:w*h],out_v[0:w*h])
4    {
5      for(int x = 1; x < h - 1; x++)
6        for(int y = 1; y < w - 1; y++) {
7          //Sobel 3x3 filter computation .......
8        }
9      //Other computation on device that may use in, out_h or out_v
10   } //out_h and out_v are copied to host
11   //.....................
12 }
```

Listing 2.10 – Accelerate the Sobel 3x3 filter with the OpenACC kernels construct

Unlike the *parallel* construct if there are not any other directives on nested loops within the *kernels region*, then these loops (or kernels) will be executed sequentially by a single device thread if the compiler fails to parallelize them.

The serial construct

The third offloading directive provided by OpenACC is the *serial* construct. This directive defines a region of the program that will be executed sequentially on the current device. When the local thread encounters a *serial* construct, one gang of one worker with a vector length of one is created to execute the *serial region* sequentially on the current accelerator. In fact, the directive *serial* is equivalent to a *parallel* construct whose clauses num_gangs, num_workers, vector_length have the integer value "1" as argument. The *serial* construct has same the clauses as the *parallel* construct, except the *num_gangs*, *num_workers*, and *vector_length* clauses.

The loop construct

Unlike OpenMP which has the *distribute*, *for*, *simd*, and *loop* directives to describe the parallelization mode associated to a loop, OpenACC only has the *loop* construct. The OpenACC's *loop* construct describes what type of parallelism (gang, worker or vector) must be used to parallelize the loop that immediately follows this directive. The *loop* directive can be combined with any of *parallel*, *kernel* and *serial* constructs. In addition to *reduction*, *collapse*, *private*, and *device_type* clauses, the *loop* construct accepts the following clauses:

- *gang*, *worker*, *vector:* their behavior depend of the parent compute construct, which can be the *parallel* construct ro the *kernels* directive.
  - If the parent of the *loop* construct is a *parallel* directive, then *gang* and *worker* clauses specify that the iterations of the associated loop(s) must be executed in parallel by distributing loop(s)'s iterations among gangs created by the *parallel* construct, and each gang's iterations are to be divided among this gang workers. The vector clause specifies that the iterations of the associated loop(s) are to be executed in vector or SIMD mode.
  - When the parent of the *loop* construct is a *kernels* directive, the gang clause specifies that the iterations of the associated loop(s) are to be executed in parallel across the gangs. The worker clause indicates that the associated loop(s)'s iterations are to be executed in parallel across the workers within a single gang. The vector clause specifies that the iterations of the associated loop(s) must be executed with vector or SIMD processing.
- *seq:* can be used to specify that the associated loop(s) is(are) to be executed sequentially.
- *auto:* allows the compiler to be freedom to choose the best parallelization strategy (either gang, or vector, or worker) for the associated loop(s).
- *tile:* that the implementation should split each loop in the loop nest into two loops, with an outer set of *tile* loops and an inner set of *element* loops.
- *independent:* must be used to inform the compiler that the iterations of the associated loop(s) are data-independent with respect to each other.

Listing 2.11 shows an example of parallelizing nested loops of the Sobel 3×3 filter kernel seen in Listing 2.10.

```
1  int main (....) {
2    //.....................
3    #pragma acc kernels copyin(in[0:w*h]) copyout(out_h[0:w*h],out_v[0:w*h])
4    {
5      #pragma acc loop independent
6      for(int x = 1; x < h - 1; x++)
7        #pragma acc loop independent
8        for(int y = 1; y < w - 1; y++) {
9          //Sobel 3x3 filter computation .......
10       }
11     //Other computation on device that may use in, out_h or out_v
12   } //out_h and out_v are copied to host
13   //.....................
14 }
```

Listing 2.11 – Accelerate the Sobel 3x3 filter with the OpenACC kernels and loop constructs

OpenACC also provides the *declare* and *routine* directives. The first creates a visible device copy of the variable while the last must be used to tell the compiler to compile a given procedure for an accelerator as well as for the host. In addition to *seq* and *device_type* clauses, the *routine* directive has also the following clauses:

- *gang:* must be used to indicate that the procedure contains, may contain, or may call another procedure that contains a *loop* construct with a *gang* clause.
- *worker:* specifies that the procedure contains, may contain, or may call another procedure that contains a *loop* construct with a *worker* clause.
- *vector:* indicates that the procedure contains, may contain, or may call another procedure that contains a *loop* directive with the *vector* clause
- *bind:* can be used to specify the name to use when calling the procedure.
- *nohost:* informs the compiler not to compile a version of this procedure for the host.

In addition to the *wait* clause, OpenACC provides a *wait* directive that causes the local thread to wait for completion of asynchronous operations on the current device or causes one device activity queue to synchronize with one or more other activity queues on the current device.

2.2.3 Memory Model

As OpenMP, OpenACC's memory model assumes that host and accelerators have separate memory. Therefore, the host thread may not be able to read or write device memory directly because

it is not mapped into the host thread's virtual memory space. All data movement between host memory and accelerator memory must be performed by the host thread. This goes through system calls that explicitly move data between the separate memories. As in OpenMP, data movement can be managed by the compiler, based on directives and clauses from the programmer.

2.2.4 Data-Mapping Directives, Clauses and Runtime Library Routines

As OpenMP, OpenACC provides directives, clauses and API for creating a device data environment and managing data transfer between host and device memories.

The directives *data*, *enter data*, *exit data* and *update* have same behavior as their OpenMP equivalents:*target data*, *target enter data*, *target exit data*, and *target update data* (see sub Section 2.1.4). Listing 2.12 shows an example of applying the OpenACC's *data* construct to Sobel 3×3 filter example seen in Listing 2.10.

```
1  int main (....) {
2    //.....................
3    #pragma acc data copyin(in[0:w*h]) copyout(out_h[0:w*h],out_v[0:w*h])
4    { //create a device data environnement
5      #pragma acc kernels loop independent tile(128,128)
6      for(int x = 1; x < h - 1; x++)
7        for(int y = 1; y < w - 1; y++) {
8          //Sobel 3x3 filter computation .......
9        }
10     //Other computation on device that may use in, out_h or out_v
11   } //out_h and out_v are copied to host
12   //.....................
13 }
```

Listing 2.12 – Example of using OpenACC data directive

Like OpenMP, OpenACC has its own runtime library routines for allocating or deallocating memory and transferring data. However, unlike OpenMP, OpenACC's data transfer routines are available in both asynchronous and synchronous modes. Listing 2.13 shows OpenACC version of Listing 2.7, when OpenMP's device memory routines are replaced by those of OpenACC.

```
1  int main (....) {
2    //.....................
3    uint8_t *d_in    = (uint8_t*)acc_malloc(w*h*sizeof(uint8_t));
4    uint8_t *d_out_h = (uint8_t*)acc_malloc(w*h*sizeof(uint8_t));
5    uint8_t *d_out_v = (uint8_t*)acc_malloc(w*h*sizeof(uint8_t));
6    acc_memcpy_to_device(d_in, idn, w * h * sizeof(uint8_t));
7    #pragma acc kernels loop independent collapse(2)
8    for(int x = 1; x < h - 1; x++)
9      for(int y = 1; y < w - 1; y++) {
10       //Sobel 3x3 filter computation .......
11     }
12   //copy back data to host
```

```
13    acc_memcpy_from_device_async(d_out_h, out_h, w*h*sizeof(uint8_t));
14    acc_memcpy_from_device_async(d_out_v, out_v, w*h*sizeof(uint8_t));
15    //other computation on CPU that uses other data than out_h/v .....
16    acc_free(d_in); //release allocated memory
17    acc_free(d_out_h); acc_free(d_out_v);
18  }
```

Listing 2.13 – Example of using OpenACC's device memory routines

Unlike OpenMP, OpenACC provides the *host_data* directive. This construct must be used to make the address of data in device memory available in host code. Furthermore, OpenACC provides the *cache* directive which must be used to specify array elements or sub-arrays that should be fetched into the "highest level" of the cache for the body of the loop. Finally, like OpenMP, OpenACC provide *atomic* directive for atomic operations.

2.2.5 OpenACC's Specifications Evolution Since the Release 1.0

OpenACC has been specially designed for accelerators programming since its first release. Although OpenACC 1.0 came with a few sets of directives, it was more complete than OpenMP 4.0.

OpenACC 1.0 initially provided the directives *kernel*, *parallel*, *loop*, *declare*, *data*, *cache*, *host_data*, *update*, and *wait*.

OpenACC 2.0 clarified many notions and introduced news clauses, directives and API routines among which:

- introduced the *enter data* and *exit data* constructs
- clarified allowable loop nesting (*gang* may not appear inside *worker*, which may not appear within *vector*)
- added *atomic* and *routine* constructs

OpenACC 2.5 The most significant of changes made by OpenACC 2.5 was the profiling interface introduction. The *init*, *shutdown* and *set* directives were introduced for device control. The *num_gangs*, *num_workers* and *vector_length* clauses were allowed on the *kernels* construct. An implicit presence check before any *copy*, *copyin* or *copyout* operation was added. Finally, asynchronous versions of device memory routines were also added.

OpenACC 2.6 The most important change made by OpenACC 2.6 was the introduction of the *serial* directive.

OpenACC 2.7 did not add any new directives. However, some specifications were clarified, the *self* clause was added to *parallel*, *kernels* and *serial* constructs, and the *readonly* modifier was added to the *copyin* clause and *cache* directive. Table 2.3 summarizes the main evolution of OpenACC features.

Table 2.3 – Summary of OpenACC's main constructs evolution. Constructs are in rows, while added clauses are in columns.

Construct	1.0 (2011)	2.0 (2013)	2.5 (2015)	2.6 (2017)	2.7 (2018)
parallel	if, async, num_gangs, num_workers, copy, create, vector_length, firstprivate, reduction, copyin, copyout, present, deviceptr, private	wait, device_type, default	default	no_create, attach	self
kernels	if, async, copy, copyin, copyout, present, create, deviceptr	wait, device_type, default	default, num_gangs, num_workers, vector_length	no_create, attach	self
loop	collapse, gang, worker, vector, seq, independent, private, reduction	auto, tile, device_type	–	–	–
declare	copy, copyin, copyout, create, present, deviceptr, device_resident	link	–	–	–
data	if, copy, copyin, copyout, create, present, deviceptr	–	–	no_create, attach	default
host_data	use_device	–	–	if, if_present	–
update	host, device, if, async	wait, self, device_type	if_present	–	–
wait	–	async	–	–	–
enter data	–	if, async, wait, copyin, create	–	attach	–
exit data	–	if, async, wait, copyout, delete	finalize	detach	–
routine	gang, worker, vector, seq, bind, device_type, nohost	–	–	–	
seriel	–	–	–	async, wait, if, copy, copyin, copyout, create, device_type, reduction, no_create, present, deviceptr	–

2.2.6 OpenACC Implementations

There are some mature compilers that fully support OpenACC. There are also many prototypes of OpenACC (first versions) implementation, but most of them are obsolete and not maintained.

- Among commercial compilers, the most up to date are probably NVIDIA PGI pgcc/pgc++ and Cray cc compilers. Indeed, OpenACC was initially developed by PGI, Cray, and NVIDIA.

- Among compilers from academic and open source communities, there is accULL [44], a prototype of OpenACC 1.0 implementation based on the Python library. accULL was the first released implementation of OpenACC which had support for ARM CPU. However, their syntax does not completely respect the OpenACC standard and it is no longer maintained today.

- The Omni OpenACC Compiler [45] is a source-to-source translator that translates C code with OpenACC directives to C code with the CUDA API which can be then compiled by NVIDIA's CUDA compiler. It supports most part of OpenACC 1.0. specifications. In Omni OpenACC Compiler, the application cannot make use of multi-dimensional grid topologies for NVIDIA GPUs. The generated CUDA C exhibits significant overhead due to inefficient loop scheduling transformations. Omni OpenACC Compiler supports only C codes and has not support for ARM CPU.

- Rose-OpenACC [46] is another implementation of OpenACC based on the ROSE compiler framework. Rose-OpenACC can only generate OpenCL code. Furthermore, only the gang and worker clauses are interpreted and mapped to work-group and work-item in OpenCL, respectively. The repository of Rose-OpenACC sources on github is no longer available at the time of this writing.

- OpenARC [47], based on the Cetus source-to-source framework, is an open-source framework that supports OpenACC 1.0 and performs source-to-source transformations, targeting devices such as GPUs. Authors proposed self-defined directives to map data to different CUDA memory spaces, including the shared memory and texture memory. However, OpenARC seems to be no longer maintained and the link to its repository no longer works at the moment of writing this document.

- IPMACC [48] is an open source framework for source-to-source translation of C/C++ code with OpenACC ($\leq$ 2.0) directives. From OpenACC directives, it can generate OpenCL or CUDA code. Its authors have also proposed new directives to exploit the scratched memory in order to close the gap between the OpenACC and OpenCL/CUDA application performance. The *parallel* construct, synchronization clause/APIs and $2D$ arrays transfer are not supported in IPMACC. We tried to install IPMACC on an Intel I7 CPU and an ARM CPU without success, the main reasons being that IPMACC was build with obsolete versions of Python and XML libraries, and many instructions used in IPMACC sources are no longer supported in recent versions of these libraries. Furthermore, IPMACC is no longer maintained since at least 2 years.

- The most recent OpenACC's compiler proposed by the research community is an extension of on OpenUH that takes C/Fortran applications and targets NVIDIA GPUs and AMD GPUs/APUs [49].

- The GCC 7, and 8 release series fully support OpenACC's 2.0 specification. However, the official releases of GCC have no support to target accelerators. Therefore, to force GCC to generate accelerators code from OpenACC directives, the offloading supports must be enabled by trying to compile GCC from sources[1]. If the target device is an NVIDIA GPU, it is necessary to build also the *nvptx-tools*[2], which is a collection of tools for use with nvptx-none GCC toolchains. Except configuration and Makefile files, the sources of the *nvptx-tools* have not been updated since at least two years.

2.3 OpenMP and OpenACC Specifications Comparison

OpenMP and OpenACC, while similar, are still quite different in their approach. We describe some of their differences in following paragraphs.

Philosophical Difference

One of the most significant differences between OpenMP and OpenACC is their philosophy. OpenMP is *prescriptive* because the OpenMP API covers only user-directed parallelization and not compiler-generated automatic parallelization. In OpenMP, the programmer has to explicitly specify which regions of code must be parallelized and how this must be done. The same must also be done for nested loops inside a parallel region. Consider the snippet of code presented in Listing 2.14, if we use only the *target* directive (line 1), the loop will be offloaded on the device and executed by a single device thread. By adding the *teams* construct, a league of *thread teams* will be created, but it's still the device master thread that will execute the loop. By inserting the *distribute* construct, loop's iterations will be distributed between *thread teams*, but only the master thread of teams will execute the *teams region*. The other threads within a teams will only participate in the *teams region* if we add the *parallel for* directive.

```
1 #pragma omp target teams distribute parallel for is_device_ptr(...)
2 #pragma acc kernels loop independent deviceptr(...)
3 for(int x = 1; x < h - 1; x++)
4   //........
```

Listing 2.14 – OpenMP and OpenACC philosophical difference

On the other hand, OpenACC is *descriptive*. Indeed, the OpenACC programming model is user-guided parallelism i.e. it allows the programmer to augment information (e.x., the absence of loop iterations interdependency, guidance on the mapping of loops onto an accelerator, etc.) available to the compilers. The basic idea of the OpenACC programming model is that it is best for the user to describe the parallelism and data motion in a more general way via directives so that the OpenACC compiler can have more freedom to map the parallelism to the hardware. Hence, the

[1] available on https://github.com/gcc-mirror/gcc/tree/openacc-gcc-8-branch
[2] available on https://github.com/MentorEmbedded/nvptx-tools/

OpenACC compiler has more performance responsibility and the quality of the implementation can greatly affect application performance. With OpenACC, to get the loop (of Listing 2.14, line 3) fully parallelized, we just have to add the *loop independent* to the *kernels* construct, thus the compiler will generate gangs of workers and distribute the loop iterations between them.

Synchronization

OpenMP has a set of directives for threads synchronization on host, such as *critical*, *barrier*, *taskwait*, *taskgroup*, *flush*, and *ordered*. OpenMP's *depend* clause allows synchronization between different *target regions*, but synchronization between threads within a *teams* or between different *teams* is not currently available. Multiple device execution queues creation is also not available in OpenMP current versions. On the other hand, intra gang synchronization is not currently available in OpenACC. However, inter gangs synchronization may be possible, since there is an implicit barrier at the end of a *kernels* and *parallel region*. OpenACC's *async* clause allows multiple device execution queues creation while the *wait* clause allows synchronization between different *kernels* or *parallel* regions.

Presence Test

OpenMP learns from OpenACC and vice verse. Indeed, in OpenACC 1.0, there were two of data motion clauses, one form for testing the presence (before transferring a data) and one for "skipping" the presence test. OpenMP 4.0 had one form of data motion clause which always checks for presence. Thus, since OpenACC 2.5, the form that skips the presence test was eliminated. Concerning the *update* directive, OpenACC 1.0 and OpenMP 4.0 both made it an "error" to do an *update* on an object that was not present on the device. OpenACC 2.0 relaxed this hindrance to programmers and added the *if_present* clause. On the other hand, OpenMP > 4.0 just makes the *update* a no-op if the object is not present.

Routines Calls, Global Data Access, and Scalar

OpenACC's *routine* and *declare* directives fill jointly the same functionality as OpenMP's *target declare* construct, but they provides most features than their OpenMP equivalent. OpenACC 1.0 made all scalars firstprivate unless overridden by the programmer, while OpenMP 4.0 made all scalars *map(inout)* on target constructs. OpenMP > 4.0 makes all scalars firstprivate by default.

Host and Device Execution

Both, OpenMP and OpenACC allow the user to specify the different target for different code regions in the same application, and within a same compilation unit. This can be done by using the *if* clause for both, the *device* clause for OpenMP and the *device_type* clause for OpenACC. This is

an important feature of these two standards since it allows the user to have a single source for both host and accelerators and assure portability.

Memory Hierarchy Management

OpenACC provides the *cache* directive to specify array elements or sub-arrays that should be fetched into the "highest level" of the cache for the body of a loop, while OpenMP 5.0 *allocate* clause and directive may address this issue. Unlike OpenMP, OpenACC current version does not provide any clause or directive to access to other memory spaces (e.x., constant memory of GPUs).

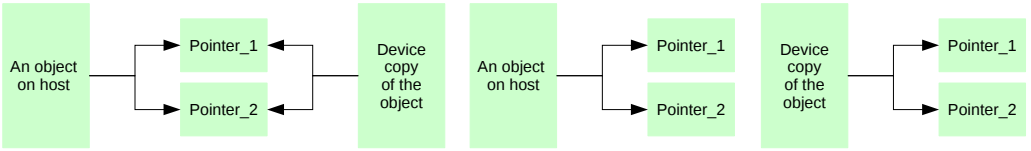
There are also many technical differences between the two specifications, the first being the *parallel* construct that starts parallel execution on the current device in OpenACC while in OpenMP it creates a team of threads that execute the region on the host unless the construct is called inside a *target region*. The OpenMP's *teams* construct is equivalent to the OpenACC's *kernels* construct only if there is a single loop nested inside the *kernels region*. OpenMP 5.0 introduced the *loop* directive, but it will be necessary to wait for OpenMP's 5.0 implementations to see if the two *loop* directives are equivalent. Table 2.4 summarizes OpenACC and OpenMP main features.

Table 2.4 – OpenACC 2.7 vs. OpenMP 5.0 for accelerator programming

OpenACC	OpenMP
Data Directives + clauses that are not common to OpenACC and OpenMP	
acc data: no_create, present, attach, default	**omp target data**: use_device_addr, device
acc enter/exit data: attach/detach	**omp target enter/exit data:** device
acc update: if_present, self	**omp target update**
acc host_data	–
acc cache	may be feasible with **omp allocate** directive
acc routine and **acc declare**	**omp declare target** – **omp end declare target**
–	**omp declare mapper**
Possible equivalence between compute offload constructs	
acc parallel (without any nested constructs)	**omp target** (without any nested constructs)
acc kernels (without any nested constructs)	**omp target** (without any nested constructs)
acc serial [loop]	**omp target** (without any nested constructs)
acc parallel loop independent gang(static:chunk_size) worker	**target teams distribute dist_schedule(chunk_size) parallel for**
acc kernels loop gang(integer) worker	**target teams distribute dist_schedule(chunk_size) parallel for**
Multiple device execution queues creation	
async(integer)	–
Inter offloaded regions synchronization	
wait(integer)	depend(in | out | inout: ...)
acc wait	–
Asynchronous behavior	
async	nowait
acc_memcpy_to|from_device_async	–

Deep Copy

The *deep copy* means copy all fields in the aggregate data object (e.x., struct, std::vector) which may include dynamically allocated memory pointed to by the fields as shown in Figure 2.8b. When a deep copy is performed, dynamically allocated memory that are pointed by the fields are also copied. The aggregate data type can be nested. The deep copy is to handle nested dynamic data structures. For example, there could be an array that is used in an offloaded region and the entire array must be copied on the device. If we suppose that this array is an array of structures, each element of that array is a structure that has a field that is another allocatable array, and each one of these allocatable arrays could have different sizes. It is extremely challenging for the OpenMP and OpenACC compilers to decide how to transfer the data in this case.



(a) Shallow copy (b) Deep copy

Figure 2.8 – Difference between Shallow copy and Deep copy

In Fortran, all of pointer and information pointed is stored in the dope vector structure which is initialized at the pointer initialization. This information includes the memory pointed's location and the size of information pointed. For Fortran applications, the compiler has the information needed to perform a full deep copy. However, this feature is not available in C and C++. Hence, for C/C++ codes, the compiler can't capture the memory size of information pointed at compile-time. Therefore, it is necessary to define directives or clauses that give the compiler indications on how to generate the transfers and avoid runtime errors.

Deep copy is the significant feature on which work the OpenMP and OpenACC committees for their future releases. For example, the PGI compilers 19.1 and newer professionals editions include an implementation of the draft OpenACC 3.0 true deep copy directives in Fortran, C, and C++.

The current versions of OpenMP(5.0) and OpenACC (2.7) perform a *shallow copy*. Shallow copy of an aggregate data object copies all of the member field values. This works well if the fields are values. However, when the fields are pointers, this leads to runtime errors since the pointers are copied on the device without the data that they point. Consequentially, both the object on CPU and its copy on device point to the same memory location as shown in Figure 2.8a.

The first solution to avoid the shallow copy problem is to perform a manual deep copy. Consider the example presented in Listing 2.15. Since the array *vect* is a "std::vector", it is a collection of three-pointers. If we copy array on the device by using the *copyin* clause, and use it inside the *parallel region* (line 13), we will get a runtime error since only the pointers have been copied, not the data they are pointing to. Thus, it is necessary to allocate two supplementary arrays (line 3 and 4) in order to get *vect* available on the device. This is done by using *enter data* and *exit data* constructs, and the instructions from 5 to 12. Although we use OpenACC directives in this

example, the same thing can be done in OpenMP.

```
1  vector<int> vect = new vector<int>[4096];
2  //initialize vect elements
3  int **temp = new int*[4096]; //device copy of vect
4  int *sizes = new int[4096]; //contains the size of each sub vector of vect
5  #pragma acc enter data create(temp[0:4096][0:0])
6  for (int i = 0; i < 4096; i++) {
7    int sze  = vect[i].size(); //get number of sub elements of vect[i]
8    sizes[i] = sze;
9    temp[i]  = vect[i].data(); //copy data on that vect[i] is pointing to
10   #pragma acc enter data copyin(temp[i:1][:sze])
11 }
12 #pragma acc enter data copyin(sizes[:4096])
13 #pragma acc parallel loop independent gang vector present(temp,sizes)
14 for (int i = 0; i < 4096; i++) //iterate on elements of vect
15   for (int j=0; j< sizes[i]; ++j) //iterate on sub elements of i
16     //use temp[i][j];
17 #pragma acc exit data delete(sizes) //end of enter data of line 12
18 #pragma acc exit data delete(temp)  //end of enter data of line 5
```

Listing 2.15 – Manual deep copy example

The second solution is to use the *unified memory access* concept as introduced in 2.1.5. In OpenMP, the user may use the newly added *requires* directive with its clause *unified_address*. Currently, OpenACC does not provide any directive or clause to do such as thing, but some implementations like PGI compilers have a flag that the user can use to get a program's data available for both host and device. As we will see in the next chapter, using this flag can degrade application performance since it increases the data transfer traffic.

Note that for physical unified memory architectures like AMD APU and NVIDIA Tegra platforms, deep copy support may not be necessary, though it may still in some cases be beneficial for performance.

2.4 OpenMP/OpenACC Suitability for GPUs Programming

In the previous section, we have seen that both OpenMP and OpenACC provides a set of features for accelerators programming without exposing a lot of details about the underlying architecture. In this section, we discussed the suitability of OpenMP and OpenACC for GPUs programming in comparison to CUDA.

Mapping to CUDA Thread Hierarchy

As seen in 1.6.1, GPU's threads are organized in grid, block and warp in CUDA. OpenACC parallelism levels can be mapped to this hierarchy:

- *num_gangs* and *gang* correspond to a CUDA grid
- *num_workers* and *worker* conceptually maps to a warp in CUDA
- *vector_length* and *vector* correspond to a CUDA block

Depending on vector length, a worker can consist of multiple warps, or multiple workers can span across a single warp. We know that a warp consisted of 32 threads. Hence, when using the *vector_length* or *vector* clause, the value specified must be a multiple of 32 as explained in 1.6.1 - optimization 8. As CUDA grids and blocks, gangs and vectors can be configured in 1 or 2 or 3D dimension.

On the other hand, in OpenMP, the *num_teams* clause corresponds to a CUDA grid while the *thread_limit* clause maps to a CUDA block. However, neither *num_teams* neither *thread_limit* can't be configured in 2 or 3 dimensions because it is not allowed to have a *teams* construct nested inside a *teams region*. The CUDA warp concept does not exist in OpenMP currents versions. As in CUDA and OpenACC, when using the *thread_limit* clause, it is recommended to set its value to a multiple of 32.

When using the number of threads management clauses (*num_gangs*, *vector_length*, *num_teams*, etc.), it is necessary to take into account the capabilities of the targeted GPU as explined in 1.6.1 - optimization 7.

Remember that in CUDA, a grid consists of one or more block(s), a block consists of one or more warp(s), and a warp is a group of 32 threads. In OpenACC, a gang consists of one or more worker(s), a worker is a vector of threads. In OpenMP, there is a league of teams of threads. Figure 2.9 depicts the thread hierarchy in CUDA, OpenACC and OpenMP. We assume that we have a grid of two blocks of 64 threads.

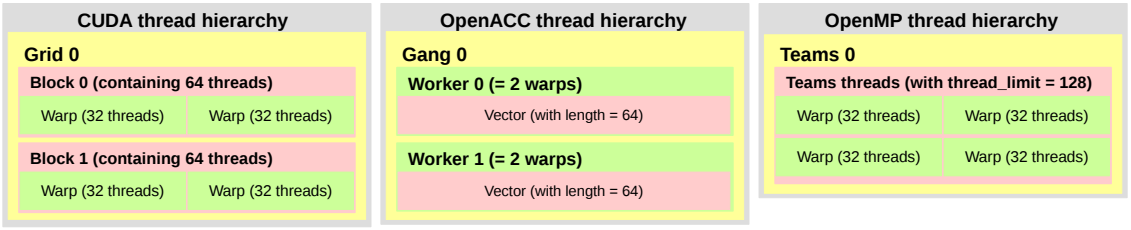


Figure 2.9 – Thread hierarchy illustration in CUDA, OpenACC and OpenMP

Memory fence

GPUs typically implement a weakly-ordered memory model that is the order in which a GPU thread writes data to memory (e.x.: shared memory, L1 cache, L2 cache, global memory). This order may not necessarily be the order in which the data is observed being written by another GPU thread. In this context, an explicit memory fence may be used to enforce some ordering on memory accesses. CUDA provides memory fence functions that can be used in kernel code to assure the

consistency of threads' (in the same block) temporal view of memory(or all memory). In OpenMP, this can be done with the *flush* directive while OpenACC does not provide any construct or API routine for this purpose.

Synchronization

CUDA provides synchronization routines to synchronize GPU threads inside a grid (e.x. sync in CUDA Cooperative groups header), or a block (e.x. __syncthreads()), or a warp (e.x. __syncwarp). Intra Block synchronization can be used to coordinate communication between the threads of the same block or to make all global and shared memory accesses made by block's threads, prior to a certain point of the kernel, are visible to all threads in the block. On the other hand, intra warp synchronization can be used to guaranty memory ordering among threads participating in the barrier.

OpenMP does not provide any mechanism (either clause, directive or routines) to make inter teams (inter blocks in CUDA terminology) synchronization. OpenMP does not also provide any mechanism to synchronize (directly at source level) threads in a *teams* (intra block and intra warp in CUDA terminology). OpenACC also does not provide support for synchronization between threads in the same vector or worker or gang.

CPU and GPU Asynchronous Execution Support

In order to prevent the CPU thread from waiting for the completion of a GPU operation (data transfer or kernel execution) before executing next instructions (as explained in 1.6.1 - optimization 2), CUDA provides asynchronous versions of the data transfer routines. OpenACC also provides asynchronous versions of its data copy routines (see an example in Listing 2.13) while OpenMP does not. Furthermore, OpenMP *nowait* clause and OpenACC *async* clause can be used to avoid the host thread wait for the completion of an offloaded region. If another part of code needs the results of an offloaded code with the asynchronous clauses, then the *depend* clause (in OpenMP) and the *wait* construct and clause (in OpenACC) can be used for synchronization purpose.

Compute and Data Transfer Overlapping - CUDA Multi-streaming

In order to efficiently use all processors in the system including host CPU and devices, some GPUs support asynchronous memory copy to or from the device concurrently with a kernel execution (as explained in 1.6.1 - optimization 2). In CUDA, this can be done by creating multiples devices execution queues (or streams). A stream is a sequence of commands that execute in order. Thus, different streams may execute their commands concurrently or out of order with respect to each other. When using multi-streaming, it is important to consider the capabilities of the targeted GPU otherwise, the application performance may be degraded.

The OpenACC's *async* and *wait* clauses allow multiple device execution queues creation when

they have been used with an argument. Their argument identifies the stream number. Hence, multi-streaming is feasible in OpenACC. The OpenACC's *wait* clause and directive allow synchronization between streams.

The OpenMP's *nowait* clause, that aims to avoid the local thread wait until the completion of the structured-block associated to a directive before executing next instructions, does not involve stream creation. Furthermore, support to create multiple device activity queues in addition to the default queue is not currently available in OpenMP. Therefore, CUDA multi-streaming is not possible with current OpenMP versions. A new clause may be added to OpenMP specifications or an argument must be allowed for the *nowait* clause (as it is the case in OpenACC *async* clause) to permit multiple device activity queues creation. In the same way, a supplementary argument must be added to the *depend* clause to permit streams synchronization.

Memory Hierarchy Management

Both OpenACC and OpenMP current versions make all scalars (used inside an offload region) *firstprivate* by default and store them in registers in priority. If the available registers are not enough, they are stored into thread local memory.

Neither OpenACC neither OpenMP does not provide support (directives, clauses or routines) to access to GPUs's texture memory. However, the implementation (i.e. the compiler) can choose to put certain data (that are only read inside the offloaded region) in the texture memory.

OpenACC provides the *cache* directive to specify that a variable should be fetched into the "highest level of cache". A variable appearing in a *cache* directive can be stored in the L1 cache, for instance, depending on the implementation. OpenACC does not provide any support to access to the constant memory of GPU.

Concerning OpenMP, with its newly added *allocate* directive and the new memory spaces (see Table 2.1), it should be possible to access not only the constant memory but also the cache L1, the texture cache and may be the shared memory.

Dynamic parallelism

As already mentioned in 1.5.2, NVIDIA GPUs support dynamic parallelism and it's allowed in the CUDA programming model. OpenMP supports dynamic parallelism but OpenACC does not. Indeed, it is not allowed to use a *kernels* or *parallel* construct inside a routine that has been annotated with *pragma acc routine*.

2.5 Summary

In this chapter, we presented an overview of the OpenMP and OpenACC programming models which are used in our research. Both OpenMP and OpenACC provide a rich set of directives

and APIs that allow users to annotate compute-intensive regions, explicitly manage data transfers, and expose loop parallelism. These features can enable non-expert programmers to portably and productively achieve performance for their applications. OpenMP and OpenACC directives hide many details of the underlying device architecture, freeing a programmer's attention for other tasks. OpenMP and OpenACC standards continue to evolve, with features for describing complex memory hierarchies, parallelism, complex data type management, support for multiple device targeting, etc. We also have seen that OpenMP and OpenACC are quite different. However, it is possible to translate between OpenMP and OpenACC, but the process may not be automatic. We discussed the suitability of OpenMP and OpenACC for GPUs programming and showed that although the two standards continually evolve and offer a large set of features, some optimizations (shared, texture and constant memory addressing, intra block and inter block synchronization, unified memory usage, etc.) necessitate using low-level languages. In the next chapter, we will propose an effective application parallelization methodology with OpenMP and OpenACC, evaluate directive-based programming models performance, and explore some performance optimization opportunities with them.

Chapter 3

Application Parallelization Methodology with OpenACC & OpenMP

Contents

In this chapter, we propose a parallelization methodology with OpenMP/OpenACC for GPU. We will start by making a non-exhaustive assessment of some relevant works that have been done on OpenMP and OpenACC for accelerator programming during the last decade in 3.1. Sections 3.2 and 3.3 give respectively a short description of hardware platforms, compilers, and benchmarks used in this work. Section 3.4 presents some problems frequently encountered in applications parallelization either on CPU or GPU. In section 3.5, we will describe the proposed parallelization methodology for parallelizing sequential programs with OpenMP/OpenACC for GPU. The proposed parallelization methodology includes severals optimization strategies such as directives combination (3.6) and loop scheduling (3.8) [12]. Next we use the proposed parallelization approach to port two well established benchmarks suites and some synthetic programs to OpenACC, OpenMP, and CUDA. We will also use other optimization strategies such as memory coalescing and data transfer management (3.7). With rigorous experimental analysis, we will then analyze how the performance can be incrementally tuned. Finally, we will analyze OpenMP and OpenACC performance evolution with workload using synthetic programs in section 3.9.

3.1 Previous work on OpenACC & OpenMP

Since the release of OpenACC 1.0 and that of OpenMP 4.0 later, many works have been done to evaluate both in term of performance, portability, and productivity. Many works have also proposed new directives and clauses for addressing some features not offered by OpenACC and OpenMP, but also tools to assist programmers for application acceleration with these two standards. In this section, we present some of the previous works that are relevant to our work.

OpenACC Performance Evaluation and New Directives Proposition

- Hoshino et al. [9] did a performance comparison between CUDA and OpenACC by porting two benchmark kernels (a matrix multiplications and a 3-D stencil) and a computational fluid dynamics application. They found that in general OpenACC is approximately 50% lower than CUDA, but for some memory-bound application, it can reach up to 98% with careful manual optimizations such as loop fusion, thread mapping and register blocking.

- J. A. Herdman [50] compared OpenACC's "parallel" and "kernels" constructs performance in Cray CCE 8.1.7, PGI 13.7, and CAPS 3.3.2 using the hydrodynamic mini-application CloverLeaf. Their results indicated that CAPS' implementation of the "parallel" construct was deficient, CCE's implementations of both constructs achieved the same performance and PGI's "kernel" construct outperformed the rest on more occasions than any other implementation. Their results also showed that a native CUDA implementation outperforms the best OpenACC by 15% to 20% and OpenCL outperforms OpenACC by 10% to 20% on NVIDIA GPU.

- Xuechao et al. [51] did an empirical investigation of program productivity comparison between OpenACC and CUDA. They found that the OpenACC programming time is 37%

shorter than CUDA, but CUDA running speed is $9x$ faster than OpenACC and OpenACC development work is not significantly affected by a previous CUDA experience.

- Ahmad et al. [52] investigated the implementation aspect of the OpenACC cache directive under NVIDIA GPUs and proposed optimization for the CUDA backend.
- Li et al. [10] have compared OpenACC and CUDA performance focusing on programming models, optimization technologies and underlying compilers. They measured and compared kernel execution times and data transfer times to/from the GPU and concluded that in terms of kernel running time, the OpenACC performance is lower than the CUDA while the data transfer time in OpenACC programs tends to be much faster than in CUDA.
- Rengan Xu [53] evaluated and compared HMPP, PGI accelerator model, and OpenACC involving several scientific applications. He found that the performance of the evaluated directive-based programming models is close enough to that of the CUDA. He also proposed new directives for OpenACC to support multiple GPUs in a single node and an analytical model-based auto-tuning framework to identify the optimal loop schedule that may be better than the default loop schedule chosen by the compiler. Table 3.1 summarizes four of works presented above.

Table 3.1 – Summary of relevant previous work on OpenACC performance evaluation

Authors	**Platform(s)**	**Compiler(s)**	**Application**	**Optimization**	**Results**
Hoshino et al. [9], 2013	Single node of the TSUBAME 2.0	PGI 12.10, Intel 11.1, HMPP 3.2, Cray 8.1 CUDA 4.1	Mat mult, 3D stencil, CFD	*kernels*, *loop*, manual loops scheduling, scalar replacement, loop fusion	In term of runtime CUDA In term of runtime CUDA vs. OpenACC: 50% on average
J.A. Herdman [50], 2014	NVIDIA GPU	Cray 8.1.7, PGI 13.7, CAPS 3.3.2	CloverLeaf	*kernels* vs. *parallel* *kernels* vs. *parallel*	- PGI *kernels* is the best; - CUDA vs. OpenACC: $15-20\%$; - OpenCL vs. OpenACC: 10% to 20%

Applications acceleration with OpenACC/OpenMP

- Ikeda et al. [54] presented a source-to-source OpenACC optimizer that automatically optimizes a histogram computation code for a graphics processing unit (GPU) and concluded that the achieved speedups over the naive method ranged from $\times 0.7$ to $\times 3.6$.
- Reza et al. [55] accelerated the computation of finite difference generated time-domain Green's functions of layered media using OpenACC. They compared the accuracy of the OpenACC version to that of the serial implementation on CPU and found that the accuracy of the problem is maintained.
- P. Alyson et al. [56] proposed OpenACC extensions to enable efficient code generation and execution of stencil applications by parallel skeleton frameworks such as PSkel. They showed that their stencil extensions may improve the performance of OpenACC in up to 28% and 45% on GPU and CPU, respectively.

OpenMP Performance Evaluation

- Martineau et al. evaluated in [57] OpenMP 4.0 effectiveness as a heterogeneous parallel programming model. They found that OpenMP 4.0 can achieve good performance while decreasing development cost. They also analyzed in [58] an implementation of OpenMP 4.5 provided in the trunk branch[1] of Clang 4.0 on an NVIDIA Kepler GPU. They used synthetic programs and three mini-applications to show the majority of issues affecting the performance of this Clang OpenMP 4.5 implementation. They suggested some potential solutions that can improve Clang code generation performances.

- Hayashi et al. [59] evaluated and analyzed IBM XL C and LLVM/Clang compilers implementation of OpenMP 4.0/4.5 on an IBM POWER8 + NVIDIA Tesla K80 platform. They used six kernels and applications taken from the PolyBench and SPEC ACCEL benchmark suites. They found that the OpenMP generated codes are in some cases faster, in some cases slower than straightforward CUDA implementations written without complicated hand-tuning.

- GT Bercea et al. [60] introduced an implicit allocation policy in the NVPTX backend of LLVM to lower implicitly shared OpenMP variables to the shared memory of the device. They evaluated the proposed scheme in cases that involve scalar variables and statically allocated arrays. Results showed that for scalar variables the pressure on shared memory is relatively low and does not negatively impact occupancy.

- Artem et al. [61] explored OpenMP 4.5 performance improvement via kernels fission, pipelining memory transfers with kernel execution and grid-geometry selection. They used programs taken from Rodinia and PolyBench benchmark suites, the same Clang OpenMP 4.5 implementation as in [58] and two NVIDIA GPUS. As a performance metric, they used the *occupancy* which is the ratio of active warps on an SM to the maximum number of active warps supported by the SM. Their results indicate that the performance gains can be significant especially in machines with larger page sizes.

- Graham et al. [11] explored the performance portability of the directives provided by OpenMP 4 and OpenACC to program various types of node architectures. Their results show that due to the slightly different interpretations of the OpenMP 4 specification, it is crucial to understand how the specific compiler being used implements a particular feature on different platforms.

- Memeti et al. [62] studied empirically the characteristics of OpenMP, OpenACC, OpenCL, and CUDA with respect to programming productivity, performance, and energy. They observed that programming with OpenACC/OpenMP/CUDA requires significantly less effort than programming with OpenCL and that OpenMP/OpenCL/CUDA have comparable performance and energy consumption on Ida system.

[1] https://github.com/clang-ykt

- B. Bastem et al. [63] introduced a tiling based programming model, implemented as a library, for GPUs that overlaps the computation and data transfers between the CPU and the GPU. Results showed what the library can handle cases where the device memory is not sufficient to hold the entire application data by staging the data as tiles. Table 3.2 summarizes four of the works presented above.

Table 3.2 – Summary of relevant previous work on OpenMP evaluation

Authors	**Platform(s)**	**Compiler(s)**	**Application**	**Optimization**	**Results**
Martineau et al. [58], 2016	NVIDIA K40m and K40c GPUs	LLVM/Clang	Synthetic programs, SNAP, TeaLeaf, CloverLeaf	Combined *target* + *teams* + *distribute* + *parallel* + *for*, loop fusion, FMA[2], manual loops scheduling	- CUDA is faster than OpenMP, - FMA, non-coherent loads, and *num_teams* setting may improve OpenMP performance, - LLVM/Clang uses many registers which can reduce performance
Hayashi et al. [59], 2016	IBM POWER8 + NVIDIA Tesla K80	IBM XL C, LLVM/Clang	6 kernels taken from PolyBench and SPEC ACCEL	Combined and non-combined constructs, loop permutation, fusion, and unrolling, FMA, *dist_schedule*, schedule(static,1)	- OpenMP codes performance $\geq$ straightforward CUDA versions - Added new constructs fo shared memory and read-only cache
Artem et al. [61], 2018	NVIDIA Titan X Pascal and P100 GPUs	LLVM/Clang as in [58]	Kernels taken from PolyBench and Rodinia	kernels fission, overlapping data transfer and computation, loop scheduling	Kernels fission, asynchronous execution, and loop scheduling can significantly improve OpenMP applications performance
Graham et al. [11], 2016	NVIDIA K20X GPUs, Xeon Phi 7210	Cray 8.5, PGI 16.5, Intel 16	Dense linear algebra kernels, Jacobi, HACCmk	Combining and un-combining *target*, *teams*, *distribute*, *parallel for*, and *simd*	- Compilers interpret differently OpenMP target constructs - The *simd* construct is portable - OpenACC is usually simpler to use than OpenMP - Compilers have same interpretation of OpenACC - Intel does not support combined OpenMP target directives

OpenACC and OpenMP Programing Models Comparison

- Guido et al. [64] ported the SPEC ACCEL benchmarks from OpenACC to OpenMP 4.5 to show how an application can be programmed with directive-based languages in a performance portable style that lets the compiler make platform-specific optimizations to achieve good performance on a variety of systems.

- Sergio et al. [65] explored the similarities and the functionality gaps between OpenACC and OpenMP programming models and presented insights into the translation process of constructs from OpenMP to OpenACC. They also presented an empirical study of performance and portability across multicore platforms and GPU accelerators for varying workload sizes.

- Wienke et al. [66] compared OpenACC and OpenMP programming models with respect to their programmability. They evaluated OpenACC and OpenMP expressiveness by pattern-based comparison. The patterns covered were map, stencil, reduction, fork-join, superscalar sequence, nesting parallelism, and geometric decomposition. They concluded that the two

standards are equivalent but OpenMP would likely achieve the best adoption in the long-term because it is such a prominent standard.

Multi-GPUs targeting with OpenACC/OpenMP

- Komoda et al. [67] presented an OpenACC compiler with the capability to execute single GPU OpenACC programs on multiple GPUs. They proposed some directives that allow programmers to express the patterns of memory accesses in the parallel loops to be offloaded.
- Kazuaki et al. [68] proposed an OpenACC transpiler to automatically use multiple GPUs. They evaluated this transpiler on four benchmarks to quantify its performance. They showed that for some applications the proposed transpiler can compete with hand written MPI code.
- Yonghong et al. [69] explored support of multiple accelerators in high-level programming models by designing extensions to OpenMP to support offloading data and computation regions to multiple devices. The proposed extensions allow for distributing data and computation among a list of devices.

Autotuner for OpenACC/OpenMP

- Montgomery et al. [70] presented an autotuner that can assist the programmer in selecting high-quality values for OpenACC *num_gangs* and *vector_length* clauses. The autotuner was evaluated on a suite of 36 OpenACC kernels and results showed that it identified values within the top 5% for 29 kernels, within the top 10% for five kernels, and within the top 25% for the remaining two. Eleven of the kernels achieved a speedup greater than $2\times$ over the compiler's defaults while the autotuner required only $7 - 11$ runs of the target program, on average.
- Makoto et al. [71] discussed the performance tunabilities of OpenACC and OpenCL. Since OpenACC cannot synchronize threads running on GPUs, they designed an additional compiler directive for thread synchronization. The additional directive can allow OpenACC to describe more tuning techniques in the same approach as OpenCL.

As can be seen, many works have been done on OpenMP and OpenACC. The work presented in this chapter brings to the state of the art the following contributions:

- We propose an effective application parallelization methodology with OpenACC and OpenMP. The proposed approach will be used to parallelize synthetics kernels and severals applications taken from well established benchmark suites.
- We analyze two OpenMP and OpenACC compilers, namely LLVM/Clang and PGI compiler, in order to understand how these compilers interpret OpenMP/OpenACC directives, the behavior, and the performance of OpenMP/OpenACC directives when the targeted accelerator is a GPU.

- We give feedback on the parallelization strategy adopted based on code patterns.
- We study the performance evolution according to kernels' workload and compare the performance of three data management methods.

3.2 Hardware platforms & compilers

As seen in Section 1.5, the two platforms used in this thesis are a laptop with an NVIDIA Quadro M2000M hosted in an Intel I7 CPU and an NVIDIA Tegra X1 SoC. However, only the Quadro M2000M GPU will be used in this chapter. Although there are many more or less mature compilers with OpenACC support (seen in 2.2.6), it is the PGI's 18.10.1 which we will use for our experiment since it is the most mature (supports all features of OpenACC 2.6 and includes an implementation of the draft OpenACC 3.0).

Concerning OpenMP, we will use the official release of LLVM/Clang 7.0 which supports all features of OpenMP 4.5. We will use the CUDA toolkit version 9.1 for CUDA versions. In this thesis, performance data will be collected on GPU with the NVIDIA profiler *nvprof* and the *chronos* library on CPU. Finally, CPU versions will be compiled with GCC 8.1. Table 3.3 presents main compilation flags used in our work.

Table 3.3 – Compilers flags used in our experiments

Compilers	**Flags**
nvcc	-use_fast_math -Xptxas -dlcm=cg -O3 -gencode arch=compute_50,code=sm_50
Clang	-O3 -v -ffast-math -ffp-contract=fast -fopenmp -fopenmp-targets=nvptx64 -Xopenmp-target -march=sm_50
PGI (for targetting CPU)	-mp -Minform=warn -fast -O3 -acc -Minfo=accel -ta=multicore
PGI (for targetting GPU)	-mp -Minform=warn -fast -O3 -acc -Minfo=accel -ta=nvidia,maxwell,cc50,fastmath,fma,lineinfo,unroll,loadcache:L1
Gcc	-mavx -Wall -fopenmp -fopenacc -funroll-loops -O3 -ftree-loop-vectorize -fopt-info-vec

3.2.1 Performance Metrics

In order to collect performance data on the GPU, we will use the NVIDIA profiler called *nvprof*. The following performance metrics will be used:

- *Execution time*

- *Instructions executed*
- *GPU occupancy:* gives the ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
- *SM efficiency:* corresponds to the percentage of time at least one warp is active on a multiprocessor

3.3 Benchmarks

For our experiments, we have considered a number of different applications from the Rodinia and PolyBench benchmark suites. We have also used applications from for image processing and synthetic programs.

The Rodinia benchmark suite, designed for performance evaluation in the context of heterogeneous parallel computing [72]; is implemented for both multi-core CPUs and GPUs using OpenMP (for CPU), CUDA, and OpenCL. Rodinia benchmark suite covers different types of application behaviors according to the Berkeley's dwarfs. Therefore they exhibit various types of computations, data access patterns, problem partitions, and optimizations. While Rodinia[3] benchmark suite provides in total 23 CUDA, 21 OpenCL and 19 OpenMP applications, we have selected 9 OpenMP and CUDA applications for our experimentation. The inclusion criteria during the selection process of applications are (1) the need to have the same application implemented in both CUDA and OpenMP, and (2) applications that are compilable and executable in our systems. Table 3.4 summarizes applications taken from the Rodinia benchmark suite.

The PolyBench collection of benchmarks contains codes for linear algebra computations, physics simulation, data-mining, stencils, etc. [73]. Although the suite consists of 30 kernels, we have selected 18 that seem relevant for our experience. Table 3.5 shows selected kernels in the PolyBench benchmark suite.

In addition to applications and kernels taken from Rodinia and PolyBench benchmarks suites, we have also selected 4 applications from the computer vision domain since our use case is an image processing algorithm. These 4 applications include a stereo disparity (SDisp) computation (taken from NVIDIA CUDA 10.0 Samples), Canny filter (Canny) [74], Harris Corner Detector (HCD) [75], and Horn & Schunck (H&S) optical flow estimation algorithm [76].

3.4 General Parallel Programming Problems

In general, parallelizing an application leads to a performance improvement. However, a programmer must consider certain problem when parallelizing an application. We describe bellow some difficulties encountered when parallelizing a code.

[3] version 3.1, available at http://lava.cs.virginia.edu/Rodinia/download_links.htm, accessed 04/03/2019

Table 3.4 – Applications taken from the Rodinia benchmark suite

Application	Dwarf	Domain	Inner loops	Description
BFS: 2 kernels	Graph Traversal	Graph Algorithms	K1: 1	Breadth-First Search algorithm traverses all the connected components in a graph.
Back-Propagation (BP): 2 kernels	Unstructured Grid	Pattern Recognition	K1: 1 K2: 1	A machine-learning algorithm that trains the weights of connecting nodes on a layered neural network.
HotSpot (HS): 1 kernel	Structured Grid	Physics Simulation	4	A 2D transient thermal modeling kernel which computes the final state of a grid of cells.
HotSpot 3D (HS3D): 1 kernel	Structured Grid	Physics Simulation	2	Same as HS but in 3D.
LavaMD (LMD): 1 kernel			3	Calculates the potential and relocation of particles within a large 3D space.
Lud (LUD): 3 kernels			K1: 4 K2: 8 K3: 6	An algorithm to decompose a matrix as the product of a lower triangular matrix and an upper triangular matrix.
Needleman -Wunsch (NW): 2 kernels	Dynamic Programming	Bio-informatics	K1: 2 K2: 2	A dynamic programming algorithm for sequence alignments, which builds up the best alignment by using optimal alignments of smaller subsequences.
SRAD	Structured Grid	Image Processing		A diffusion method used in ultrasonic and radar imaging applications.
Heartwall (HW)				

Table 3.5 – Applications taken from the PolyBench benchmark suite

Application	Domain	Number of inner loops	Application	Domain	Number of inner loops
2mm	Linear algebra	2	mvt	Linear algebra	1
3mm	Linear algebra	2	symm	Linear algebra	2
atax	Linear algebra	1	syr2k	Linear algebra	2
bicg	Linear algebra	1	trmm	Linear algebra	2
correlation	Data-mining	1	adi	Stencil	2
covariance	Data-mining	1	durbin	Linear algebra	2
fdtd-2d	Stencil	1	gramschmidt	Linear algebra	3
gemm	Linear algebra	2	seidel-2d	Stencil	2
gemver	Linear algebra	1	trisolv	Linear algebra	1
gesumv	Linear algebra	1			

Data sharing

When parallelizing a code region, variables (global or local) can be either shared or private. If a variable is shared, then there exists one instance of this variable which is shared among all threads. Instead, if a variable is private, then each thread has its own local copy of this variable. For example, function arguments are by default shared either in CUDA, OpenACC or OpenMP.

Variables declared outside of OpenMP/OpenACC offloading regions are also shared. On the other hand, variables declared locally in a CUDA kernel or an OpenACC/OpenMP offloading region are private. Loop iteration variables are also private.

```
1  void dataSharing(int &sum, int N, int *Tab, std::vector<int> new_list) {
2    sum = 0,
3    #pragma omp parallel for
4    for(int i = 0; i < N; i++) { //i is private by default
5      int a = 0; //a is private
6      //instructions using a ....
7      sum += Tab[i]; //Tab, sum, and new_list are shared
8      new_list.push_back(i);
9      //....
10   }
11 }
```

Listing 3.1 – Data sharing example

Concurrent accesses to variables that are shared among all threads require synchronization mechanisms in order to avoid the crash of the program at runtime, data race, hazards, etc. Consider the example of Listing 3.1. Parallelizing the loop of line 4 with OpenMP (pragma of line 3) leads to a runtime crash due to the instruction of line 8. Lists from the *std* library being safe by construction, we should not have more than one thread trying to add a new element to *new_list* as it is the case in Listing 3.1. One solution can be to protect the *push_back* instruction with a critical section. However, we should get any speedup from OpenMP in this case because there will be contention for the critical section. A faster solution would be for every thread to have its own list, and get those lists merged after the loop finishes.

A data race may occur when two threads access concurrently to the same memory location (at least one in writing) and there is no synchronization that is mandating any particular order among these accesses. For example the instruction of line 7, which sums *Tab* elements, may produce a wrong result. Since the order of loop iterations execution may change for each call of *dataSharing*, the value of sum also may be different. A solution can be to force the updates of *sum* to be mutually exclusive, by using the OpenMP/OpenACC *atomic* clause. However, as for *critical*, using *atomic* can hurt performance due to the cost of synchronization. Another simplest and efficient solution should be to use the *reduction* clause of OpenMP/OpenACC.

Hazards occur when data is accessed in parallel. These may or may not be problematic. For example, when multiple threads write data to the same memory cell, it is undetermined, which data will be stored in the end. However, if all threads write the same value, it does not matter as the result is always the same. To prevent hazards efficiently, atomic operations can be used.

Dependency

Cases where data is read from a cell that was previously overwritten by another thread can be problematic. Typically loops can have two types of dependence : loop-carried dependency and

loop-independent dependency. In loop-carried dependency, dependence exists across iterations. Therefore, if the loop is removed the dependence no longer exists. On the other hand, dependence exists within an iteration in loop-independent dependency. Thus, if the loop is removed, the dependence still exists.

3.4.1 Problems Specific to GPU Programming

General parallel programming problems discussed above are independent of the parallelization paradigm used as well as the architecture targeted. However, there are certain problem that are specific to GPU programming. We briefly present bellow five of them.

1. *Reduction* – The reduction of Listing 3.1 can be parallelized on GPU with OpenACC, OpenMP, and CUDA. Listing 3.2 shows the OpenMP/OpenACC and CUDA versions. As can be seen in this example, with OpenMP and OpenACC the reduction is managed by the compiler. On the other hand, the reduction is entirely programmed by the programmer. First, input data are loaded from global memory to shared memory. Next, the reduction is done in shared memory, and then the result of each block in saved in global memory. After copying back *g_odata*, the last step is to sum partial sums from each block on CPU.

```
1  sum = 0;
2  #pragma omp target parallel for reduction(+: sum) map(sum) ...
3  #pragma acc kernels loop independent reduction(+: sum) copy(sum) ...
4  for(int i = 0; i < N; i++)
5    sum += Tab[i];
6  //CUDA version
7  __global__ void reductionCUDA(int *g_idata, int *g_odata, int N) {
8    cg::thread_block cta = cg::this_thread_block();
9    __shared__ int sdata[];
10   int tid = threadIdx.x; //index in shared mem
11   int i = blockIdx.x*blockDim.x + threadIdx.x; //index in global mem
12   sdata[tid] = (i < n) ? g_idata[i] : 0; //laod to shared mem
13   cg::sync(cta); //synchronize threads
14   for (int s = blockDim.x / 2; s > 0; s >>= 1) {
15     if (tid < s) sdata[tid] += sdata[tid + s];
16     cg::sync(cta);
17   }
18   // write result for this block to global mem
19   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
20 }
```

Listing 3.2 – Data sharing example

2. *Scan* – A scan operation can be inclusive or exclusive. The inclusive scan takes a binary operator $\oplus$ and an array of N elements $[A_0, A_1, ..., A_{N-1}]$ and returns the array $[A_0, (A_0 \oplus A_1), ..., (A_0 \oplus A_1 \oplus ... \oplus A_{N-1})]$. Exclusive scan is defined similarly, but shifts the output and

uses an identity value I as the first element. A loop containing a scan operation has a loop-independent dependency. OpenMP 5.0 provides the *scan* directive. However, OpenACC has not any construct for scan operation purpose.

3. *Extern library routines calls* can not be used inside OpenMP/OpenACC and CUDA kernels. Consider the vector *new_list* of Listing 3.1. In addition to the problem of concurrent accesses to *new_list*, the routine *push_back* is not available to CUDA/OpenMP/OpenACC.

4. *Recursive routines* are not allowed in CUDA/OpenMP/OpenACC.

5. *SIMD intrinsics* (e.g. SSE, AVX, NEON) are not portable. For instance, if the serial version of the code to be offloaded contains SSE instructions, then it is necessary to rewrite these SSE instructions in pure scalar instruction before adding OpenMP/OpenACC directives.

3.5 Methodology of Parallelizing with OpenMP and OpenACC

The main benefits of programming GPUs using a directive-based programming model are achieving performance by simply adding directives to parallel code regions in an existing program. For applications taken in Rodinia, we ported them in OpenACC and OpenMP from their CPU OpenMP versions. Concerning the CUDA versions, we used the optimized CUDA codes already available in Rodinia. For kernels that come from PolyBench, Canny, HCD, and H&S we consider the CPU C version as the starting point to develop the OpenACC, OpenMP and CUDA versions. Since SDisp comes from NVIDIA, we ported their CPU C++ version to OpenACC and OpenMP. We propose the following methodology for parallelizing sequential code with OpenMP/OpenACC:

1. Profile the sequential code in order to identify compute-intensive parts which are typically loops. When performance bottlenecked regions have been identified, verify if loops' bounds are countable. If they are not countable, perform necessary modification to make them countable. If loops iterators are not allowed in OpenMP/OpenACC, for example the std::vector iterator, replace them by classic allowed iterators.

2. Inventory all data, such scalar variables, arrays, and pointers, used in these performance bottlenecked regions. Analyze the type of arrays and pointers inventoried. If some of these arrays/pointers have a complex type like structures containing dynamic allocated arrays of another structure, then it is necessary to add the required instructions to perform a manual deep copy as seen in 2.3. The instructions for the manual deep copy must be inserted before the performance bottlenecked region, and replace all references to the old variables (arrays/pointers) by their device copies.

3. Optimize data structures and array access pattern to efficiently use the device memory (subsection 1.6.1 - Optimization 5). For instance, accessing data in the global memory in a coalesced way, i.e. consecutive threads should access consecutive memory address. This may require some loop optimizations like loop permutation, or transforming the data layout that will change the memory access pattern.

4. Remove data dependency in compute-intensive parts issue by restructuring the code and making private scalar variables as much as possible. However, the programmer must take care of data race situations (as explained in above in Section 3.4).

5. Once all modifications of the target code region have be done, the next step is the data management. As explained in Sections 2.1.4 and 2.2.4, there are three ways to manage data in OpenACC/OpenMP: copy clauses, data directives, and memory management routines (either those of OpenMP/OpenACC or those of CUDA API).

   - If arrays/pointers used in the code region to be parallelized do not require a manual deep copy and are not permanent, i.e. they are only used inside the routine were they have been declared, then a single device data environment must be created by using a data directive (e.g. *omp target data*, *acc data*, and *enter data − exit data*) or the copy clauses if there is a single structured block code to be offloaded. By creating a single device data environment, redundant data movement is avoided. If there are some arrays/pointers that must be updated on the host as illustrated in the Listing 3.3, the *update* directive can be used.

```
1  int main (....) {
2    //variables declarations and instanciation ....
3    #pragma acc data copyin(.....) .....
4    {
5      #pragma acc kernels ......
6      for(.....) {/*...........*/}
7      #pragma acc update ......... //update some data on the host
8      //use them on the host
9      #pragma acc update ......... //update them on the device
10     #pragma acc kernels ......
11     for(.....) {/*reuse them ...........*/}
12   }
13   //......
14 }
```

Listing 3.3 – Example of device data environment containing code executed on CPU

   - If arrays/pointers used in the code region to be parallelized do not require a manual deep copy and are permanent, i.e. they are used across several routines, then:
     - if these data are members of a C++ class, allocated in the class constructor, and released in the class destructor, then the *enter data* and *exit data* directives must be privileged to coy them to and from the GPU. Indeed, this solution does not require for instance to replace a host variable reference by its equivalent on the device since both refer to the same variable. The *update* directive can be used if these data must be updated on the host.
     - Whether these data are members of a class or not, OpenACC/OpenMP memory management routines can also be used.

- If arrays/pointers used in the code region to be parallelized require a manual deep copy, the only solution currently available is to perform the manual deep copy through the *enter data* and *exit data* directives (an example can be seen in Listing 2.15) or memory management routines.

Depending on the application to be parallelized, one of these three methods may be privileged. Of course, it is also possible to use them jointly in the same application. However, the time consumed by copy clauses, data directives, and memory management routines must be considered. We will analyze this question in Section 3.7.

6. Once the necessary actions for data management have been done and the required modifications to ensure that the code region to be parallelized exhibits the most parallelism (as recommended in subsection 1.6.1 - Optimization 1), the next step is to insert compute offloading directives. The directive to insert depend on the pattern of the target code region as well as the signification of the directive itself (or action performed by the compiler when it encounters this directive). We will analyze the interpretation of OpenMP/OpenACC directives by two compilers and the impact of the directive used on the offloaded code region performance in Section 3.6.

7. Both OpenACC and OpenMP provide clauses to set the total number of threads, i.e. the sizes of the grid and the block, that will execute the offloaded regions. It can happen that the grid and block sizes choose by OpenACC/OpenMP compilers is not optimal, in that case the programmer should optimize the threads number manually by using these clauses. This should lead to improvement in speedup. However, the programmer must consider the capacity of the targeted GPU as recommended in subsection 1.6.1 - Optimization 7 and 8.

8. Advanced optimization can be use:

   (a) *Loops fusion* − When loop are perfectly nested, using the *collapse* clause can improve the offloaded code region performance. If the directive-based parallelization method used is OpenACC, the *tile* clause can also be used.

   (b) *Kernels fusion* − It can happen that the offloaded code regions, taken individually, do not contain enough computation. In such case, it may be beneficial to merge these offloaded code regions into one big kernel.

   (c) *Overlapping computation and data transfer* − It can happen that the application to be parallelized contains several sequential part that can be execution on the host concurrently with the regions offloaded on the GPU. Asynchronous clauses, directives and memory routines can be used in such situation. This may improve application overall performance by avoiding of blocking the host execution (as recommended in subsection 1.6.1 - Optimization 2).

   (d) *Multiple device execution queues* − If the application to be parallelized contains multiples regions that can be executed concurrently on the GPU, then two or more streams (or device execution queues) can be created with the OpenACC' *async(integer)* clause.

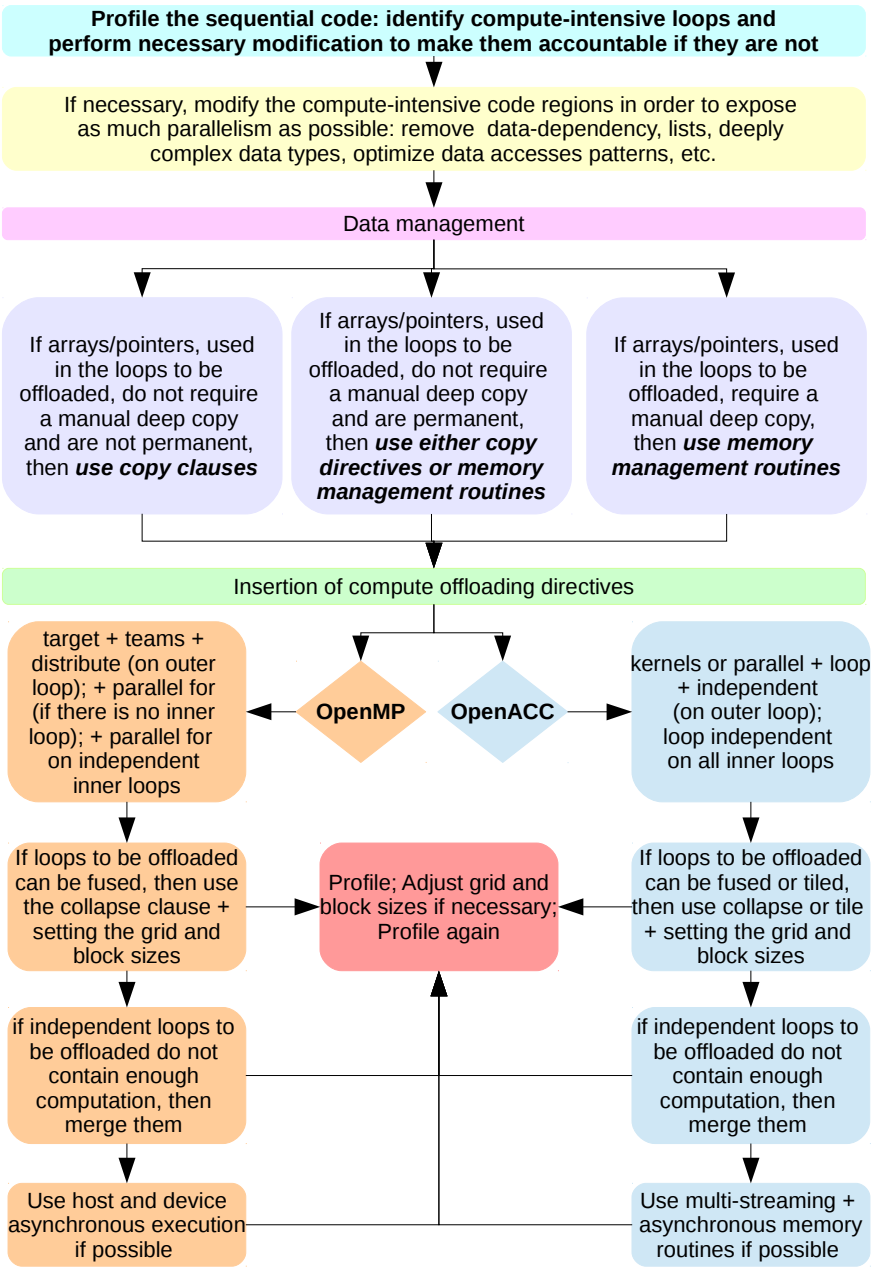


Figure 3.1 – Main steps of the methodology of parallelizing with OpenMP and OpenACC

Figure 3.1 summarizes the parallelization approach discussed above. Some of the above steps need to be applied repeatedly along with profiling and feedback information provided by compilers and profilers. The practices and optimization techniques applied vary depending on the original structure of an application. Some of those techniques are developed in the following sections. While some points (1, 3, 4) discussed above have been used for optimizing parallel program on CPUs, applying them on GPUs pose different challenges, particularly when using them in large code bases.

3.6 OpenMP and OpenACC constructs impact on performance

Both OpenMP and OpenACC provides a rich set of directives for computation offloading on an accelerator (e.g., GPU), which allows a user to gradually increase its application performance by testing difference constructs combination. We have seen in Section 2.3 that OpenMP is *prescriptive*, i.e. the compiler only parallelizes if the user requests it. On the other hand, OpenACC compilers are supposed to do the necessary parallelization from the general information provided by the programmer. OpenMP directives can be used separately or grouped to form combined constructs (see Figure 2.4). In the same way, OpenACC directives can also be uncombined or combined. According to the structure of the code to be offloaded, the user can test severals combination of OpenMP/OpenACC directives.

In this section, we deepen the work presented in our paper [12]. First, we will analyze PGI implementation of OpenACC 2.7 and LLVM/Clang implementation of OpenMP 4.5 by evaluating the performance of the Sobel 3×3 filter (used in previous Chapters) parallelized with different OpenACC/OpenMP directives. The performance used for this evaluation are the kernel execution time, the number of registers used per thread, the number of executed instruction, the number of line of code (LOC) in the generated PTX file, the occupancy, and the SM efficiency. We also report the grid and block sizes generated by the compilers. To be sure that the observed results are independent of the input image size, we use four images of different sizes. Next, we will present the main results for applications seen in Section 3.3. Note that, in this section, we do not optimize the loop scheduling, we let the compiler choose the optimal grid and block sizes.

Analysis of PGI implementation of OpenACC

In order to analyze the PGI implementation of OpenACC, we parallelize the Sobel 3×3 filter with 12 configuration of OpenACC directives. Listing 3.4 shows the *parallel* construct variants while Listing 3.5 presents the different cases for the *kernels* directive.

```
1  #pragma acc parallel deviceptr(in,...) //will be called Par
2  for(int x = 1; x < h - 1; x++)
3    for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
4  #pragma acc parallel deviceptr(in,...) //will be called ParLpUnc
5  #pragma acc loop
6  for(int x = 1; x < h - 1; x++)
7    #pragma acc loop
8    for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
9  #pragma acc parallel loop deviceptr(in,...) //will be called ParLp
10 for(int x = 1; x < h - 1; x++)
11   #pragma acc loop
12   for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
13 //will be called ParLpInd
14 #pragma acc parallel loop independent deviceptr(in,...)
15 for(int x = 1; x < h - 1; x++)
16   #pragma acc loop independent
17   for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
```

```
18 //will be called ParLpIndCol
19 #pragma acc parallel loop independent collapse(2) deviceptr(in,...)
20 for(int x = 1; x < h - 1; x++)
21   for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
22 //will be called ParLpIndTil
23 #pragma acc parallel loop independent tile(32, 32) deviceptr(in,...)
24 for(int x = 1; x < h - 1; x++)
25   for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
```

Listing 3.4 – OpenACC parallel construcs

```
1  #pragma acc kernels deviceptr(in,...) //will be called Krn
2  for(int x = 1; x < h - 1; x++)
3    for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
4  #pragma acc kernels deviceptr(in,...) //will be called KrnLpUnc
5  #pragma acc loop
6  for(int x = 1; x < h - 1; x++)
7    #pragma acc loop
8    for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
9  #pragma acc kernels loop deviceptr(in,...) //will be called KrnLp
10 for(int x = 1; x < h - 1; x++)
11   #pragma acc loop
12   for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
13 //will be called KrnLpInd
14 #pragma acc kernels loop independent deviceptr(in,...)
15 for(int x = 1; x < h - 1; x++)
16   #pragma acc loop independent
17   for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
18 //will be called KrnLpIndCol
19 #pragma acc kernels loop independent collapse(2) deviceptr(in,...)
20 for(int x = 1; x < h - 1; x++)
21   for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
22 //will be called KrnLpIndTil
23 #pragma acc kernels loop independent tile(32, 32) deviceptr(in,...)
24 for(int x = 1; x < h - 1; x++)
25   for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
```

Listing 3.5 – OpenACC kernels construcs

Table 3.6 shows the Sobel 3×3 filter execution time obtained for OpenACC *parallel* and *kernels* constructs variants on the Quadro M2000M GPU. Colored cells shows the best execution time for *parallel* and *kernels* variants. Table 3.7 presents related performances to the PGI generated code properties while Table 3.8 shows the PGI compiler default formula for thread mapping.

For the Sobel 3×3 filter, globally we observe that:

- Parallelizing a code region with the OpenACC's ***parallel*** (Par, lines $1 - 3$ of Listing 3.4) and ***kernels*** (Krn, lines $1 - 3$ of Listing 3.5) directives, used in this way, have the same performance (execution time, number of registers used per thread, grid and block sizes,

Table 3.6 – Sobel 3×3 filter execution time (in millisecond) for various OpenACC directives

parallel	512×512	640×480	1344×372	1342×1110	*kernels*	512×512	640×480	1344×372	1342×1110
Par	93.258	109.2	177.48	536.64	Krn	93.407	109.07	177.7	536.72
ParLpUnc	0.04388	0.05159	0.08157	0.25714	KrnLpUnc	93.613	109.49	178.33	537.17
ParLp	0.04195	0.04965	0.08082	0.25276	KrnLp	0.22098	0.221	0.17248	0.64291
ParLpInd	0.04198	0.04959	0.0802	0.25168	KrnLpInd	0.04753	0.05622	0.09028	0.26805
ParLpIndCol	0.06087	0.0699	0.11333	0.33721	KrnLpIndCol	0.05988	0.06992	0.11361	0.33571
ParLpIndTil	0.06932	0.08022	0.13185	0.39206	KrnLpIndTil	0.06892	0.08037	0.13214	0.39228

Table 3.7 – Sobel 3×3 filter performance for various OpenACC directives

				Executed instructions				
Configuration	Registers	Occupancy (%)	SM efficiency (%)	512×512	640×480	1344×372	1342×1110	PTX LOC
Par	32	1.563	20	8597593	10077209	16396193	49026797	451
ParLpUnc	31	92.97	86.52	485520	548744	830280	2486352	118
ParLp	32	91.965	84.34	485520	548744	830280	2486352	118
ParLpInd	31	92.139	86.81	485520	548744	830280	2486352	118
ParLpIndCol	23	89.441	85.15	894253	1048431	1706933	5103822	102
ParLpIndTil	29	83.899	91.04	1054912	1236080	2046408	6038928	121
Krn	32	1.563	20	8597593	10077209	16396193	49026797	451
KrnLpUnc	32	1.563	20	8598611	10078163	16396931	49029011	450
KrnLp	29	8.535	89.57	314688	368720	599718	1793064	517
KrnLpInd	29	86.966	8654	768608	900600	1464876	4374384	127
KrnLpIndCol	23	90.364	87.18	894253	1048431	1706933	5103822	102
KrnLpIndTil	29	83.913	91.56	1054912	1236080	2046408	6038928	121

Table 3.8 – OpenACC directives default loop scheduling for Sobel 3×3 filter

Constructs	Grid$[x, y, z]$	Block$[x, y, z]$	Sobel filter loops parallelization
Par, Krn, KrnLpUnc	$[1, 1, 1]$	$[1, 1, 1]$	no loop is parallelized
ParLpUnc, ParLp, ParLpInd	$[OutLoopSize, 1, 1]$	$[128, 1, 1]$	outer loop: gang inner loop: vector(128)
ParLpIndCol, KrnLpIndCol	$[OutLoopSize * InnLoopSize/Block.x, 1, 1]$	$[128, 1, 1]$	outer loop: gang, vector(128) inner loop: collapsed
ParLpIndTil, KrnLpIndTil	$[OutLoopSize * InnLoopSize/Block.x, 1, 1]$	$[tileH * tileW, 1, 1]$	outer loop: gang, vector(128) inner loop: tiled
KrnLp	$[InnLoopSize/Block.x, 1, 1]$	$[128, 1, 1]$	outer loop: sequential inner loop: gang vector(128)
KrnLpInd	$[InnLoopSize/Block.x, OutLoopSize/Block.y, 1]$	$[32, 4, 1]$	outer loop: gang, vector(4) inner loop: gang, vector(32)

GPU occupancy, SMs efficiency, executed instructions, and PTX code). However, if they are applied to a structured block containing nested loops as can be seen in Listing 3.6, then they may not have the same GPU occupancy, SMs efficiency, and executed instructions count. Of course, the PTX code and the number of registers used per thread will be different.

```
1 #pragma acc parallel deviceptr(...)
2 {
3   for(int x = 1; x < h - 1; x++) {/*may contains inner loops...*/}
4   for(int x = 1; x < h - 1; x++) {/*may contains inner loops...*/}
5 }
```

Listing 3.6 – Structured block of code

- Parallelizing a code region with the OpenACC's ***parallel*** and ***loop*** constructs in two independent directives (ParLpUnc, lines $4-5$ of Listing 3.4), or in a combined mode (ParLp, line 9 of Listing 3.4), and adding the ***independent*** clause (ParLpInd, lines $14-17$ of Listing 3.4) have approximatively same performance. This induces two important remarks:
  - whether *parallel* and *loop* are combined or not, we get the same performance,
  - if we are not sure that our loops are data independent, we do not have to take the risk of having data races by adding the *independent* clause since PGI compiler can determine the absence of data dependency by itself.

  By simply adding the *loop* construct, instead of having a single thread, we have now $(ImageHeight-2)\times 128$ threads (as can be seen in Table 3.8) that execute the offloaded code. As a consequence, the Sobel filter is speeding up by a factor of $2126.19\times$ compared to the Par version.

- Loops collapsed with the OpenACC's ***parallel loop independent collapse(n)*** (ParLpIndCol, line 19 of Listing 3.4) and ***kernels loop independent collapse(n)*** (KrnLpIndCol, line 19 of Listing 3.5) composite constructs produce the same performance. Thus these two directives are interchangeable. However, if we remove the *independent* clause, they will no longer be equivalent. Indeed, the PGI compiler fails to parallelize the *kernels* variant in that case, it generates only a single device thread. By collapsing the Sobel filter loops, we get a speedup factor of $1562.94\times$ with respect to the Par/Krn version.

- Loop(s) tiled with the OpenACC's ***parallel loop independent tile(n, n)*** (ParLpIndTil, line 23 of Listing 3.4) and ***kernels loop independent tile(n, n)*** (KrnLpIndTil, line 23 of Listing 3.5) composite constructs have the same performance. In the absence of the *independent* clause, the *kernels* version is not parallelized. The PGI compiler fails to compile the code when $(128, 128)$ is used as tile size for both *parallel* and *kernels* variants, while $(64, 64)$ produce erroneous results. By tiling by $(32, 32)$, the execution time is improved by a factor of $1355.74\times$ compared to the Par/Krn version.

- The OpenACC's ***kernels*** directive (Krn) and ***kernels*** plus ***loop*** constructs used in a non-combined fashion (KrnLpUnc, lines $4-5$ of Listing 3.5) are equivalent. Thus the *kernels* and *loop* constructs (without any other loop clauses) must be combined in order to get a parallelized code.

- Combining ***kernels*** and ***loop*** (KrnLp, line 9 of Listing 3.5) improves the execution time by a factor of $695.33\times$ with respect to the Par/Krn version.
- Using the OpenACC's ***kernels loop independent*** composite construct (KrnLpInd, lines $14-17$ of Listing 3.5) improves more the Sobel filter execution time due to the *independent* clause. Indeed, the PGI compiler parallelizes both the outer and the inner loops of the Sobel filter, thus we get one thread per pixel. The speedup factor is $1968.98\times$ compared to the Par/Krn version.
- KrnLpInd and ParLpInd have approximatively the same execution time. Indeed the speedup between KrnLpInd and ParLpInd versions of the Sobel 3×3 filter is $1.114\times$ on average.
- The *collapse* and *tile* clauses reduce the number of registers used per thread.

Analysis of LLVM/Clang implementation of OpenMP 4.5

In order to analyze the LLVM/Clang implementation of OpenMP 4.5, we offloaded the Sobel 3×3 filter with 12 configurations of OpenMP directives. Listing 3.7 shows the *target parallel* variants while Listing 3.8 presents the different cases for the *target teams* directive.

```
#pragma omp target is_device_ptr(in,...) //will be called Tgt
for(int x = 1; x < h - 1; x++)
  for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
#pragma omp target is_device_ptr(in,...) //will be called TgtPrUnc
#pragma omp parallel
for(int x = 1; x < h - 1; x++)
  for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
#pragma omp target parallel is_device_ptr(in,...) //will be called TgtPr
for(int x = 1; x < h - 1; x++)
for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
//will be called TgtPrFr
#pragma omp target parallel for is_device_ptr(in,...)
for(int x = 1; x < h - 1; x++)
  for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
//will be called TgtPrFrIO
#pragma omp target parallel for is_device_ptr(in,...)
for(int x = 1; x < h - 1; x++)
  #pragma omp parallel for
  for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
//will be called TgtPrFrIOSch
#pragma omp target parallel for schedule(static,1) is_device_ptr(in,...)
for(int x = 1; x < h - 1; x++)
  #pragma omp parallel for schedule(static,1)
  for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
//will be called TgtPrFrCol
#pragma omp target parallel for collapse(2) schedule(static,1) \
is_device_ptr(in,...)
for(int x = 1; x < h - 1; x++)
  for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
```

Listing 3.7 – OpenMP target parallel variants

```
#pragma omp target teams is_device_ptr(in,...) //will be called TgtTm
for(int x = 1; x < h - 1; x++)
  for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
//will be called TgtTmDs
#pragma omp target teams distribute is_device_ptr(in,...)
for(int x = 1; x < h - 1; x++)
  for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
//will be called TgtTmDsPrFr
#pragma omp target teams distribute is_device_ptr(in,...)
for(int x = 1; x < h - 1; x++)
  #pragma omp parallel for
  for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
//will be called TgtTmDsPrFrSch
#pragma omp target teams distribute is_device_ptr(in,...)
for(int x = 1; x < h - 1; x++)
  #pragma omp parallel for schedule(static,1)
  for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
//will be called TgtTmDsPrFrIO
#pragma omp target teams distribute parallel for is_device_ptr(in,...)
for(int x = 1; x < h - 1; x++)
  #pragma omp parallel for
  for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
//will be called TgtTmDsPrFrCol in the remaining
#pragma omp target teams distribute parallel for collapse(2) schedule(
    static,1) is_device_ptr(in,...)
for(int x = 1; x < h - 1; x++)
  for(int y = 1; y < w - 1; y++) {/*Sobel 3x3 filter computation ...*/}
```

Listing 3.8 – OpenMP target teams variants

Table 3.9 – Sobel 3×3 filter execution time (in millisecond) for various OpenMP directives

Directives	512×512	640×480	1344×372	1342×1110	*Directives*	512×512	640×480	1344×372	1342×1110
Tgt	130.83	153.26	249.41	746.4	TgtTmUnc	272.5	319.17	521.24	951.5
TgtPrUnc	166.8	195.44	318.22	1553.85	TgtTm	271.5	318.04	518.26	1548.4
TgtPr	138.17	161.94	263.78	789.61	TgtTmDs	2.0866	2.2692	3.9265	11.661
TgtPrFr	2.1634	2.71	4.6391	13.896	TgtTmDsPrFr	0.148	0.1505	0.182	0.584
TgtPrFrIO	2.971	3.5712	5.1383	17.277	TgtTmDsPrFrSch	0.144	0.1469	0.177	0.581
TgtPrFrIOSch	2.9632	3.5622	5.1275	17.281	TgtTmDsPrFrIO	0.764	0.921	1.7344	2.2745
TgtPrFrCol	1.3111	1.537	2.5072	8.913	TgtTmDsPrFrCol	0.395	0.472	0.701	2.1451

Table 3.9 shows the Sobel 3×3 filter execution time obtained for OpenMP *target parallel* and *target teams* constructs variants on the Quadro M2000M GPU. Colored cells shows the best execution time for *target tparallel* and *target tteams* variants. Table 3.10 presents related performances to the

LLVM/Clang generated code properties while Table 3.11 shows the LLVM/Clang compiler default formula for thread mapping.

Table 3.10 – Sobel 3×3 filter performance for various OpenMP directives

Directives	Registers	Occupancy (%)	SM efficiency (%)	PTX LOC	Directives	Registers	Occupancy (%)	SM efficiency (%)	PTX LOC
Tgt	32	6.25	20.00	174	TgtTmUnc	32	79.94	98.97	177
TgtPrUnc	32	6.25	20.00	435	TgtTm	32	84.84	94.03	174
TgtPr	32	6.24	20.00	115	TgtTmDs	32	98.36	99.58	275
TgtPrFr	39	6.09	19.99	236	TgtTmDsPrFr	47	60.68	98.84	640
TgtPrFrIO	70	6.24	19.99	485	TgtTmDsPrFrSch	47	60.64	98.84	643
TgtPrFrIOSch	70	6.24	19.99	485	TgtTmDsPrFrIO	70	11.36	95.87	486
TgtPrFrCol	32	6.24	19.99	239	TgtTmDsPrFrCol	32	89.25	99.67	242

Table 3.11 – LLVM/Clang default loop scheduling for Sobel 3×3 filter

Constructs	Grid$[x, y, z]$	Block$[x, y, z]$	Sobel filter loops parallelization
Tgt	$[1, 1, 1]$	$[128, 1, 1]$	no loop is parallelized
TgtPrUnc	$[1, 1, 1]$	$[128, 1, 1]$	no loop is parallelized
TgtPr	$[1, 1, 1]$	$[128, 1, 1]$	no loop is parallelized
TgtPrFr	$[1, 1, 1]$	$[128, 1, 1]$	inner loop iterations are distributed across the 128 threads of the Block
TgtPrFrIO	$[1, 1, 1]$	$[128, 1, 1]$	loops iterations are distributed across the single thread of the Grid (outer loop) and the 128 threads of the Block
TgtPrFrIOSch	$[1, 1, 1]$	$[128, 1, 1]$	make each thread of the TgtPrFrIO case must execute only one iteration in a round robin fashion way
TgtPrFrCol	$[1, 1, 1]$	$[128, 1, 1]$	outer and inner loops are merged to form a single iteration space which is distributed across the 128 thread of the Block
TgtTmUnc	$[128, 1, 1]$	$[128, 1, 1]$	no loop is parallelized, outer and inner loops iterations are excuted by the device master thread
TgtTm	$[128, 1, 1]$	$[128, 1, 1]$	no loop is parallelized, outer and inner loops iterations are excuted by the device master thread
TgtTmDs	$[OutLoopSize, 1, 1]$	$[128, 1, 1]$	outer loop iterations are distributed teams, and only the teams master thread executes them
TgtTmDsPrFr	$[OutLoopSize, 1, 1]$	$[128, 1, 1]$	outer loop: same thing as in TgtTmDs, inner loop iterations are executed by teams threads (128)
TgtTmDsPrFrSch	$[OutLoopSize, 1, 1]$	$[128, 1, 1]$	inner loop: each teams thread execute one iteration
TgtTmDsPrFrIO	$[OutLoopSize/Block.x, 1, 1]$	$[128, 1, 1]$	
TgtTmDsPrFrCol	$[OutLoopSize* InnLoopSize/Block.x, 1, 1]$	$[128, 1, 1]$	outer and inner loops are merged, the iterations are distributed across teams, and each teams thread executes one iteration in a round robin fashion way

For the Sobel 3×3 filter, globally we observe that:

- The OpenMP's ***target*** (Tgt, line 1 of Listing 3.7) directive must not be used alone.
- The OpenMP's ***target*** directive used with the ***parallel*** construct in a non-combined fashion (TgtPrUnc, lines $4 - 5$ of Listing 3.7) or a combined mode (TgtPr, line 8 of Listing 3.7):

their using must be avoided. They decrease the application performance because the offloaded code is not parallelized and the OpenMP *parallel* directive has an additional cost when it is used in combination with the *target* construct. Indeed, by inspecting the PTX file, we found that LLVM/Clang implementation of OpenMP 4.5 for accelerators programming uses internal routines (e.g. __kmpc_kernel_prepare_parallel, __kmpc_kernel_parallel, __kmpc_for_static_fini, etc.). In fact, for each OpenMP construct LLVM/Clang has an internal routine. The instructions for computing the parameters of these routines as well as their calls are included in the PTX code of the offloaded region. These routines are used by LLVM/Clang for preparing the offloaded code, sending its parameters to device master thread, threads creation and their management, adding the kernel code to device execution queue, synchronization, etc. These routines (depending on the constructs used) are systematically called in each kernel.

- The OpenMP's ***target parallel for*** composite construct (TgtPrFr, line 12 of Listing 3.7) is better than Tgt, TgtPrUnc, and TgtPr. With TgtPrFr the inner loop is parallelized. As result, the Sobel filter is speeding up by a factor of $56.13\times$ compared to the Tgt version.

- Using the ***parallel for*** construct on both outer and inner loop at the same time (as it is the case in TgtPrFrIO, TgtPrFrIOSch, and TgtTmDsPrFrIO) must be avoided since this leads to a performance loss.

- The OpenMP's ***target parallel for collapse(n)*** composite construct (TgtPrFrCol, line 26 of Listing 3.7) must be used instead of TgtPrFr when loops can be collapsed. By using, TgtPrFrCol, both outer and inner loops parallelized. TgtPrFrCol improves the Sobel filter performance with a factor of $95.68\times$ compared to the Tgt version.

- The ***teams*** directive (TgtTm, line 1 of Listing 3.8) must not be used alone. Always add to it the ***distribute*** construct (TgtTmDs, line 5 of Listing 3.8). By adding the ***distribute*** directive, the iterations of the associated loop(s) are distributed across teams master threads. As a results, TgtTm and TgtTmUnc decrease the Sobel filter performance while TgtTmDs speeds up it by a factor of $64.44\times$ with respect to the Tgt version.

- The inner loop(s) of a ***target teams distribute*** construct may be parallelized with the ***parallel for*** directive (TgtTmDsPrFr). Thus, TgtTmDsPrFr speeds up the Sobel filter performance with a factor of $1137\times$ compared to the the Tgt version.

- Adding the clause ***schedule(static, 1)*** to a ***parallel for*** directive inside a *teams region* prevents *teams threads* from having un-coalesced memory accesses. Having coalesced memory accesses can improve GPU applications performance (as seen in Chapter 1: 1.6.1 – Optimization 5).

- LLVM/Clang implementation uses 128 as default block size.

3.6.1 Obtained Results

From lesson learned by porting the Sobel 3×3 to OpenACC and OpenMP, we used the parallelization methodology discussed in Section 3.5 (steps $1 - 6$ and 8a) to parallelize on GPU the applications presented in 3.3. We used the following directives:

- OpenACC
  - ***parallel loop independent*** (on outer loop and ***loop independent*** on inner loop): ParLpInd
  - ***parallel loop independent collapse(n)***: ParLpIndCol
  - ***parallel loop independent tile(n, n)***: ParLpIndTil
  - ***kernels loop independent*** (on outer loop and ***loop independent*** on inner loop): KrnLpInd
  - ***kernels loop independent collapse(n)***: KrnLpIndCol
  - ***kernels loop independent tile(n, n)***: KrnLpIndTil
- OpenMP
  - ***target parallel for***: TgtPrFr
  - ***target parallel for collapse(n)*** : TgtPrFrCol
  - ***target teams distribute*** (on outer loop) plus ***parallel for*** on inner loop : TgtTmDsPrFr
  - TgtTmDsPrFr plus the ***schedule*** clause : TgtTmDsPrFrSch
  - ***target teams distribute parallel for collapse(n)*** : TgtTmDsPrFrCol

OpenACC

Table 3.12 summarizes main optimization used in CUDA versions and those of OpenACC. The CUDA versions of kernels taken from PolyBench benchmark suite are straightforward implementations without any complicated optimization.

Figures 3.2a, 3.2c, and 3.2b present OpenACC versions execution time speed-down with respect to CUDA codes on the Quadro M2000M GPU. On all these three figures, the speed-down is great than $1.65\times$. As previously observations made on the Sobel 3×3 filter:

- ParLpIndCol and KrnLpIndCol versions have same execution time for 28 of the 31 applications we used for our performance evaluation. For the remaining three applications (namely SDisp, Canny, and H&S), the KrnLpIndCol version is at least $2.6\times$ slower than the ParLpIndCol variant.

Table 3.12 – Optimization made in CUDA version vs. OpenACC

Application	CUDA	ParLpInd/KrnLpInd	Par/KrnLpIndCol	Par/KrnLpIndTil
BP	Shared memory	*parallel/kernels + loop independent*	collapse(2)	tile(n, m)
BFS		*parallel/kernels + loop independent*		tile(n)
HS	Shared memory	*parallel/kernels + loop independent*	collapse(2)	
HS3D		*parallel/kernels + loop independent*	collapse(3)	
LMD	Shared memory	*parallel/kernels + loop independent*		tile(n)
LUD	Shared memory	*parallel/kernels + loop independent*		
NW	Shared memory	*parallel/kernels + loop independent*		
SRAD	Shared memory	*parallel/kernels + loop independent*	collapse(2)	tile(n, m)
HW	Shared memory	*parallel/kernels + loop independent*	collapse(2)	
SDisp	Shared memory Texture, SIMD	*parallel/kernels + loop independent*	collapse(2)	tile(n, m)
Canny	Shared memory	*parallel/kernels + loop independent*	collapse(2)	tile(n, m)
HCD	Shared memory	*parallel/kernels + loop independent*	collapse(2)	tile(n, m)
H&D	Shared memory	*parallel/kernels + loop independent*	collapse(2)	tile(n, m)

- ParLpIndTil and KrnLpIndTil produce approximatively the same execution time for 29 of the 31 applications we used for our performance evaluation. For the remaining two applications (namely SDisp and H&S), the *parallel* variant performance is best than tat of the *kernels* version.

As can be seen on Figures 3.2a, 3.2c, and 3.2b, OpenACC codes speed down compared to CUDA hand optimized codes (applications of Table 3.12 except BP and SRAD) can be low as $1.115\times$, but also hight as $30.357\times$. However, for the SDisp kernel where CUDA version is optimized with the shared and texture memory but also SIMD instructions, OpenACC speed down reaches $125.572\times$. On the other hand, OpenACC codes speed down compared to unoptimized CUDA codes (kernels coming from the PolyBench benchmark suites) can be low as $1.048\times$, but also hight as $241.435\times$.

Concerning the comparison between the *kernels* and *parallel* composite constructs (ParLpInd and KrnLpInd):

- As our previous observations for the Sobel 3×3 filter, ParLpInd and KrnLpInd have approximatively the same execution time for 18 of the 31 applications used for this performance evaluation. Listing 3.9 presents some patterns corresponding to these 19 applications. PGI compiler performs the same optimization for codes pattern described in Listing 3.9.

```
1 //pattern 1 (e.g HS, HW, HCD, SRAD)
2 #pragma acc kernels/parallel loop independent ...
3 for(int i = ...; i < ...; i...) {
4   #pragma acc loop independent
5   for(int j = ...; j < ...; j...)
```

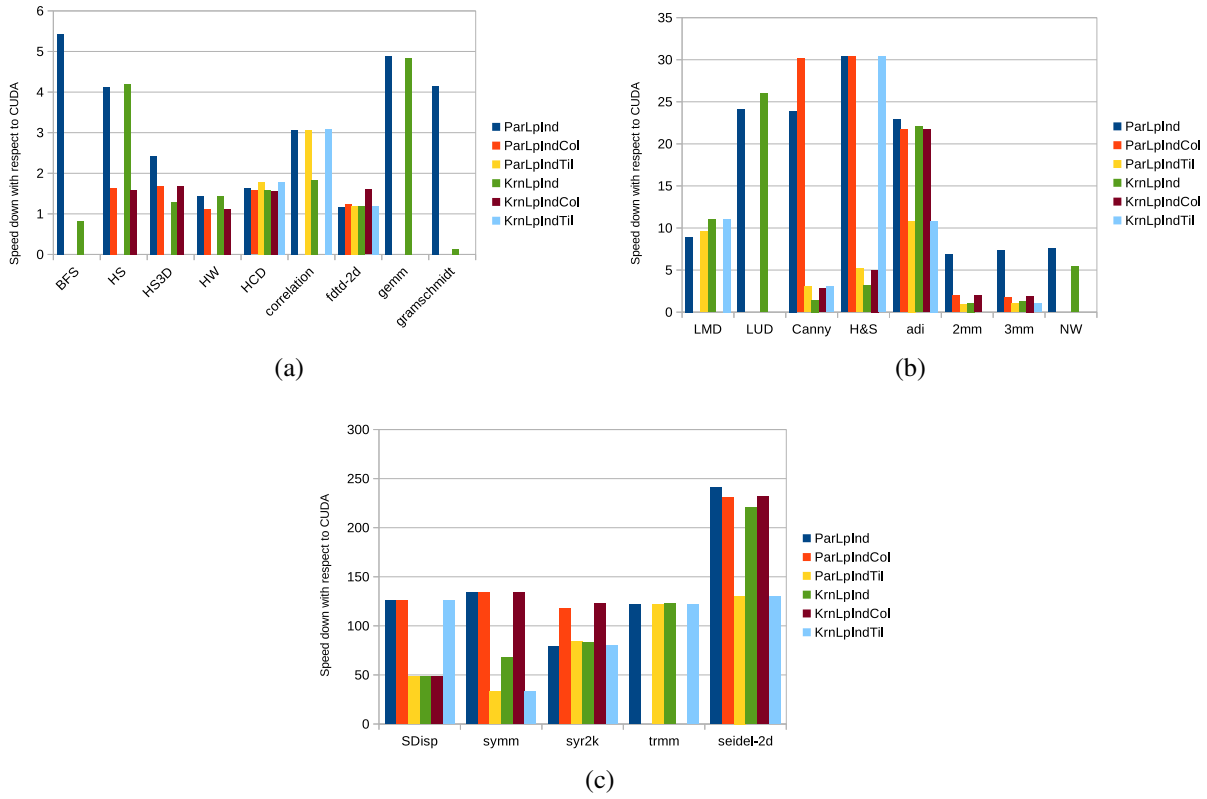


Figure 3.2 – OpenACC speed down with respect to CUDA on the Quadro M2000M GPU

```
6     {/*window based operations with no other nested loops*/}
7   }
8   //pattern 2 (e.g fdtd-2d, adi, gemm, LUD)
9   #pragma acc kernels/parallel loop independent ...
10  for(int i = ...; i < ...; i...) {
11    #pragma acc loop independent
12    for(int j = ...; j < ...; j...) {/*...*/}
13    #pragma acc loop independent
14    for(int j = ...; j < ...; j...) {
15      //may be some other instructions
16      #pragma acc loop independent
17      for(int j = ...; j < ...; j...) {/*...*/}
18    }
19    //....
20  }
21  //pattern 3 (e.g syr2k, trmm, covariance, LMD)
22  #pragma acc kernels/parallel loop independent ...
23  for(int i = ...; i < ...; i...) {
24    #pragma acc loop independent
25    for(int j = ...; j < ...; j...) {
26      // somes instructions
27      for(int k = ...; k < ...; k...) {/*......*/}
```

```
28      //.......
29    }
30  }
```

Listing 3.9 – Code pattern type for which ParLpInd and KrnLpInd have same runtime

- For the remaining 13 applications, KrnLpInd is faster than ParLpInd. Listing 3.10 shows three examples of code patterns for which the KrnLpInd version has a lower execution time than the ParLpInd variant. Consider pattern 1 (lines 2 to 8) of the code below, the PGI compiler parallelizes this pattern according to whether the parent directive is *parallel* or *kernels*:

```
1  //pattern 1 (e.g first and second kernels of correlation, BFS)
2  #pragma acc kernels/parallel loop independent ...
3  for(int i = ...; i < ...; i...) { //somes instructions ...
4    #pragma acc loop independent //(may be reduction(....))
5    for(int j = ...; j < ...; j...)
6      {/*may be a reduction operation, a atomic operation, ...*/}
7    //somes instructions but no other nested loops ...
8  }
9  //pattern 2 (e.g third kernel of correlation)
10 #pragma acc kernels/parallel loop independent ...
11 for(int i = ...; i < ...; i...)
12   #pragma acc loop independent
13   for(int j = ...; j < ...; j...) {/*some point to point operations*/}
14 //pattern 3 (e.g Canny, HCD, SDisp)
15 #pragma acc kernels/parallel loop independent ...
16 for(int i = ...; i < ...; i...)
17   #pragma acc loop independent
18   for(int j = ...; j < ...; j...) { //....
19     for(int k = ...; k < ...; k...) { //may contain some atomic operation
20       #pragma acc loop independent collapse(2)
21       for(int l = ...; l < ...; l...)
22         for(int m = ...; m < ...; m...) {/*window based operations...*/}
23     }
24   }
```

Listing 3.10 – Code pattern type for which ParLpInd and KrnLpInd have not same performance

  - *parallel* – the outer loop is parallelized with *gang*; the inner loop is parallelized with *vector*; if the inner loop contains some reduction operations, then an implicit *reduction* is generated.
  - *kernels* – outer loop is parallelized with *gang* and *vector*; the inner loop remains sequential.

For the second pattern presented in Listing 3.10 (lines 10 to 13), when the parent construct is the *kernels* construct, PGI compiler parallelizes both outer and inner loops with *gang* and *vector*.

- When there are multiple nested loops, as it is the case in the pattern 3 of Listing 3.10 (lines 15 to 24), the PGI compiler optimizes KrnLpInd and ParLpInd codes differently:
  - *parallel* – outer loop is optimized as *gang*, loops of line 18 and 19 remain sequential, and the collapsed loops of lines 21 and 22 are optimized as vector.
  - *kernels* – both outer and inner loops (line 16 and 18) are optimized as *gang* and *vector*, loops of line 19, 21, and 22 remain sequential.

For the 13 applications for which KrnLpInd is faster than ParLpInd, *parallel* typically requires 3.376× the execution time of *kernels* on average.

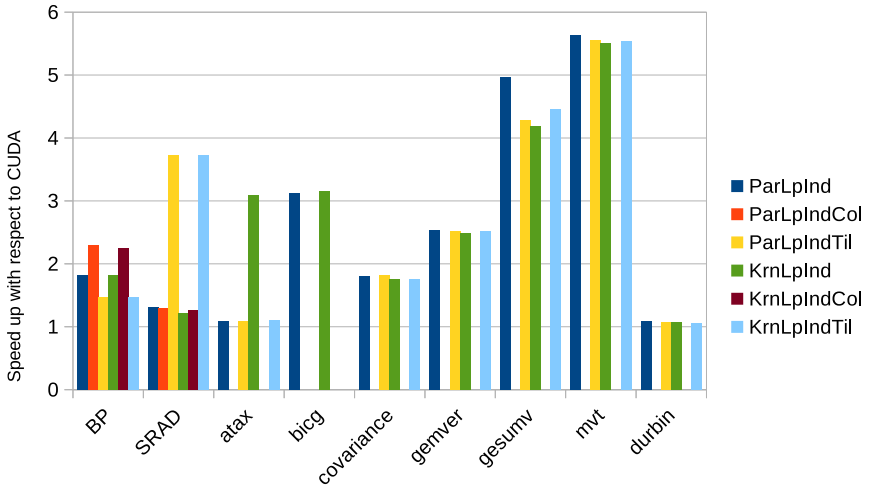


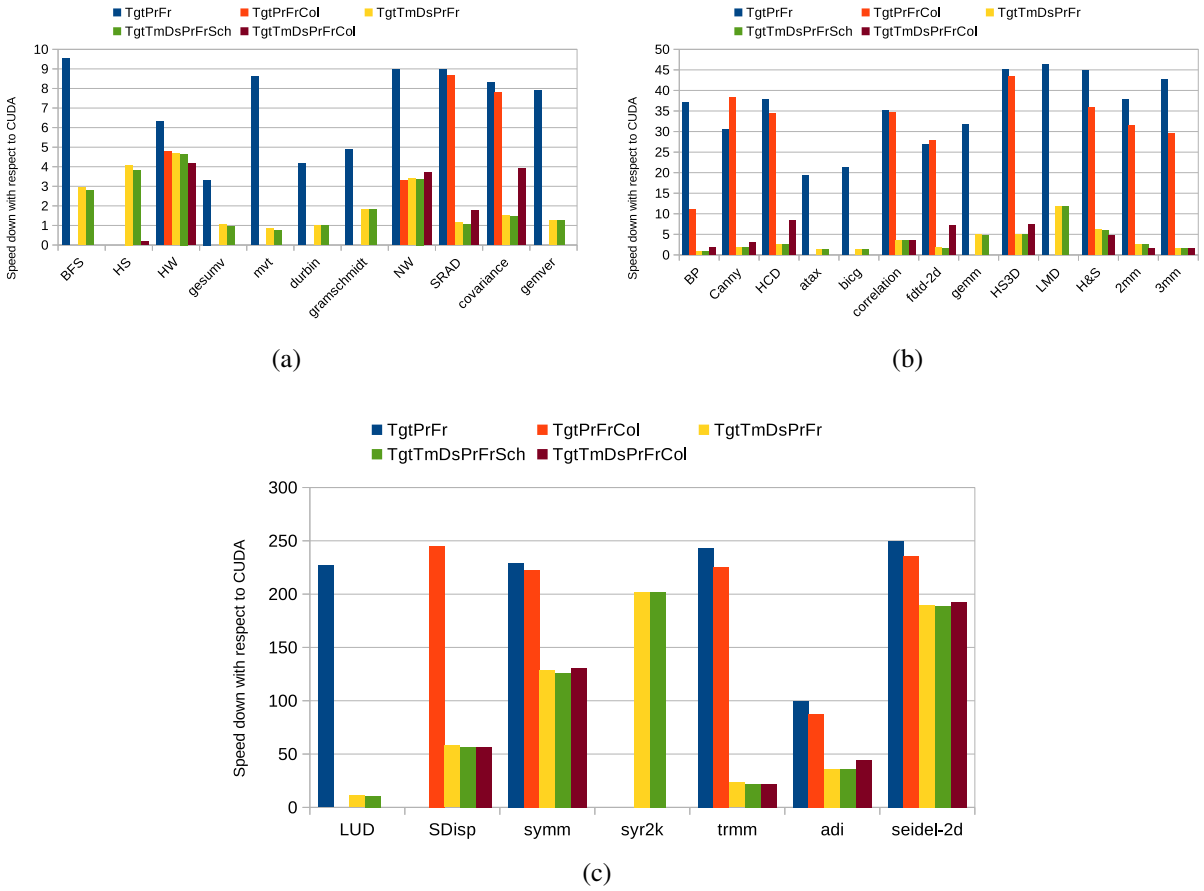
Figure 3.3 – OpenACC speed up with respect to CUDA on the Quadro M2000M GPU

Figure 3.3 shows the 9 kernels for that OpenACC execution time is less than that of CUDA. Although BP and SRAD CUDA versions have been optimized by using the shared memory, they are slower than the OpenACC variants.

In summary, the speed down factor between OpenACC codes and CUDA versions (optimized or not) can be very high for certain applications. These applications are generally either memory bound or have several nested loops which prevent the compiler from properly parallelizing them. Results obtained with this performance evaluation are consistent with observations made in [9], [50], and [53]. The *kernels loop independent* construct may outperform the *parallel loop independent* construct for some code patterns.

OpenMP

In order to evaluate OpenMP offloading features performance we parallelized the applications presented in 3.3 with OpenMP. We used the steps 1 to 6 seen in Section 3.5. The optimizations presented in Table 3.12 for CUDA versions remain unchanged in the experiment made here. The CUDA versions of kernels taken from PolyBench benchmark suite are straightforward implementations without any complicated optimization.



(a) (b)

(c)

Figure 3.4 – OpenMP speed down with respect to CUDA on the Quadro M2000M GPU

Figures 3.4a, 3.4b, and 3.4c present OpenMP versions execution time speed-down with respect to CUDA codes on the Quadro M2000M GPU. On all these three figures, the speed-down is great than $1.777\times$. As previously observations made on the Sobel 3×3 filter:

- TgtPrFrCol is fast than TgtPrFr. TgtPrFrCol improves applications performance with a speed up factor of $1.44\times$ on average. Thus, TgtPrFrCol must be used instead of TgtPrFr when loops can be collapsed.
- Using the *schedule(static, 1)* clause on a *parallel for* directive inside a *teams region* prevents

teams threads from having un-coalesced memory accesses. However, the performance gain is really not significant.

As can be seen on Figures 3.4a, 3.4b, and 3.4c, OpenMP codes speed down compared to CUDA hand optimized codes (applications of Table 3.12 except BP) can be low as $1.777\times$, but also hight as $57.94\times$. However, for the SDisp kernel where CUDA version is optimized with the shared and texture memory but also SIMD instructions, OpenMP (TgtPrFrCol) speed down reaches $245.39\times$. On the other hand, OpenMP codes speed down compared to unoptimized CUDA codes (kernels coming from the PolyBench benchmark suites) can be low as $1.385\times$, but also hight as $249.689\times$.

Unlike OpenACC, OpenMP *teams* variants have a significant speedup with respect to CUDA only for the HS application as can be seen on Figure 3.5.

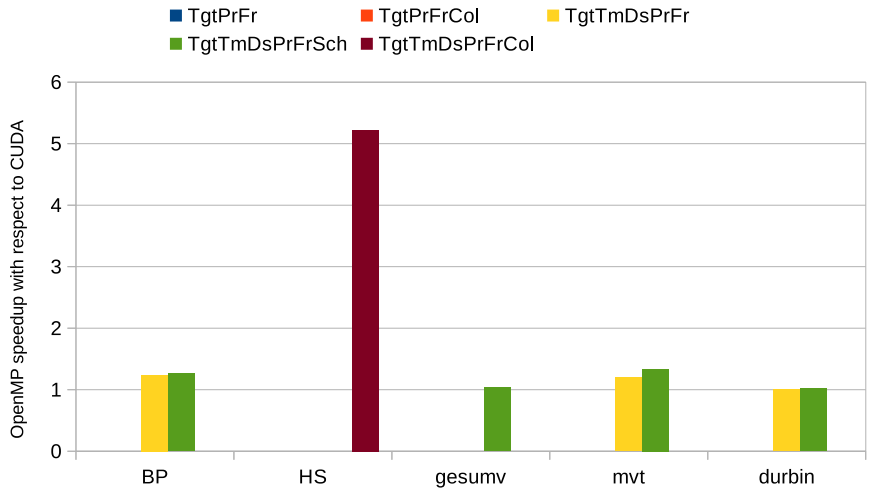


Figure 3.5 – OpenMP speedup with respect to CUDA on the Quadro M2000M GPU

In summary, the speed down factor between OpenMP codes and CUDA versions (optimized or not) can be very high for certain applications. As for OpenACC, these applications are generally either memory bound or have several nested loops. Results obtained with this performance evaluation are consistent with observations made in [9], [59]. The *target teams distribute* construct combined with the *parallel for* directive typically provides best performance than the *target parallel for* construct.

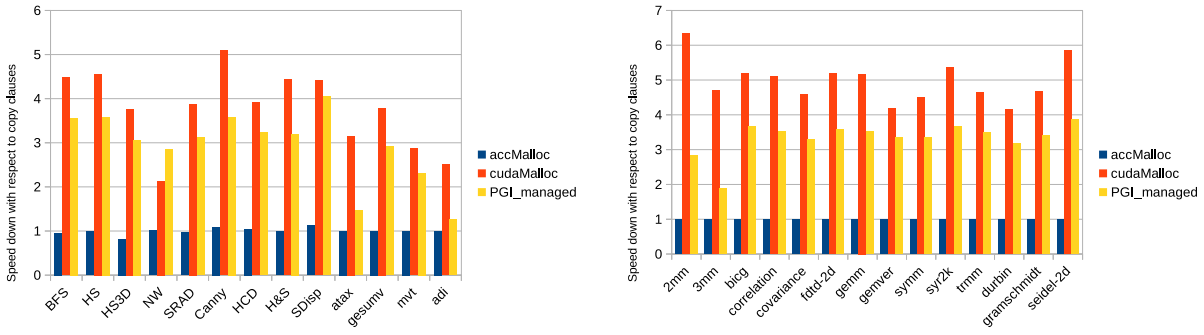
3.7 Memory allocation methods performance comparison

As discussed in Section 3.5 – step 5, there are three ways to manage data transfer between host and device memory in both OpenMP and OpenACC. The three ways may have not the same execution time. In this section, we are going to evaluate the performance of these three ways to manage data transfer. As both OpenMP and OpenACC provide interoperability with CUDA API, we will also use CUDA memory management routines. The data management operations measured in this part are:

- memory allocation (e.g. *create*, *map(alloc:)*, *acc_malloc*, *cudaMalloc*, ...),
- memory deallocation (e.g. *acc_free*, *cudaFree*, *omp_target_free*),
- data transfer from host to device (e.g. *copyin*, *map(to:)*, *acc_memcpy_to_device*, *cudaMemcpy*, etc.),
- data transfer from device to host (e.g. *copyout*, *map(from:)*, *acc_memcpy_to_device*, *cudaMemcpy*, ...)

OpenACC

Figures 3.6a and 3.6b shows the speed down of the OpenACC memory routines, the CUDA memory management routines, and the *tesla:managed* option of PGI compiler with respect to OpenACC copy clauses. Note that the *tesla:managed* option of PGI compiler allows the usage of CUDA Managed Memory by making all data used inside offloaded codes available on both host and device (e.g GPU).



(a) (b)

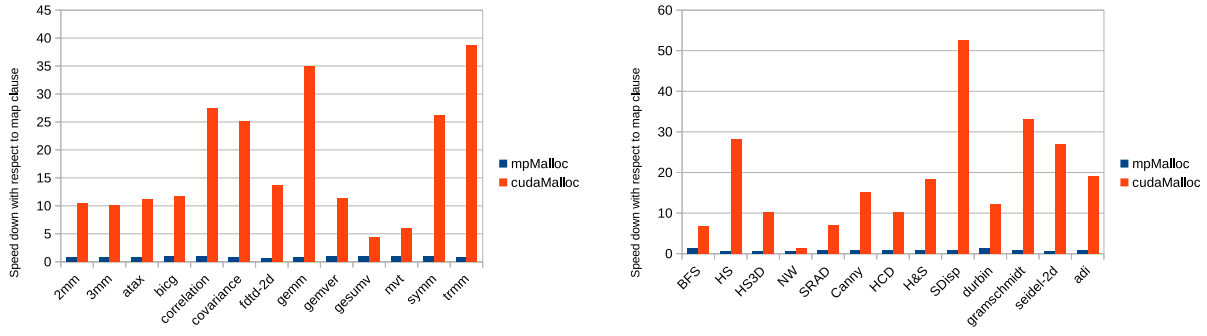
Figure 3.6 – OpenACC memory allocation methods performance on the Quadro M2000M GPU

We observed that OpenACC copy clauses and memory routines are replaced at runtime by many other low-level CUDA memory routines. However, as can be seen in Figures 3.6a and 3.6b, OpenACC memory routines have the same execution time as the copy clauses. CUDA memory management routines require $4.372\times$ of the copy clauses (or OpenACC memory routines) runtime on average. On the other hand, the *tesla:managed* option of PGI compiler takes $3.135\times$ the runtime of copy clauses.

In summary, OpenACC copy clauses, as well as its memory routines, have a lower execution time than CUDA memory management routines and the *tesla:managed* option of PGI compiler. The programmer may use either copy clauses or OpenACC memory routines as much as possible instead of CUDA routines. The *tesla:managed* option of PGI compiler must be used with precaution since this unified memory feature is not a part of OpenACC standard and other OpenACC implementations may not have this feature.

OpenMP

Figures 3.7a and 3.7b shows the speed down of the OpenMP memory routines and the CUDA memory management routines. As it is the case with OpenACC, we observed that OpenMP copy clauses and memory routines are replaced at runtime by many other low-level CUDA memory routines. However, as can be seen in Figures 3.7a and 3.7b, CUDA memory management routines require $18.165\times$ of the copy clauses runtime on average.



(a) (b)

Figure 3.7 – OpenMP memory allocation methods performance on the Quadro M2000M GPU

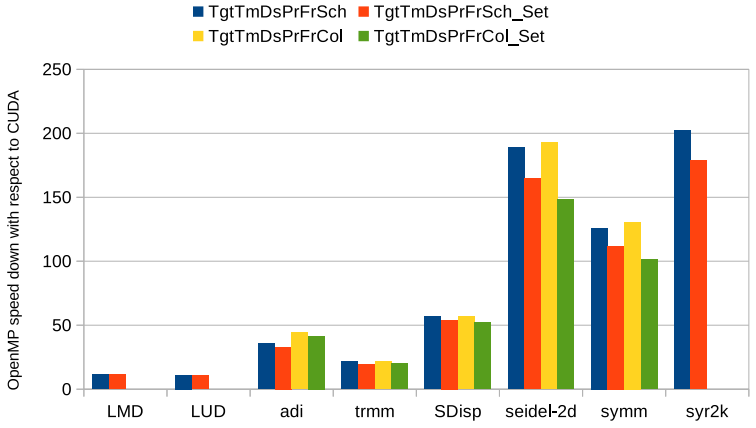
Unlike OpenACC, OpenMP memory routines have not the same execution time as the copy clauses. Indeed, OpenMP copy clauses requires $1.261\times$ OpenMP memory routines execution time. On the other hand, OpenMP copy clauses and memory management routines have a lower execution time than those of OpenACC.

3.8 Optimization via work distribution: thread mapping

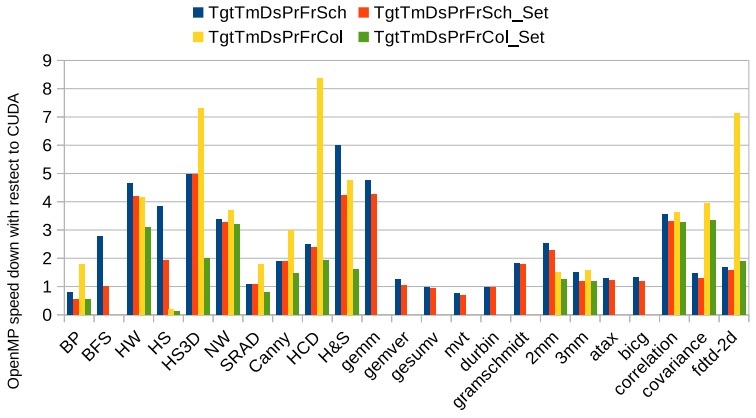
As seen in Chapter 2, both OpenACC and OpenMP provides clauses allowing the user to manually set the number of thread to execute an offloaded code. However, it may happen that the number chosen by the compiler is not the most optimal. In this case, the programmer should adjust the number of threads using the clauses provided for this purpose (e.g. *gang*, *vector*, *num_teams*, etc.).

We have seen in Table 3.11 that LLVM/Clang chooses always 128 as block size regardless of the code and the directives used. For kernels where threads used many registers, this default block size can cause performance loss by causing the register splitting phenomena. In that case, the user must reduce the number of threads per block by setting the *thread_limit* clause.

Figures 3.8a and 3.8a show the obtained results when we manually set the grid and block sizes in OpenMP. In general, assisting the compiler in the choice of the grid and block sizes in OpenMP may improve applications performance. This performance improvement factor can be really significant when the collapse clause is used.



(a)



(b)

Figure 3.8 – Setting grid and block sizes: OpenMP speed down with respect to CUDA on the Quadro M2000M GPU

3.9 OpenACC and OpenMP performance Evolution with Workload

Code generation quality and cost, global memory access pattern and workload are factors which can affect OpenACC and OpenMP performance. Benchmark suites (e.g Rodinia, PolyBench) are good for general performance evaluation. However, they not allow to distinguish the time spent in the management of the kernel (eg placement in device execution queue, threads management, loop index management, ...) of the time devoted to the kernel execution (eg computation, memory accesses, etc.). In order to analysis OpenMP and OpenACC compilers code generation quality and cost, global memory access pattern and kernel workload impact on performance, we will use two simple synthetics kernels: vectors and matrix additions. We are going to gradually increase the number of instructions in the offloaded region loops. In order to prevent the compiler from using

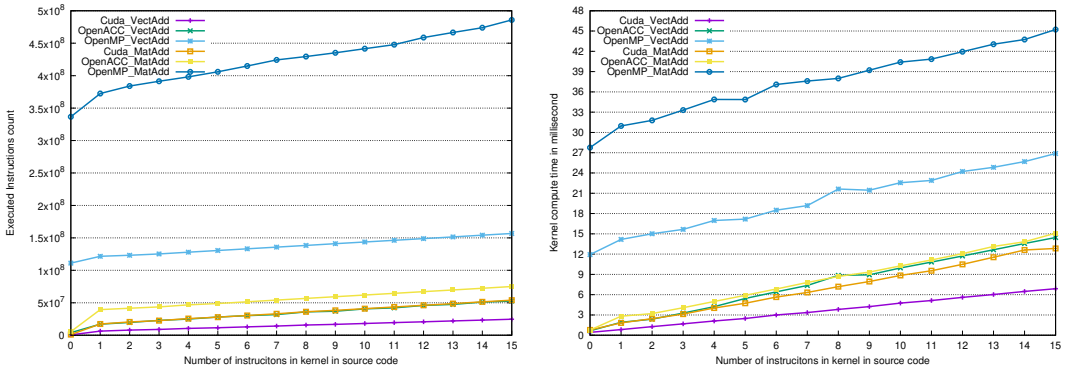
temporal variables, we will use different memory addresses. Listing 3.11 shows the macro used to gradually increase the number of instructions to be executed in loops and the CUDA, OpenACC, and OpenMP versions of the vectors addition kernel.

```
1  //vector addition, "i" is the loop index, "u" is the address offset
2  #define instr(u)    *(C + i + (u)) = *(A + i + (u)) + *(B + i + (u));
3  #define LOOP(i) \
4  LOOP0(i)    \
5  LOOP1(i)    \
6  LOOP2(i)    \
7  LOOP3(i)
8  #define LOOP0(i)  if(i&0x1) instr(0);
9  #define LOOP1(i)  if(i&0x2) {instr(-1); instr(1);}
10 #define LOOP2(i)  if(i&0x4) {instr(-3); instr(-2); instr(2); instr(3);}
11 #define LOOP3(i)  if(i&0x8) {instr(-7); instr(-6);instr(-5); instr(-4); \
12                              instr(7); instr(6);instr(5); instr(4);}
13 //CUDA version
14 __global__ void vectCUDA(int* A, int* B, int* C) {
15   int i=blockDim.x*blockIdx.x+threadIdx.x;
16   if(i < 7 || i > (N - 7)) return;
17   LOOP(NOMBRE);
18 }
19 //OpenACC version
20 void vectOpenACC(int* A, int* B, int* C) {
21   #pragma acc kernels loop independent deviceptr(A, B, C)
22   for(int i = 7; i < N - 7; i++) {
23     LOOP(NOMBRE);
24   }
25 }
26 //OpenMP version
27 void vectOpenMP(int* A, int* B, int* C) {
28   #pragma omp target teams distribute parallel for is_device_ptr(A, B, C)
29   for(int i = 7; i < N - 7; i++) {
30     LOOP(NOMBRE);
31   }
32 }
```

Listing 3.11 – The vectors addition kernel with the macro used for workload evaluation

Figures 3.9b and 3.9a show the vectors and matrix addition kernels compute time and executed instructions evolution with workload. In results presented here, vectors and matrix sizes are 4096×4096 and data type is *float*.

In terms of execution time, we found that the speedup between the access in $1D$ and that $2D$ is on average 0.524 for OpenMP, 0.907 for OpenACC and 0.526 for CUDA. On the other hand, for the executed instructions count, this speedup is worth 0.324 for OpenMP, 0.609 for OpenACC and 0.355 for CUDA. We note that these results are independent of data type and arrays size. Compute time and executed instructions count evolution with the workload is fairly stable for both OpenACC, OpenMP and CUDA. This confirms that the additional cost associated with code generation is not significant.



(a) Executed instructions GPU

(b) Execution time

Figure 3.9 – Evaluation of workload impact on performance on the Quadro M2000M GPU

As can be seen in Figure 3.9b and 3.9a, local memory access pattern does not affect performance for OpenACC if there is enough work in the kernel. In CUDA data access in $2D$ instead of $1D$ can have a significant impact on performance as the workload in the kernel increases. On the other hand, in OpenMP, accessing global memory data in a $2D$ fashion has a significant impact on kernel performance compared to $1D$ access.

As said in 3.6 and as can be seen in the figure 3.9a, OpenMP versions execute 2 to $8\times$ instructions more than CUDA versions and 2 to $6\times$ more than OpenACC versions. This shows that the current implementation of OpenMP in LLVM/Clang still needs improvement in order to generate better PTX code.

3.10 Conclusion

In this chapter, we analyzed PGI compiler implementation of OpenACC and OpenMP 4.5 support in LLVM/Clang. Our results show that both OpenACC/OpenMP' support, in PGI/Clang compiler, comply with the official specifications of OpenACC and OpenMP standards. We proposed an efficiently parallelization methodology with OpenACC/OpenM. The proposed methodology has been used to parallelize kernels and applications taken from publics benchmark suites. The obtained results indicate that:

- The directives can be combined in various ways and this has a significant impact on the final performance of the offloaded code.
- It is insufficient to simply insert *kernels* or *parallel* or *target* to inform the compiler that that a particular code region must be compiled for execution on a device (e.g. GPU).
- It is highly essential to combine offloading directives (ie *kernels*, *parallel*, *serial*, and *target*) with loop parallelization constructs (eg *loop*, *teams distribute*, *parallel for*). Although current compilers are mature and perform several optimizations, the user may provide them

more information through loop parallelization constructs' clauses (eg *collapse*, *tile*, *independent*, etc.) in order to obtain an optimized code.

- Exploring optimization techniques, several of those discussed in this chapter, to not only give the compiler adequate information to perform the necessary transformations, but also perform transformations that can exploit the target hardware efficiently. Other optimization (eg caching read-only data, asynchronous execution, multi-streaming), which will be discussed in the next chapters, may be helpful to improve the performance of an OpenMP/OpenACC application.

- There is, in general, a systematic overhead in the kernel launch for OpenMP, but for most applications, it's not a big issue provided the kernel has a sufficient workload.

Programmability and Code Portability

Programming heterogeneous systems can be simplified using OpenACC or OpenMP. Their main advantages are that they do not require significant changes to the original source code, the development time is relatively low, they can target multiples devices from different platform manufacturers, leading to considerably lesser code maintenance. However, in order to achieve good performance, the user's intervention is required to manually apply certain code transformations. This is because the compiler is not intelligent enough yet to determine certain information (e.g. absolute absence of data dependency, optimal loop scheduling, data movement optimization, etc.). Furthermore, the OpenMP API does not cover compiler-generated automatic parallelization, so the user must give the compiler as many hints as possible. Currently, there is no compiler support that can effectively utilize the registers and shared memory in GPU that play an important role in GPUs. There is also no to access to the texture memory. Data movement can be costly, it has been the user's responsibility to choose the necessary data management method in order to get the best performance.

Performance Portability

Achieving performance portability can be quite delicate. Different architectures (whether or not from the same manufacturers) may demand distinct programming requirements. Merely considering the two NVIDIA GPUs used in this thesis (see their description in Sections 1.5.3 and 1.5.4). To avoid having inactive SMs and thus exploit all available computing resources on the GPU, the grid size must be a multiple of 5 because the Quadro M2000M GPU have five SMs. However, the optimal size of the grid on the Tegra X1 GPU must be a multiple of 2. As seen in Table 1.3, the Quadro M2000M GPU have more cores, higher memory capacity, and bandwidth than the Tegra X1 GPU. Thus, transferring a large data once and having a single kernel that has a significant workload may be suitable for the Quadro M2000M GPU but not for the Tegra X1 GPU. Instead, it may be preferable to split the large data and the kernels into data chunks and small kernels in order to overlap data transfer and computation.

Performance portability is not only an issue with just the architecture but also an issue that different compilers can provide a different implementation for an OpenACC/OpenMP directive/clause. Moreover, the quality of the compilation matters significantly. This is especially true for OpenACC because as seen in 2.3, the basic principle of the OpenACC programming model is the user describes the parallelism and data motion in a more general and the compiler must perform necessary optimization to map parallelism to the underlying hardware. For example, the OpenACC *kernels* directive allows the compiler to choose the loop-scheduling technique to be applied i.e. analyze and schedule each loop level to *gang*/*worker*/*vector*. A compiler can use its own technique to schedule the loop nest to nested gang, worker, and vector, and this is typically not part of OpenACC standard. As a result, the performance obtained using the kernels directive can be different for different compilers. On the contrary, the code that uses *parallel* directive is more portable since this allows the user to have control over explicitly adopting the loop scheduling. Another example if the number of teams created by an OpenMP compiler. Typically, LLVM/Clang generates a grid size of 128 for a *teams* construct, but when adding the *distribute* directive, the grid size depends on the size(s) of the loop(s) associated to the *distribute* directive. On the other hand, whether the *distribute* directive is added to the *teams* construct or not, the Cray compiler generates a grid size equal to the size of the outer loop.

Caches Optimization

We have already seen that NVIDIA GPUs have several types of memory (global, constant, shared, L1 and L2 caches, texture, registers, ...) and that caches and registers are faster than the global memory. On Maxwell GPUs, L2 cache is used to cache accesses to local or global memory, including temporary register spills. Global memory accesses are always cached in L2. However, data that is read-only for the entire lifetime of the kernel can also be cached in the unified L1/texture cache. Caching read-only data in L1 may improve application performance. Nevertheless, the compilers might not always be able to detect that the read-only condition is satisfied for some data because of pointers aliasing issues. Although the usage of both L1 and L2 is controlled by the hardware and they are not manageable by the programmer, it can mark pointers used for loading read-only data with both the *const* and the *__restrict__* qualifiers. In the next chapter, we will discuss the *__restrict__* qualifier and another code optimization method in order to help compilers to cache read-only data into the L1 cache.

Chapter 4

Towards Pointer Aliasing on GPU

Contents

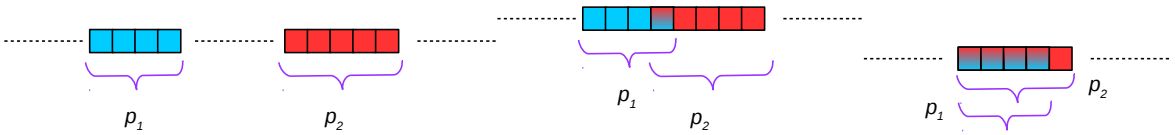
NVIDIA GPUs have several types of memory (global, constant, shared, L1 and L2 caches, texture, registers, ...). We have already mentioned that caches and registers are faster than the global memory. Data that are read-only for the entire lifetime of the kernel can be cached in the unified L1/texture cache of the GPU in order to improve application performance by reducing global memory accesses. Although PGI, LLVM/Clang and NVIDIA compilers are mature and powerful, they sometimes fail to detect that the read-only condition is satisfied for some data due to the presence of pointers. Thus, pointers can hinder compiler optimization. Indeed, it is hard to know where pointers are pointing and compilers must be conservative in their presence. As result, the programmer's intervention is often required to help compiler for loading read-only data in the L1 cache. In this chapter, we will start by giving an introduction to pointer aliasing and it's impact on compiler optimization and therefore on application performance (Section 4.1.). Next, we will present existing solutions in Section 4.2. Since many works had been done on pointer aliasing problem, we will present some of them in Section 4.3. In order to help GPU programmer to optimize their code and tell the compiler that the pointers are not aliased, we propose two static analysis tools in this chapter. Since our tools are based on a static analyzer framework, namely Frama-C, we will give a brief introduction on this environment in Section 4.4. The next sections 4.6 and 4.5 explain the design of our two statics analysis tools published in our works [77, 78]. We evaluate our tools in Section 4.7 and present the obtained results.

4.1 Pointer Aliasing

The major difficulty of the C-like languages (e.x. C, C++, CUDA, etc.) comes from the freedom they offer developers to manipulate memory by the means of pointers. These memory manipulations through the pointers must be analyzed in order to be able to determine read and write accesses. Determining statically (at compile-time) for each pointer the list of locations to which it will point at the runtime is an undecidable problem [79]. The difficulty comes from the different execution paths that can not be predicted as well as the dynamic memory allocation whose success can only be verified at runtime. At runtime, a pair of pointers may have the following behaviors, depicted in Figure 4.1.

- *No Alias:* p_1 and p_2 are disjoint. They do not reference overlapping memory regions (Sub Figure 4.1a).
- *Alias:* p_1 and p_2 reference regions that overlap. In this case, we can distinguish between partial alias (Sub Figure 4.1b) where p_1 and p_2 overlap in some way but do not start at the same address and must alias (Sub Figure 4.1c) where p_1 and p_2 alias and start at the same address.

Although pointers are powerful and convenient, their presence in a program creates a real uncertainty about the data accessed by the instructions. Variables are modified by dereferencing pointers. In the absence of information on the memory location to which the pointer points, it is necessary to analyze the program under the assumption that all the variables of the program can be read or written by each dereference. This uncertainty on pointer targets can hinder several code optimizations



(a) noalias (b) partial alias (c) must alias

Figure 4.1 – Simple representation of memory regions in case of no, partial and must aliasing. Each □ represents a memory slot

made by compilers like code motion, instruction rescheduling, dead code suppression, constant propagation, etc.

Code motion consists of interchanging some statements or moving others. This optimization is beneficial especially in loops where loop invariant code motion allows removing unchanged statements from the loop by hoisting or sinking them and therefore to avoid useless executions. Code motion is also used to improve data locality that optimizes memory and cache accesses. Listing 4.1 shows an example for possible loop invariant code motion. In this example, there is a read from *ptr* and a write in *ptr_2* in the *while* loop. It is obvious that *ptr* is never changed inside the loop since there is no writing in *ptr*, and *ptr* and *ptr_2* access disjoint pieces of memory. Therefore, if the compiler knows this information, it can move out the line 8 of the loop.

```
1  int *ptr = (int*)malloc(4 * N * sizeof(int));
2  int *ptr_1, *ptr_2, *a;
3  *ptr = 255;
4  *a   = 128;
5  ptr_1 = ptr + N;
6  ptr_2 = ptr + 4 * N;
7  while(ptr_2 > ptr_1) {
8    a = *ptr;
9    *ptr_2 = 4;
10   ptr_2--;
11 }
```

Listing 4.1 – Example for possible loop invariant code motion

Instruction rescheduling means interchanging some program statements. This option can be profitable to improve data locality or to enable further optimizations. Listing 4.2 presents an example where instruction can be rescheduled. If the compiler knows that *A* and *B* are not alias, it can interchange the first and second loop to improve data locality since *B* is written in the first loop (line 2) and then read in the third loop (line 6). It also can merge, in a second pass, the filling of *B* and *C* in the same loop.

```
1  //original version
2  for(int i = 0; i < N; i++)
3    B[i] = initVar;
4  for(int i = 0; i < (3 * N); i++)
5    A[i] = ....;
6  for(int i = 0; i < N; i++)
```

```
7   C[i] = B[i];
8  //interchanging the first and second loop to improve data locality
9  for(int i = 0; i < (3 * N); i++)
10   A[i] = ....;
11 for(int i = 0; i < N; i++)
12   B[i] = initVar;
13 for(int i = 0; i < N; i++)
14   C[i] = B[i];
15 //rescheduling by merging the second and third loop
16 for(int i = 0; i < (3 * N); i++)
17   A[i] = ....;
18 for(int i = 0; i < N; i++)
19   B[i] = initVar;
20   C[i] = initVar;
```

Listing 4.2 – Example for possible instruction reschduling

Dead code suppression consists of remove useless statements (generally assignments) such as overwritten variables or writes that are never read. For example, let us consider again the code provided in Listing 4.2. If *B* and *C* start at the same location and therefore must alias, thus there is no need to fill in table *C* at line 7 because it is already done in the first loop at line 3 which fills table B. Loop at line 7 could be removed.

Constant propagation consists of substitute the values of known constants in expressions at compile time. Considering the example provided in Listing 4.3, to determine the effect of executing the instruction at line 3, the compiler needs to know where variable *ptr* might point. For example, if compiler has precise information and knows that *ptr* points to *x*, then it can determine that *x* has the constant value 100 at line 4. Similarly, if the compiler is certain that *ptr* does not point to *x*, then it can determine that *x* has the constant value 0 at line 4. By contrast, if the compiler has no precise information, then it must assume that *ptr* might or might not point to *x*. In this last case, the compiler can not determine the value of *x* at statement 4.

```
1 int32_t x = 0;
2 int32_t *ptr;
3 //.....
4 *ptr = 100;
5 write(x);
```

Listing 4.3 – Example for possible constant propagation

4.1.1 Problem Statement

Let us consider the CPU C++ code of the *gesummv* kernel (from PolyBench benchmark suite) provided in Listing 4.4. Without further knowledge or special hardware support, the compiler must assume that A, B, x, y and tmp might refer to the same memory region or overlapping regions. Hence, the loop cannot be parallelized or software-pipelined because it has to be ensured

that an update of $tmp[i]$ is performed before the next value of $x[j+1]$ is loaded for example. This constraint also prevents the compiler from generating loads for multiple array elements of A, B and x at the same time since the subsequent store to $tmp[i]$ might modify elements of arrays A, B or x. Therefore, the compiler will compile this code inefficiently because it must conservatively assume the pointers alias.

```
1 void gesummvCPU(float *A, float *B, float *x, float *tmp, float *y) {
2   for (int i = 0; i < NI; i++) {
3     for (int j = 0; j < NJ; j++) {
4       tmp[i] += A[i * NJ + j] * x[j];
5       y[i]   += B[i * NJ + j] * x[j];
6     }
7     y[i] = ALPHA * tmp[i] + BETA * y[i];
8   }
9 }
```

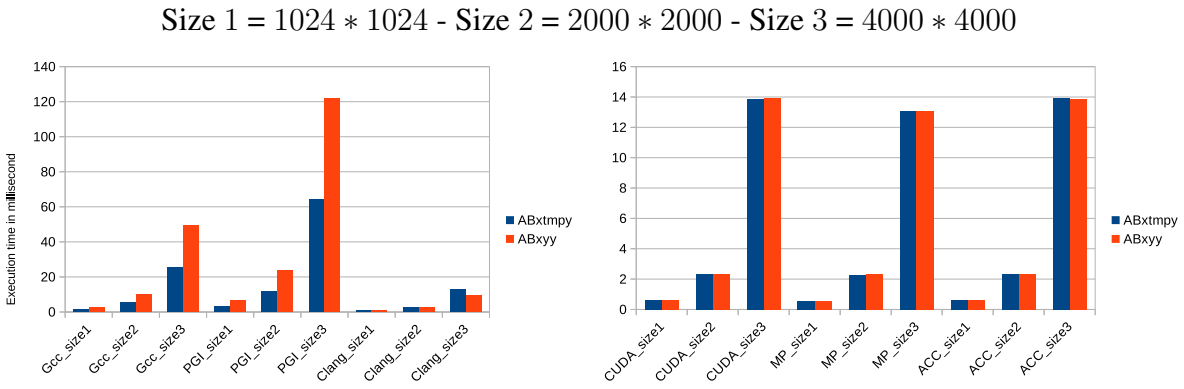
Listing 4.4 – Gesumv kernel CPU C++ code

Listing 4.5 shows a simple implementation of *gesummv* kernel in CUDA (lines 1 to 9), OpenMP (lines 10 to 20) and OpenACC (lines 21 to 30) of the CPU C++ version seen in Listing 4.4. Now let's evaluate the performance of the CPU'version on an Intel I7 CPU and GPU'versions on the Quadro M2000M GPU by considering two cases: (1) calling *gesummv* kernel with five distinct pointers *(A, B, x, tmp, y)*; (2) calling *gesummv* kernel with the same pointer for its last two arguments *(A, B, x, y, y)*. Of course the two calls will not give the same computation results, but it does not matter in the present case. Compilers as well as compilation flags being used are those presented in Table 3.3 in Chapter 3. For each version, the execution time presented on Figure 4.2 is the average of 100 executions.

```
1  __global__ void gesummvCUDA(float *A, float *B, float *x,
2                              float *tmp, float *y) {
3    int i = blockIdx.x * blockDim.x + threadIdx.x;
4    if (i > NI) return;
5    for(int j = 0; j < NJ; j++) {
6      tmp[i] += A[i * N + j] * x[j];
7      y[i] += B[i * N + j] * x[j];
8    }
9    y[i] = ALPHA * tmp[i] + BETA * y[i];
10 }
11 void gesummvMP(float *A, float *B, float *x, float *tmp, float *y) {
12   #pragma omp target teams distribute parallel for schedule(static, 1) \
13   is_device_ptr(A, B, x, tmp, y)
14   for (int i = 0; i < NI; i++) {
15     for (int j = 0; j < NJ; j++) {
16       tmp[i] += A[i * NJ + j] * x[j];
17       y[i]   += B[i * NJ + j] * x[j];
18     }
19     y[i] = ALPHA * tmp[i] + BETA * y[i];
20   }
21 }
```

```
22 void gesummvACC(float *A, float *B, float *x, float *tmp, float *y) {
23   #pragma acc kernels loop independent deviceptr(A, B, x, tmp, y)
24   for (int i = 0; i < NI; i++) {
25     for (int j = 0; j < NJ; j++) {
26       tmp[i] += A[i * NJ + j] * x[j];
27       y[i]   += B[i * NJ + j] * x[j];
28     }
29     y[i] = ALPHA * tmp[i] + BETA * y[i];
30   }
31 }
```

Listing 4.5 – Simple implementation of gesummv kernel in CUDA/OpenACC/OpenMP



(a) Intel I7 CPU

(b) Quadro M2000M GPU

Figure 4.2 – Execution time of calling the *gesummv* kernel with ABxtmp and ABxyy

We can see in Figure 4.2a that when Gcc or PGI compiler is used, calling *gesummv* kernel with the same pointer for *tmp* and *y* requires on average $1.96\times$ the execution time of the case where it is called with five distinct pointers. On the other hand, when the compiler used is Clang, the two call contexts have the same execution time. By inspecting assembly files generated by the three compilers we discovered that Gcc and PGI unroll the *j* loop 8 times, but PGI uses Fused Multiply-Add (FMA) instructions in *j* loop while Gcc does not. ON the other hand, Clang does not unroll the *j* loop. None of the three keep *tmp[i]* and *y[i]* in temporary registers for the duration of the loop *j*.

Unlike the CPU case, as we can see in Figure 4.2b, the call context of the *gesummv* kernel has no impact on its performance on GPU whether with CUDA or OpenACC or OpenMP. By inspecting the PTX generated for CUDA, OpenACC and OpenMP we found that the *j* loop is not unrolled and thereby for each iteration of this loop, the current thread must load from global memory *A[i * NJ + j]*, *B[i * NJ + j]*, *tmp[i]*, *y[i]*, and *x[j]* (two times); make two FMA operations; and store *tmp[i]* and *y[i]* to global memory. For line 8 (see Listing 4.5), *tmp[i]* and *y[i]* are reloaded again from global memory to perform an FMA operation and then save *y[i]* to global memory.

As result, calling a GPU kernel with aliased or non aliased pointers produce the same compute

time. Thus, it may be useful in the absence of aliasing to inform the compiler that kernel pointers arguments are not aliased, so that it can generate more optimized code.

4.2 Existing Solutions

To mitigate the problem posed by pointers, many solutions have been proposed. The first solution consists of alias analyzing which is performed by compilers. When compiling a code, alias analysis is typically performed many times by the compiler. In this section, we give a brief introduction to alias analysis. The two other solutions discussed in this section must be done by the programmer at source level.

4.2.1 Alias Analysis

To ensure that the correct code is generated in the presence of aliases, compilers have to perform an alias analysis to determine the aliases in the program. Alias analysis (or "pointer analysis" or "points-to analyses") attempts to discover, statically (at compile-time), the possible values of a pointer at runtime. Typically, this information is used to optimize programs or prove their correctness, which can be done by eliminating dead code and null pointers, parallelizing, rescheduling, code motion, detecting bugs due to array out-of-bounds accesses, etc. Many approaches that trade precision for speed, speed for precision, or even attempt to balance between both, have been proposed and improved in time and space.

Although the literature of alias analysis is abundant and many works (e.g: [80, 81, 82]) have been done in the last few decades, the research community has not yet solved pointer alias analysis satisfactorily. Many alias analyzer are implemented in mainstream compilers, but the results of these analyzers are often inaccurate. Pointer analysis imprecision is a severe problem while it prevents the compiler from optimizing some code where there is no aliasing. There are many challenges for alias analysis including:

- Complexity that is huge in space and time. Indeed, every pointer should be compared with every other pointer at every program point and potentially considering all program paths to that point.
- Accuracy, how few pairs of pointers are reported while remaining correct.
- Coding corner cases, pointer arithmetic (*p++), casting, function pointers, long-jumps, etc.
- Deciding if analyze the whole program, library code or not, optimizing at link-time only or not, ...

Alias analysis can be classified as *flow-sensitive* or *flow-insensitive*, depending on whether statements order information is used during the analysis. A flow-insensitive analysis produces sets of memory locations that pointers may points to at any point of the program. For sequence of code

from line 1 to line 3 of Listing 4.6, the output of a *flow-insensitive* analyzer can be: p may points to a or b. On the other hand, a *flow-sensitive* analysis computes alias information at every point of the program. For example, a *flow-sensitive* approach is capable to determine that between line 2 and 3 of Listing 4.6, p points to a, and after line 3, p points b. By not considering the order of statements, and therefore computing a conservative summary, a *flow-insensitive* analysis can be more efficient, but less accurate than a *flow-sensitive* analysis [83, 84].

In addition to *flow-sensitivity*, there are several other design options including *context sensitivity* and aggregate modeling. If different calling contexts are considered, the analysis is *context-sensitive* otherwise it is *context-insensitive*. *Context-sensitive* analysis considers the calling context (caller) when analyzing the target of a function call (callee). A *context-sensitive* alias analyzer needs to have a way to create an abstract description for f (in Listing 4.6, line 5), so that every time it is called, it can apply the calling context to the abstract description. *Context-sensitive* alias analyzer provides a finer grain model of the static code hence results in higher precision.

```
1  int a, b, *p;
2  p = &a;
3  p = &b;
4  int c, d, *x;
5  void f() { *x++; }
6  void main() {
7    x = &c;
8    f();
9    x = &d;
10   f();
11 }
12 struct { int a; b; } x, y;
13 int A[10];
```

Listing 4.6 – Example code for alias analyzer property

Another property is the filed modeling. If structures or arrays elements are treated as individual locations, the analysis is *field-sensitive* otherwise it is *field-insensitive*. *Field-sensitive* approach models each field of each struct variable. Considering Listing 4.6, for the structure defined at line 12, a *field-sensitive* alias analyzer create four nodes (pointer, variable or memory location): $x.a$, $x.b$, $y.a$ and $y.b$. The same principle applies when dealing with arrays, for instance the array defined at line 13, *field-sensitive* approach creates 10 nodes: $A[0]$, $A[1]$, ..., $A[10]$. On the other hand, *field-insensitive* approach models each struct variable (e.x. 2 nodes $x.*$ and $y.*$ for the struct defined at line 12 of Listing 4.6) but does not model their fields.

4.2.2 Languages Supports

To palliate the problem posed by pointers, some programming languages furnish developers with high-level constructs that they can use to tell compilers that variables do not alias each other. Therefore the C programming language, since the C99 Standard, features the *restrict* keyword, that can be used by the programmer to give the compiler information about aliasing. In fact, it is

applied to a pointer p to say that only p or a pointer derived from it can access that memory region during its lifetime. Hence, if p is a restricted pointer then, any access to p through any other means may result in undefined behavior. Therefore, if the arguments of *gesummv* routine, as showing in 4.7, have been annotated with the *restrict* qualifier, thereby allowing the compiler to perform more aggressive optimization, such as instructions scheduling, loop invariant code motion, register promotion, redundant load/store elimination, etc. The same thing can be applied to GPU variants since NVCC, PGI and Clang compilers support the *restrict* keyword.

```
1  void gesummvRestrict(float *__restrict__ A, float *__restrict__ B,
2  float *__restrict__  x, float *__restrict__  tmp, float *__restrict__  y) {
3    for (int i = 0; i < NI; i++) {
4      for (int j = 0; j < NJ; j++) {
5        tmp[i] += A[i * NJ + j] * x[j];
6        y[i]   += B[i * NJ + j] * x[j];
7      }
8      y[i] = ALPHA * tmp[i] + BETA * y[i];
9    }
10 }
```

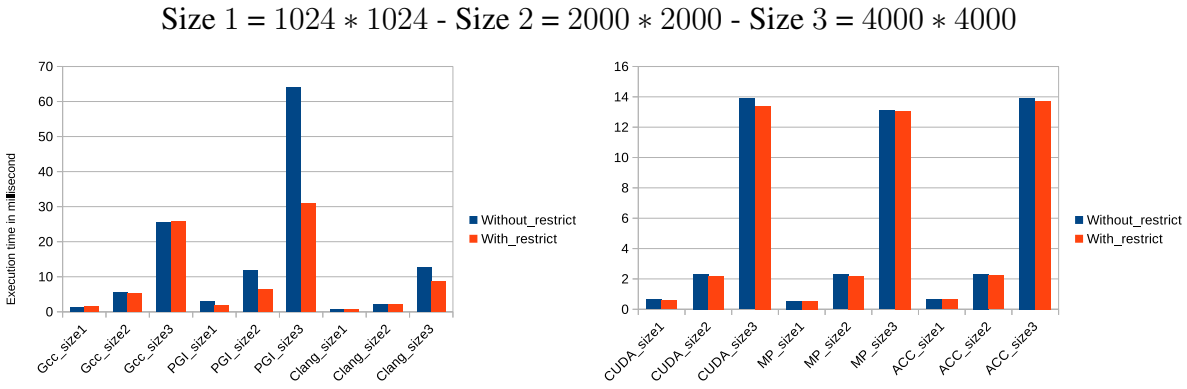
Listing 4.7 – A simple example of gesumv kernel

Figure 4.3 presents the performance result when using *restrict* keyword as seen in Listing 4.7. We can observe that informing the compiler that kernel arguments do not alias through the *restrict* keyword has no significant impact on execution time globally.

However, there is an exception when using PGI compiler on CPU code (without any directives for parallelization as seen on Listing 4.7). Indeed, by inspecting the assembly file and the feedback given by PGI during the compilation, we found that in addition to *j* loop unrolling and using FMA operations, PGI generated 2 prefetches instructions, and it is thanks to these prefetching instructions that we obtained a speedup of $1.89\times$ on average. When using clang as compiler and working on large array (e.x. *size 3*), the performance gain with the *restrict* type qualifier can be significant ($1.45\times$ for *size 3*).

4.2.3 Scalar Replacement

Another solution is to perform, at the source code level, a scalar replacement either manually or automatically. The goal of scalar replacement or register promotion is to identify repeated accesses made to the same memory address, either within an iteration or across iterations, and to remove the redundant accesses by keeping the data in registers. This is done by identifying sections of the code in which it is safe to place the redundantly accessed data in a register [85, 86, 87, 88]. Thus, before entering such section, the value redundantly accessed data is *promoted* (i.e. loaded) from its memory location to a register. Within the section, references to this value are rewritten to refer to the register. Upon exit from the section, the value is *demoted* (i.e. stored) to a memory location. By doing this, the compiler strongly reduces or completely eliminates partial redundant loads and stores, and thereby improve code performance. Hence, register promotion is among the



(a) Intel I7 CPU

(b) Quadro M2000M GPU

Figure 4.3 – Execution time of the *gesummv* kernel when the *restrict* keyword is added to its pointers arguments. Note that the kernel is called with distinct pointers, otherwise we will get an undefined behavior.

most important functions performed by an optimizing compiler. Scalar replacement as well as pointer alias analysis is widely used in compiler optimization.

During compilation, the compiler makes several decisions of which one of the most important relates to the storage of values. Indeed, the compiler must determine, for each value, where it will reside at run-time. For simplicity, suppose there are only two choices: in memory and in a register. As already mentioned in 1.1, registers accesses are much faster than memory accesses, so it is generally desirable to keep values in registers. Thus, modern compilers try to keep as many values in registers as possible. However, the presence of pointers can prevent retention of a value in a register across statement boundaries. Indeed, in the absence of specific knowledge about the set of variables that can be referenced by each pointer, the compiler is forced to treat references to any storage that the pointer might possibly address in this conservative fashion. Therefore, it can be interesting to aid compiler by performing scalar replacement at the source level.

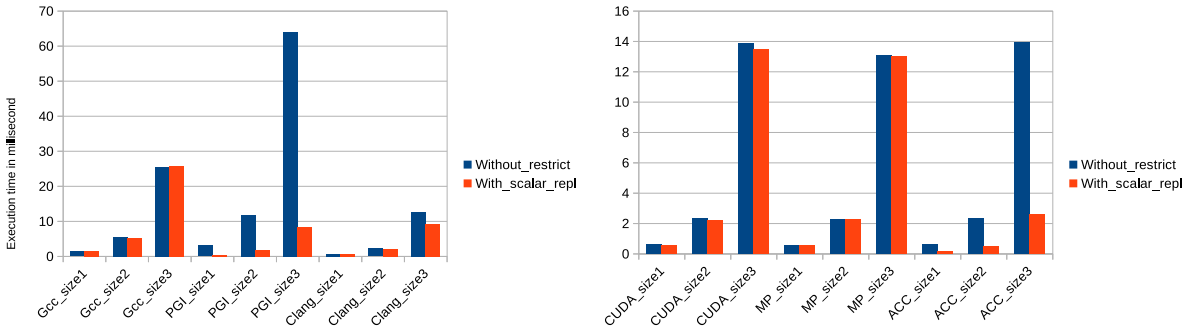
Considering the *gesummv* routine showed in 4.4, given that *tmp[i]* and *y[i]* are independent of the *j* loop, they can be move out and replaced by a scalar without consequences on the reliability of the computation results (assuming that *gesummv* will never be called with aliased pointers). Listing 4.8 shows a new version of *gesummv* kernel when scalar replacement is applied.

```
1  void gesummvScalarRep(float *A, float *B, float *x, float *tmp, float *y) {
2    for (int i = 0; i < NI; i++) {
3      float yy = y[i];
4      float tmptmp = tmp[i];
5      for (int j = 0; j < NJ; j++) {
6        tmptmp += A[i * NJ + j] * x[j];
7        yy     += B[i * NJ + j] * x[j];
8      }
9      y[i] = ALPHA * tmptmp + BETA * yy;
10   }
11 }
```

Listing 4.8 – A simple example of gesumv kernel

Figure 4.4 shows performance results obtained when *tmp[i]* and *y[i]* are replaced by scalar in *j* loop in order to remove redundant loand and store of them. When performing scalar replacement in source code, we obtain the same performance as using the *restrict* keyword except for PGI compiled versions. By inspecting Gcc, Clang and NVCC generated assemble and PTX, we found that there are the same as those generated for *restrict* variants. Concerning PGI compiler, when performing scalar replacement in source code: (1) On CPU, it vectorizes the *j* loop, generates 3 prefetch instructions and uses FMA. The speedup, with respect to the initial version reported on Figure 4.2, is $7\times$ on average. (2) On GPU, it parallelizes the *j* loop and generates implicit reduction for *tmptmp* and *yy* and the obtnaid speedup is $4.94\times$ on average.

Size 1 = $1024 * 1024$ - Size 2 = $2000 * 2000$ - Size 3 = $4000 * 4000$



(a) Intel I7 CPU

(b) Quadro M2000M GPU

Figure 4.4 – Execution time *gesummv* kernel when using scalar replacement in source code.

4.2.4 Summary

Pointers are very important in C-like languages, particularly in GPU programming paradigms since the only way to recover the output result of a GPU kernel is to pass it to him as a pointer. One or more pointers might point to the same memory location at runtime and determine at compile time the precise location of each variable of a program is an undecidable problem. Many works were being done on pointer aliasing problem and many alias analyzers are implemented in mainstream compilers, and despite all this, the presence of pointers in a program can prevent some optimization of the compiler. Indeed, pointers introduce a degree of uncertainty into the results of static analysis. This uncertainty, coupled with separate compilation (as it is the case with GPU codes compilers), forces compilers to treat variables with potentially exposed addresses quite conservatively. As a result, compilers need more information to be able to compile efficiency codes where pointers are used.

The *restrict* type qualifier can be used by the programmer to give the compiler information about aliasing. However, programmers must use this keyword with precaution, to avoid data race, since the compilers will assume that marked pointers with this type qualifier are never aliased in the program and thereby does not perform any verification to ensure that the aliasing information received is true. Although the *restrict* keyword has been available for several years already, it remains less used by programmers and its insertion is left to the programmer. The task of inserting the *restrict* qualifier is, in general, tedious and prone to errors especially since the C does not perform any verification to ensure that *restrict* keyword is not misplaced.

Scalar replacement, or register promotion, attends to reduce memory accesses by keeping in registers reused data for a certain part of a program, and thereby improve code performance. Unlike CPUs, to support massively parallel computations, accelerators such as GPGPUs offer an expansive set of registers, larger than even the L1 cache, to hold the temporary state of each thread. Scalar variables are the most likely candidates to be assigned to these registers by the compiler. As suggested by the observed performance gain on scalar replacement version with OpenACC, register promotion can be a key enabling optimization for effectively improving the utilization of register files on accelerator devices and thereby substantially reducing the cost of memory operations.

Our preliminary evaluation of *restrict* keyword and scalar replacement on the *gesummv* kernel example showed that the performance gain when adding *restrict* keyword to kernel pointers arguments is not really significant for CPU code when the used compiler if Gcc and Clang, and for CUDA and OpenMP versions. However, PGI compiled code and OpenACC version results are interesting, particularly when applying scalar replacement. Since the release of the OpenACC and OpenMP 4.0 standards as seen in 3.1, many works (e.x. [9, 10, 57, 58, 11, 62]) have been done to evaluate their performance against those of CUDA and OpenCL. Other works (e.g. [52, 59, 56]) suggested some optimizations, but these optimizations were essentially based on the implementation of new directives. Moreover, many efforts have been done to improve OpenACC and OpenMP compilers in order to generate more optimized code. However, like other mainstream compilers, they still fail to recognize even the simplest opportunities for reuse of subscripted variables in the absence of precise pointer aliasing information. All this motivated us to develop a static analysis tool for the automatic insertion of the *restrict* keyword and automatic scalar replacement in source level. In next sections, we present our static analysis tool and evaluate the performance of the *restrict* type qualifier on both CPU and GPU, and those of scalar replacement also on the PolyBench benchmark suite.

4.3 Previous work

In this section we are going to provide a brief revue of the works that have been done on alias analysis as well as scalar replacement.

4.3.1 Alias Analysis

Alias analysis is one of the most used techniques that aim to optimize languages with pointers. It is no surprise that this topic has received much attention in many fields including compilation and program verification. The literature of alias analysis is abundant. For a quite exhaustive bibliography, see for instance [89]. In this dissertation, we shall be limited to related works that are the most relevant for our study of interest.

The most popular algorithm for context-insensitive, flow-insensitive, iterative and constraints-based points-to analysis is known as inclusion-based or Andersen-style analysis [80]. For every pointer variable, it computes the set of abstract locations the pointer may point to. The representation used in Andersen's algorithm is such that one points-to graph represents the entire program and each node represents exactly one location. Within this analysis, the assignments of the current program are seen as constraints. For functions, Andersen's analysis generates constraints for the formal and actual parameters. Table 4.1 shows Andersen constraints while Listing 4.9 shows a simple example code for these constraints. The complexity of Andersen's points-to analysis is $O(n^3)$ where n is the number of nodes or pointers. The time required to run Andersen's points-to analysis grows exponentially as the program size grows since the number of nodes tends to grow as program size increases [90]. Many works have been done to improve Andersen's algorithm and many of them, for instance, Steensgaard's algorithm, are implemented in mainstream compilers (e.x. LLVM, Gcc). Like Andersen's analysis, Steensgaard [91] analysis is unification-based, inter-procedural, flow-insensitive, context-insensitive and field-insensitive. It also transforms programs into constraints and solves the sets of constraints to obtain points-to results. The main difference is that instead of collecting subset constraints, it collects equivalence constraints. These constraints are much simpler, but produce less precise model.

Table 4.1 – Four Andersen constraint types

Type	**Assignment**	**Constraint**	**Meaning**
Base	$a = \&b$	$a \supseteq b$	$loc(b) \in pts(a)$
Simple	$a = b$	$a \supseteq b$	$pts(a) \supseteq pts(b)$
Complex	$a = *b$	$a \supseteq *b$	$\forall v \in pts(b), pts(a) \supseteq pts(v)$
Complex	$*a = b$	$*a \supseteq b$	$\forall v \in pts(a), pts(v) \supseteq pts(b)$

```
1  int* foo(int *x) {
2    x = malloc(4*sizeof(int));
3    return x;
4  }
5  int main() {
6    int *p, *q, i, j;
7    i = 1; j = 2;
8    p = \&i;        //<------ p ⊇ i
9    q = \&j;        //<-------q ⊇ j
10   p = q;         //<-------p ⊇ q
11   int *A, *B;    //<-------p ⊇ i, j
12   A = foo(B); //<-------x ⊇ b ≡ x = b
```

```
13    return 0;    //<-------a ⊇ x ≡ a = x
14  }
```

Listing 4.9 – Example for Andersen constraints

Sperle et al. presented in [92] three different techniques to disambiguate pointers used as arguments of functions. Their first technique relies on the static alias analysis already available in mainstream compilers to perform pointer disambiguation and the others combine static bound inference with code cloning, hence, extending the reach of pointer disambiguation. Maalej et al. introduced in [82] a new technique to disambiguate pointers, which relies on a less-than analysis. Their alias analysis uses the observation that if $p1$ and $p2$ are two pointers, such that $p1+$ *size of array pointed* $< p2$, then they cannot alias. They designed in [93] another alias analysis algorithm that uses a combination of less-than analysis and classical range analysis to show that two pointers cannot dereference the same memory location.

4.3.2 Scalar Replacement

The original scalar replacement algorithm was proposed by Carr-Kennedy in [85] more than 20 years ago. Carr-Kennedy's algorithm finds opportunities for reuse of subscripted variables and replaces the references involved by references to temporary scalar variables. This algorithm handles two special instances of the scalar replacement problem very well: (1) repeated accesses made within the same loop iteration in code having arbitrary conditional control-flow, and (2) code with repeated accesses made across iterations in the absence of conditional control-flow. For the first problem, Carr-Kennedy's algorithm relies on Partial Redundancy Elimination (PRE), while for the second algorithm, it relies on dependence analysis and rotating scalar values (see example in Listing 4.10). Carr-Kennedy's algorithm steps are: (1) Dependence graph construction, (2) Control flow analysis, (3) Availability analysis, (4) Reachability analysis, (5) Potential generator selection, (6) Anticipability analysis, (7) Dependence graph marking, (8) Name partitioning, (9) Register pressure moderation, (10) Reference replacement, (11) Statement insertion analysis, (12) Register copying, (13) Code motion, and (14) Initialization of temporary variables.

The algorithm is complex, requires perfect dependence information to be applicable and operates only on loop bodies without any backward conditional flow. Further, the algorithm performs its profitability analysis on name partitions, where a name partition consists of references that share values. If a name partition is selected for scalar replacement, all the memory references in that name partition will get scalar replaced, otherwise, none of the accesses in the name partition is scalar replaced.

```
1  //original loop before scalar replacement
2  for (int i = 2; i < NI; i++)
3    X[i] += X[i-2];
4  //loop becomes after optimization with scalar replacement
5  int x0 = X[0];     //invariant x0 = X[i-2]
6  int x1 = X[1];     //invariant x1 = X[i-1]
7  for (int i = 2; i < NI; i++) {
```

```
8    int x2  = X[i];
9    x2 += x0;
10   X[i] = x2;
11   x0 = x1;    //rotate scalar values
12   x1 = x2;
13 }
```

Listing 4.10 – Example for scalar replacement with Carr-Kennedy's algorithm

Since then, several works have been done to improve Carr-Kennedy's algorithm in many aspects. Surendran et al. presented in [94] new algorithms for scalar replacement and dead store elimination based on Array SSA (Static Single Assignment) form. Byoungro et al. described in [87] an algorithm for scalar replacement that can exploit reuse opportunities across multiple loops. Budiu et al. presented in [86] a simplified Carr-Kennedy inter-iteration register promotion algorithm to handle a number of dynamically executed memory accesses. In their approach, the compiler generates a flag represented by a single bit that is associated with each value to be scalarized, as well as code that dynamically updates the flag. The flag can be inspected at run time to avoid redundant load operations, and their algorithm ensures that only the first load and last store take place. The fact that this algorithm inserts additional control flow statements in the code could lead a thread divergence if running on GPU. Their approach is not suitable for GPU codes.

Andión et al. presented in [95] a scalar replacement algorithm (based on standard compiler transformations) for optimizing computation regions specified using the HMPP directive interface. Tian et al. presented in [96] an extension to the classical scalar replacement algorithm for optimizing registers usage in OpenACC codes. Their approach is based on feedback information regarding register utilization and a memory latency-based cost model to select which array references should be replaced by scalar references.

4.4 Frama-C

Frama-C[1] is a static analyzer for C code. It provides its users with a collection of plug-ins that perform static analysis, deductive verification, and testing, for safety- and security-critical software [97]. The Frama-C platform gathers several analysis techniques into a single collaborative extensible framework. The platform is based on a common kernel, which hosts analyzers as collaborating plug-ins. The Frama-C plug-ins are programming in the *OCaml* language.

Frama-C kernel is based on a modified version of the C Intermediate Language (CIL). CIL is a front-end for C that parses ISO C99 programs into a normalized representation. For instance, *for* loops are replaced by equivalent *while* loops (with additional conditional and break statements to leave the loop body), normalized expressions have no side-effects, conditional statements with compound conditions are unfolded into multiple conditionals (with one non-compound condition each), etc. Listing 4.11 shows the *gesummv* kernel in CIL normalized representation (which is the

[1] https://frama-c.com/

output of the command *frama-c -print gesummv.c*).

```
1  void gesummvCILNorm(float *A, float *B, float *x, float *tmp, float *y) {
2    int i;
3    int j;
4    i = 0;
5    while (i < n) {
6      j = 0;
7      while (j < n) {
8        *(tmp + i) += *(A + (i * n + j)) * *(x + j);
9        *(y + i) += *(B + (i * n + j)) * *(x + j);
10       j ++;
11     }
12     *(y + i) = alpha * *(tmp + i) + beta * *(y + i);
13     i ++;
14   }
15   return;
16 }
```

Listing 4.11 – CIL normalized version of *gesummv* kernel

Frama-C extends CIL to support other features such as ACSL (ANSI/ISO-C Specification Language) annotations. This modified CIL front-end produces the C + ACSL Abstract Syntax Tree (AST), an abstract view of the program shared among all analyzers (or plug-ins). The AST assigns unique identifiers for statements and blocks that can be used for the program counter, it also keeps line numbers which is very useful for us to perform modifications in the source file after the analysis. Frama-C uses, in general, the default C compiler available on the host system (usually *gcc*) to pre-process the input C files before normalizing it and generating the AST.

In addition to the AST, the kernel provides several general services for helping plug-in development and providing convenient features to Frama-C's end-user. For example, it provides a visitor mechanism facilitates crawling through the AST. In general, writing a Frama-C plug-in requires to visit the AST to compute information for some C constructs. There are two different ways to do that in Frama-C: through a direct recursive descent or by using the Frama-C visitor. It is the latter that we will use in the design of our static analyzer tool. A visitor is a class with one method per type of the AST, whose default behavior is to just call the method corresponding to each of its children. By inheriting from the visitor, and redefining some of the methods, one can perform actions on selected parts of the AST, without the need to traverse the AST explicitly.

Frama-C includes several ready-to-use plug-ins for the static analysis of C code. We use the results of *value* plug-in in our analyzer. The Value Analysis (in short value) plug-in is a forward data-flow analysis based on the principles of abstract interpretation. For each instruction of the program, the information inferred by the plug-in are twofold: a flag indicating the possibility that the execution of the instruction may fail at runtime or at least invoke an undefined behavior; and for each memory location, an over-approximation of the values it may contain.

4.5 Design of a Scalar Replacement Tool: RPromF

As introduced in 4.2.3, scalar replacement or register promotion improves program performance by identifying sections of the code in which it is safe to place the redundantly accessed data in a register. Furthermore, scalar replacement can be a key enabling optimization for effectively improving the utilization of register files and shared memory on GPUs and thereby substantially reducing the number of redundant memory accesses. Thus, we decided to design a static analysis tool named RPromF, for **R**egister **Prom**otion with a **F**rama-C plug-in, that automatically performs register promotion at source code level without requiring any intervention from the programmer.

Note that static alias analysis result might not be true at runtime and that an optimized function with register promotion and the original version (e.x. Listing 4.4 and Listing 4.8) are equivalent and produce the same computation result only in absence of alias. Thus, it would be prudent to keep the original version of the function unchanged and to make optimization in a clone of this function. In this way, the clone version will be called if actuals pointers arguments don't point to overlapping memory regions, otherwise, the original version is called.

RPromF, based on Frama-C, includes six main steps: (i) identification of the function to be optimized; (ii) analyze identified functions to check that there is no alias in their body; (iii) if there is no alias then clone them, analyze clone's instructions to retrieve data reuse information and perform scalar replacement; (iv) analyze callers of functions that have been cloned and optimized in order to verify if they are called with aliased pointers; (v) and if their actuals parameters are free of alias, add instruction to be able to used the clone whenever possible.

To simplify the analysis, we make following assumptions:

- *For OpenACC/OpenMP codes:* (1) Offloaded region codes are in separate functions of the *main* for example. (2) Devices pointers are defined in the caller of functions containing offloading regions. (3) Device memory management is make either with copy clauses or runtime library routines. In all cases, pointers appearing in offloaded code regions are not defined, allocated or initialized in the body of the current function, instead they are its' formal arguments.
- *For CUDA codes:* Device pointers are defined, allocated and initialized (if data transfer is needed) in the function calling the CUDA kernels.
- Concerning CPU code (without any *#pragma acc* or *#pragma omp*), as for CUDA, OpenACC and OpenMP codes, the loops in which the register promotion must be performed are in a separate function of the *main*. All variables appearing in the body of such function are formal arguments.
- For GPU codes, there is no dynamic parallelism (CUDA global kernel launching another global kernel) or call to device routines (defined with *#pragma acc routine* or *#prama omp target declare*).
- There is a single source file, with extension *.c* for OpenMP/OpenACC/CPU versions or *.cu* for CUDA codes, for the whole program.

Since Frama-C is a static analyzer for C code,it does not support any keyword or type that is not part of the C standard. There is a Frama-C plug-in in development so that Frama-C can analyze C++ codes. However, at the moment of this writing, Frama-C does not support either CUDA specific keyword and types, either OpenACC and OpenMP directives. As already said, Frama-C normalizes, the code to be analyzed, in CIL representation, for example, all *for* loops are turned in *while* loops thus denaturing the source code. Hence, to analyze CUDA, OpenACC and OpenMP codes, it is necessary to do some text processing tasks on the source file before and after the analysis with Frama-C. RPromF can be divided into three parts: the pre-processing, the analysis, and the post-processing. In the following subsections, we present these three parts.

4.5.1 Pre-Processing Module

The pre-processing task, performed by a script programmed in Perl, consist of:

- creating a copy, named *source_name_copy.c*, of the input file thus source file remains unchanged,
- if the input file is a C file, finding functions that contain lines beginning with *#pragma omp target* or *#pragma acc kernels* or *#pragma acc parallel*, and then adding the suffix *_acc* (for OpenACC codes) or *_mp* (for OpenMP versions) at the end of identified functions's names.
- if the input file is a CUDA file, identifying CUDA kernels while searching lines containing *__global__ void*, and then removing *__global__* and adding the suffix *_cud* at the end of kernels's names,
- commenting all lines beginning with *#pragma acc* or *#pragma omp* if input file is a C file,
- if the input file is a CUDA file, hiding all CUDA specific keywords, types or API functions. For instance, comment lines containing grid and bloc sizes definition, turn all *cudaMalloc*/*cudaFree* into C *malloc*/*free*, remove all *«...»*, replace thread position computations formulas by constant values, ...
- and commenting all lines where non common libraries are included (e.x. openacc.h, accelmath.h, cuda_runtime.h, etc.).

Note that the pre-processing task conserves the original program structure and line number information, no line addition or deletion. This is very important so that post-processing can make the necessary changes in the right places in the original version of the code.

4.5.2 Analysis module

Because the optimized version of a function with the scalar replacement is equivalent to its original version only if there is no alias and marking aliased pointers as restricted may result in undefined

behavior, we run an alias analysis before making any changes in the source code. For that purpose, we implemented in Frama-C a simple and fast alias analyzer inspired by the Andersen-style analysis [80] and the concepts of the *basicaa* pass of LLVM[2]. In our implementation, we considered two pointers operations: taking the address of a variable (e.g.: $p = \&val$) and assignments (e.g.: $p = q$, $p = Tab$ where p and q are pointers and Tab a constant array). Algorithm 1 shows the pseudo-code of the alias analyzer.

Algorithm 1 Alias analyzer plug-in

```
1:  for each kernel definition do
2:      get its' formal parameters list
3:      for each instruction where l-value is pointer do
4:          if r-value match with pointer assignments or address taking then
5:              if l-value is a formal parameter or there is a formal in the r-value then
6:                  increment the counter of aliased formal argument pointers
7:              else if l-value is a global var then
8:                  if its address is not taken and there is a formal in the r-value then
9:                      increment the counter of aliased formal argument pointers
10:                 end if
11:             end if
12:         end if
13:     end for
14:     if the counter of aliased formal argument pointers is equal to zero then
15:         add this kernel with its' callers list to an hash table that will be queried later
16:     end if
17:     for each formal parameter of this kernel do
18:         if current formal parameter is not a l-value in any instruction then
19:             save this kernels name and formal parameter information into an hash table
20:         end if
21:     end for
22: end for
```

To implement the alias analyzer plug-in, we redefined two visitors available in Frama-C. The first is the *vglob_aux* visitor which allows us to analyze functions definition. The second is the instruction visitor called *vinstr* with which we can analyze all instructions of a program. Note that our implementation does not use any control flow or dependence graph information, it only uses the AST and the results of the *value* plug-in.

As can be seen in Algorithm 1, each kernel definition is analyzed. We start by tracking all pointers assignment and address taking instructions. If any formal parameter does not appear in such instructions, then this kernel arguments are alias free. Next, we go through the formal parameters list of each kernel and check if there are read-only data. The informations of the kernels that formal parameters are not aliased and the pointers that are only read are saved into two hash tables.

[2] https://llvm.org/docs/AliasAnalysis.html#the-basicaa-pass

Next, for each kernel, we analysis its' callers to verify if actual parameters are alias free. If the parameters passed to a kernel are not aliased in all callers of this kernel, then we can start the next step that consist of identifying redundant memory accesses.

In order to retrieve repetitive accesses made to the same memory location, we implemented a simplified version of Carr-Kennedy' algorithm [85]. We start by computing the Control Flow Graph (CFG) in order to be able to determine loops informations (e.g. loop identifier, beginning and ending lines's numbers, nested loops, ...). We redefined the Frama-C's statements visitor to analyze only the loops. Thus we compute the CFG and analyze loops, next we redefined the Frama-C left-value visitor to analyze only left-values that are pointers. We also redefined the Frama-C' expression visitor to analyze the right-values. Next, we re-analyze statements which are loops in order to find the loop index variable name, type and initialization value. In the next step, we redefined the Frama'C instructions visitor the instructions which are neither loops nor conditional control-flow (e.g. if-else and switch case statements). The goal of this instructions visitor is to find loop-invariant addresses for stores, but also for loads. Once we have the results of previous redefined visitors, we can generate the necessary instructions that will allow the post-processing module to perform the scalar replacements. Consider the example of the *gesumv* kernel seen in Listing 4.4, the output of the analysis looks like instructions showed in Listing 4.12. Figure 4.5 summarizes all operations performed during the analysis step.

```
1 Add before 3 declare float tmptmp
2 Add before 3 declare float yy
3 Add before 3 assignment tmptmp tmp[i]
4 Add before 3 assignment yy y[i]
5 Change line 4 substitute tmp[i] tmptmp
6 Change line 5 substitute y[i] yy
7 Change line 7 substitute tmp[i] tmptmp
8 Change line 7 substitute y[i] yy
```

Listing 4.12 – Partial output of the analysis module

4.5.3 Post-processing module

According to the results of the analysis made with Frama-C on the file generated by the pre-processing step (described in 4.5.1), changes must be made to the source code. These changes are performed by another Perl program. This program takes as input the output file of our Frama-C plug-in and the initial source file. It starts by generating a copy of the source file and then performs the requested modifications (see example in Listing 4.12) in the the output file of our Frama-C plug-in. For example, if the source file is a CUDA program then the output file of the RECKA plug-in is used and the changes made to the source file are: (i) clone a CUDA kernel whose formal parameters (as well as actual parameters) are not aliased; (ii) add a suffix to the clone's name; perform necessary modifications (declare new scalar variables, initialize them, comment lines, replace old redundant accessed addresses references by new variable, ...); and (iii) change this kernel call site by adding an if statement allowing to call either the clone or the original version

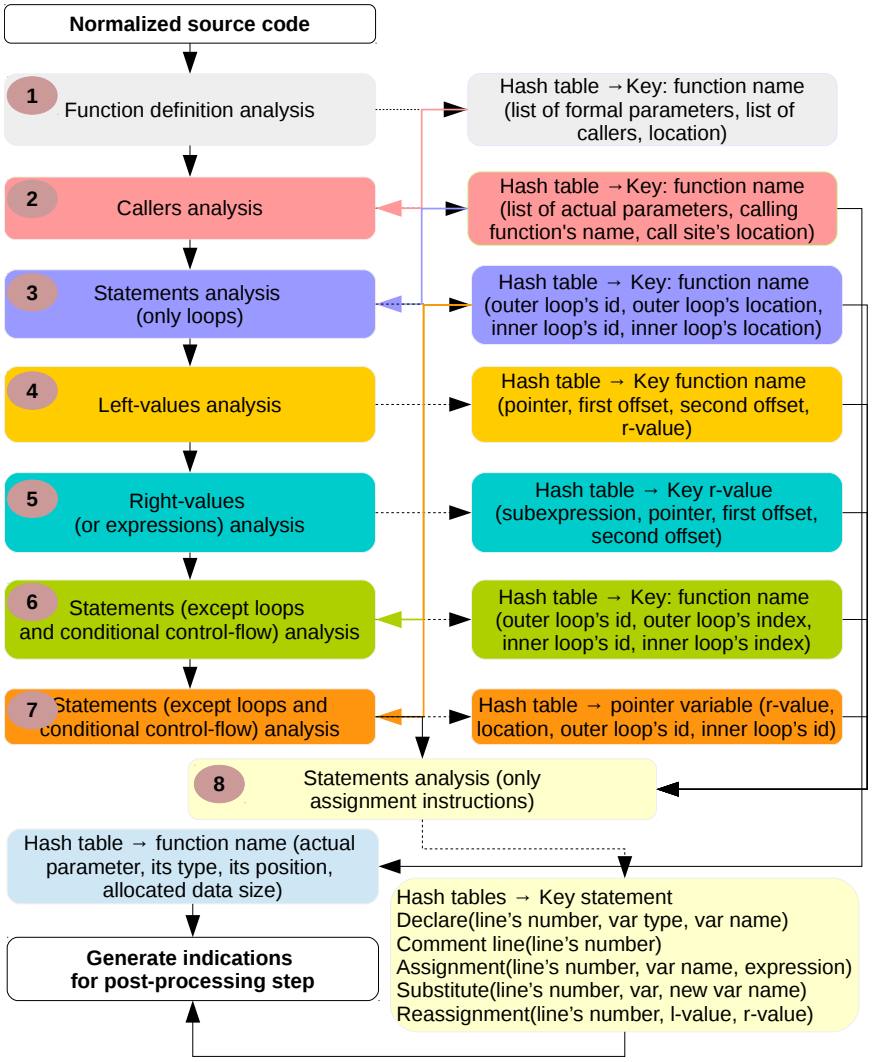


Figure 4.5 – Functional diagram of the scalar replacement tool's analysis module

depending on the runtime less-than check result. For the runtime less-than check test, we used the methods described in [82, 93].

4.6 Design of an Automatic Restrict Keyword Insertion Tool

As introduced in 4.2.2, adding the *restrict* and *const* type qualifiers can improve the GPU kernels' performance by caching read-only data in the L1 cache, removing redundant loads, and allowing the compiler to perform more aggressive optimizations. In this section, we present the main step of the design of a static analysis tool named RECKA (REstrictification of CUDA Kernel pointers Arguments) that can insert the *restrict* keyword in the source code without requiring any intervention from the programmer. Note that it can also analyze OpenMP and OpenACC programs. More

specifically, RECKA works as follows: (i) an alias analysis is performed on the kernels; (ii) if the kernel's formal parameters are alias free, then this kernel instructions are analyzed in order to identify data that are read-only; (iii) the kernel's callers are analyzed in order to determine if the kernels actual parameters are also alias free; (iv) if the kernel actual parameters are alias free, then the indications are generated to clone the kernel, rename the clone, add the *restrict* keyword to formal parameters pointers that are not read-only or *const restrict* otherwise.

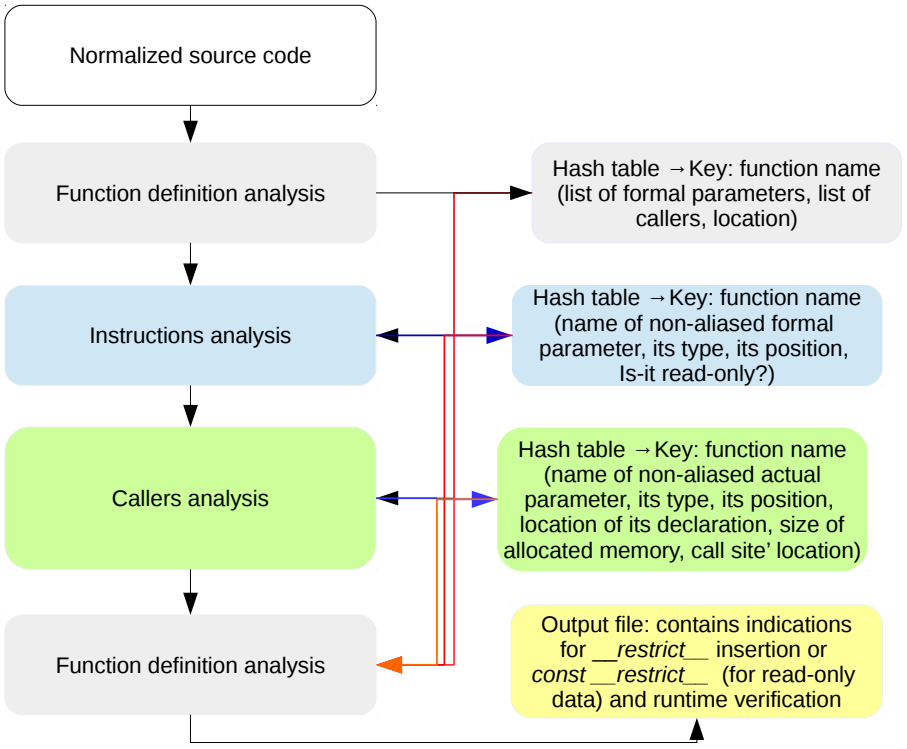


Figure 4.6 – Functional diagram of the restrict keyword insertion tool's analysis module

As did for the scalar replacement tool, we start by performing a preprocessing task. We use the module described in Subsection 4.5.1 for that purpose. Next, the analysis is performed by a Frama-C plug-in. As depicted in Figure 4.6, we redefined the Frama-C' function definition visitor to analyze kernels body. This analyze consist of verifying that no formal argument has its address taken or is accessed through another pointer. We have also redefined the Frama'C' instruction visitor. This visitor allow us to analyze all instructions and determine read-only data. Once we have verified that the kernel formal parameters are alias free and determined read-only data, we analyze this kernel callers. The goal of analyzing the kernel's callers is to be sure that the actuals arguments of this kernel are also alias free. In last step, we re-analyze kernel definition by using the results of previous analysis in order to generate instructions to perform the *restrict* as well as *const restrict* type qualifiers insertion. The post-processing step (See Subsection 4.5.3) is done by the same Perl program that we have used in the scalar replacement tool. Figure 4.7 shows an overview of the restrict keyword insertion tool.

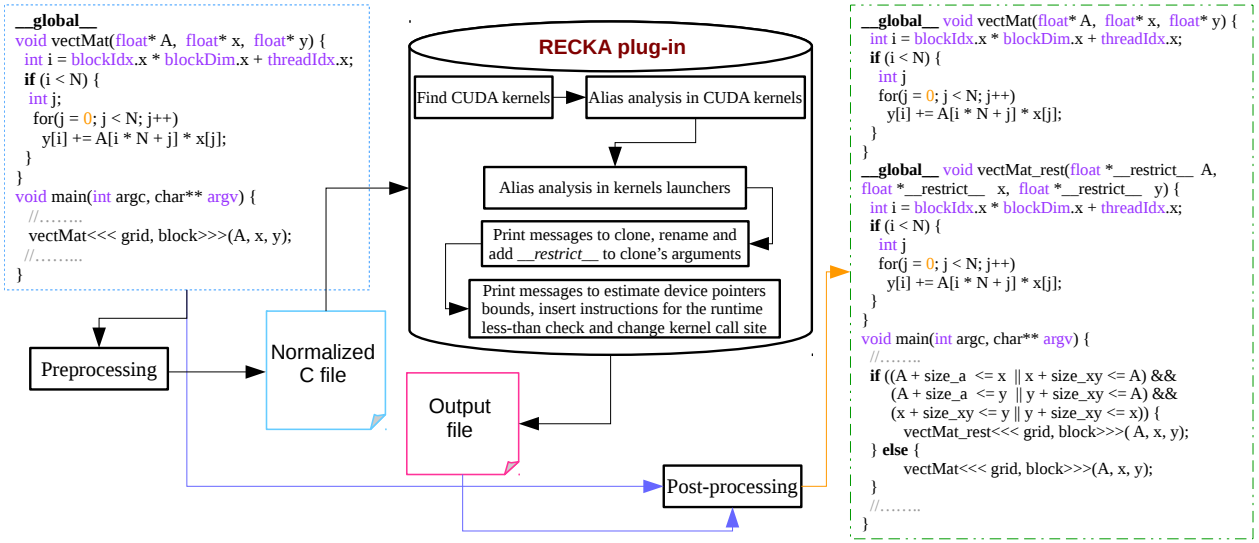


Figure 4.7 – An overview of RECKA

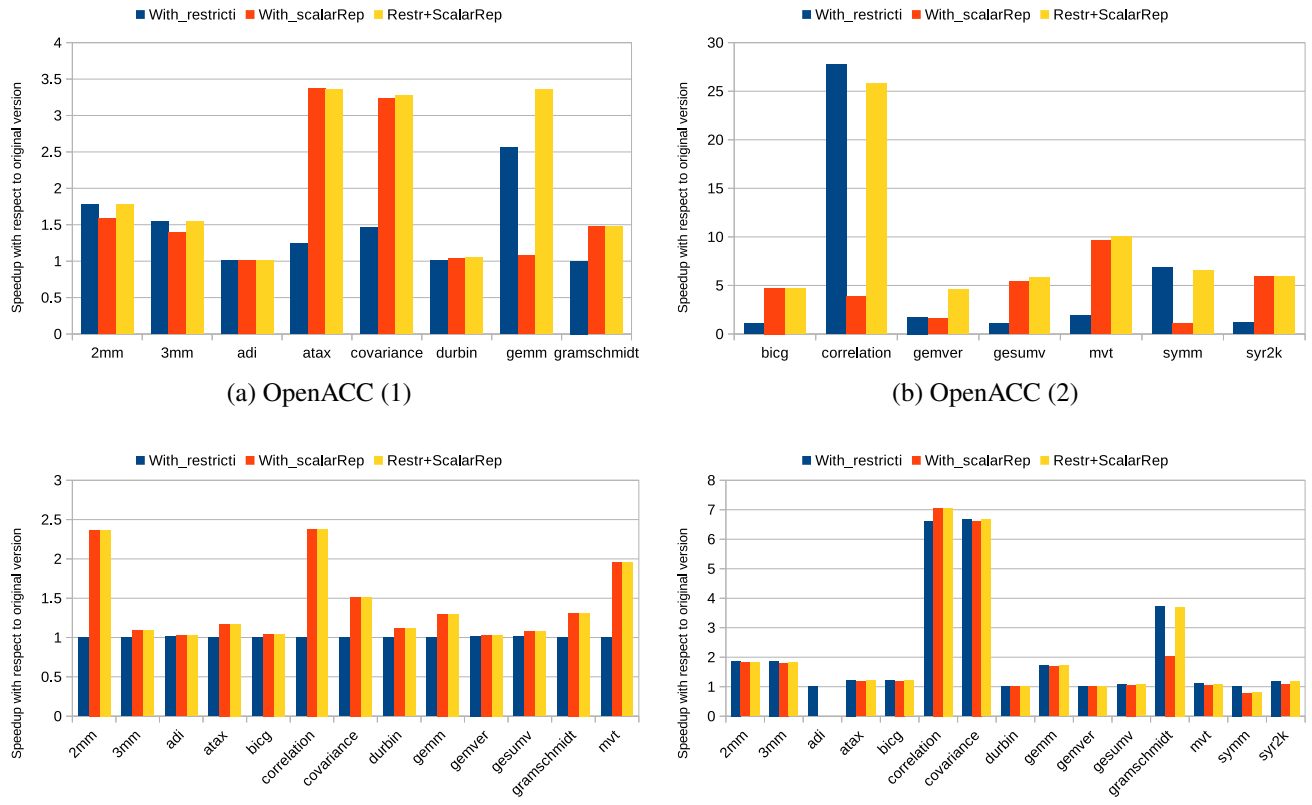
4.7 Tools Evaluation

In order to assess the performance improvement factor du to the *restrict* keyword insertion and scalar replacement we used the two static analysis tools described above to analyze kernels taken from the PolyBench benchmark suite. The generated codes have been compiled with compilers seen in Table 3.3.

Sub-figures of Figures 4.8 show the speedup of the generated codes with respect to original versions. First, we showed that our two scalar analysis tools are successfully able to perform the necessary analysis as well as to insert the needed modifications (the *restrict* keyword or scalar replacement) without any misplacement. Since both the *restrict* keyword and scalar replacement can be applied to a same kernel, we manually insert the *restrict* type qualifier and performed the scalar replacement in a third version. Since our goal is to evaluate the impact of the *restrict* keyword and scalar replacement on application performances, we did not compare OpenACC or OpenMP against CUDA.

With OpenACC, we observed that adding the *restrict* keyword can improve the application performance with a speedup factor of $1.42\times$ on average in comparison to original version. However, this speedup factor is relatively high for the *correlation* kernel where it is $27.71\times$. Indeed, this kernel has several false pointers aliasing issues. Thus, by indicating to the compiler that pointers are free alias, it can perform more aggressive optimizations. On the other hand, the scalar replacement performed at source level leads to a performance improvement with a speedup factor of $3.19\times$ on average. However, this speedup factor is more interesting for the *gesumv*, *mvt*, and *syr2k* kernels where severals memory addresses are redundantly accessed. The versions where both the *restrict* keyword and the scalar replacement have been applied produce the best results. Indeed, the speedup factor for this last case is $4.79\times$ on average with respect to the original version.

Unlike OpenACC, we observed that both the *restrict* keyword as well as the scalar replacement



(a) OpenACC (1)

(b) OpenACC (2)

(c) OpenMP

(d) CUDA

Figure 4.8 – Quadro M2000M: Speedup with respect to original versions

do not improve significantly the application performance in OpenMP and CUDA. In OpenMP, the speedup factor is $1.03\times$ on average for the *restrict* type qualifier and $1.41\times$ for the scalar replacement as well as the third version (combination of *restrict* and scalar replacement). On the other hand, with CUDA the speedup factor is $1.64\times$ on average for the *restrict* keyword and $1.71\times$ for the scalar replacement.

4.8 Conclusion

In this chapter, we discussed the pointer aliasing issues and its impact on application performance on GPU. We have implemented two new static analysis tools based on the Frama-C framework. These two tools have been designed in order to insert the *restrict* keyword and to perform *scalar replacement* at source code level. We evaluated the two static analysis tools as well as the impact of the *restrict* type qualifier and the scalar replacement optimization on application performance. The results show that the performance gain with the *restrict* keyword as well as the *scalar replacement* is not really significant in general. However, since these two optimization do not degrade applica-

tion performance, the programmer may use them in order to help compilers in their optimization steps.

Chapter 5

Use Case: a Stereo-Vision based Visual Odometry Application

Contents

Benchmarks used in previous chapters allowed us to explore performance improvement opportunities with OpenMP and OpenACC, but also to assess their performance. However, real-world applications are more complex than those found in benchmarks in general. Thus, we will applicate the parallelization methodology proposed in Chapter 3 to a visual odometry application. First, we give a brief background on visual odometry in general in Section 5.1. Next, we describe the use case application accelerated in this chapter in Sections 5.2, 5.3, and 5.4. As explained in Section 3.5 – Strategy 1, before parallelizing an application, it is necessary to profile and analysis it in order to identify most time consuming codes. Thus, the use case is profiled in Section 5.5. Next we analyze the CPU based implementation of identified compute-intensive region codes is order to determine the changes needed 5.6 to adapt these codes for parallelization 5.7. After adapting them for parallelization, we describe the acceleration strategies (Sections 5.8 and 5.9) and evaluate their performance (Section 5.10).

5.1 Visual Odometry

Knowledge of the ego-vehicle's motion state is essential for assessing the collision risk in Advanced Driver Assistance Systems (ADASs) or autonomous driving. Vision Odometry (VO) is one of the robust techniques used for vehicle localization [98, 99]. VO consists of estimating the pose of an agent (e.g., vehicle, human, and robot) by using only a stream of images acquired from a single or from multiple cameras attached to this agent [100]. Indeed, images contain a sufficient amount of meaningful information (color, texture, shape, etc.) to estimate the movement of a camera in a static environment.

VO applications VO has a wide range of applications including robotics, automotive, and wearable computing [100]. VO is applied in many types of mobile robotic systems, such as ground, underwater, aerial, and space robots. VO is mainly used for navigation and to reach targets efficiently as well as to avoid obstacles while driving. Moreover, VO plays a significant role in autonomous underwater vehicles and coral-reef inspection systems given that the GPS signal degrades or becomes unavailable in underwater environments [99]. In the automotive industry, VO also plays a big role since it is widely used in numerous ADAS systems, such as vision-based assisted braking systems [101, 99].

VO systems classification VO can be classified according to the type of camera used. Stereo, monocular, stereo or monocular omnidirectional, and RGB-D cameras, can be used for VO purposes. Systems using a binocular camera are considered *stereo VO systems* while those using a monocular camera are considered *monocular VO systems*. A binocular camera has, in general, two lenses with a separate image sensor for each lens. Given that information on the third dimension (i.e., depth) can be extracted from a single frame, the image scale can be immediately and instantaneously retrieved because the size of the stereo baseline is fixed and known, thereby resulting in an efficient and accurate triangulation process. However, binocular cameras require more calibration effort than monocular cameras, and errors in calibration directly affect the motion estimation

process. Using a monocular camera mitigates the effect of calibration errors in motion estimation. Low cost and easy deployment are generally the main motivations for using the monocular camera in a VO application. Monocular VO systems, compared with stereo VO systems, are essentially good for small robotics because they conserve the space of the baseline between the pair of stereo cameras [99].

VO challenges The main challenges in VO systems are mainly related to computational cost and light and imaging conditions [99]. For VO to work efficiently, sufficient illumination and a static scene with enough texture should be present in the environment to allow an apparent motion to be extracted [100]. Furthermore, consecutive frames should be captured by ensuring that they have sufficient scene overlap. In areas that have a smooth and low-textured surface floor, directional sunlight and lighting conditions are highly considered, leading to non-uniform scene lighting. Moreover, shadows from static or dynamic objects or from the vehicle itself can disturb the computation of pixel displacement and thus result in erroneous displacement estimation. In monocular VO system, if the surface is uneven, the image scale will fluctuate, and the image scaling factor will be difficult to estimate. The accuracy of real-time stereo VO is a function of image resolution, meaning the resolution is typically reduced to achieve faster performance [99, 102].

Approaches of Visual Odometry

Estimating the position of an agent (e.x., mobile robot, vehicle, etc.) with VO can generally be approached in three ways: through a feature-based approach, an appearance-based approach, or a hybrid of feature- and appearance-based approach [99].

Feature-based approach involves extracting image features (such as corners, lines, and curves) between sequential image frames, matching or tracking the distinctive ones among the extracted features, and finally estimating the motion. In this approach, matching an image with a previous one is accomplished by comparing each feature in both images. Afterward, the displacement is obtained by calculating the velocity vector between the identified pairs of points. In the case of stereo VO system, the extracted features from the first frame are matched with the corresponding points in the second frame, thus providing the 3D position of the points in space. The camera motion is estimated based on feature displacement where relative pose of the camera can be estimated by finding the geometric transformation between two images acquired by the camera using a set of corresponding feature points.

Appearance-based or Texture-based approach tracks the changes in the appearance of acquired images and the intensity of pixel information therein. It focuses on the information extracted from the pixel intensity. The camera motion and vehicle speed can be estimated using optical flow. Optical flow algorithm uses the intensity values of the neighboring pixels to compute the displacement of brightness patterns from one image frame to another.

Hybrid of feature- and appearance-based approach The feature-based approach is suitable for textured scenarios. However, this approach fails to deal with low-textured environments of a single pattern (e.g., sandy soil, asphalt, and concrete). The few salient features that can be detected and tracked in these low-textured environments make the feature-based approach inefficient in such environments. By contrast, the appearance-based approach is more robust and superior to feature tracking methods in low-textured. Hence, in some scenarios, the hybrid approach is the best solution which is a combination of feature- and appearance- based approaches.

5.1.1 Stereo Matching

Stereo matching denotes the problem of finding dense correspondences in pairs of images in order to perform the 3D reconstruction. In general, stereo matching approaches can be divided into two broad categories: global approaches and local approaches [103]. At the border of these two approaches, we find semi-global approaches which are a combination of global and local approaches. Semi-global methods do a complete search while using local methods.

Global approaches treat disparity assignment as a problem of minimizing a global energy function for all disparity values. A global approach is typically formulated as an energy minimization process with two terms in the objective function. The first term, called "data term", penalizes solutions that are inconsistent with the target data. On the hand, the second term, called "smoothness term", which enforces the piecewise smoothing assumption with neighboring pixels. The disparity map is produced by assigning similar depth values to neighboring pixels. Although global approaches provide a dense and accurate disparity map, optimizing global methods is a complex NP problem, making it very difficult to use these methods in a real-time application [103].

Local approaches or region-based approaches perform the matching by searching for the homologous pixel in a window centered on the current pixel in both left and right images. The corresponding cost is calculated either by minimization or by maximization over the disparity range according to the cost function. In these approaches, the quality of the resulting disparity map is highly dependent on the proper choice of the size of the search window. A small window would cause bad mapping and a large window would increase computing time and memory consumption. In the following equations, we give three examples of cost functions used in local approaches. Note that (x, y, d) represent the coordinates in the disparity map, (x, y) the coordinates of the current pixel, d the disparity value, and w the size of the search window:

- The Absolute Difference (AD) aggregates the difference in intensity value between a pixel in the left image (I_l) and the corresponding pixel in the right image (I_r). A formulation of AD is given by equation 5.1.

$$AD(x, y, d) = |I_g(x, y) - I_d(x - d, y)| \tag{5.1}$$

- The Square Difference (SD) aggregates the square of the differences between a reference pixel in the left image (I_l) and the candidate pixel in the right image (I_r). SD is formulated bu the equation 5.2.

$$AD(x, y, d) = |I_g(x, y) - I_d(x - d, y)|^2 \tag{5.2}$$

- Sum of Absolute Differences (SAD) sums the absolute difference between the intensity of each pixel in a given window and that of the target pixel in the window. Its expression is provided by the equation.

$$SAD(x, y, d) = \sum_{(x,y)\in w} |I_g(x, y) - I_d(x - d, y)| \tag{5.3}$$

- Sum of Squared Differences (SSD) sum the squares of the differences between the center pixel in the search window and all other pixels in the same window. It has a computing complexity higher than that of the SAD and is more sensitive to outliers and noise. Its formulation is the following:

$$SSD(x, y, d) = \sum_{(x,y)\in w} |I_g(x, y) - I_d(x - d, y)|^2 \tag{5.4}$$

5.2 Use Case Presentation

Our use case is a VO algorithm for dense 3D reconstruction in real time [102]. We chose this algorithm based on two important criteria. First, it is considered as one of the most efficient and robust VO algorithms which has been widely cited in the literature. Indeed, this algorithm citation count is estimated to 745 on Google Scholar and 345 on IEEE Xplore at the moment of this writing. Second, the algorithm is compute bound and memory bound. As depicted in Figure 5.1, this algorithm's pipeline consists of four stages: scene flow, egomotion estimation, dense stereo matching and 3D reconstruction. The VO part of this algorithm consists of the two stages: scene flow and egomotion estimation.

First, in order to obtain the scene flow, features are matched between four images, namely the left and right images of two consecutive frames. Next, from the features matched, the egomotion or camera motion is computed by minimizing the sum of re-projection errors.

For obtaining dense disparity maps, the ELAS [104] method is used. ELAS algorithm computes first a sparse disparity map providing the disparity of some pixels (which can be robustly matched due to their texture and uniqueness), and then uses the fist sparse disparity map computed to estimate the disparity of all pixels in order to generate a dense disparity map at the end.

The 3D reconstruction step is done by re-projecting reconstructed 3d points of the previous frame into the image plane of the current frame. In case a point falls onto a valid disparity, both 3D points are fused by computing their 3D mean. Since this last step is relatively simple and is not compute intensive, we do not study it in this work. Instead, we focus our work on the first two stages, i.e. the scene flow computation and egomotion estimation. They will be described in detail in sections

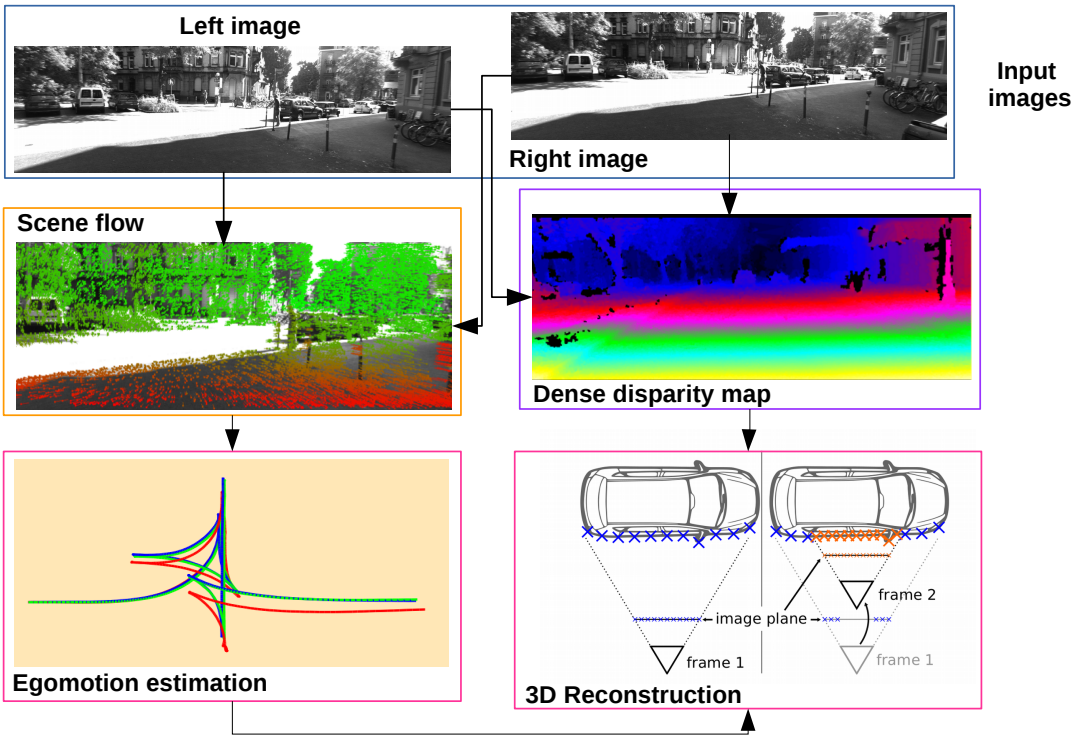


Figure 5.1 – The use case algorithm overview

5.3 and 5.4 respectively. Note that, the VO method proposed by authors of [102] can be applied in both monocular and stereo VO systems. However, in this work, we present only the stereo VO system case.

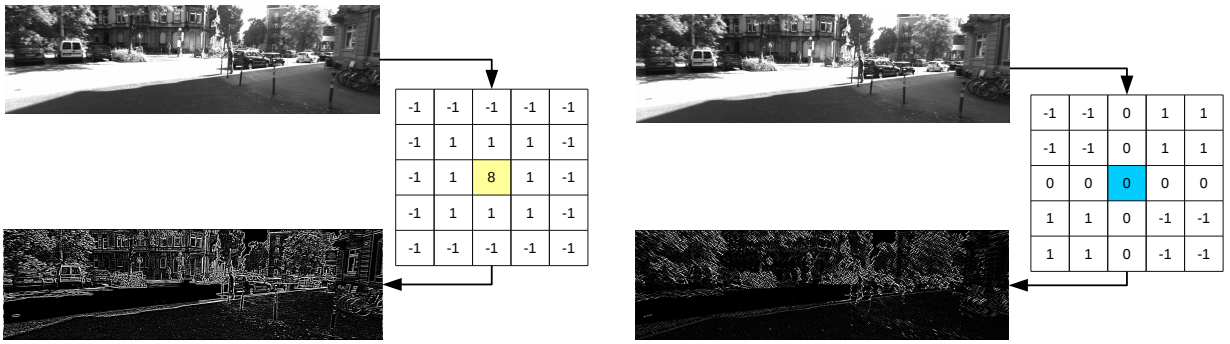
5.3 Scene Flow Description

The scene flow, computed from the left and right images of two consecutive frames, includes two main part: features detection and description (5.3.1) and features matching (5.3.2). Note that, for accuracy and reduction of the computational cost purpose, authors have chosen to perform features detection and their matching in two passes, but they can also be done in a single pass. However, we will consider that they are being done in two passes in the remainder of this work. Features detection and their matching are described below.

5.3.1 Feature Detection and Description

Feature detection and description are essential components of various computer vision algorithms, thus they have received considerable attention in the last decades. Several feature detectors and descriptors (e.g. Speeded-Up Robust Features Descriptor (SURF) [105], Harris detector [75]) have been proposed in the literature with a variety of definitions for what kind of points in an image is

potentially interesting [106]. Despite a large number of efficient features detectors and descriptors, authors chose to use a much simpler method. Indeed, for detecting features, input images (current left and right images) are first filtered with 5×5 blob and corner masks, as given in Figure 5.2.



(a) Blob (b) Corner

Figure 5.2 – Blob and Corner masks

Next, a non-maximum- and non-minimum-suppression algorithm is applied on filters images in order to find feature candidates which belong to one of four classes (i.e., blob max, blob min, corner max, corner min). The non-maximum- and non-minimum-suppression algorithm used corresponds to the algorithm 4 described in [107]. Algorithm 2 shows the pseudo code for non-maximum suppression. Since featured detection and description is done in two passes, authors chose for the first pass the value of 9 as neighborhood size while 4 is used for the second pass. A greater value is used for the first pass in order to get a small set features candidates which will be matched during the first step of the matching.

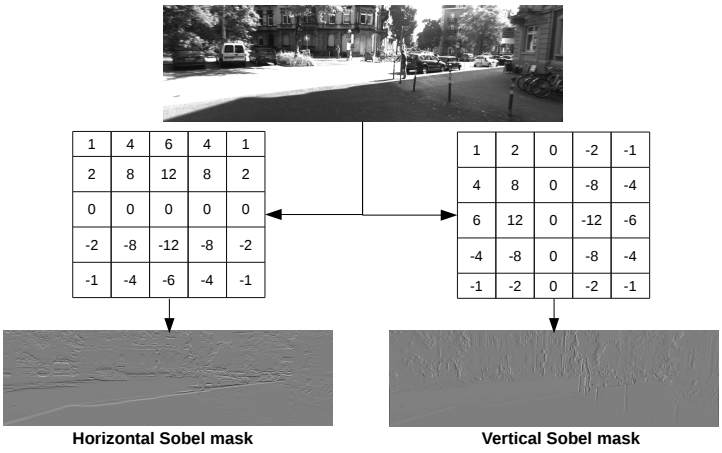
Once features candidates are detected, a descriptor is associated with each feature. Since these descriptors are based on Sobel filter results, 5×5 horizontal and vertical Sobel masks are used to filter input images as depicted in Figure 5.3a.

Features candidates descriptors are computed by concatenating Sobel filter responses using the layout given in 5.3b. In fact, for each feature candidate, 16 pixels from Sobel horizontal gradient and 16 pixels for the vertical gradient are chosen in a window of size 11×11. The colored cells on Figure 5.3b correspond to the positions of these 16 pixels to be chosen in the 11×11 window while the window's center corresponds to the coordinates of the current feature candidate. The number (32) and the position of chosen pixels in Sobel gradient images have been empirically determined by authors [102]. For speed-up the matching, the Sobel responses are computed on 8 bits. Hence 32×8 bits is required for each feature to store its descriptors. For facilitating next step implementation, the 32 descriptors of each feature are grouped by 4 and stored in an integer (of size 4 bytes) variable named d_i as depicted on Figure 5.4.

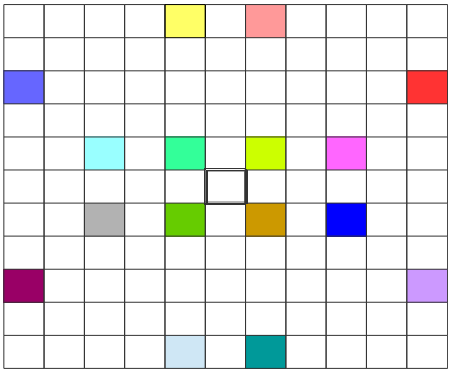
Figure 5.5 shows the complete schematic block diagram of features detection and description. Note that all computations performed in this block diagram are done twice: for the current left image and for the current right image. Features candidates spare and dense lists and Sobel gradients images of previous left and rights images must be stored in order to be used in the matching phase.

Algorithm 2 Non-maximum suppression pseudo code (algorithm 4 in [107])

for all (i, j) $\in n, 2n+1, ...2 \cap [0, W-n] \times [0, H-n]$ **do** ▷ H & W are input image sizes
    (mi, mj) $\longleftarrow$ (i, j);
    for all (i2, j2) $\in [i, i+n] \times [j, j+n]$ **do** ▷ n is the neighborhood size
        if image(i2, j2) $>$ image(mi, mj) **then**
            (mi, mj) $\longleftarrow$ (i2, j2);
        end if
    end for
    for all (i2, j2) $\in [mi-n, mi+n] \times$
        $[mj-n, mj+n] - [i, i+n] \times [j, j+n]$ **do**
        if image(i2, j2) $>$ image(mi, mj) **then**
            goto failed;
        end if
    end for
    MaximumAt (mi,mj); ▷ A maximum has been found, save it
    failed:
end for



(a) Sobel filter masks



(b) Feature descriptor

Figure 5.3 – Sobel filter mask and the layout used for computing feature descriptor

5.3.2 Feature Matching

After features candidates have been detected and their descriptors have been computed, the next step is their matching between left and right images two consecutive frames. This is done in six steps described below.

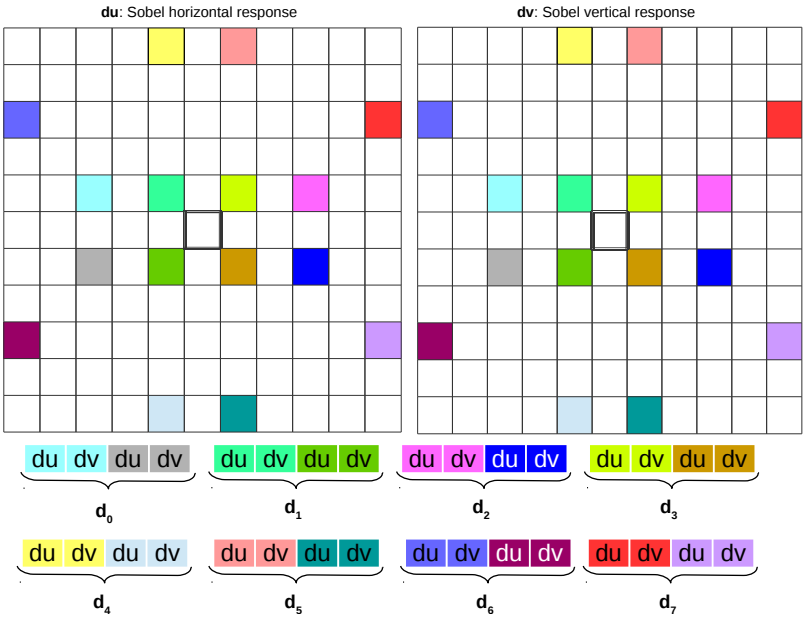


Figure 5.4 – Feature descriptor computation

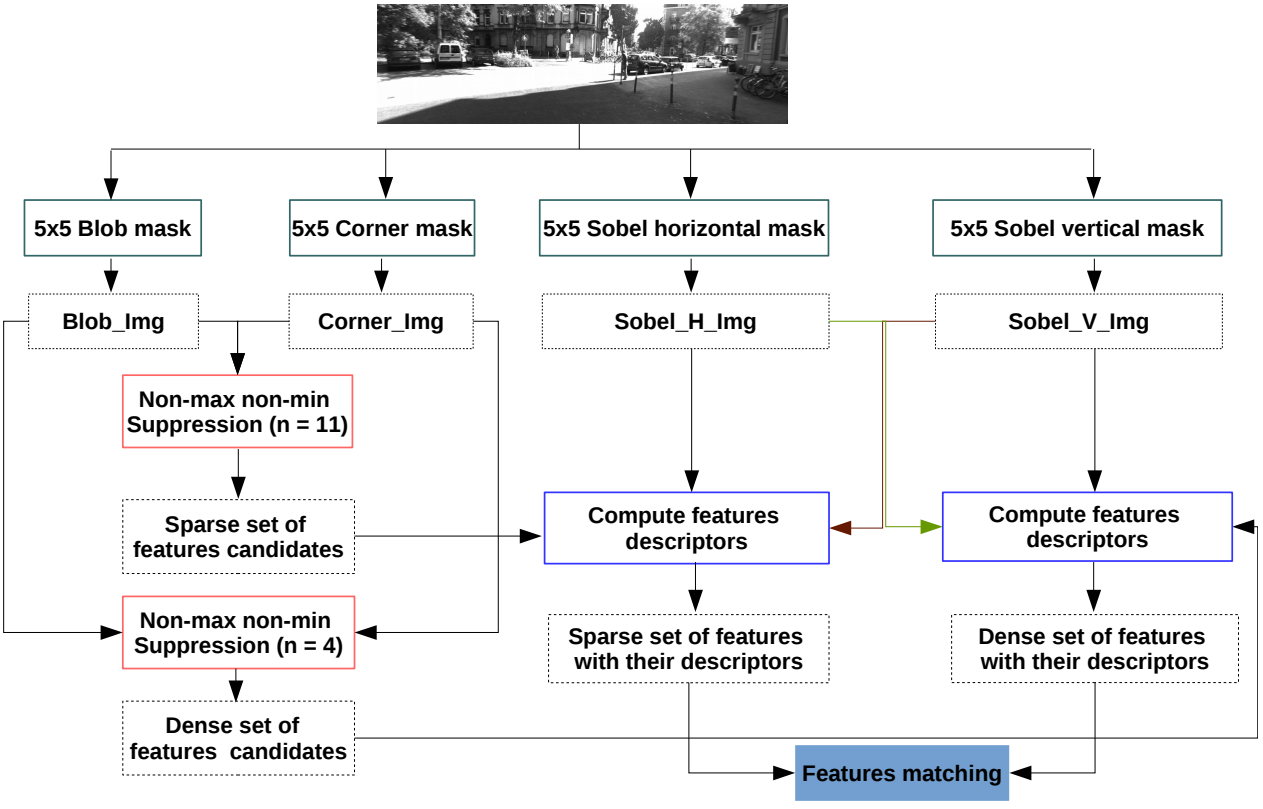


Figure 5.5 – Complete work flow of features detection and description

Step 1: Matching first pass

It is achieved by matching features candidates in a *circle*. Formally, this consists by starting from all feature candidates (detected during the first pass of non-maximum and non-minimum suppression) in the current left image:

1. find the best match in the previous left image within a $M \times M$ search window,
2. in the previous right image,
3. in the current right image,
4. in the current left image again.

If the last feature coincides with the first feature, then the *circle match* is accepted. The matching cost function used to find the best match is the SAD (5.3). This first matching provides a sparse set of features matched.

Step 2: First rejection of outliers

Taking the sparse set of features matched, sporadic outliers are removed by establishing neighborhood relations as edges of a 2D Delaunay triangulation on the feature locations in the current left image. The Delaunay triangulation [108] is a type of partitioning of a set of points positioned in a plane formed of triangles whose vertices are objects, and which together constitute a partition of the convex envelope of these objects. The condition to retain a match during Delaunay triangulation is that the match must be supported by at least two neighboring matches, where a match is supporting another match if its disparity falls within some threshold τ_{disp}.

Step 3: Build statistics

After the rejection of outliers, some statistics are build in order to speed up the second pass of matching. This is achieved by assigning each feature in the current left image to a 50×50 pixel bin of an equally spaced grid. Given all sparse feature matches, the minimum and maximum displacements for each bin are computed. Those statistics are used to locally narrow down the final search space, leading to faster matching and a higher number of matches at the same time.

Step 4: Matching second pass

Taking the search spaces computed in the previous step, match features candidates (detecting during the second pass of non-maximum and non-minimum suppression)) in the current left image. As for the first pass, the matching is done in a *circle* and a feature is considered to be matched only if the last feature coincides with the first feature. This matching provides a dense set of features matched.

Step 5: Second rejection of outliers

Again, sporadic outliers are removed by the mean of a 2D Delaunay triangulation.

Step 6: Refinement

Once rejection of outliers is completed, post-processing, naming refinement can be applied. The goal of refinement is to further improve feature localization. To do that, for each feature matched, four descriptors are computed from the left and right Sobel gradient images of two consecutive frames. Unlike, the descriptor used in 5.3.1, the descriptor used here is a subset of 16 pixels empirically localized in Sobel gradients images within a 5×5 window as depicted in Figure 5.6. Once descriptors are computed, a simply matching (using the SAD as a cost function) is performed for each feature matched between current left image and previous left image; next between the current left image and current right image; and current left image and previous right image. If a minimal disparity is found, the feature matched is updated.

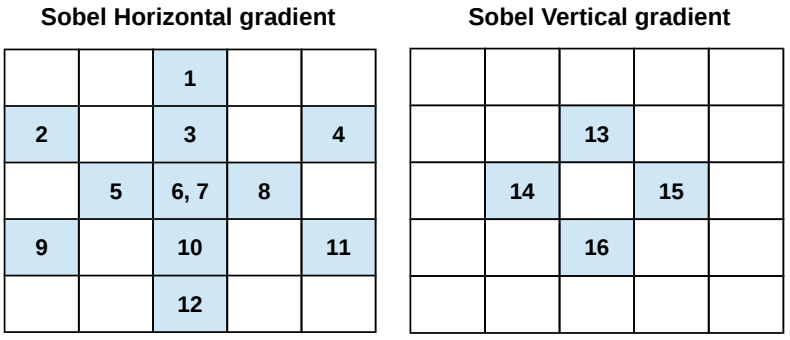


Figure 5.6 – Descriptor used for refinement

5.4 Egomotion Estimation Description

Given all *circular* features matched from the feature matching step, the camera motion is computed by minimizing the sum of re-projection errors and refining the obtained velocity estimates by means of a Kalman filter. First, bucketing is used to reduce the number of features and spread them uniformly over the image domain. Next, feature points are projected from the previous frame into 3D via triangulation using the calibration parameters of the stereo camera rig. Assuming squared pixels and zero skew, the re-projection into the current image is given by:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f & 0 & c_u \\ 0 & f & c_v \\ 0 & 0 & 1 \end{pmatrix} \left[(R(r)t) \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} - \begin{pmatrix} s \\ 0 \\ 0 \end{pmatrix} \right] \tag{5.5}$$

with

- homogeneous image coordinates $(u\ v\ 1)^T$
- focal length f
- principal point (c_u, c_v)
- rotation matrix $R(r) = R_x(r_x)R_y(r_y)R_z(r_z)$
- translation vector $t = (t_x\ t_y\ t_z)^T$
- 3d point coordinates $X = (x\ y\ z)^T$
- and shift $s = 0$ (left image), s =baseline (right image).

Let now $\pi^{(l)}(X;r,t) : \Re^3 \longrightarrow \Re^2$ denote the projection implied by Equation 5.5, which takes a 3D point X and maps it to a pixel $x_i^{(l)} \in \Re^2$ on the left image plane. Similarly, let $\pi^{(r)}(X;r,t)$ be the projection onto the right image plane. Using Gauss-Newton optimization, Expression 5.6 is iteratively minimized with respect to the transformation parameters (r,t).

$$\sum_{i=1}^{N} \parallel x_i^{(l)} - \pi^{(l)}(X;r,t) \parallel^2 + \parallel x_i^{(r)} - \pi^{(r)}(X;r,t) \parallel^2 \tag{5.6}$$

Here $x_i^{(l)}$ and $x_i^{(r)}$ denote the feature locations in the current left and right images respectively. The required Jacobians $J_{\pi^{(l)}}$ are readily derived from Equation 5.5. To be robust against outliers, the estimation approach is wrapped into a RANSAC (RANdom SAmple Consensus) [109] scheme, by first estimating (r,t) for 50 times independently using 3 randomly drawn correspondences. All inliers of the winning iteration are then used for refining the parameters, yielding the final transformation (r,t). On top of this estimation procedure a standard Kalman filter is placed, assuming constant acceleration. To this end, first the velocity vector $v = (rt)^T/\Delta_t$ is obtained as the transformation parameters divided by the time between frames Δ_t. The state equation is given by

$$\begin{pmatrix} v \\ a \end{pmatrix}^{(t)} = \begin{pmatrix} I & \Delta_t I \\ 0 & I \end{pmatrix} \begin{pmatrix} v \\ a \end{pmatrix}^{(t-1)} + \epsilon \tag{5.7}$$

and the output equation reduces to

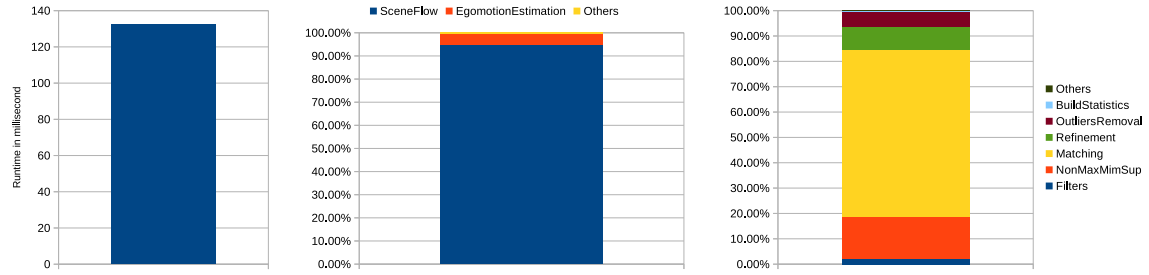
$$\frac{1}{\Delta_t} \begin{pmatrix} r \\ t \end{pmatrix}^{(t)} = \begin{pmatrix} I & 0 \end{pmatrix} \begin{pmatrix} v \\ a \end{pmatrix}^{(t)} + \nu \tag{5.8}$$

since v is directly observed. Here, a denotes acceleration, I is the 6×6 identity matrix and ϵ and ν represent Gaussian process and measurement noise, respectively.

5.5 Profiling and Analysis of Original Authors Version

The authors of the VO algorithm described in [102] provided a CPU based implementation. The source code is available on their web site[1]. In previous sections, we have seen that this VO algorithm consists of two part: the scene flow and the egomotion estimation.

In order to identify the most time-consuming functions, we profiled authors' CPU based implementation (as discussed in Section 3.5 – Step 1). In the remaining, *original CPU based version* will also means authors' CPU based implementation. We use the C++ *chrono* library to retrieve the execution times of all functions. Profiling data are collected on the Intel I7 CPU. Codes are compiled with the Gcc/++−8.1 compiler with same flags as those seen in Table 3.3. Additionally, we also use the flag *-msse3* since author's version includes SSE $2/3$ instructions. In our experiment and for the remaining, we use the sequence *2010_03_09_drive_0019* of the Karlsruhe dataset[2]. This sequence consists of 372 pairs of stereo images of size 1344×372. Figure 5.7 presents the performance of the VO algorithm. Note that we run the algorithm on the entire sequence and take the average of functions' execution time.



(a) Global time per frame (b) Repartition (c) Scene flow

Figure 5.7 – Performance of the original CPU based implementation on Intel I7

As can be seen in Figure 5.7, the VO algorithm requires 132.466 milliseconds per frame on average on our Intel I7 CPU. On the other hand, this same algorithm takes 466.91 milliseconds on the Tegra X1 ARM CPU. This performance may not meet the real-time requirements found in most ADAS systems for example. Sub Figure 5.7b shows that the most time-consuming part of this VO algorithm is the scene flow part. Indeed, 95% of the global execution time corresponds to the scene flow part. In Sub Figure 5.7b, we means by *Others*, input images loading, memory allocation/deallocation, etc.

Looking in detail at the scene flow part as depicted on sub Figure 5.7c, we find that non-maximum- and non-minimum-suppression, feature matching, and refinement are the three most time-consuming routines. Here, *Others* corresponds to memory allocation and deallocation operations. Hence, it is these three functions that we will seek to parallelize in order to accelerate the VO algorithm. The time consuming by filters (Sobel, blob, and corner) computation is not significant.

[1] http://www.cvlibs.net/software/libviso/
[2] http://www.cvlibs.net/datasets/karlsruhe_sequences/

However, we will also parallelize them later since they are the input of the non-maximum- and non-minimum-suppression.

5.6 Original CPU Based Scene Flow Implementation

Authors chose to implement Sobel and corner filters using SSE instruction while the blob filter is in pure C++. Sobel gradients are computed on $8-$bit in order to speed up the matching and the refinement. Algorithm 3 shows the authors' CPU based pseudo-code for the non-maximum- and non-minimum-suppression algorithm. As we can see, loops of lines 1 and 2 iterate over blob (If1) and corner (If2) filters results images.

Once maximums and minimums of blob and corner filters results have been found, the next step is the computation of their descriptors. Algorithm 4 shows the pseudo code of the function that computes features candidates descriptors. In fact, there are no computation in this function, only load (from Sobel gradients) and store (in d_i) memory operations.

After the computation of features candidates descriptors, features candidates are copied from the spare and dense vectors of features candidates to integer arrays as depicted on Algorithm 5.

Algorithm 6 shows the pseudo code corresponding to the work flow of features detection and description (as seen on Figure 5.5). Note that, this sequence of operations is done twice : one for the current left image and one for the current right image.

As explained in Subsection 5.3.2, each feature candidate is matched in a circle. Algorithm 7 shows the pseudo of the matching cost function used. The main computation part is from line 8 to 22 and the most computation instruction is SAD. Authors chose to implement it using SSE instructions as shown in Listing 5.1. Since Sobel responses are quantized on $8-$bit and SSE registers are on $128-$bits, only two instructions (lines 5 and 6) are needed for the SAD computation.

```
1 __m128i xmm1 = _mm_load_si128((__m128i*)(m1+step_size*i1+4));
2 __m128i xmm2 = _mm_load_si128((__m128i*)(m1+step_size*i1+8));
3 __m128i xmm3 = _mm_load_si128((__m128i*)(m2+step_size*(*i2_it)+4));
4 __m128i xmm4 = _mm_load_si128((__m128i*)(m2+step_size*(*i2_it)+8));
5 xmm3 = _mm_sad_epu8 (xmm1,xmm3);
6 xmm4 = _mm_sad_epu8 (xmm2,xmm4);
7 xmm4 = _mm_add_epi16(xmm3,xmm4);
8 cost = (double)(_mm_extract_epi16(xmm4,0)+_mm_extract_epi16(xmm4,4));
```

Listing 5.1 – SAD implementation in SSE

Algorithm 8 shows the function that performs features matching. Before to match features candidates, a search range is computed for each of them. Thus, four std::vector arrays are allocated (line 1), one for each features candidates list. These arrays are filled in line 2 computing minimal row and column of the bin to which each feature candidate belongs. Next, by iterating over all features candidates of the previous left image, each feature is matched in a circle from line 6 to line 9. As explained in 5.3.2, a feature is retained only if the last feature coincides with the first feature (line

Algorithm 3 Non-maximum and non-minimum suppression: Original CPU based pseudo code

Require: If1, If2, W, H, n, threshold, feature_list ▷ feature_list is a C++ std vector
1: **for all** i = n + margin; i < W − n − margin; i + = n +1 **do** ▷ margin = 9
2: **for all** j = n + margin; j < H − n − margin; j + = n +1 **do**
3: f1mini = i; f1minj = j; f1maxi = i; f1maxj = j; ▷ f1 denotes blob while
4: f2mini = i; f2minj = j; f2maxi = i; f2maxj = j; ▷ f2 corresponds to corner
5: **for all** i2 = i; i2 <= (i + n); i2++ **do**
6: **for all** j2 = j; j2 <= (j + n); j2++ **do**
7: ▷ Instructions of lines 4 to 6 of Algorithm 2 four times
8: **end for**
9: **end for**
10: **for all** i2 = f1mini − n; i2 <= min(f1mini + n, W −1− margin); i2++ **do**
11: **for all** j2 = f1minj − n; j2 <= min(f1minj + n, H −1− margin); j2++ **do**
12: **if** If1(i2, j2) < If1(f1mini, f1minj) **then** goto failed;
13: **end for**
14: **end for**
15: **if** If1(f1mini, f1minj) ≤ − threshold **then** ▷ the threshold is set to 50
16: feature_list.push_back(f1mini, f1minj, If1(f1mini, f1minj), 0);
17: **end if**
18: failed:
19: **for all** i2 = f1maxi − n; i2 <= max(f1maxi + n, W −1− margax); i2++ **do**
20: **for all** j2 = f1maxj − n; j2 <= max(f1maxj + n, H −1− margax); j2++ **do**
21: **if** If1(i2, j2) < If1(f1maxi, f1maxj) **then** goto failed;
22: **end for**
23: **end for**
24: **if** If1(f1maxi, f1maxj) ≥ − threshold **then**
25: feature_list.push_back(f1maxi, f1maxj, If1(f1maxi, f1maxj), 1);
26: **end if**
27: failed:
28: Same code from line 10 to 14, replace f1 by f2
29: **if** If2(f2mini, f2minj) ≤ − threshold **then**
30: feature_list.push_back(f2mini, f2minj, If2(f2mini, f2minj), 2);
31: **end if**
32: Same code from line 19 to 23, replace replace f1 by f2
33: **if** If2(f2maxi, f2maxj) ≥ − threshold **then**
34: feature_list.push_back(f2maxi, f2maxj, If2(f2maxi, f2maxj), 3);
35: **end if**
36: **end for**
37: **end for**
Ensure: feature_list

Algorithm 4 Descriptors Computation: Original CPU based pseudo code

Require: feature_list, I_du, I_dv, H, W ▷ du: horizontal and dv: vertical Sobel gradients

1: **for all** vector<feature_strt>::iterator it = feature_list.begin(); it ! = feature_list.end(); it++ **do**
2: Extract current feature candidate coordinates i and j from feature_list;
3: Compute d_0 to d_7 as depicted on Figure 5.4
4: **end for**

Ensure: feature_list

Algorithm 5 Copying features candidates to a classic array : Original CPU based pseudo code

Require: spare_feature_list, dense_feature_list

1: Allocate an array of integer type and size $sizeof(feature\_str) * spare\_feature\_list.size()$;
2: Allocate an array of integer type and size $sizeof(feature\_str) * dense\_feature\_list.size()$;
3: **for all** vector<feature_strt>::iterator it iterating over spare_feature_list **do**
4: Copy current feature candidate from spare_feature_list to array allocated on line 2;
5: **end for**
6: **for all** vector<feature_strt>::iterator it iterating over dense_feature_list **do**
7: Copy current feature candidate from dense_feature_list to array allocated on line 3;
8: **end for**

Ensure: arrays of line 2 and 3

Algorithm 6 Features detection and description work flow: Original CPU based pseudo code

Require: Current left image (I1), H, W

1: sobel_SSE ▷ Sobel 5×5 SSE2/3 based implementation
2: corner_SSE ▷ Corner 5×5 SSE2/3 based implementation
3: blob ▷ Blob 5×5 pure C++ based implementation
4: vector<feature_strt> spare_feature_list;
5: vector<feature_strt> dense_feature_list;
6: feature_list_spare = Algorithm 3 ▷ first pass of non-max/min suppression with n = 9
7: feature_list_spare = Algorithm 4 ▷ compute features descriptors
8: feature_list_dense = Algorithm 3 ▷ second pass of non-max/min suppression with n = 3
9: feature_list_dense = Algorithm 4 ▷ compute features descriptors
10: feature_list_spare, feature_list_dense = Algorithm 5

Ensure: feature_list_spare, feature_list_dense, Sobel gradients images

Algorithm 7 Matching cost function: Original CPU based pseudo code

Require: feature_list_1, current feature candidate, feature_list_2, index_list, minimal_disp_index ▷ and search_ranges if second pass of matching

1: Initialize minimal_cost to 10000000;
2: Extract current feature candidate coordinates (u1, v1) from feature_list_1;
3: Load feature descriptors from feature_list_1 into two SSE $128-$bit registers;
4: Compute disparity search range for feature to be matched;
5: **if** it's the second matching pass **then**
6: Restrict disparity search range; ▷ using search_ranges (computed by Build statistics step)
7: **end if**
8: **for all** u_bin = u_bin_min; u_bin $\leq$ u_bin_max; u_bin++ **do**
9: **for all** v_bin = v_bin_min; v_bin $\leq$ v_bin_max; v_bin++ **do**
10: Extract from index_list the index corresponding to this feature;
11: **for all** iterate over std::vector index_list[index] **do**
12: Extract current feature candidate coordinates (u2, v2) from feature_list_2;
13: **if** u2 and v2 are in disparity search range **then**
14: Load feature descriptors from feature_list_2 into two SSE $128-$bit registers;
15: cost = Compute SAD between descriptors of first registers (line 4) and second
16: registers (line 10); ▷ SAD is implemented using SSE instructions
17: **if** cost $<$ minimal_cost **then**
18: minimal_cost = cost; minimal_disp_index = index;
19: **end if**
20: **end if**
21: **end for**
22: **end for**
23: **end for**

Ensure: minimal_disp_index

10 to line 15).

Algorithm 8 Features Matching: Original CPU based pseudo code

Require: feature_list_sparse (4) of previous and currents left and right images ▷ instead, feature_list_dense if it is the second pass of features matching

1: Allocates 4 arrays of type std::vector for storing bin index vectors of the 4 feature_list_sparse element;
2: Fill the four precedent allocated arrays;
3: **for all** i1p = 0; i1p < n1p; i1p++ **do** ▷ n1p = size of previous left image's feature_list_sparse
4: Extract current feature coordinates from previous left image's feature_list_sparse
5: Compute row and column of statistics bin to which current feature belongs;
6: Algorithm 7; ▷ find best match between previous left and right images
7: Algorithm 7; ▷ find best match between previous and current right images
8: Algorithm 7; ▷ find best match between current left and right images
9: Algorithm 7; ▷ find best match between current and previous left images
10: **if** match found in line 6 = match found in line 9 **then**
11: Extract coordinates corresponding to best matches found in line 7 and 8;
12: **if** if those of line 7 are greater than those of line 8 **then** ▷ means disparities are positive
13: spare_points_matched_list.push_back(4 couples of (u, v, feature index));
14: ▷ couples correspond to best matches found from line 6 to 9
15: **end if**
16: **end if**
17: **end for**

Ensure: spare_points_matched_list ▷ instead, dense_points_matched_list if it is the second pass of features matching

During the refinement, matched points are matched again in order to improve features localization, i.e. features coordinates in previous left, current right and previous right images. For that purpose, the SAD is computed within a window of size 5×5 as shown in Algorithm 9.

Algorithm 10 shows the pseudo code of the function that performs the refinement. The computational part, here, is the matching instructions from line 2 to line 4.

5.7 Scene Flow Adaptation for Parallelization

In order to parallelize the scene flow either on CPU or GPU, we have to make some changes (as discussed in Section 3.5 – Step 1). We present the main modifications made in the following subsections.

Algorithm 9 Refinement cost function: Original CPU based pseudo code

Require: point matched coordinates (u1, v1, u2, v2), Sobel gradients for Img1 and Img2

Check if point matched coordinates are within margin;
Compute descriptors of (u1, v1) in Sobel gradients of Img1 as depicted on Figure 5.6;
Load previous descriptors computed into an SSE $128-$bit register;
Create a static array of size 25 for storing cost values;
for all dv = 0 to 5 **do**
    for all du = 0 to 5 **do**
        Compute descriptors of (u2+du-2, v1+dv-2) in Sobel gradients of Img2;
        Load previous descriptors computed into an SSE $128-$bit register;
        cost[dv$*5+$du] = Compute SAD between descriptors of line 3 and those of line 8;
                                ▷ SAD is implemented using SSE instructions
    end for
end for
for all i $= 0$ to 25 **do**
    Find minimum cost and corresponding index;
end for
Update u2 and v2;

Ensure: updated u2 and v2

Algorithm 10 Refinement: Original CPU based pseudo code

Require: dense_points_matched_list, Sobel gradients of previous and currents left and right images

for all std::vector iterator ite $= 0$ to size of dense_points_matched_list **do**
    Algorithm 9; ▷ Img1 and Img2 = current and previous left images
    Algorithm 9; ▷ Img1 and Img2 = current left and right images
    Algorithm 9; ▷ Img1 current left image and Img2 = previous right image
    dense_points_matched_list.push_back(ite);
end for

Ensure: dense_points_matched_list

Remove Lists

We have already seen in Section 3.4 that the parallelization of a code containing operations of adding new elements to a list requires the use of atomic operations. We also know that critical sections and atomic operations degrade performance, especially when there is not enough workload in loops. Therefore, we first replace all lists by classic arrays (as discussed in Section 3.5 – Step 2) allocated dynamically. We can't use static arrays since we do not know in advance the number of features candidates that will be found or that of points that will be matched. Replacing std::vector by classics arrays will allow us to perform the manual deep copy (as discussed in Section 3.5 – Step 1) when we will parallelize our application on GPU, but also the problem due to the *push_back* routine' call (as discussed in Sections 3.4 – Problem 3).

Rewrite Filters

Although Sobel and corner filters are vectorized using SSE instructions, this can run only on an Intel CPU. The SIMD intrinsics (e.g. SSE, AVX) have the disadvantage of not being portable. Therefore, we have to port Sobel and corner filters from SSE instructions to those of NEON in order to run our use case on the Tegra X1 CPU. NEON is the SIMD intrinsics for ARM CPUs. We must also rewrite Sobel and corner filters in pure C++, which can be parallelized later with OpenMP/OpenACC and ported to CUDA for GPU execution.

Non-maximum and non-minimum suppression Algorithm Adaptation

The non-maximum and non-minimum suppression algorithm can be parallelized in several ways. We are going to present two strategies in which the features description part (Algorithm 4) is integrated.

First way to parallelize the non-maximum and non-minimum suppression algorithm

In this first strategy, we use two arrays that have the same sizes as input images and will replace the lists (sparse_feature_list and dense_feature_list in Algorithm 3) used in the original version. Since features candidates are a small subset of the pixels constituting the input images, we change the set of features classes from $\{0, 1, 2, 3\}$ to $\{10, 11, 22, 23\}$ which will allow us to determine later which element of these arrays corresponds to a feature. Since the feature description algorithm (4) is memory bound, we compute descriptors in the same loops (lines 1 and 2 of Algorithm 3) where they were detected. Once features are detected and described, we go through the features arrays to count the total number of features candidates. Next, features candidates are copied into new arrays as seen in Algorithm 5. Listing 5.2 resumes this first strategy implementation.

```
1 //allocate Tab, an array of size W*H and initialize it to 0
2 for(int i = n+margin; i < H-n-margin; i += n+1)
3   for(int j = n+margin; j < W-n-margin; j += n+1) {
```

```
4     //set min and max to current i and j (lines 3-4 of Algorithm 3)
5     //find min and max in I21 and If2 within a window (5-9 of Algorithm 3)
6     //if there is no other min arround in If1 (10-14 of Algorithm 3)
7     if(If1[f1mini*W+f1minj] ≤ − threshold) //a feature candidate is found {
8       //compute descriptors (line 3 of Algorithm 4)
9       Tab[i*W+j] ⟵ {10, If1[f1mini*W+f1minj], f1mini, f1minj, d0,..,d8}
10    }
11    //if there is no other maximum arround in If1 (19-23 of Algorithm 3)
12    if(If1[f1maxi*W+f1maxj] ≥ − threshold) {//a feature candidate is found
13      //compute descriptors (line 3 of Algorithm 4)
14      Tab[i*W+j+1] ⟵ {11, If1[f1maxi*W+f1maxj], f1maxi, f1maxj, d0,..,d8}
15    }
16    //do the same thing for minumax and maximum in If2
17  }
18 int sum_features = 0;
19 for(int i = 0; i < H; i++)
20   for(int j = 0; j < W; j++)
21     if(Tab[i*W+j] != 0) sum_features++;
22 features_list = new array of integers, sizeof(feature_str) * sum_features
23 for(int i = 0; i < H; i++)
24   for(int j = 0; j < W; j++)
25     if(Tab[i*W+j] != 0) //copy features from Tab to features_list
```

Listing 5.2 – First version of features detection and description

Second way to parallelize the non-maximum and non-minimum suppression algorithm

The second strategy goal is to maximize parallel execution by structuring the code in a way that it exposes as much parallelism as possible. This strategy allows also to reduce our use case memory footprint. Thus, we start by allocating a simple *unsigned char* array which has the same size as input images. As can be seen in Listing 5.3 (lines 4, 7, 9 and 11), features candidate index are set either to 10, or 11, or 22, or 23. Next, we compute the number of features per line and save it in a temporal *short* array (from lines 14 to 17 of Listing 5.3). As in the first strategy, the total number of features candidate is computed. We also compute the addresses range to features candidate per line (from lines 18 to 24 of Listing 5.3). Last, we compute descriptors of each feature candidate and save features data into the features array (from lines 25 to 36 of Listing 5.3).

```
1  uint8_t *Tab = new array of size W*H and initialize its elements with 0
2  for(int i = n+margin; i < H-n-margin; i += n+1)
3    for(int j = n+margin; j < W-n-margin; j += n+1) {
4      //lines 4 to 6 of Listing 5.2
5      if(If1[f1mini*W+f1minj] ≤ − threshold) Tab[f1mini*W+f1minj] = 10;
6      //line 11 of Listing 5.2
7      if(If1[f1maxi*W+f1maxj] ≥ − threshold) Tab[f1mini*W+f1minj] = 11;
8      //lines 4 to 6 of Listing 5.2 (If1 is replaced by If2)
9      if(If2[f2mini*W+f2minj] ≤ − threshold) Tab[f2mini*W+f2minj] = 22;
10     //line 11 of Listing 5.2 (If1 is replaced by If2)
11     if(If2[f2maxi*W+f2maxj] ≥ − threshold) Tab[f2mini*W+f2minj] = 23;
```

```
12 }
13 //get the number of features candidate per line
14 short *lineSum = new array of size H and initialize it to 0
15 for(int i = 0; i < H; i++)
16   for(int j = 0; j < W; j++)
17     if(Tab[i*W+j] != 0) lineSum[i]++;
18 //compute the addresses range to features candidate per line
19 access_range = array of type (struct{int bing, ind}), size H*sizeof(struct)
20 int sum_features = 0;
21 for(int i = 0; i < H; i++) {
22   //compute beginning and ending of addresses of features in this line
23   sum_features += lineSum[i];
24 }
25 features_list = new array of integers, sizeof(feature_str) * sum_features
26 for(int i = 0; i < H; i++) {
27   if(lineSum[i] != 0) {
28     //get addresses range to features cadidate in current line
29     for(int j = 0; j < W; j++) {
30       if(Tab[i*W+j] != 0) {
31         //compute this feature descriptors
32         //save the feature data into features_list
33       }
34     }
35   }
36 }
```

Listing 5.3 – Second version of features detection and description

This second implementation consumes less memory than the first, and unlike the first version (from lines 23 to 25 of Listing 5.2), feature data storage (from lines 25 to 36 of Listing 5.3) can be done in parallel since we know the starting and ending access addresses to features per line.

Feature Matching Algorithm Adaptation

We make the following modification in the feature matching algorithm:

- Allocate a new array of size features candidate list of the current left image. This new array has the same type as the point matched structure. However, we added a supplementary boolean variable which must be set to true if a feature has been matched.

- The index vectors allocated and filled on lines 1 and 2 of Algorithm 8 must be copied in classic arrays for two main reasons. First, most sub-vectors of these index vectors are empty. Thus, by copying only non-empty sub-vectors of these index vectors in classic arrays, we can speed up the matching cost computation (loop of the line 11 in Algorithm 7). Second, this will allow us to avoid the problem of shallow copy (seen in 2.3) when we will parallelize the matching on GPU.

- As we did with the Sobel and corner filters, we port the SAD cost function computation (line 15 of Algorithm 7) from SSE to NEON for execution on ARM CPU. Note that the SAD instruction (line 5 of Listing 5.1) is not available in NEON. Therefore, we must explicitly implement the SAD in NEON. We have also rewritten the SAD in scalar C++ in order to parallelize it with OpenACC/OpenMP and CUDA for execution on GPU. Listing 5.4 shows the corresponding C++ version of the SAD seen in Listing 5.1.

```
1 uint8_t *desc_temp_m1 = (uint8_t*)(m1 + step_size * i1 + 4);
2 uint8_t *desc_temp_m2 = (uint8_t*)(m2 + step_size * new_k[i2_it] + 4);
3 double cost = 0.0;
4 for (int j = 0; j < 32; j++)
5   cost += (double)abs(desc_temp_m1[j] - desc_temp_m2[j]);
```

Listing 5.4 – Pure C++ implementation of SAD

Therefore, our adapted implementation of matching cost function (Algorithm 7) is the same as the original except:

  - Lines 3 and 14 are changed to declaring and initializing two char pointers to the first descriptor (d_0) of each feature (lines 1 and 2 of Listing 5.4).
  - Line 15 is replaced by lines 3 to 5 of Listing 5.4.

Algorithm 11 Adapted implementation of features matching function

Require: feature_list_sparse (4) of previous and currents left and right images ▷ instead, feature_list_dense if it is the second pass of features matching

1: Lines 1 and 2 of Algorithm 8;
2: Allocate new array, and fill them with the index vectors;
3: feature_array = array having same size and type as feature_list_sparse of previous left image;
4: Initialize feature_array to zero;
5: **for all** i1p = 0; i1p < n1p; i1p++ **do** ▷ n1p = size of previous left image's feature_list_sparse
6: Lines 4 to 9 of Algorithm 8;
7: feature_array = lines 10 to 15 of Algorithm 8;
8: **end for**
9: Copy valid elements of feature_array into the spare_points_matched_list vector;

Ensure: spare_points_matched_list ▷ instead, dense_points_matched_list if it is the second pass of features matching

Note that, after the matching step we copy matched points into lists (std::vector). This is required since the next steps (build statistics, outliers removal, and egomotion estimation) use them and we will no longer modify them. Algorithm 11 shows our adapted implementation of the features matching function.

Refinement Algorithm Adaptation

As for the matching cost function, the instructions for loading descriptors and computing the SAD (lines 3, 8 and 9 of Algorithm 9) have to be programmed in scalar C++ instead of SSE for execution on GPU, and in NEON in order to target NVIDIA Tegra X1 CPU. Furthermore, we copy points matched into a classic array of the same type and size as *dense_points_matched_list*. Thus it is this new array that will be used in line 5 of Algorithm 10. Then, after the refinement is done, we copy back points matched into the *dense_points_matched_list* vector, which will be used during the egomotion estimation phase.

5.8 Scene Flow Parallelization on CPU

The scene flow pipeline can be parallelized on CPU in several ways. We present three of them in the following paragraphs.

Parallelization of Authors CPU based implementation Since the inputs images are independent, the workflow depicted in Algorithm 6 can be run in parallel on both images. This is done by exploiting task parallelism through OpenMP *parallel sections* construct. We will call this parallelized version *OriginalOMP* for the remaining of this chapter. Filters, features detection and matching will be parallelized as follow:

- *Filters* – They are in SSE for Intel CPU and NEON for ARM CPU. Filters results of left and right images are computed in parallel by creating two *OpenMP section* as can be seen in Listing 5.5.

```
1 #pragma omp parallel sections num_thread(2)
2 {
3   #pragma omp section
4   {/*compute filters results for current left image*/}
5   #pragma omp section
6   {/*compute filters results for current right image*/}
7 }
```

Listing 5.5 – Filters parallelization with OpenMP sections

- *Features detection and description* – This corresponds to the Algorithm 3. Since features detection and description is performed twice (spare and dense) for both current left and right images, we create two OpenMP *parallel sections* in order to balance the workload. Thus, the pseudo code of Listing 5.6 is run twice.

```
1 #pragma omp parallel sections num_threads(2)
2 {
3   #pragma omp section //for current left image
4   {
```

```
5       //do non-maximum and non-minimum suppression (Algorithm 3)
6       //compute descriptors (Algorithm 4)
7       //copy features to feature array (Algorithm 5)
8     }
9     #pragma omp section
10    {/*same thing as describing in above section for right image*/}
11  }
```

Listing 5.6 – FIlters parallelization with OpenMP sections

- *Features matching* − The matching cost (SAD) is in SSE for Intel CPU and NEON for ARM CPU. We parallelize the main loop of Algorithm 8 (line 3) with the OpenMP *parallel for* construct. Since matched points are stored in a list, the instruction of line 13 must be mutually exclusive (as seen in Section 3.4) by using OpenMP *critical section* construct. We can't use the *atomic* directive since the *push_back* operation is not implemented in OpenMP.

- *Refinement* − This step remains sequential since the main loop (line 1 of Algorithm 10) iterate on the points matched vector and std::vector::iterator is not allowed in OpenMP loop. Figure 5.8 depicts the work flow of the parallelized version of authors CPU based implementation.

Parallelization of our Adapted Version with OpenMP Feature detection, description, and matching present severals instruction level parallelism. Our adapted versions will be parallelized with OpenMP as follow. OpenMP versions will be called C++OMP and C++SimdOMP (for matching and refinement where the SAD is implemented with SIMD intrinsics).

- *Filters* − are parallelized with the OpenMP work-sharing *parallel for* constructs.

- *Features detection and description* − we parallelized the two versions presented in 5.7:

  - *First version* (Listing 5.2): the loops of lines 2 and 3 are parallelized with the OpenMP work-sharing *parallel for* constructs. We do not parallelize the other nested loops because they contain atomic operations and excessive use of the *atomic* construct may hurt performance. Loops at lines 19 and 20 are parallelized with the OpenMP *parallel for collapse(2) reduction(+:)* composite construct. The last two loops (lines 23 − 24) that copy features candidates from a large array into a small array remain sequential.
  - *Second version* (Listing 5.3): loops of lines 2 and 3 are parallelized with the OpenMP work-sharing *parallel for* constructs. The loops of lines 15 − 16 that compute the total number of features detected are parallelized with *parallel for collapse(2) reduction(+:)*. Loop of line 21 remains sequential because the compute of addresses range of features candidate per line present a loop-independent dependency (seen 3.4). We also parallelized the loop of line 26 with the OpenMP *parallel for* directive.

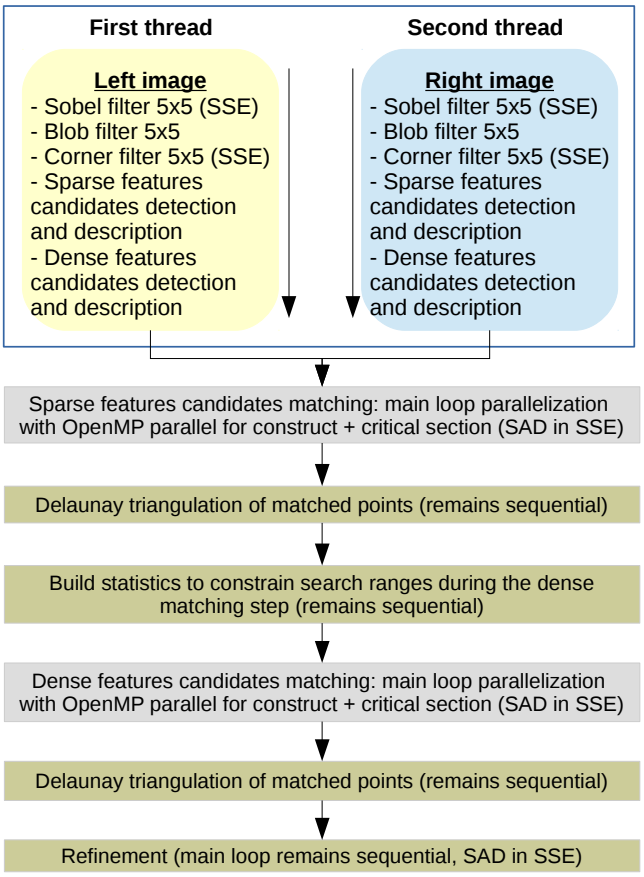


Figure 5.8 – Work flow of the parallelized version of authors CPU based implementation

- *Featuring matching* (Algorithm 11) – line 2 is parallelized using OpenMP *parallel sections*. Loops of lines $4-5$ are parallelized with the OpenMP *parallel for* directive. The copy of points matched (line 9) from an array to a std::vector is done sequentially because the copy is done through a *push_back* operation which is atomic. The scalar C++ version of SAD computation (lines $4-5$ of Listing 5.4) is parallelized with the *parallel for reduction(+:)* construct.
- *Refinement* (Algorithm 10) – the main loop (at line 1) is parallelized with the OpenMP *parallel for* directive. Figure 5.9 depicts the work flow of our adapted version parallelized with OpenMP.

Parallelization of our Adapted Version with OpenACC Feature detection, description, and matching are parallelized with OpenACC using the same approaches (described above) that we used with OpenMP. We replaced the *omp parallel for* by *acc kernels loop independent* and *acc parallel loop independent* composite constructs. OpenACC versions will be called as fellow:

- C++SimdACCKr and C++SimdACCPr for matching and refinement where the SAD is

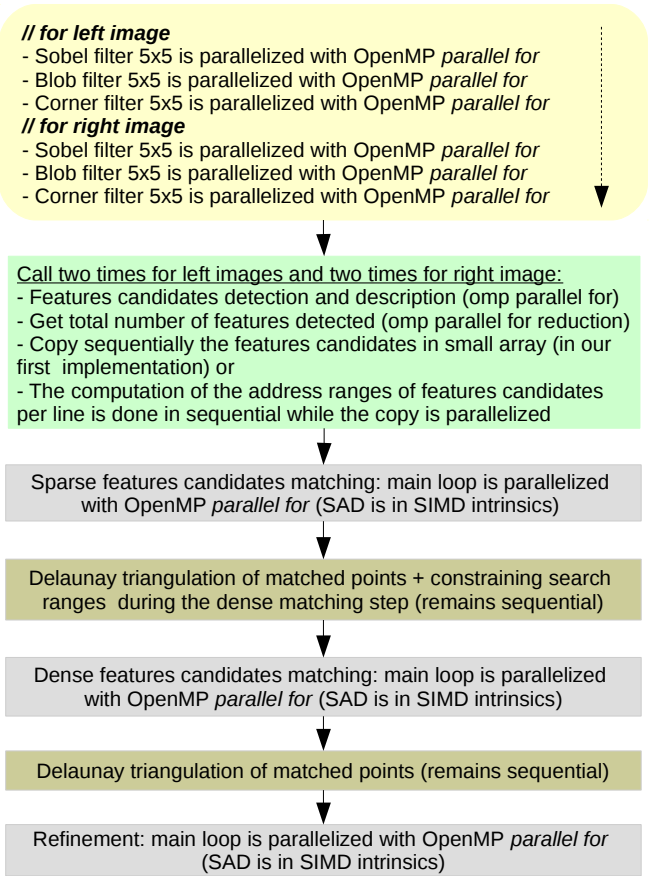


Figure 5.9 – Work flow of the parallelized version of authors CPU based implementation

implemented with SIMD intrinsics;

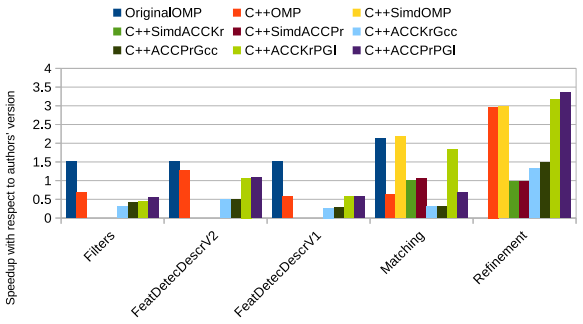
- C++ACCKr and C++ACCPr for C++ scalar versions. The "Kr" is for *kernels* variants and "Pr" for those of *parallel*.

5.8.1 Obtained Results

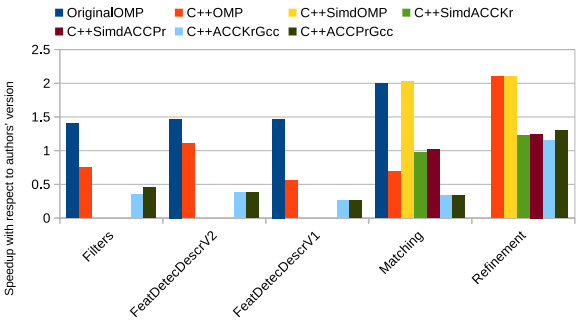
We evaluate the performance of new versions of scene flow on CPUs (Intel I7 and NVIDIA Tegra X1 CPU). The GCC compiler version 8.1 is used to compile the OpenMP. Both GCC and PGI (community version 18.11.1) compilers are used on OpenACC versions except the SIMD variants that are compiled only with Gcc because our version of PGI has not support for SSE instructions. We used the flags seen in Table 3.3. However, the flag *-mcpu=cortex-a57+simd+fp* has been added to GCC flags for the NVIDIA Tegra X1 CPU.

Figures 5.10a and 5.10b show the performance speedup obtained on Intel I7 and NVIDIA Tegra X1 CPUs for the scene flow computation. We make following observations:

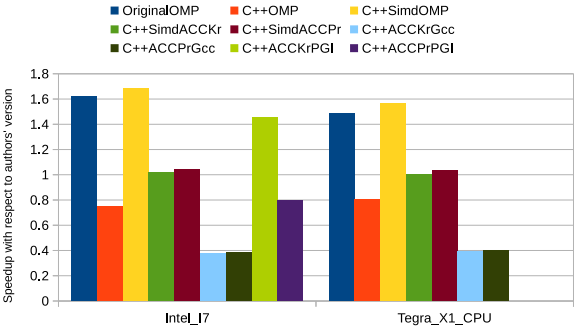
- *Filters* – C++ variants have a speed-down factor of $2.129\times$ on average in comparison to



(a) Intel I7



(b) Tegra X1 CPU



(c) Overall performance

Figure 5.10 – Scene flow performance on CPU: Speedup with respect to authors' version

authors version. This result is not surprising because as already mentioned Sobel and corner filters are fully implemented in SIMD intrinsics (SSE for Intel CPU and NEON for ARM CPU) the authors' version. Although the auto-vectorization capability of compilers has improved, writing a vectorized code with SIMD intrinsics remains more effective. Combining task parallelism with manual vectorization improves filters performance with a factor of $1.48\times$ on average.

- *Features detection and description* – The authors' version combined with OpenMP task parallelism produces the best performance with a speedup factor of $1.489\times$ on average. Authors' version, where a std::vector is used to store features candidates, has a lower execution time than our adapted versions because we store features candidates in an array of the same size as input images. Note that features candidates represent only at most 4% of the elements of the array containing them. Thus, we spend a significant in the step of copying features candidate from a larger array to a smaller one. Our first parallelization strategy (Listing 5.2) is less efficient than the second strategy since this last is entirely parallelized as explained in Section 5.7.

- *Features matching* – Although the OriginalOMP version contains a mutually exclusive instruction, we obtained a speedup factor of $2.087\times$ on average. This is due to the fact that

there is sufficient work inside the loop containing the critical section, which makes it possible to hide the latencies due to the synchronization. Our adapted C++ version combined with SIMD intrinsics (for the matching cost computation) and parallelized with OpenMP produces the same performance as the OriginalOMP version. The most interesting result is the speedup ($1.826\times$) obtained with our adapted C++ version parallelized with the OpenACC *kernels loop independent* construct and compiled with PGI compiler. Note that this last version does not contain any SIMD instructions. As seen in Chapter 3 (Subsection 3.6.1), when there are multiples nested loops, the OpenACC *parallel loop independent* is less efficient than the *kernels* variant (see an example pattern 3 in Listing 3.10). The performance speed down between the *parallel* and the *kernels* versions is $3.106\times$. This factor is consistent with those obtained in Subsection 3.6.1, which was $3.376\times$.

- *Refinement* − The scalar C++ version as well as the SIMD variant have same performance. This result can be explained by:
  - the SAD is computed for only 16 elements (using two $128-$bit register for SSE version or two static arrays for the scalar variant), and
  - the sizes of loops containing the SAD computation (lines $5-6$ of Algorithm 9) are known at compile time.

  As a result, GCC compiler vectorizes by itself the scalar version and unroll the outer loops, which leads to the achievement of performance equal to that of the SIMD version of the SAD computation. In the same way, the scalar versions parallelized with OpenACC and compiled by PGI produce also the same performance as the SIMD variant. In comparison to the author version the speedup factor obtained, with our adapted C++ variant parallelized with OpenMP and OpenACC (compiled with PGI), is $3.116\times$ on average on the Intel I7 CPU. On the other hand, this speedup factor for C++ variant parallelized with OpenMP on the Tegra X1 CPU is $2.087\times$.
- GCC compiler implementation of OpenACC is not yet mature. When targeting CPU and the used compiler is GCC, OpenMP should be used instead of OpenACC.

Figure 5.10c presents the overall performance speedup obtained on Intel I7 and NVIDIA Tegra X1 CPU for the VO algorithm presented in 5.2. Our adapted C++ version combined with SIMD intrinsics (C++SimdOMP) produces the best performance gain. The pure scalar C++ version, i.e. C++ACCKrPGI, parallelized with OpenACC *kernels loop independent* composite construct and compiled with PGI compiler produces also an interesting result. Note that in general, Intel x86 processor outperform ARM processors in term of execution time [110, 111]. We observed a speed down factor of $2.75\times$ between the Tegra X1 CPU and the Intel I7 CPU.

5.9 Scene Flow Parallelization on GPU

The parallelization of scene flow on CPU allowed us to improve the overall performance of the VO algorithm presented in 5.2 with a speedup factor of $1.57\times$ on average without significant impact on

computation result accuracy. However, parallelizing features detection, description, and matching algorithms on GPU should allow us to further improve the overall performance of our use case application. This will also allow us to see if we obtain the same results and observations as those previously obtained in Chapters 3 and 4 when OpenMP and OpenACC are used to accelerate a complex application. We have already seen in Section 5.7 all necessary algorithmic modifications in order to adapt features detection, description, and matching algorithms for parallelization. In this section, we will parallelize our adapted scalar C++ version with OpenMP, OpenACC, and CUDA for GPU execution. Note that there is no redundant same memory address accesses in the scene flow algorithm. Thus, the scalar replacement optimization can not be used here. Although, the performance improvement factor obtained by adding the *restrict* keyword is not really significant as seen in Chapter 4, we will add the *restrict* type qualifier to pointers that are alias free. We will also use the *const* type qualifier for read-only data.

5.9.1 Offloading Scene Flow on GPU with OpenMP

In order to offload the scene flow computation, we use our adapted C++OMP version. We have already seen in Section 3.5 usual optimization strategies to use for offloading an application with OpenMP/OpenACC. Since we use the C++OMP version, steps $1-4$ (finding compute-intensive parts, removing lists and deeply complex data types, removing data dependency, optimizing data accesses patterns) of the parallelization methodology proposed in Section 3.5 have already been done in Section 5.8. Passing from our C++ version parallelized with OpenMP *parallel for* construct for CPU execution to an OpenMP offloaded version does not requires many code changes.

First, we have to manage data transfer between the CPU and the GPU by using either data-mapping clauses, or OpenMP device memory routines, or CUDA API routines (as discussed in Section 3.5 $-$ Step 5). We have seen in Section 5.3 that Sobel filter results of current left and right images are used for features description, but they are also used, just like Sobel results of previous left and right images, in the refinement step. In the same way, features candidates lists of previous and current images are used in the feature matching step. Thus, in order to optimize data transfer and keep the code readable, we use OpenMP device memory routines for permanent data while *map* clauses are used for temporal data management. Figure 5.11 shows an example of the parallelization of the VO algorithm on GPU.

Second, We have to replace all *parallel for* construct by OpenMP offloading directives (as discussed in Section 3.5 $-$ Step 6). We present in following paragraphs, different implementations of scene flow offloading on GPU with OpenMP.

Basic Implementations As first implementations, we parallelize filters, features detection and description, matching, and refinement with the TgtPrFr, TgtTmDsPrFr, and TgtTmDsPrFrSch composite constructs (as we did in Section 3.6.1):

- *Filters* $-$ Outer and inner loops are both parallelized in the TgtTmDsPrFr and TgtTmDsPrFrSch versions, while only outer loop is parallelized in the TgtPrFr version.

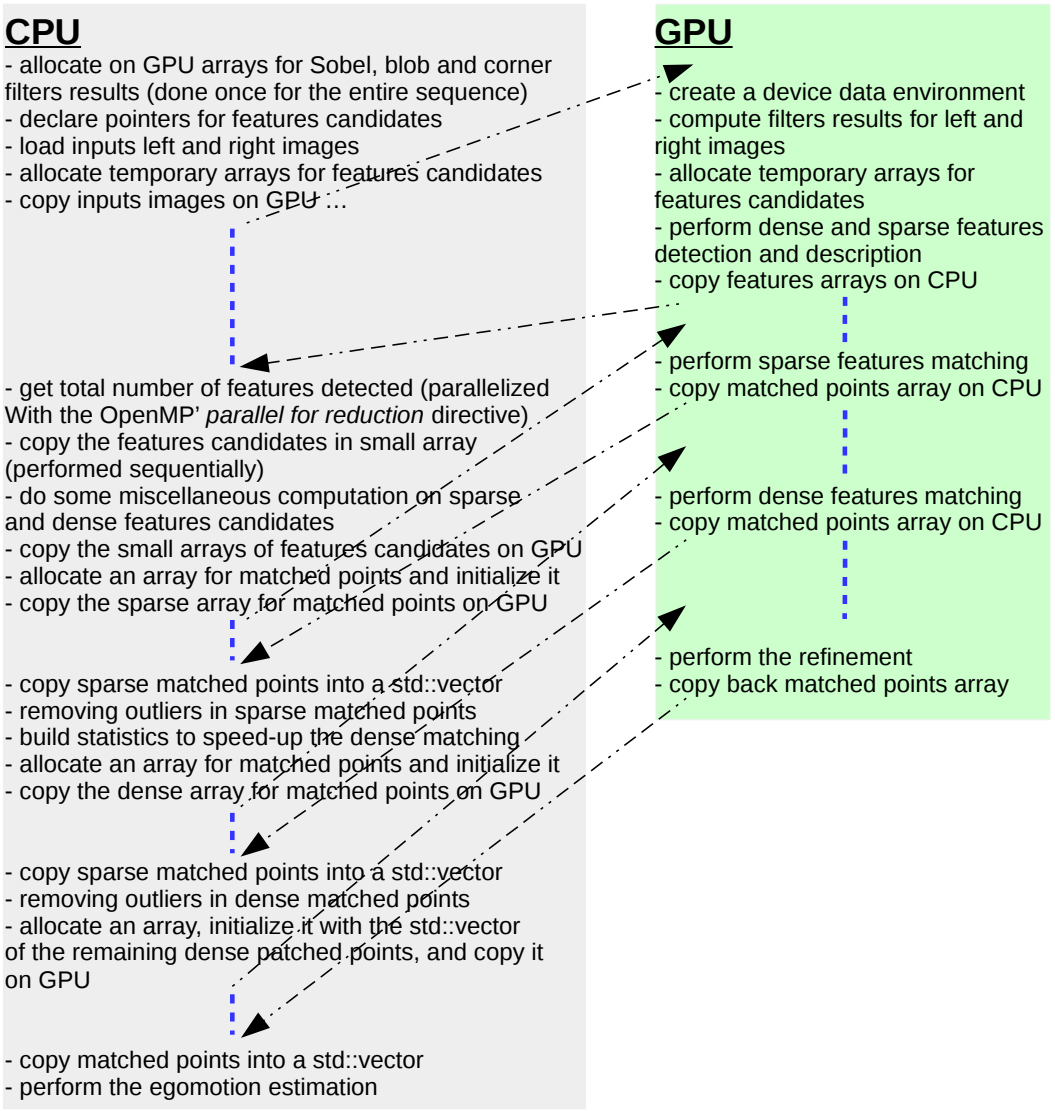


Figure 5.11 – Work flow of the parallelized version of authors CPU based implementation

- *Features detection and description:*

  – First version − Only the outer loop (line 2 of Listing 5.2) is parallelized in all versions (TgtPrFr, TgtTmDsPrFr, and TgtTmDsPrFrSch) because the LLVM/Clang compiler fails to compile the code when parallelizing the inner loop (line 3 of Listing 5.2). As in the CPU version, the other nested loops remain sequential. The computation of the total numbers of features detected remains unchanged as in the CPU version. The copy of features candidates from a large array to a smaller one (lines 18 to 25 of Listing 5.2) is also done on CPU sequentially (as explained in Section 5.8).

  – Second version − As in the first implementation, Only the outer loop (line 2 of Listing 5.3) is parallelized in all versions (TgtPrFr, TgtTmDsPrFr, and TgtTmDsPrFrSch). The

other nested loops remain sequential. In the computation of the number of features per line (lines 13 to 17 of Listing 5.3), we parallelized only the outer loop. We can parallelize the inner loop with the *parallel for reduction* construction but, as seen in the Subsection 3.6, this degrades application performance. As in the CPU version, the loop of line 21 remains sequential because the compute of addresses range of features candidate per line present a loop-independent dependency. Thus, this step is done on CPU. The copy of features candidates from a large array to a smaller one (lines 25 to 36) step is also offloaded by parallelizing the outer loop.

- *Features matching* − We use the *declare target* directive to inform the compiler that the matching cost computation routine (Algorithm 7, with the SAD implemented in scalar C++) will be called inside a *target region*. Thus, the compiler will generate a device version of this routine. In Algorithm 11, we offload only the loop of line 3 which is the main compute-intensive part.

- *Refinement* − As in the feature matching step, the *declare target* directive is used to tell LLVM/Clang that it must generate a device version for the cost computation routine (Algorithm 9). We parallelize the loop at line 1 of our adapted version of Algorithm 10.

Loop Fusion Optimization We have seen in Chapter 3 that loop collapsing combined with manual setting of grid and block sizes improves OpenMP offloaded code performance. Thus, we parallelize filters and first kernel in both first and second implementation of feature detection and description with the TgtPrFrCol_Set and TgtTmDsPrFrCol_Set combined directives (Section 3.5 − Step 8a).

Kernels Fusion Optimization As seen in Section 3.9, there is an additional cost related to the launch of OpenMP offloaded code. However, this additional cost is negligible when there is a significant workload in the kernel. Thus, we merge the Sobel, corner and blob filters kernels into one kernel (Section 3.5 − Step 8b). This may also reduce memory traffic (i.e. loads from global memory) since the three filters are applied to the same image in the same iteration, thus data are loaded once. We can also perform features detection and description for left and right images in the same loops in order to increase the workload. However, this increases also memory traffic. Thus, the performance gain in the features detection and description can be low. We will call kernels fusion optimization versions *KernFusOpt* afterward. Concerning the features matching kernel, we move the matching cost computation routine (Algorithm 7) inside the matching kernel and completely unroll the SAD loop (lines 4 − 5 of Listing 5.4).

Asynchronous Execution Optimization We know that features detection and description is done in two steps for both left and right images. Given that our adapted implementations of features detection and description contain sequential part executed on CPU, we use OpenMP *nowait* and *depend* clauses to overlap CPU and GPU executions. Note that these two clauses do not create new device execution queue, they only allow us to avoid waiting for GPU execution ends before

executing CPU part (Section 3.5 – Step 8c). Listing 5.7 a pseudo for asynchronous execution optimization applied to our firs implementation of features detection and description (Listing 5.2). The *depend* clause used at line 6, 14, and 16 ensure that the three first kernels (lines 2 to 6) have finished before starting any work on sparse features candidates arrays on the CPU. We do not need to add a *depend* clause to lines 21 and 23 since there is an implicit barrier at the end of the *single region*, this leaves enough time for the last two kernels to finish. Features matching and refinement routines also can be optimized by combining kernel splitting with asynchronous execution since the points matched are copied into a std::vector after the end computations.

```
1  int var;
2  #pragma omp target ..... nowait .....
3  {/*compute filters results for current left image*/}
4  #pragma omp target ..... nowait .....
5  {/*compute filters results for current right image*/}
6  #pragma omp target ..... nowait .....
7  {/*sparse features detection and description for left images*/}
8  #pragma omp target ..... nowait ..... depend(out: var)
9  {/*sparse features detection and description for right images*/}
10 #pragma omp target ..... nowait .....
11 {/*dense features detection and description for left images*/}
12 #pragma omp target ..... nowait .....
13 {/*dense features detection and description for right images*/}
14 #pragma omp single
15 { //compute sparse features total number and copy them into a small array
16   #pragma omp task depend(in: var)
17   {/*left images lines 19-25 of Listing 5.2*/}
18   #pragma omp task depend(in: var)
19   {/*right images lines 19-25 of Listing 5.2*/}
20 }
21 #pragma parallel sections nowait num_threads(2)
22 { //compute dense features total number and copy them into a small array
23   #pragma omp section
24   {/*.....*/}
25   #pragma omp section
26   {/*.....*/}
27 }
```

Listing 5.7 – Asynchronous execution with first version of features detection and description

5.9.2 Offloading Scene Flow on GPU with OpenACC

As we did with OpenMP, we must add data management instructions by using either data-mapping clauses, or OpenACC device memory routines, or CUDA API routines, or the *tesla:managed* flag of PGI compiler. We use the OpenACC device memory routines for the same reasons evoked above for OpenMP. Apart from adding device memory allocations and data transfer instructions, the CPU C++ACCKr/C++ACCPr versions remain unchanged. We present in the following paragraphs, different implementations of scene flow offloading on GPU with OpenACC.

Basic Implementation corresponds to the C++ACCKr (or KrnLpInd) and C++ACCPr (or ParLpInd) versions:

- *Filters* − We adopt the same approach used in OpenMP (parallelizing outer and inner loops).
- *Features detection and description* − Unlike in OpenMP, outer and inner loops of the first kernel of our two implementations (line 2 − 3 of Listing 5.2 and 5.3) are parallelized. The others nested loops remain sequential but we add to them the *loop independent seq* directive to avoid that the PGI compiler thinks there are dependencies and abstains from making certain optimizations like loop unrolling.
- *Feature matching and refinement* − As we did with OpenMP, the costs routines (Algorithms 7 and 9) are declared as device routines by using the *acc routine* construct. However, instead in OpenMP, we parallelize inner loops of these routines with the *loop independent seq* construct for the matching cost routine and the *loop independent collapse(2)* for the refinement cost function.

Loop Fusion Optimization As for OpenMP, loop collapsing combined with manual grid and block sizes setting may improve performance. Thus we can fuse outer and inner loops of filters, and those of the first kernel in both first and second implementation of feature detection and description (Section 3.5 − Step 8a). This optimization versions are called ParLpIndCol and KrnLpIndCol.

Loop Tiling Optimization Instead of merging outer and inner loops in filters and the first kernel in our two implementations of feature detection and description, we can tile them (Section 3.5 − Step 8a). The tiled versions are called ParLpIndTil and KrnLpIndTil.

Kernels Fusion Optimization Although there is not overhead in kernel launch in OpenACC, kernels fusion may improve overall performance by reducing global memory accesses. Thus, we merging filters into one kernel (Section 3.5 − Step 8b). We also merge the feature detection and description kernel using the same approach that is used with OpenMP.

Multi-Streaming Optimization Unlike OpenMP, OpenACC *async* and *wait* clauses allow multiple device execution queues creation. This is also called multi-streaming in CUDA. We optimize filters and features detection and description routines by creating multiple devices execution queues in order to overlap data transfer with concurrent kernels execution (Section 3.5 − Step 8d). Listing 5.8 shows OpenACC multi-streaming implementation of the feature detection and description example seen in Listing 5.7.

```
1 #pragma acc ............ async(1) .....
2 {/*compute filters results for current left image*/}
3 #pragma acc ............ async(2) .....
4 {/*compute filters results for current right image*/}
5 #pragma acc ............ async(1) .....
```

```
6  {/*sparse features detection and description for left images*/}
7  #pragma acc ............ async(2) .....
8  {/*sparse features detection and description for right images*/}
9  #pragma acc ............ async(1) .....
10 {/*dense features detection and description for left images*/}
11 #pragma acc ............ async(2) .....
12 {/*dense features detection and description for right images*/}
13 #pragma parallel sections nowait num_threads(2)
14 { //compute sparse and dense features total number and copy them into a
       small arrays
15   #pragma omp section
16   {
17     #pragma acc wait(1)
18     //......left images
19   }
20   #pragma omp section
21   {
22     #pragma acc wait(2)
23     //......right images
24   }
25 }
```

Listing 5.8 – OpenACC multi-streaming example

5.9.3 Parallelizing Scene Flow with CUDA

Unlike OpenMP and OpenACC where we simply add few instructions to the CPU scalar C++ version, accelerating scene flow on GPU with CUDA requires an important rewriting effort. Indeed, in addition to data transfer management, we must port filters, features detection, description, matching, and refinement routines from C++ to CUDA. We present in the following paragraphs, different implementations of scene flow offloading on GPU with CUDA.

Basic Implementation As first implementation, we port the CPU C++OMP version to CUDA. The Sobel, corner, and blob filters basic implementations are similar to those of the Sobel filter 3×3 seen in Section 1.6 - Listing 1.1. In the same way, we port our two implementations (Listing 5.2 and Listing 5.3) of features detection and description algorithms to CUDA. Since the nom-minimum nom-maximum suppression (Algorithm 3) is done only for a few subset pixels of blob and corner filters results, we must add supplementary instruction in the kernel code to determine the index of threads that must perform the computation. As in OpenMP and OpenACC, the cost functions in matching and refinement are translated to CUDA device callable kernels by using the *__device__* identifier.

First Optimization We use shared memory optimization since there are many data reuse in Sobel, corner, and blob filters. CUDA API provides a SIMD intrinsic implementation of the SAD

computation, thus we use this instruction instead the scalar version of SAD (lines $4-5$ of Listing 5.4). Since there is no data reuse in the matching main loop, we do not use the shared memory here. Unlike the features matching kernel, the Sobel filter results of the current left image are reused three times, we store the 16 descriptors of each point matched into an array allocated in the shared memory. This prevents the current thread from reloading this data from the global memory. Note, however, that there is no reuse of data between threads.

Kernels Fusion Optimization As we did with OpenMP and OpenACC, we merge the shared memory versions of filters into one kernel. In this new version, pixels of input image are loaded once into the shared memory and reused by threads in the same block to compute the three filters' results. We modify the feature detection and description implementations in order to compute results for left and right image in the same kernel. This results in an increase in the amount of computation performed but also in the number of the global memory' accesses since we process 8 inputs (filters' results of current left and right image) in the same kernel.

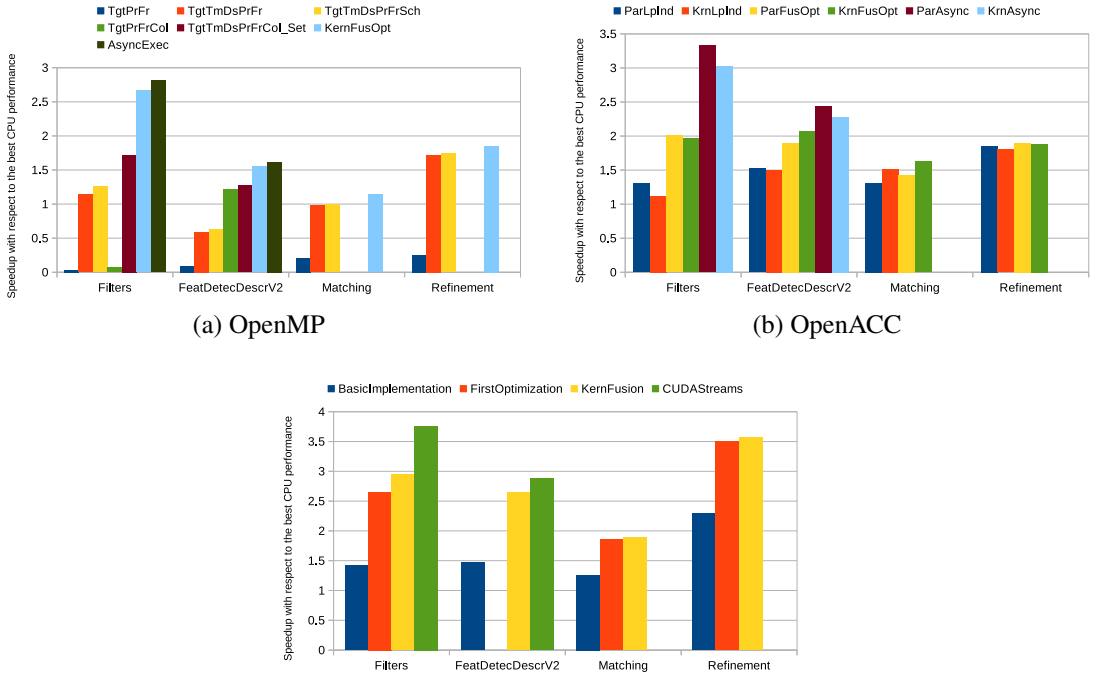
Multi-Streaming Optimization We have already mentioned that CUDA API provides routines that allows the programmer to create multiples device execution queue or streams. As has been explained in 1.6.1 - optimization 2, concurrent kernels execution may improve application performance. Thus, we optimize filters and features detection and description routines by using multi-streaming as we did with OpenACC.

5.10 Performance Results on GPU

We discussed, in the previous Section 5.9, different options to parallelize the scene flow computation on GPU with OpenMP, OpenACC, and CUDA. In this section, we present the obtained results on the Quadro M2000M GPU and the NVIDIA Tegra X1 GPU. We use compilers as well as compilation flags seen in Table 3.3. Note that, all read-only data are marked with the *const __restrict__* type qualifier, the scalar replacement has been used as much as possible, and not read-only arrays are marked with the *__restrict__* keyword. The performance, reported on the figures below, include kernels execution time, temporary GPU memory allocation and deallocation, as well as data transfer between device and host.

Figures 5.12a, 5.12b, and 5.12c present the performance results obtained on the NVIDIA Quadro M2000M GPU for OpenMP, OpenACC and CUDA respectively. On the other hand, Figures 5.13a and 5.13b present the results obtained on the NVIDIA Tegra X1 GPU. We make the following observations:

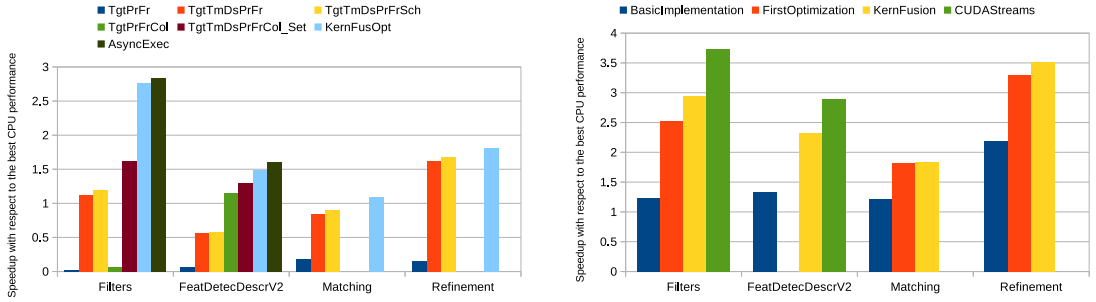
- *Filters* – As previously observed in Chapter 3 (subsection 3.6.1, Listing 3.9), the OpenACC' *parallel* and *kernels* directives have approximatively the same performance. The performance gap between OpenMP' *target parallel for* variants and its *target teams* versions is really significant.



(a) OpenMP (b) OpenACC

(c) CUDA

Figure 5.12 – Scene flow performance on the NVIDIA Quadro M2000M GPU



(a) OpenMP (b) CUDA

Figure 5.13 – Scene flow performance on the NVIDIA Tegra X1 GPU

– *OpenMP* – Loops collapsing (Section 3.5 – Step 8a) combined with the manual setting of the grid and block sizes produce a best performance than the non-collapsed version. Computing the results of the Sobel, blob, and corner filters in the same loops (Section 3.5 – Step 8b) improve significantly the performance in comparison to the collapsed and non-collapsed versions. Note that loops are collapsed and the grid and block sizes have been setted in the filters merged version. Asynchronous execution does not significantly improve the overall performance. This result is not surprising since all GPU operations are simply queuing in a single device execution queue and are executed one

by one. Typically, optimized parallelization with OpenMP requires $1.34\times$ the performance of the hand-optimized CUDA code for convolution type applications.

- *OpenACC* – Loop collapsing (Section 3.5 – Step 8a) combined with the manual setting of the grid and block sizes produces typically the same performance as the non-collapsed version for window-based operations. Thus, we do not plot the results for the collapsed and tiled versions on the Figure 5.12b. The performance gain with kernels fusion (Section 3.5 – Step 8b) is less significant than with OpenMP. On the other hand, the multi-streaming (Section 3.5 – Step 8d) significantly improves the overall performance. Optimized parallelization with OpenACC requires on average $1.13\times$ the performance of the hand-optimized CUDA code for convolution type applications.
- *CUDA* – Using shared memory improve significantly the performance of filters. As we observed with OpenACC, the performance gain with kernels fusion is less significant than with OpenMP. On the other hand, concurrent kernels execution through multiple device execution queues creation leads to an improvement of the overall performance.

- *Features detection and description* – This routine is essentially memory bound. The overall speedup obtained is not significant with OpenMP while those obtained with OpenACC and CUDA is great than $1.5\times$.
  - *OpenMP* – The TgtPrFr version does not produce any speedup, instead it significantly degrade the kernel performance. The TgtTmDsPrFr also does not produce any speedup here due to the fact that LLVM/Clang fails to compile the code when both outer and inner loops of the features detection kernel (lines $1-2$ of Listing 5.3) are parallelized. As result, only the outer loop is parallelized as we discussed in Section 5.9.1. Surprisingly, the TgtPrFrCol_Set and TgtTmDsPrFrCol_Set versions produce approximatively the same performance. Unlike the result observed with filters, kernels fusion does not significantly improve the overall performance. This result is not surprising since the kernel itself is very memory bound, thus the speedup gain can not compensate the latencies due to memory accesses.
  - *OpenACC* – As previously observed with filters, kernels fusion improve the application performance, but the performance gain is not significant for the same reasons mentioned above. The multi-streaming used with the *parallel* directive produce the best performance. However, the performance gap between the multi-streaming versions of *parallel* and *kernels* is not really significant.
  - *CUDA* – In opposite of results observed with filters, kernels fusion significantly improves the overall performance. The multi-streaming also improves the global performance. Typically, for memory-bound kernel, OpenMP requires $1.77\times$ of the CUDA version runtime while OpenACC requires $1.18\times$ on average.

- *Matching* – The OpenMP' versions do not produce any significant speedup. This result is due to the fact that the features matching routine contains several deeply nested loops. This routine has also many irregular memory access. As result, the LLVM/Clang compiler fails to generate an efficient code for this kernels. The PGI compiler also fails to generate optimized

code for this routine despite the fact that we used all optimization clauses provided by the OpenACC' standard. We obtained a performance improvement with the CUDA's version which uses the CUDA API' SIMD intrinsic for the SAD computation. For the matching routine, optimized OpenMP code requires on average $1.66\times$ simple CUDA version runtime while OpenACC requires $1.16\times$.

- *Refinement* – The performance gain with the simple CUDA version is great than optimized versions of OpenMP and OpenACC. As previously observed with the features matching kernel, using the CUDA API' SIMD intrinsic for the SAD computation improves significantly this routine performance. For the refinement routine, optimized OpenMP code requires on average $1.93\times$ simple CUDA version runtime while OpenACC requires $1.88\times$. Figure 5.14 shows the best OpenMP, OpenACC, and CUDA speedup with respect to the best CPU performance.

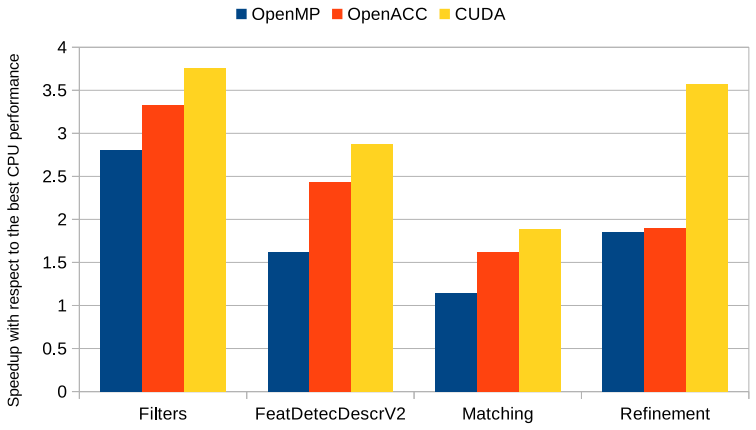


Figure 5.14 – Best OpenMP, OpenACC, and CUDA speedup with respect to the best CPU performance

5.11 Conclusion

In this chapter, we used the parallelization strategy presented in Section 3.5 to parallelize a complex application with OpenMP and OpenACC. We have also ported this application to CUDA. As previously observed in Chapter 3, OpenACC and OpenMP directives can be combined in various ways and this has a significant impact on the final performance of the offloaded code. This is specially true for OpenMP where the *target teams* variants are more efficient than the *target parallel for* versions in general.

Programmability

Code parallelization with OpenMP and OpenACC requires less development time than with CUDA since they do not require to rewrite the CPU code and they are easier to learn and use. However, these three parallelization approaches have some points in common:

- Code restructuring is necessary in order to maximize parallel execution, reduce latency, and optimize memory accesses performance.

- Manual true deep copy must be programmed by the user when complex data structures and lists are used in CPU version.

OpenACC/OpenMP interoperability with CUDA can be useful in some case. For example, the asynchronous versions of OpenMP data transfer routines are not yet available. When a program contains several consecutive data transfers to/from the GPU, memory can be allocated with OpenMP memory allocation routines while *cudaMemcpyAsync* can be used for transferring data to GPU memory. Another example is the setting to zero of all elements of an array. Currents versions of OpenACC and OpenMP do not provide a *memset* routine. Thus, the array can be allocated with OpenMP/OpenACC memory routine and the initialization can be done with *cudaMemset* instead of allocating and initializing data on CPU and then copy them to GPU.

Data management can be tricky with OpenMP and OpenACC when using copy clauses in some situations. Typically OpenMP/OpenACC data motion clauses indicate two things to the compiler: the number of bytes (generally equal to array size multiply by the size of the array in byte) to be copied/allocated, and the starting address. The compiler only needs to know the type of the pointer/array. Consider the array allocated at the line 2 in Listing 5.9. The first kernel (line 4 to 6) will fail at runtime due to an out of bound memory access since only $number\_of\_features * sizeof(int)$ bytes have been copied on GPU. The compiler does not generate any warning or error message to alert the programmer, and this kind of error is often difficult to identify, especially if the array has been allocated to another function elsewhere. The second kernel (line 7 to 9) not fails at runtime since we copying the required size.

```
1 int data_size = sizeof(features_struct) * number_of_features;
2 int *array = (int*)malloc(data_size); /*fill array on CPU*/
3 #pragma acc kernels loop copyin(array[0:number_of_features]) ...
4 for(int i = 0; i < number_of_features; i++)
5   {/*use features_candidate_array*/}
6 #pragma acc kernels loop copyin(array[0:(data_size/sizeof(int))]) ...
7 for(int i = 0; i < number_of_features; i++)
8   {/*use features_candidate_array*/}
```

Listing 5.9 – Example of common user error with OpenMP/OpenACC copy clauses

Performance

OpenMP and OpenACC programs can achieve performance comparable to that of non-complicated optimized CUDA codes. However, when the offloaded region code is computed/memory bound and contain several nested loops (e.g the stereo disparity SDisp kernel seen in Chapter 3, the features matching kernel), LLVM/Clang and PGI compiler quickly reach their limit. In such cases, CUDA must be used. We observed that OpenACC provides the best performance than OpenMP in general.

Chapter 6

Conclusion

For higher power efficiency and computation throughput purpose, all computing systems, from mobile to supercomputers, are becoming increasingly heterogeneous and massively parallel. Modern heterogeneous systems combine on the same integrated circuit different types of processors including a host CPU, one or more GPUs, and others accelerators. GPUs can achieve significant performance. However, achieving this performance requires to fully exploit the underlying architecture. In order to take advantage of GPUs computing potential, legacy sequential applications must be modified to expose as much parallelism as possible.

Parallelizing an application on GPU requires to find the best trade-off between the programming cost, the final code's portability and the speedup gain. Low-level languages, strongly related to the architecture, allow expert programmers to hand-tune their codes to exploit the full capabilities of the GPU. However, the programmer must have substantial knowledge of GPUs architecture in order to achieve good performance. On the other hand, directive-based programming models aim to provide interfaces for exploiting the power of GPUs in a more productive and portable manner. Theoretically, OpenMP and OpenACC allow programmers to provide directives to the compiler without requiring significant modification or adaptation of underlying code itself. It happens from our experiments that without enough information, the compiler is not able to do the optimizations as well as the user who is an expert in both the ported application and the architecture. Even though the compiler can apply some optimizations automatically, it may not achieve the expected speedup as the compiler does not have a full view of the whole application. Because of these reasons, there is generally a performance gap between the codes accelerated with OpenACC/OpenMP and those hand-optimized with CUDA/OpenCL. Therefore, more optimization research is required in order to achieve competitive performance with OpenMP and OpenACC. The main contribution of this work is to provide feedback that will help programmers for speeding up efficiently their legacy sequential codes on GPU with OpenMP and OpenACC. To reach this goal, several research issues have been discussed in this work. The contributions of this dissertation are summarized as follows:

1. We analyzed and compared the main features of OpenMP and OpenACC. This allows us to provide a clearer understanding of these two standards. Indeed, it is very important to understand what each directive does so that the programmer can give as many information

as possible to the compiler, especially when using OpenMP. We also saw that although being similar, OpenMP and OpenACC are quite different. However, it is possible to translate between them, but the process may not be automatic.

2. We studied the suitability of OpenMP and OpenACC for GPUs programming in comparison to features available in CUDA. We saw that basics optimizations available in CUDA (e.g. thread management, asynchronous execution, and dynamic parallelism) can be done with OpenMP/OpenACC. However, low-level threads inter-synchronization is not currently available neither in OpenACC nor in OpenMP. Although both OpenACC and OpenMP continue to evolve and offer a large set of features, some optimizations (e.g. shared, texture and constant memory accesses, SIMD intrinsics, etc.) still require the use of low-level languages.

3. We analyzed the PGI compiler implementation of OpenACC and the OpenMP 4.5 support in LLVM/Clang in order to understand how these two compilers interpret OpenACC and OpenMP features. We explored different performance improvement opportunities with OpenACC and OpenMP by parallelizing synthetic kernels, and thirty applications with OpenACC, OpenMP, and CUDA. It has been shown the directives can be combined in various ways and this has a significant impact on the final performance of the offloaded code. This is specially true for OpenMP where the *target teams* variants are more efficient than the *target parallel for* versions. It is highly recommended to combine offloading directives with loop parallelization constructs and to explore optimization techniques (e.g. loop fusion, tiling, and scheduling, kernels fusion, asynchronous execution, etc.) in order to achieve the best performance. We compared OpenACC *parallel* and *kernels* construct and identified some code patterns where the *kernels* directive must be used instead of *parallel* and vice versa. Data movement between CPU and GPU memory can be costly, it is the user's responsibility to choose the necessary data management method (copy clauses, OpenMP/OpenACC memory routines, or CUDA APIs) in order to optimize data transfer cost.

4. Caching read-only data in the L1 cache may improve GPU application performance. Since the compilers might not always be able to detect that the read-only condition is satisfied for some data because of pointers aliasing issues, we proposed two static analysis tools that perform automatically at source level type qualifier (*restrict* keyword) insertion and scalar promotion to solve aliasing issues and help compilers to put more data in L1 cache and registers. We evaluated our tools on the PolyBench benchmark suite.

5. We accelerated a visual odometry application with OpenMP, OpenACC, and CUDA. We saw that code parallelization with OpenMP and OpenACC requires less development time than with CUDA since they do not require to rewrite the CPU code and they are easier to learn and use. However, these three parallelization approaches have some points in common: code restructuring is necessary in order to maximize parallel execution, reduce latency, and optimize memory accesses performance; manual true deep copy must be programmed by the user when complex data structures and lists are used in CPU version. OpenACC/OpenMP interoperability with CUDA is useful in some cases (e.g. setting all elements of an array to a specific value and asynchronous data transfer). OpenMP and OpenACC programs can

achieve performance comparable to that of non-complicated optimized CUDA codes. However, when the offloaded region code is computed/memory bound and contain several nested loops, LLVM/Clang and PGI compiler, as well as OpenMP and OpenACC, features quickly reach their limit. In such cases, CUDA must be used. We observed that OpenACC provides the best performance than OpenMP in general.

In the future work, based on our application porting experiences, lots of research still can be done to further simplify GPU programming and improve the performance using OpenMP and OpenACC. Currently, it is the user's responsibility to specify which computation regions are offloaded to the device. We plan to extend our static analysis tools to do some analysis for all loop nests and then calculate the computational intensity for each of the loop nests, and finally decide whether it is profitable to offload those loop nests. We plan to add to our static analysis tools a supplementary module that should be able to indicate to the user with parallelization strategy should be used based on the code pattern analysis. Currently, our tools can analyze only C code, the C++ frontend for Frama-C is under development, thus in future work, we can update our tools in order to be able to analyze C++ code. Concerning the data part, the programmer's burden can be further reduced if the compiler can analyze the definition and use of all data and give some hints to the user on how the data directives should be added. In addition, since one of the goals of OpenACC and OpenMP is the portability among different types of accelerators, the current directives set does not fully utilize all hardware features of GPU. Therefore the compiler may perform some GPU specific optimizations when applying the loop transformation underneath.

Appendix A

Résumé Substanciel en Français

Dans le présent chapitre, nous fournissons une synthèse de notre travail en français. Nous commençons par introduire la problématique adressée dans ce travail ainsi que nos motivations. Nous discuterons également des objectifs de ce travail de recherche. Ensuite, nous présenterons les grandes lignes des différents travaux réalisés au cours de cette thèse. Nous présenterons également un récapitulatif des principaux résultats obtenus ainsi que quelques conclusions.

A.1 Contexte, Problématique et Motivations

Les premiers ordinateurs n'avaient pas de processeurs massivement parallèle. Ainsi pendant plusieurs années, les logiciels avaient été principalement développés pour les architectures de type $x86$ d'Intel. Cela a fonctionné pendant de nombreuses années, car les progrès technologiques permettaient d'améliorer les performances des processeurs sans modifier l'architecture de manière significative. Cependant, ces progrès avaient ralenti et l'obtention de gain de performance en fonction du nombre de transistors intégrés était devenue un objectif critique [5, 6]. Ainsi d'autres méthodes avaient été proposées pour continuer à améliorer les performances des processeurs. L'une de ces propositions était le calcul parallèle. Les architectures parallèles peuvent aller de plusieurs processeurs complexes à des milliers de processeurs plus simples fonctionnant en parallèle. De nos jours il existe plusieurs fabricants d'architectures parallèles. Ainsi, le marché regorge de toutes sortes processeurs, allant de cœurs de calcul peu nombreux mais rapides (comme c'est le cas dans les CPU) aux processeurs dotés de milliers de cœurs plutôt lents (par exemple les GPU). On retrouve également des processeurs spécialement conçus pour un objectif spécifique. Ainsi cours de la dernière décennie, il y a eu une transition continue des processeurs multicœurs vers des architectures massivement parallèles. Cette transition a été mise en évidence par l'utilisation croissante des processeurs graphiques (appelés GPU) en tant qu'accélérateurs de calcul à usage général.

En outre, les architectures matérielles sont constamment modifiées et améliorées, introduisant de nouvelles fonctionnalités, des fonctionnalités modifiées ou supprimées. NVIDIA est l'un des gros fournisseurs de GPU. Les GPU NVIDIA ont une hiérarchie de mémoire complexe avec une série

de différents caches. Au cours des cinq dernières générations de GPU NVIDIA, de nombreux changements importants ont été appliqués à cette hiérarchie de mémoire, de sorte que le code écrit pour les générations précédentes ne fonctionne généralement pas aussi efficacement que sur les générations plus récentes. Par exemple, les architectures Fermi de NVIDIA permettent aux utilisateurs d'ajuster de manière dynamique la quantité de mémoire dédiée au cache L1 pour chaque fonction (ou kernel) s'exécutant sur la GPU. L'architecture Kepler avait ajouté plus d'options pour l'ajustement de la taille de ce cache L1. Dans la génération Maxwell, cette fonctionnalité a été entièrement supprimée. Ainsi, dans trois générations consécutives des GPU NVIDIA, le comportement a été constamment modifié. Un autre exemple est celui des processeurs vectoriel dans les CPU. Des extensions vectorielles avancées (AVX) 1.0 ont été ajoutées dans les processeurs Intel I7 $- 2700K$ tandis que les processeurs I7 $- 4765T$ ont le format AVX 2.0. AVX n'étant pas compatible avec les versions antérieures, les instructions AVX 2.0 ne peuvent pas être utilisées sur des processeurs plus anciens. Les programmeurs doivent donc vérifier explicitement les capacités du processeur sur lequel leur logiciel est exécuté.

Les architectures massivement parallèles regroupant un ou plusieurs accélérateurs (par exemple les GPU, les Xeon Phi d'Intel, les DSP, les FPGA, ...) en plus du CPU sur une même puce sont de plus en plus utilisées. La situation actuelle des systèmes embarqués est marquée par deux événements importants: l'explosion du parallélisme et la montée en puissance de systèmes hétérogènes. Par exemple, dans le contexte des systèmes d'aide à la conduite, la nécessité d'un traitement complexe de la reconstitution de l'environnement amène plusieurs fabricants à proposer des circuits intégrant différents processeurs. Ceux-ci contiennent généralement entre 4 et 16 processeurs ARM, un ou plusieurs GPU, DSP, processeur vectoriel et processeur spécialisé. Par exemple, la carte intégrée NVIDIA Tegra X1 contient une GPU et 8 processeurs ARM (dont 4 Cortex A57 et 4 Cortex A53), le Soc TDA2 de Texas OMAP contient 4 processeurs ARM (dont 2 Cortex A15 et 2 Cortex M4), un DSP, une GPU IVA HD et 4 processeurs analytiques EVE. Ainsi, tous les systèmes informatiques, du mobile au super calculateur, deviennent des ordinateurs massivement parallèles et hétérogènes pour une efficacité énergétique et un débit de calcul supérieurs.

Bien que les architectures hétérogènes ont théoriquement une puissance de calcul impressionnante avec une consommation électrique raisonnable, leur programmation de manière efficace reste un problème ouvert [7]. La puissance de traitement parallèle fournie par des architectures hétérogènes s'accompagne de la nécessité d'écrire un code efficace permettant à tous les processeurs de résoudre un problème ensemble. En effet, la diversité des composants de systèmes hétérogènes et les différents niveaux de parallélisme rendent leur programmation basée sur des outils et même des paradigmes totalement différents (intrinsics SIMD, OpenMP, OpenACC, CUDA, OpenCL, etc.). Un autre défi de l'informatique hétérogène consiste à répartir la charge de travail entre les différents cœurs afin d'obtenir les meilleures performances avec la plus faible consommation d'énergie. Il n'existe pas de mécanisme de parallélisation automatique et le portage d'une application complexe sur de tels systèmes peut représenter des mois de travail.

Les GPU peuvent atteindre des performances significatives pour certaines catégories d'applications (par exemple les algorithmes de vision par ordinateur et l'algèbre linéaire dense). Néanmoins, cette performance ne repose pas seulement sur un effort important de programmation et d'analyse du code, mais également sur une bonne connaissance de l'architecture des GPUS. Afin de tirer parti

du potentiel informatique des GPUs, des modifications importantes doivent être apportées aux applications séquentielles. Les API de bas niveau existantes telles que CUDA et OpenCL obligent généralement les utilisateurs à être des programmeurs experts et à restructurer le code en grande partie. De plus, CUDA et OpenCL offrent un certain nombre de fonctionnalités d'optimisation des performances, car l'architecture est directement accessible à l'utilisateur, ce qui permet d'obtenir d'excellentes performances, mais ajoute de la complexité pour les développeurs d'applications. Les codes CUDA et OpenCL sont généralement optimisés pour une architecture spécifique. Cela conduit à un processus de développement logiciel moins productif et plus sujet aux erreurs qu'il est difficile d'adopter par le HPC et les applications en croissance rapide pour les marchés des véhicules intelligents.

Une autre approche consisterait à utiliser des modèles de programmation de haut niveau basés sur des directives, tels qu'OpenACC [8] et OpenMP[4], pour paralléliser des applications sur des accélérateurs. Ce genre de modèles de programmation permettent à l'utilisateur d'insérer les directives dans un code source Fortran ou C/C++ existant, permettant ainsi à une partie de leur code de s'exécuter sur l'accélérateur. À l'aide de directives, les programmeurs peuvent indiquer aux compilateurs d'effectuer certaines transformations et optimisations sur les régions de code annotées. L'utilisateur peut insérer des directives de manière graduelle pour paralléliser et optimiser un programme, permettant ainsi une migration productive pour les applications héritées. Le principal avantage de l'approche basée sur les directives est qu'elle offre une abstraction de programmation de haut niveau simplifiant ainsi la maintenance du code et améliorant la productivité. En outre, les modèles de programmation basés sur des directives peuvent offrir un bon compromis entre la productivité, la portabilité et la performance.

Depuis l'apparition de la première version d'OpenACC dans 2011 et de celle d'OpenMP 4.0 en juillet 2013, de nombreux travaux ont été réalisés pour les évaluer en termes de performance, de portabilité et de productivité. De nombreux travaux ont également proposé de nouvelles directives et clauses pour aborder certaines fonctionnalités qui ne sont actuellement pas proposées par OpenACC et OpenMP, mais également des outils pour aider les programmeurs à paralléliser leurs applications avec ces deux standards. Cependant, l'écart de performances entre les programmes accélérés avec OpenACC et OpenMP et ceux parallélisés avec les langages de bas-niveau comme CUDA et OpenCL indique qu'il est nécessaire de poursuivre les travaux de recherches afin de réduire cet écart.

A.2 Objectifs

OpenACC et OpenMP étant des modèles de haut niveau, les compilateurs traduisent généralement les directives d'OpenMP/OpenACC en un langage de bas niveau (par exemple l'assembleur, le PTX, CUDA, ou OpenCL). CUDA et OpenCL, deux langages de bas niveau, permettent à l'utilisateur d'appliquer l'optimisation de son choix. Mais cela nécessite que l'utilisateur comprenne parfaitement l'architecture sous-jacente afin que les optimisations appliquées puissent utiliser efficacement l'architecture. OpenACC et OpenMP nécessitent donc que le compilateur applique ces optimisations automatiquement. Toutefois, faute d'informations suffisantes, le com-

pilateur n'est pas en mesure d'effectuer les optimisations qu'un utilisateur expert (de l'application à paralléliser et de l'architecture architecture cible) pourrait le faire. Même si le compilateur peut appliquer automatiquement certaines optimisations, il peut ne pas atteindre l'accélération prévue, car il n'a pas une vue complète de l'application. Pour toutes ces raisons là, il existe généralement un écart de performance important entre les codes accélérés avec OpenACC/OpenMP et ceux optimisés manuellement avec CUDA/OpenCL. Ainsi, l'objectif principale de notre travail est d'identifier les principales causes de ces écarts de performance et d'indiquer un certain nombre d'optimisations permettant de réduire ces écarts de performance. Cela permettrait aux futurs programmeurs d'accélérer efficacement leurs applications sur GPU en utilisant les modèles de haut niveau basés sur des directives. Pour atteindre cet objectif, plusieurs travaux de recherche ont été effectués dans le cadre de cette thèse. Nous décrivons dans les sections les grandes lignes de ces travaux.

A.3 Analyse des Standards OpenMP et OpenACC

Pour aboutir à des gains de performance intéressants avec OpenMP et OpenACC, il est crucial de bien comprendre les implications de chaque directive. Ainsi, nous allons analyser dans cette section les deux standards. Nous les comparerons également pour bien comprendre les différences et similarités existant entre OpenMP et OpenACC afin d'aider les utilisateurs de ces deux standards à porter efficacement leurs programmes d'OpenMP vers OpenACC et inversement. Nous étudierons également la question de leur adaptabilité pour la programmation des GPUs.

OpenMP

OpenMP est sans aucun doute le standard le plus utilisé depuis plusieurs années pour la programmation parallèle des processeurs à mémoire partagée. OpenMP 4.0[43] a étendu le modèle de programmation de mémoire partagée OpenMP pour prendre en charge les accélérateurs (GPU, Intel Xeon Phi, ...) en introduisant de nouvelles directives permettant de balancer une tache sur un accélérateur. Depuis la publication d'OpenMP 4.0 en juillet 2013, OpenMP n'a cessé d'évoluer. La version actuelle est la 5.0[4] qui a été publiée en novembre 2018.

Dans le modèle d'exécution d'OpenMP pour les accélérateurs, le programme principal s'exécute sur l'hôte CPU et ce dernier peut déléguer l'exécution de certaines parties à l'accélérateur. En règle générale, chaque périphérique possède ses propres fils d'exécution (ou threads), qui sont distincts des threads hôtes ou de ceux des autres accélérateurs. Concrètement, lorsque le thread local du CPU rencontre une directive *target*, il crée une nouvelle tâche qui sera exécutée sur l'accélérateur cible. Cette nouvelle tâche englobe le bloc code qui vient juste après la directive. Ensuite, le thread local du CPU alloue de la mémoire sur l'accélérateur cible, initie le transfert des données, transfère le code de la générés à l'accélérateur, passe les arguments de la tâche générée (ou kernel), place le kernel dans la file d'attente de l'accélérateur cible, puis attends que ce dernier ait fini d'exécuter la tâche, puis rapatrie les résultats sur la CPU et libère la mémoire allouée. Le programmeur peut donner plus de détails (comme l'identifiant de l'accélérateur cible, comment est-ce que

les variables scalaires doivent être copiées sur l'accélérateur, les tableaux à copier sur ou depuis l'accélérateur, s'il faut bloquer ou non le thread local CPU jusqu'à la fin de l'exécution de la tâche sur l'accélérateur, etc.) au compilateur au travers des clauses que la directive *target* accepte.

La directive *target* implique la création d'un seul thread sur l'accélérateur et cela n'a évidemment pas de sens étant donné que l'accélérateur à des centaines de processeurs disponibles. Ainsi, OpenMP fournit d'autres directives que le programmeur peut utiliser pour indiquer au compilateur de créer plusieurs threads et comment repartir le travail entre ces threads. Les principales directives sont *teams*, *distribute*, et le traditionnel *parallel for*. La directive *teams* indique au compilateur de créer un certain nombre de groupe de threads. La directive *distribute* informe le compilateur qu'il doit repartir de façon égale les iterations de la boucle, associée à cette direction, entre les groupes de threads qu'il a créé à partir de la directive *teams*. Enfin la directive *parallel for* indique que tous les threads de tous les groupes doivent participer à l'exécution de la tâche générée.

Le modèle de mémoire adoptée par OpenMP pour la programmation des accélérateurs suppose que l'hôte CPU et l'accélérateur peuvent avoir des mémoires distinctes. Par conséquent, le transfert de données entre les mémoires de l'hôte CPU et de l'accélérateur peut être explicitement géré. Ainsi, lorsque le thread local CPU rencontre la directive *target* il crée un environnement mémoire qui couvre le bloc de code suivant cette directive. Lorsqu'une tâche est générée pour un périphérique cible, les références à la variable d'origine de l'hôte font référence à la variable correspondante dans l'environnement de données du périphérique. Dans ce contexte, si une variable d'origine n'est pas actuellement mappée et qu'aucune variable correspondante n'existe dans l'environnement de données du périphérique, l'accès à la variable d'origine peut entraîner une erreur d'accès mémoire et entraîner le crash du programme. OpenMP fournit un certain nombre de directives, clauses et appels de fonctions pour la copie des données vers ou depuis la mémoire de l'accélérateur cible.

OpenACC

OpenACC[8] est lui aussi un modèle de programmation parallèle basé sur des directives. Contrairement à OpenMP, OpenACC a été spécialement conçu dès la base pour la programmation des accélérateurs. Sa première version 1.0 est sortie en 2011. La version actuelle est le 2.7. OpenACC utilise le même modèle d'exécution et de mémoire qu'OpenMP. Contrairement à OpenMP où seule la directive *target* permet le balancement d'une tâche sur l'accélérateur, OpenACC fournit trois directives permettant de générer des tâches pour le périphérique cible.

La première directive qu'OpenACC fournit pour le balancement d'une tâche sur l'accélérateur est *parallel*. Avec cette directive, le programmeur peut prescrire explicitement les optimisations à faire lors de la génération de la tâche à balancer sur l'accélérateur. Par contre avec la directive *kernels*, le compilateur est seul maître du choix de la meilleure stratégie d'optimisation pour le bloc de code auquel s'applique la directive. Enfin, avec la troisième directive *serial*, le programmeur indique au compilateur de générer une taĉhe qui sera exécutée par un seul thread sur l'accélérateur. Tout comme les directives d'OpenMP, *kernels*, *parellel* et *serial* acceptent un certains nombres de clauses que le programmeur peut utiliser pour fournir davantage d'information au compilateur.

En plus des directives *kernels*, *parellel* et *serial* qui permettent d'indiquer au compilateur les parties

qui doivent être exécutées sur un accélérateur, OpenACC fournit également la directive *loop* qui permet de fournir au compilateur des informations sur les boucles imbriquées dans la tâche cible et l'aider dans leur parallélisation.

OpenACC fournit lui aussi de nombreuses directives, clauses et appels de fonctions pour copier des données dans ou depuis la mémoire de l'accélérateur cible.

Comparaison entre OpenMP et OpenACC

OpenMP et OpenACC, bien que similaires, sont encore très différents dans leur approche. L'une des différences les plus significatives entre OpenMP et OpenACC est leur philosophie. OpenMP est prescriptive car l'API OpenMP couvre uniquement la parallélisation dirigée par l'utilisateur et non la parallélisation automatique entreprise par le compilateur. Dans OpenMP, le programmeur doit spécifier explicitement quelles régions de code doivent être parallélisées et comment cela doit être effectué. La même chose doit également être faite pour les boucles imbriquées dans une région parallèle. Considérons le fragment de code présenté dans Listing A.1. Si nous utilisons uniquement *target* (à la ligne 1), la boucle sera exécutée par un seul thread sur l'accélérateur. En ajoutant *teams*, un certain nombre de groupe de threads sera créé, mais seul le thread principal de l'accélérateur exécutera la boucle. En insérant *distribution*, les itérations de la boucle seront réparties entre les différents groupes de threads créés, mais seul le thread chef de chaque groupe exécutera les itérations assignées à son groupe. Les autres threads d'un groupe ne participeront à l'éxécution que si nous ajoutons la directive *parallel for*.

```
1 #pragma omp target teams distribute parallel for is_device_ptr(...)
2 #pragma acc kernels loop independent deviceptr(...)
3 for(int x = 1; x < h - 1; x++) {/*........*/}
```

Listing A.1 – OpenMP and OpenACC philosophical difference

Par contre, OpenACC est descriptive. En effet, le modèle de programmation OpenACC est un parallélisme guidé par l'utilisateur, c'est-à-dire qu'il permet au programmeur de donner le plus d'information possible au compilateur pour l'aider à choisir la stratégie de parallélisation adéquate. L'idée de base du modèle de programmation OpenACC est qu'il est préférable que l'utilisateur décrive le parallélisme et le mouvement des données de manière plus générale via des directives afin que le compilateur OpenACC puisse avoir plus de liberté pour mapper le parallélisme sur le matériel. Par conséquent, le compilateur OpenACC a davantage de responsabilités en termes de performances et la qualité du compilateur peut grandement affecter les performances des applications. Avec OpenACC, pour que la boucle de Listing A.1 soit complètement parallélisée, il suffit d'ajouter le *loop independent* à la directive *kernels*.

OpenMP et OpenACC permettent à l'utilisateur de spécifier l'accélérateur cible pour différentes régions de code dans la même application et au sein d'une même unité de compilation. Cela peut être fait en utilisant la clause *if* pour les deux, la clause *device* pour OpenMP et la clause *device_type* pour OpenACC. C'est une caractéristique importante de ces deux standards car elle

permet à l'utilisateur d'avoir une source unique pour l'hôte et les accélérateurs et d'assurer la portabilité.

OpenACC fournit la directive textit cache pour spécifier des éléments de tableau ou des sous-tableaux à charger dans le cache le plus proche des coeurs de calcul pour le corps d'une boucle, tandis que la clause/directive *allocate* d'OpenMP 5.0 devrait permettre d'accomplir cette même fontionalité. Contrairement à OpenMP, la version actuelle d'OpenACC ne fournit aucune clause ou directive permettant d'accéder à d'autres espaces mémoire (mémoire constante des GPU, par exemple).

Il existe également de nombreuses différences techniques entre les deux spécifications, la première étant la directive *parallel* qui lance une exécution parallèle sur un accélérateur dans OpenACC, tandis que dans OpenMP, il crée une équipe de threads qui exécutent une region de code sur l'hôte sauf si elle est appelé à l'intérieur d'une directive *target*. La directive *teams* d'OpenMP est équivalente à la construction *kernels* d'OpenACC uniquement s'il existe une boucle unique imbriquée dans la *kernels region*. OpenMP 5.0 a introduit la directive *loop*, mais il faudra attendre les implémentations d'OpenMP 5.0 pour voir si les deux directives *loop* sont équivalentes.

A.3.1 Adaptabilité d'OpenMP et OpenACC pour la programmation des GPUs

Correspondance avec l'hiérarchie des threads en CUDA En CUDA les threads sont d'abord regroupés par groupe de 32, appelé *warp*. Les *warp* forment des blocs et les bocs des grilles. En OpenACC aussi il y a trois niveaux dans l'hiérarchie des threads à savoir *gang*, *vector* et *worker*. Ainsi, en OpenACC, *gang* correspond à la grille en CUDA, *vector* au bloc et *worker* au warp. Par contre en OpenMP, il n'y a que deux niveaux dans l'hiérarchie des threads à savoir *num_teams* et *thread_limit*. Ainsi en OpenMP *num_teams* correspond à la grille en CUDA et *thread_limit* au block. Tout comme en CUDA, les *gang*, *vector* et *worker* en OpenACC peuvent être configurés en $1D$, $2D$ ou $3D$. Mais cela n'est pas possible avec OpenMP où *num_teams* et *thread_limit* ne peuvent être configurés qu'en $1D$.

Synchronisation OpenMP dispose d'un ensemble de directives pour la synchronisation des threads sur l'hôte CPU, telles que *critical*, *barrier*, *taskwait*, *taskgroup*, *flush*, and *ordered*. La clause *depend* d'OpenMP permet la synchronisation entre différentes kernels, mais la synchronisation entre les threads au sein d'un groupe de threads ou entre différents groupes n'est pas disponible. La création de multiples files d'attente d'exécution de l'accélérateur n'est pas non plus disponible dans les versions actuelles d'OpenMP. OpenACC non plus ne fournit pas de mécanisme de synchronisation intra-groupe. Cependant, la synchronisation entre les groupes peut être possible, car il existe une barrière implicite à la fin d'une région *kernels* ou *parallel*. La clause *async* d'OpenACC permet la création de multiples files d'attente d'exécution de périphériques tandis que la clause *wait* permet de synchroniser différentes régions *kernels* ou *parallel*. Concrètement, ni OpenMP ni OpenACC ne fournissent à ce jour de directives, ou clauses ou appels de fonctions correspondant aux fonctions de synchronisations (ex: __syncthreads(), __syncwarp) disponibles

dans CUDA.

Exécution asynchrone entre l'hôte CPU et la GPU Afin d'empêcher le thread d'attendre la fin d'une opération GPU (transfert de données ou exécution d'un kernel) avant d'exécuter les instructions suivantes, CUDA fournit des versions asynchrones de ces fonctions de transfert de données. OpenACC fournit également des versions asynchrones de ses routines de copie de données, contrairement à OpenMP. De plus, les clauses OpenMP *nowait* et OpenACC *async* peuvent être utilisées pour éviter que le thread de l'hôte n'attende l'achèvement d'une tâche GPU. Si une autre partie du code nécessite les résultats d'un code déchargé avec les clauses asynchrones, la clause *depend* (en OpenMP) et la directive/clause *wait* (en OpenACC) peuvent être utilisées à des fins de synchronisation.

Création de plusieurs file d'attente d'exécution sur la GPU Également appelé multi-streaming en CUDA, la clause *async* d'OpenACC permettent de créer plusieurs files d'attentes d'exécution et la directive/clause *wait* permet de les synchroniser. Par contre en OpenMP, le multi-streming n'est pas possible avec les versions actuelles.

A.4 Méthodologie de parallélisation avec OpenMP/OpenACC

Nous avons proposé une approche de parallélisation avec OpenMP et OpenACC pour aider les programmeurs à tirer profit de ces deux standards. Nous avons appliqué la méthodologie proposée sur 30 applications et kernels pris dand les suites de benchmark Rodinia et PolyBench afin d'évaluer les performances d'OpenACC et OpenMP en comparaison de celles de CUDA. Les versions OpenACC ont été compilées avec le compilateur de PGI tandis que les versions OpenMP ont été compilées avec le compilateur LLVM/Clang. Quand aux versions CUDA, nous avons utlisé le CUDA toolkit version 9.1. Nous avons effectué nos évaluations de performance sur deux cibles: un PC portable ayant une CPU Intel I7 et une GPU NVIDIA Quadro M2000M; un carte embarqué NVIDIA Tegra X1 ayant 8 processeurs ARM regroupé par lot de 4 et une GPU NIVDIA de génération Maxwell. Nous avons essayé plusieurs combinaisons des directives d'OpenACC et OpenMP afin de voir quel impact la manière d'utiliser les directive peut avoir sur les performances d'une application.

Suite aux résultats obtenus nous avons fait les constats suivant:

- Avec OpenMP ou OpenACC, les directives peuvent être combinées de plusieurs manières et cela a un impact significatif sur les performance du code exécuté sur la GPU.
- C'est insuffisant d'insérer simplement les *kernels* or *parallel* or *target* pour indiquer au compilateur d'une zone de code sera exécuté sur la GPU.
- Bien que les compilateurs actuels sont efficace, il est crucial que l'utilisateur leurs fournisse le plus d'information possible afin de les aider à paralléliser le code au mieux. Les directives

de balancement sur GPU (ex: *kernels*, *parallel*, *serial*, and *target*) doivent être combinées avec les directives relatives à la parallélisation des boucles (ex: *loop*, *teams distribute*, *parallel for*).

- L'exploration de plusieurs stratégies d'optimisation (ex: privatisation des variables, indiquer l'absence de dépendence de données en OpenACC, fusion des boucles, fusion des kernels, division des boucles en plus petites, exécution asynchrone, indiquer le nombre de threads à créer, ...) permet d'aboutir à de meilleures performance avec OpenMP et OpenACC.
- Il y a généralement un surcoût lié au lancement des tâches sur la GPU avec OpenMP, mais pour la plupart des applications, ce n'est pas un gros problème si le kernel a une charge de travail suffisante.
- Les clauses de copies de même que les appels de fonctions de transfer de données en OpenMP et OpenACC consomment moins de temps que les fonctions CUDA. Ainsi, l'utilisateur doit utiliser de préférence les clauses de copies ou à défaut les fonctions de gestion de mémoire fournies par OpenMP et OpenACC.
- Un code manuellement bien optimisé en CUDA reste en général plus performant que les versions OpenMP et OpenACC. Celà est spécialement vrai pour les tâches ayant une forte intensité arithmétique et contenant plusieurs boucles profondement imbriquées. Dans ce genre de cas les compilateurs d'OpenMP et OpenACC ne réussit qu'à paralléliser les boucles les plus externes.

A.5 Propostion de deux outils d'analyse statique pour optimiser l'utilisation des caches

Les GPUS ont plusieurs types de mémoires (ex: mémoire globale, cache L1, cache L2, mémoire partagée, les registres, ...). Sur les GPU Maxwell, les accès à la mémoire locale ou globale, y compris les débordements de registre temporaires sont mis en cache L2. Les accès mémoire globaux sont toujours mis en cache dans la L2. Toutefois, les données en lecture seule pour toute la durée de vie du kernel peuvent également être mises en cache dans le cache unifié L1/texture. La mise en cache des données en lecture seule dans la L1 peut améliorer les performances des applications. Néanmoins, les compilateurs ne peuvent pas toujours détecter que la condition de lecture seule est satisfaite pour certaines données en raison du problème d'aliasing des pointeurs. Bien que l'utilisation de L1 et de L2 soit contrôlée par le matériel et que le programmeur ne puisse pas la gérer, il peut marquer les pointeurs utilisés pour charger des données en lecture seule avec les qualificatifs *const* et *__restrict__*. Nous avons ainsi développé deux outils d'analyse statique, l'un permettant l'insertion automatique du mot clé *__restrict__* et l'autre effectuant le remplacement par scalaire. Nous avons évaluer les performances des deux outils sur le benchmark PolyBench. Les résultats indiquent:

-

A.6 Application d'OpenMP et OpenACC sur une application complexe d'odométrie visuelle

Les suites de benchmarks ne sont en général pas aussi complexes que les applications réelles. Ainsi nous avons appliqué la méthodologie de parallélisation proposée à une application réelle d'odométrie visuelle. La connaissance de l'état de mouvement d'un véhicule est essentielle pour évaluer le risque de collision dans les systèmes avancés d'aide à la conduite (ADAS) ou la conduite autonome. L'odométrie visuelle (VO) est l'une des techniques robustes utilisées pour la localisation de véhicules [98, 99]. La VO consiste à estimer la position d'un agent (par exemple, un véhicule, un humain ou un robot) en utilisant uniquement un flux d'images acquis à partir d'une ou de plusieurs caméras embarquées sur cet agent [100]. En effet, les images contiennent une quantité suffisante d'informations significatives (couleur, texture, forme, etc.) pour permettre l'estimation du mouvement d'une caméra dans un environnement statique.

L'algorithme d'odométrie visuelle [102] que nous avons choisi de paralléliser comporte plusieurs regions de codes à forte intensité arithmétique. Cet algorithme est considéré comme l'un des algorithmes VO les plus efficaces et les plus robustes, largement cité dans la littérature. En effet, le nombre de citations de cet algorithme est estimé à 745 sur Google Scholar et à 345 sur IEEE Xplore au moment de la rédaction de cet document. Le pipeline de cet algorithme comprend quatre étapes: le calcul du flux de scène, l'estimation de mouvement, la mise en correspondance stéréo et la reconstruction 3D. Nous avons profilé cet algorithme et découvert que les parties qui prennent le plus de temps sont le calcul du flux de scène et la mise en correspondance stéréo. Par conséquent, ce sont ces deux étapes que nous avons parallélisé.

A.7 Conclusions et perspectives

Le portage d'une application sur une GPU nécessite de trouver le meilleur compromis entre les coûts de programmation, la portabilité du code final et le gain de performance. Les langages de bas niveau, étroitement liés à l'architecture, permettent aux programmeurs experts d'ajuster leurs codes à la main pour exploiter toutes les capacités du GPU. Cependant, le programmeur doit avoir une connaissance approfondie de l'architecture des GPU pour obtenir de bonnes performances. D'autre part, les modèles de programmation basés sur des directives visent à fournir des interfaces permettant d'exploiter la puissance des GPU de manière plus productive et plus portable. Théoriquement, OpenMP et OpenACC permettent aux programmeurs de fournir des directives au compilateur sans nécessiter de modification ou d'adaptation importante du code sous-jacent. Il résulte de nos expériences que sans informations suffisantes, le compilateur n'est pas en mesure de faire les optimisations aussi bien que l'utilisateur qui est un expert à la fois de l'application portée et de l'architecture cible. Même si le compilateur peut appliquer automatiquement certaines optimisations, il peut ne pas atteindre l'accélération prévue, car il n'a pas une vue complète de l'application. Pour ces raisons, il existe généralement un écart de performance entre les codes accélérés avec OpenACC / OpenMP et ceux optimisés manuellement avec CUDA / OpenCL. Par

conséquent, davantage de recherches d'optimisation sont nécessaires pour atteindre des performances compétitives avec OpenMP et OpenACC. La principale contribution de ce travail est de fournir un retour d'information qui aidera les programmeurs à accélérer efficacement leurs codes sur GPU avec OpenMP et OpenACC. Pour atteindre cet objectif, plusieurs questions de recherche ont été abordées dans ce travail. Les contributions de cette thèse sont résumées comme suit:

- Nous avons analysé et comparé les principales fonctionnalités d'OpenMP et d'OpenACC. Cela nous a permis de montrer que la compréhension des actions de chaque directive est nécessaire pour donner le plus d'informations possible au compilateur, en particulier lorsqu'il utilise OpenMP. Nous avons montré que, bien qu'étant similaires, OpenMP et OpenACC sont assez différents et que le passage d'un code OpenACC à un code OpenMP nécessite plus qu'un simple changement des directives.

- Nous avons étudié l'adéquation d'OpenMP et d'OpenACC pour la programmation de GPU par rapport aux fonctionnalités disponibles dans CUDA. Nous avons vu que les optimisations de base disponibles dans CUDA (ex: la gestion des threads, l'exécution asynchrone et le parallélisme dynamique) peuvent être effectuées avec OpenMP/OpenACC. Toutefois, la synchronisation des threads de bas niveau n'est actuellement disponible ni dans OpenACC ni dans OpenMP. Bien que OpenACC et OpenMP continuent d'évoluer et offrent un grand nombre de fonctionnalités, certaines optimisations (accès partagés, accès à la mémoire de texture et à la mémoire constante, instructions SIMD, etc.) nécessitent toujours l'utilisation de langages de bas niveau.

- Nous avons exploré différentes possibilités d'amélioration des performances avec OpenACC et OpenMP en parallélisant des kernels synthétiques et trente applications avec OpenACC, OpenMP et CUDA. Il a été démontré que les directives peuvent être combinées de différentes manières, ce qui a un impact significatif sur les performances finales du code déchargé. Ceci est spécialement vrai pour OpenMP où les variantes de *target temas* sont plus efficaces que les versions de *target parallel for*. Il est vivement recommandé de combiner les directives de déchargement avec les directives de parallélisation de boucle et d'explorer des techniques d'optimisation (ex: fusion de boucle, division de boucle, fusion de kernels, exécution asynchrone, etc.) afin d'obtenir les meilleures performances. Nous avons comparé les diretives *parallel* et *kernels* d'OpenACC et identifié certains modèles de code dans lesquels la directive *kernels* doit être privilégiée et inversement. Le transfert de données entre la CPU et la mémoire du GPU peut être coûteux, il appartient à l'utilisateur de choisir la méthode de gestion de données nécessaire (clauses de copie, routines de mémoire OpenMP / OpenACC ou API CUDA) afin d'optimiser les coûts de transfert de données.

- Charger les données en lecture seule dans le cache L1 peut améliorer les performances d'une application GPU. Étant donné que les compilateurs ne sont pas toujours en mesure de détecter qu'un tableau est en lecture seule en raison du problème de l'aliasing des pointeurs, nous avons proposé deux outils d'analyse statiques qui permettent d'insérer automatiquement le mot clé *restrict* et le remplacement par scalaire pour résoudre les problèmes d'alias et aider les compilateurs à mettre plus de données dans le cache L1 et les registres.

- Nous avons accéléré une application visuelle d'odométrie avec OpenMP, OpenACC et CUDA. Nous avons vu que la parallélisation du code avec OpenMP et OpenACC nécessite moins de temps de développement qu'avec CUDA, car ils ne nécessitent pas de réécrire le code séquentiel et sont plus faciles à apprendre et à utiliser. Cependant, ces trois approches de parallélisation ont des points communs: la restructuration du code est nécessaire pour maximiser l'exécution en parallèle, réduire le temps de latence et optimiser les performances des accès mémoire; la copie profonde manuelle doit être programmée par l'utilisateur lorsque des structures de données complexes et des listes sont utilisées dans la version séquentielle. L'interopérabilité OpenACC/OpenMP avec CUDA est utile dans certains cas (par exemple lors de la mise à zéro de tous les éléments d'un tableau). Les programmes parallélisés avec OpenMP et OpenACC peuvent atteindre des performances comparables à celles des codes CUDA moyennement optimisés. Cependant, lorsque le bloc de code à parallélisé à une forte intensité de calcul, beaucoup d'accès mémoire et contient plusieurs boucles imbriquées, le compilateur LLVM/Clang et PGI, ainsi que les fonctionnalités OpenMP et OpenACC, atteignent rapidement leur limite. Dans de tels cas, CUDA doit être utilisé. Nous avons observé que OpenACC fournit les meilleures performances que OpenMP en général.

Dans les travaux futurs, sur la base de nos expériences de portage d'applications, il reste encore beaucoup à faire pour simplifier davantage la programmation GPU et améliorer les performances avec OpenMP et OpenACC. Actuellement, il incombe à l'utilisateur de spécifier les régions de code qui doivent être parallélisées sur GPU. Nous prévoyons d'étendre nos outils d'analyse statiques afin d'effectuer une analyse de toutes les boucles, puis de calculer l'intensité arithmétique pour chaque boucle afin d'indiquer au programmeur s'il est rentable de balancer ces boucles que la GPU. Nous prévoyons d'ajouter à nos outils d'analyse statique un module supplémentaire qui devrait pouvoir indiquer à l'utilisateur quelle stratégie de parallélisation devrait être utilisé en fonction de l'analyse des patterns de code. Actuellement, nos outils ne peuvent analyser que du code C, l'interface C++ de Frama-C étant en cours de développement, nous pourrons ainsi mettre à jour nos outils afin de pouvoir analyser le code C++.

Bibliography

[1] NVIDIA. Whitepaper - nvidia's next generation cuda compute architecture: Fermi. https://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, 2009. Accessed: 2019-01-20.

[2] NVIDIA. Whitepaper - nvidia tegra x1: Nvidia's new mobile superchip. https://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf, 2015. Accessed: 2019-01-21.

[3] NVIDIA. Cuda c programming guide. https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf, oct 2018. Accessed: 2019-02-01.

[4] OpenMP Architecture Review Board. Openmp application programming interface. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, nov 2018. Accessed: 2019-02-06.

[5] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report, UC Berkeley, 2006.

[6] National Research Council, on Engineering, Division Physical Sciences, Science, Computer Telecommunications Board, Committee on Sustaining Growth in Computing Performance, S.H. Fuller, and L.I. Millett. *The future of computing performance: Game over or next level?* 04 2011.

[7] Sparsh Mittal and Jeffrey S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, July 2015.

[8] OpenACC-Standard. The openacc application programming interface. https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf, nov 2018. Accessed 2019-02-20.

[9] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd applica-

tion. *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, jun 2013.

[10] Xuechao Li and Po-Chou Shih. Performance comparison of cuda and openacc based on optimizations. In *Proceedings of the 2018 2Nd High Performance Computing and Cluster Technologies Conference*, HPCCT 2018, pages 53–57, New York, NY, USA, 2018. ACM.

[11] M. Graham Lopez, Véronica Vergara Larrea, Wayne Joubert, Oscar Hernandez, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Towards achieving performance portability using directives for accelerators. *Third Workshop on Accelerator Programming Using Directives*, nov 2016.

[12] Rokiatou Diarra, Alain Merigot, and Bastien Vincke. Exploring performance improvement opportunities in directive-based gpu programming. In *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 82–87. IEEE, 10 2018.

[13] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[14] *Cramming more components onto integrated circuits*, volume 38, apr 1965.

[15] *Moore's law governs the silicon revolution*, volume 86. IEEE, jan 1998.

[16] Kunle Olukotun. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Morgan and Claypool Publishers, 1st edition, 2007.

[17] Intel xeon phi coprocessor: Developer's quick start guide. https://software.intel.com/sites/default/files/managed/ee/4e/intel-xeon-phi-coprocessor-quick-start-developers-guide.pdf. Accessed: 2019-01-14.

[18] Andreas Olofsson. Epiphany-v: A 1024 processor 64-bit RISC system-on-chip. *CoRR*, abs/1610.01832, 2016.

[19] Norm Jouppi. Google supercharges machine learning tasks with tpu custom chip. https://cloud.google.com/blog/products/gcp/google-supercharges-machine-learning-tasks-with-custom-chip, may 2016. Accessed 2019-03-8.

[20] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[21] Vidal Rafael, Marcos ArocaLuiz, and Gonçalves Garcia. Towards green data centers: A comparison of x86 and arm architectures power efficiency. In *Journal of Parallel and Distributed Computing*, pages 1770–1780. ELSEVIER, dec 2012.

[22] Sparsh Mittal. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Comput. Surv.*, 48(3):45:1–45:38, February 2016.

[23] Ashish Venkat and Dean M. Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. *SIGARCH Comput. Archit. News*, 42(3):121–132, June 2014.

[24] NVIDIA. Nvidia tesla v100 gpu architecture: The world's most advanced data center gpu. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017. Accessed: 2019-01-21.

[25] Khronos OpenCL Working Group. The opencl specification. https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf, oct 2018. Accessed: 2019-02-01.

[26] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP'08 Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, 2008.

[27] Copyright Khronos Group 2019. The opencl specification. https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf, feb 2019.

[28] Ronan Keryell, Maria Rovatsou, and Lee Howes. Sycl specification: Sycl integrates opencl devices with modern c++. https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf, 2018.

[29] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp: A hybrid multi-core parallel programming environment. *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 01 2007. Accessed: 2019/02/01.

[30] CAPS. Caps openacc compiler: Hmpp workbench 3.2. https://www.olcf.ornl.gov/wp-content/uploads/2012/10/HMPPOpenACC-3.2_ReferenceManual.pdf, 2012. Accessed: 2019-02-01.

[31] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. *Innovative Parallel Computing (InPar)*, may 2012.

[32] Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jimenez-Gonzalez, Carlos Alvarez, and Xavier Martorell. Exploiting parallelism on gpus and fpgas with ompss. In *Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy Efficient HPC Systems*, ANDARE '17, pages 4:1–4:5, New York, NY, USA, 2017. ACM.

[33] Mehdi AMINI. *Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators*. PhD thesis, École nationale supérieure des mines de Paris, 2012. Accessed: 2016-11-02.

[34] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[35] Dmitry Mikushin, Nikolay Likhogrud, Eddy Z. Zhang, and C Bergström. Kernelgen – the design and implementation of a next generation compiler platform for accelerating numerical models on gpus. *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 1011–1020, 11 2014.

[36] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Aurangzeb, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel P. Midkiff. The cetus source-to-source compiler infrastructure: Overview and evaluation. *Int. J. Parallel Program.*, 41(6):753–767, December 2013.

[37] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: A compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44(4):101–110, February 2009.

[38] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. Dawncc: Automatic annotation for data parallelism and offloading. *ACM Trans. Archit. Code Optim.*, 14(2):13:1–13:25, May 2017.

[39] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, volume 48, pages 519–530. ACM SIGPLAN, jun 2013.

[40] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. Hipacc: A domain-specific language and compiler for image processing. *In Transactions on Parallel and Distributed Systems (TPDS)*, 27(1):210–224, 2016.

[41] The Khronos Vulkan Working Group. Vulkan 1.1.101 - a specification. https://www.khronos.org/registry/vulkan/specs/1.1/pdf/vkspec.pdf. Accessed: 2019-02-24.

[42] Pierre Estérie, Joel Falcou, Mathias Gaunard, Jean-Thierry Lapresté, and Lionel Lacassagne. The numerical template toolbox: A modern c++ design for scientific computing. *Journal of Parallel and Distributed Computing*, 74, 07 2014.

[43] OpenMP Architecture Review Board. Openmp application programming interface. https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf, jul 2013. Accessed: 2019-02-06.

[44] Ruyman Reyes, Iván López-Rodr'iguez, Juan Fumero, and Francisco de Sande. accull: An openacc implementation with cuda and opencl support. volume 7484, pages 871–882, 08 2012.

[45] Akihiro Tabuchi, Masahiro Nakao, and Mitsuhisa Sato. A source-to-source openacc compiler for cuda. In Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, Stephen L. Scott, and Josef Weidendorfer, editors, *Euro-Par 2013: Parallel Processing Workshops*, pages 178–187, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[46] Tristan Vanderbruggen and John Cavazos. Generating opencl c kernels from openacc. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, pages 9:1–9:10, New York, NY, USA, 2014. ACM.

[47] Seyong Lee and Jeffrey S. Vetter. Openarc: Extensible openacc compiler framework for directive-based accelerator programming study. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pages 1–11, Piscataway, NJ, USA, 2014. IEEE Press.

[48] Ahmad Lashgar, Alireza Majidi, and Amirali Baniasadi. Ipmacc: Open source openacc to cuda/opencl translator. *CoRR*, abs/1412.1127, 2014.

[49] Xiaonan Tian. *A COMPILER OPTIMIZATION FRAMEWORK FOR DIRECTIVE-BASED GPU COMPUTING*. PhD thesis, University of Houston: Faculty of the Department of Computer Science, apr 2016.

[50] J. A. Herdman, Wayne P. Gaudin, Oliver Perks, D. A. Beckingsale, A. C. Mallinson, and Stephen A. Jarvis. Achieving portability and performance through openacc. *2014 First Workshop on Accelerator Programming using Directives*, pages 19–26, 2014.

[51] Xuechao Li, Po-Chou Shih, Jeffrey Overbey, Cheryl Seals, and Alvin Lim. Comparing programmer productivity in openacc and cuda: An empirical investigation. *International Journal of Computer Science, Engineering and Applications (IJCSEA)*, 6(5), oct 2016.

[52] Ahmad Lashgar and Amirali Baniasadi. Openacc cache directive: Opportunities and optimizations. *Third Workshop on Accelerator Programming Using Directives*, nov 2016.

[53] Rengan Xu. *OPTIMIZING THE PERFORMANCE OF DIRECTIVE-BASED PROGRAMMING MODEL FOR GPGPUS*. PhD thesis, University of Houston: Faculty of the Department of Computer Science, apr 2016.

[54] Kei Ikeda, Fumihiko Ino, and Kenichi Hagihara. An openacc optimizer for accelerating histogram computation on a gpu. *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, feb 2016.

[55] Reza Miri and Mohsen Ghaffari-Miab. Fast computation of finite difference generated time-domain green's functions of layered media using openacc on graphics processors. In *2017 Iranian Conference on Electrical Engineering (ICEE)*, pages 1596–1599. IEEE Conferences, 07 2017.

[56] Alyson Pereira, Rodrigo Rocha, Márcio Castro, Luís Góes, and Mario Dantas. Extending openacc for efficient stencil code generation and execution by skeleton frameworks. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pages 719–726. IEEE Conferences, 07 2017.

[57] Matt Martineau, Simon McIntosh-Smith, and Wayne Gaudin. Evaluating openmp 4.0's effectiveness as a heterogeneous parallel programming model. *IEEE International Parallel and Distributed Processing Symposium Workshops*, may 2016.

[58] Matt Martineau, Simon McIntosh-Smith, Carlo Bertolli, Arpith C. Jacob, Samuel F. Antao, Alexandre Eichenberger, Gheorghe-Teodor Bercea, Tong Chen, Tian Jin, Kevin O'Brien, Georgios Rokos, Hyojin Sung, and Zehra Sura. Performance analysis and optimization of clang's openmp 4.5 gpu support. *7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, nov 2016.

[59] Akihiro Hayashi, Jun Shirako, Ettore Tiotto, Robert Ho, and Vivek Sarkar. Exploring compiler optimization opportunities for the openmp 4.x accelerator model on a power8+gpu platform. *Third Workshop on Accelerator Programming Using Directives*, nov 2016.

[60] Gheorghe-Teodor Bercea, Carlo Bertolli, Arpith C. Jacob, Alexandre E. Eichenberger, Alexey Bataev, Georgios Rokos, Hyojin Sung, Tong Chen, and Kevin O'Brien. Implementing implicit openmp data sharing on gpus. *CoRR*, abs/1711.10413, 2017.

[61] Artem Chikin, Tyler Gobran, and José Nelson Amaral. *OpenMP Code Offloading: Splitting GPU Kernels, Pipelining Communication and Computation, and Selecting Better Grid Geometries: 5th International Workshop, WACCPD 2018, Dallas, TX, USA, November 11-17, 2018, Proceedings*, pages 51–74. 01 2019.

[62] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kolodziej, and Christoph Kessler. Benchmarking opencl, openacc, openmp, and cuda: Programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, ARMS-CC '17, pages 1–6, New York, NY, USA, 2017. ACM.

[63] Burak Bastem, Didem Unat, Weiqun Zhang, Ann Almgren, and John Shalf. Overlapping data transfers with computation on gpu with tiles. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 171–180. IEEE, 08 2017.

[64] Guido Juckeland, Oscar Hernandez, Arpith C. Jacob, Daniel Neilson, Verónica G. Vergara Larrea, Sandra Wienke, Alexander Bobyr, William C. Brantley, Sunita Chandrasekaran, Mathew Colgrove, Alexander Grund, Robert Henschel, Wayne Joubert, Matthias S. Müller, Dave Raddatz, Pavel Shelepugin, Brian Whitney, Bo Wang, and Kalyan Kumaran. From describing to prescribing parallelism: Translating the spec accel openacc suite to openmp target directives. In Michela Taufer, Bernd Mohr, and Julian M. Kunkel, editors, *High Performance Computing*, pages 470–488, Cham, 2016. Springer International Publishing.

[65] Sergio Pino, Lori Pollock, and Sunita Chandrasekaran. Exploring translation of openmp to openacc 2.5: Lessons learned. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Conferences, 05 2017.

[66] S. Wienke, C. Terboven, J. C. Beyer, and M. Mũller. A pattern-based comparison of openacc and openmp for accelerator computing. In *Euro-Par2014 Parallel Processing Workshops*, pages 812–823. Springer, 2014.

[67] Toshiya Komoda, Shinobu Miwa, Hiroshi Nakamura, and Naoya Maruyama. Integrating multi-gpu execution in an openacc compiler. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 260–269, Washington, DC, USA, 2013. IEEE Computer Society.

[68] Kazuaki Matsumura, Mitsuhisa Sato, Taisuke Boku, Artur Podobas, and Satoshi Matsuoka. Macc: An openacc transpiler for automatic multi-gpu use. In Rio Yokota and Weigang Wu, editors, *Supercomputing Frontiers*, pages 109–127, Cham, 2018. Springer International Publishing.

[69] Yonghong Yan, Pei-Hung Lin, Chunhua Liao, Bronis R. de Supinski, and Daniel J. Quinlan. Supporting multiple accelerators in high-level programming models. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, pages 170–180, New York, NY, USA, 2015. ACM.

[70] Calvin Montgomery, Jeffrey L. Overbey, and Xuechao Li. Autotuning openacc work distribution via direct search. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, XSEDE '15, pages 38:1–38:8, New York, NY, USA, 2015. ACM.

[71] Makoto Sugawara, Shoichi Hirasawa, Kazuhiko Komatsu, Hiroyuki Takizawa, and Hiroaki Kobayashi. A comparison of performance tunabilities between opencl and openacc. In *Proceedings of the 2013 IEEE 7th International Symposium on Embedded Multicore/Manycore System-on-Chip*, MCSOC '13, pages 147–152, Washington, DC, USA, 2013. IEEE Computer Society.

[72] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization*, oct 2009.

[73] Ohio State University. Polybench/c: the polyhedral benchmark suite. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, 2019. Accessed 2019-03-05.

[74] J Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, June 1986.

[75] Chris Harris and Mike Stephens. A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference*, pages 147–151, 1988.

[76] Berthold K.P. Horn and Brian G. Schunck. Determining optical flow. Technical report, Cambridge, MA, USA, 1980.

[77] Rokiatou Diarra. Towards automatic restrictification of cuda kernel arguments. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 928–931, New York, NY, USA, 2018. ACM.

[78] Diarra Rokiatou, Alain Merigot, and Bastien Vincke. [engineering paper] recka and rpromf: Two frama-c plug-ins for optimizing registers usage in cuda, openacc and openmp programs. In *IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 187–192. IEEE, 09 2018.

[79] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[80] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, may 1994.

[81] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, Washington, DC, USA, 2011. IEEE Computer Society.

[82] Maroua Maalej, Vitor Paisante, Pedro Ramos, Laure Gonnord, and Fernando Magno Quintão Pereira. Pointer disambiguation via strict inequalities. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 134–147, Piscataway, NJ, USA, 2017. IEEE Press.

[83] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 1–14, New York, NY, USA, 1997. ACM.

[84] Michael Hind and Anthony Pioli. Which pointer analysis should i use? *SIGSOFT Softw. Eng. Notes*, 25(5):113–123, August 2000.

[85] Steve Carr and Ken Kennedy. Scalar replacement in the presence of conditional control flow. *Softw. Pract. Exper.*, 24(1):51–77, January 1994.

[86] Mihai Budiu and Seth Copen Goldstein. Inter-iteration scalar replacement in the presence of conditional control flow. In *3rd Workshop on Optimizations for DSO and Embedded Systems*, San Jose, CA, Mar 2005.

[87] Byoungro So and Mary Hall. Increasing the applicability of scalar replacement. In Evelyn Duesterwald, editor, *Compiler Construction*, pages 185–201, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[88] John Lu and Keith D. Cooper. Register promotion in c programs. *SIGPLAN Not.*, 32(5):308–319, May 1997.

[89] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of Workshop on Program Analysis For Software Tools and Engineering*, pages 54–61. ACM, 2001.

[90] Manu Sridharan and Stephen J. Fink. The complexity of andersen's analysis in practice. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, pages 205–221, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[91] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[92] Victor Hugo Sperle Campos, Péricles Rafael Alves, Henrique Nazaré Santos, and Fernando Magno Quintão Pereira. Restrictification of function arguments. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 163–173, New York, NY, USA, 2016. ACM.

[93] Maroua Maalej, Vitor Paisante, Fernando Magno Quinto Pereira, and Laure Gonnord. Combining range and inequality information for pointer disambiguation. *Sci. Comput. Program.*, 152(C):161–184, January 2018.

[94] Rishi Surendran, Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. Inter-iteration scalar replacement using array ssa form. *Cohen A. (eds) Compiler Construction. CC 2014. Lecture Notes in Computer Science*, 8409, 2014.

[95] José M. Andión, Manuel Arenaz, François Bodin, Gabriel Rodríguez, and Juan Touriño. Locality-aware automatic parallelization for gpgpu with openhmpp directives. *International Journal of Parallel Programming*, 44(3):620–643, Jun 2016.

[96] Xiaonan Tian, Dounia Khaldi, and Deepak Eachempati. Optimizing gpu register usage: Extensions to openacc and compiler optimizations. *45th International Conference on Parallel Processing (ICPP)*, aug 2016.

[97] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Form. Asp. Comput.*, 27(3):573–609, May 2015.

[98] David Nistér, Oleg Naroditsky, and James R. Bergen. Visual odometry. *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, 1:I–I, 2004.

[99] O. A. Aqel Mohammad, H. Marhaban Mohammad, Saripan M. Iqbal, and Bt. Ismail Napsiah. Review of visual odometry: types, approaches, challenges, and applications. oct 2016.

[100] Davide Scaramuzza and Friedrich Fraundorfer. Visual odometry : Part i: The first 30 years and fundamentals. *IEEE ROBOTICS & AUTOMATION MAGAZINE*, dec 2011.

[101] Davide Scaramuzza and Friedrich Fraundorfer. Visual odometry : Part ii: Matching, robustness, optimization, and applications. *IEEE ROBOTICS & AUTOMATION MAGAZINE*, jun 2012.

[102] Andreas Geiger, Julius Ziegler, and Christoph Stiller. Stereoscan: Dense 3d reconstruction in real-time. pages 963 – 968, 07 2011.

[103] Rostam Affendi Hamzah and Haidi Ibrahim. Literature survey on stereo vision disparity map algorithms. *Journal of Sensors*, 2016, 2016.

[104] Andreas Geiger, Martin Roser, and Raquel Urtasun. Efficient large-scale stereo matching. In *Asian Conference on Computer Vision (ACCV)*, 2010.

[105] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Computer Vision and Image Understanding*, 110:346–359, 06 2008.

[106] Shimiao Li. A review of feature detection and match algorithms for localization and mapping. *IOP Conference Series: Materials Science and Engineering*, 231:012003, sep 2017.

[107] Alexander Neubeck and Luc Van Gool. Efficient non-maximum suppression. volume 3, pages 850–855, 01 2006.

[108] B. Delaunay. Sur la sphère vide. a la mémoire de georges voronoï. In *Bulletin de l'Académie des Sciences de l'URSS. Classe des sciences mathématiques et na*, volume 6, pages 793–800, 1934. Accessed 30/05/2019.

[109] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981.

[110] Mateusz Jarus, Sébastien Varrette, Ariel Oleksiak, and Pascal Bouvry. Performance evaluation and energy efficiency of high-density hpc platforms based on intel, amd and arm processors. In Jean-Marc Pierson, Georges Da Costa, and Lars Dittmann, editors, *Energy Efficiency in Large Scale Distributed Systems*, pages 182–200, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[111] Gaurav Mitra, Beau Johnston, Alistair P. Rendell, Eric McCreath, and Jun Zhou. Use of simd vector operations to accelerate application code performance on low-powered arm and intel platforms. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1107–1116, Washington, DC, USA, 2013. IEEE Computer Society.

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

Titre : Parallélisation Automatique pour Systèmes Hétérogènes Embarqués

Mots clés : Architecture hétérogène, programmation parallèle, optimisation, analyse static, traitement d'image

Résumé : L'utilisation d'architectures hétérogènes, combinant des processeurs multicœurs avec des accélérateurs tels que les GPU, FPGA et Intel Xeon Phi, a augmenté ces dernières années. Les GPUs peuvent atteindre des performances significatives pour de nombreuses applications. Néanmoins, pour atteindre ces performances avec des API de bas niveau comme CUDA, il est nécessaire de bien connaître l'architecture des GPUs et d'appliquer des optimisations complexes, parfois non portables. D'autre part, les modèles de parallélisation à base directives offrent une abstraction de haut niveau du matériel sous-jacent, simplifiant ainsi la maintenance du code et améliorant la productivité. Ils permettent aux utilisateurs d'accélérer leurs codes séquentiels sur les GPUs en insérant simplement des directives. Les compilateurs d'OpenACC/OpenMP ont la lourde tâche d'appliquer les transformations et optimisations nécessaires pour générer des codes parallèles. Bien que ces compilateurs soient matures, le code généré peut ne pas atteindre l'accélération prévue. Ainsi, il existe généralement un écart de performance important entre les codes accélérés avec OpenACC/OpenMP et ceux optimisés manuellement avec CUDA/OpenCL. Afin d'aider les programmeurs à accélérer efficacement leurs codes séquentiels avec les modèles basés sur des directives, cette thèse aborde plusieurs problématiques de recherche. Nous avons étudié les modèles de parallélisation OpenACC et OpenMP et proposé une méthodologie efficace de parallélisation d'applications avec OpenACC/OpenMP. Notre expérience de portage d'applications a révélé qu'il était insuffisant d'insérer simplement des directives de déchargement OpenMP/OpenACC pour informer le compilateur qu'une région de code particulière devait être compilée pour être exécutéé sur un accélérateur. Il est essentiel de combiner les directives de déchargement avec celles de parallélisation de boucle. Nous démontrons que les modèles de parallélisation OpenMP/OpenACC peuvent atteindre de meilleures performances avec un effort de programmation moindre. Nous discutons également du problème d'alias des pointeurs dans les codes GPU et proposons deux outils d'analyse statiques qui permettent d'insérer automatiquement les qualificateurs de type et le remplacement par scalaire dans le code source.

Title : Automatic Parallelization for Heterogeneous Embedded Systems

Keywords : heterogeneous architectures, parallel programming, optimization, static analysis, computer vision

Abstract : The use of heterogeneous architectures, combining multi-core processors with accelerators such as GPUs, FPGAs and Intel Xeon Phi, has increased in recent years. GPUs can achieve significant performance for many applications. However, to achieve this performance with low level APIs like CUDA, it is necessary to know the architecture of GPUs and apply complex optimizations, sometimes not portable. On the other hand, directive-based programming models provide high-level abstraction of the underlying hardware, simplifying code maintenance and improving productivity. They allow users to parallelize their sequential codes by simply inserting directives. The OpenACC/OpenMP compilers have the difficult task of applying the transformations and optimizations needed to generate parallel codes. Although these compilers are mature, the generated code may not achieve the expected acceleration. Thus, there is usually a significant performance gap between accelerated codes with OpenACC/OpenMP and those optimized manually with CUDA/OpenCL. In order to help programmers effectively accelerate their sequential codes with directive-based models, this thesis addresses several research issues. We studied OpenACC and OpenMP programming models and proposed an efficient methodology for application parallelization with directive-based programming approaches. Our experience of porting applications revealed that it was not enough to simply insert OpenMP/OpenACC unload directives to inform the compiler that a particular code region needed to be compiled to run on an accelerator. It is essential to combine the unloading directives with those of loop parallelization. We demonstrate that OpenMP/OpenACC parallelization models can achieve better performance with less programming effort, but OpenMP/OpenACC compilers quickly reach their limit when the downloaded region code has a high arithmetic intensity, requires a very high number of access to global memory and contains several nested loops. In such cases, low level languages should be used. We also discuss the problem of pointers aliasing in GPU codes and propose two static analysis tools that automatically insert the restrict keyword and perform scalar replacement in source code.

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