UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Compiler and Runtime Based Parallelization & Optimization for GPUs

Guray Ozen

Ph.D. Thesis
Department of Computer Architecture - DAC
Universitat Politècnica de Catalunya - UPC

Advisors
Eduard Ayguadé
Jesús Labarta

November 2017

If you optimize everything,
you will always be unhappy.

To my mother. . .

Acknowledgements

This thesis would not be possible without the support of many people who have helped me in ways both large and small. First of all, I would like to thank my advisors, Jesús Labarta and Eduard Ayguadé, for their support, patience, and especially their trust. I am extremely grateful to meet with Jesus when I was MSc student; the homework I took at his course made me start a journey towards compilers. Up till today, he thought me how to find important topics and how to be methodological as a true engineer. I am very thankful to Eduard, who thought me how to communicate with the ideas effectively and how to express ideas in writing. They gave me a freedom to learn and investigate the areas that interest me. I am sure that there not many advisors would allow their Ph.D student to go off and do development on various topics to support their research.

I would like to thank the people working in the Computer Sciences department at Barcelona Supercomputing Center; for their insights and expertise in technical matters, and for their support. I also want to thank all the members of the Programming models team. In particular, I thank Sergi Mateo, Xavier Teruel, Jan Ciesko and Vishal Mehta.

I shall also mention and express my gratitude to Kathryn O'Brien and Kevin O'Brien along with his team, advanced compiler technology, from IBM T.J. Watson Research Center. I had the opportunity of working with an exceptional compiler team that allowed me to know how things work in industry. I firstly would like to Carlo Bertolli, who was my mentor during my internship. I also met a number of colleagues. Alexandre Eichenberger, Arpith Jacob – thank you all for the helpful moments and interesting conversations.

I would like to thank Michael Wolfe, for being my mentor during my internship at PGI group of NVIDIA Corp. I had a great and productive time thanks to Michael's always positive attitude and enthusiasm even though I was an impatient rookie. He gave me much priceless advice and I have learned many things from him about compilers. At NVIDIA, I also benefited from a great working environment and met a number of colleagues that made my stay even more enjoyable. I thank Annemarie Southwell, who was my supervisor, for the helpful moments.

My gratitude is also to Ayal Zaks and Michael Wolfe for reviewing this thesis. They kindly helped me improve its quality with their detailed analysis and brilliant comments, suggestions and discussion.

Barcelona, 2017 Guray Ozen.

Abstract

Graphics Processing Units (GPU) have been widely adopted to accelerate the execution of HPC workloads due to their vast computational throughput, ability to execute a large number of threads inside SIMD groups in parallel and their use of hardware multithreading to hide long pipelining and memory access latencies. There are two APIs commonly used for native GPU programming: CUDA, which only targets NVIDIA GPUs and OpenCL, which targets all types of GPUs as well as other accelerators. However these APIs only expose low-level hardware characteristics to the programmer. So developing applications able to exploit the dazzling performance of GPUs is not a trivial task, and becomes even harder when they have irregular data access patterns or control flows.

Several approaches have been proposed to help simplify accelerator programming. Models like OpenACC and OpenMP are intended to solve the aforementioned programming challenges. They take a directive based approach which allows the users to insert non-executable directives that guide the compiler to handle the low-level complexities of the system. However they have a performance gap with native programming models as their compiler does not have comprehensive knowledge about how to transform code and what to optimize.

This thesis targets directive-based programming models to enhance their capabilities for GPU programming. The thesis introduces a new dialect model, which is a combination of OpenMP and OmpSs. It also includes several extensions and the MACC infrastructure, a source-to-source compiler targeting CUDA developed on top of BSC's Mercurium compiler and able to support the new dialect model. The new model allows the use of multiple GPUs in conjunction with the vector and heavily multithreaded capabilities in multicore processors automatically. Moreover, it introduces new clauses to make use of on-chip memory efficiently. Secondly the thesis focusses on code transformation techniques and proposes the LazyNP method to support nested parallelism for irregular applications such as sparse matrix operations, graph and graphics algorithms. The method efficiently increases thread granularity for the code region where nested parallelism is desired. The compiler generates code to dynamically pack kernel invocations and to postpone their execution until a bunch of them are available. To the best of our knowledge, LazyNP code transformation was the first successful code transformation method related to nested directives for GPUs. Finally, the thesis conducts a thorough exploration of conventional loop scheduling methods on GPUs to find the advantage

and disadvantages of each method. It then proposes the concept of optimized dynamic loop scheduling as an improvement to all the existing methods.

The contributions of this thesis improve the programmability of GPUs. This has had an outstanding impact on the whole OpenMP and OpenACC language committee. Additionally, our work includes contributions to widely used compilers such as Mercurium, Clang and PGI, helping thousands of users to take advantage of our work.

Contents

List of Figures

List of Tables

1 Introduction

In recent years, a revolution has been occurring in the world of computer architecture research and development. Performance improvements based on clock frequency scaling are no longer a viable option due to physical limits of power consumption and the ability to dissipate heat. This has brought new challenges in processor design to find the most efficient energy consumption/performance ratio.

After these limits were first hit, central processor unit (CPU) architecture underwent a drastic evolution from single core to multicore. Processor vendors have done a lot of investment to find out how to increase performance without increasing clock frequency. As a result, even the simplest CPU today incorporates more than one core. Parallel systems have become mainstream at CPU architecture and are present even on mobile devices. However multicores don't provide enough parallelism for many workloads, and so over time researchers have started to explore other ways to increase parallelism such as using massively parallel accelerators as well as multicores.

The new vogue in computer architecture is accelerators, which are mostly used to supplement special functions of the host processor (CPU). They have gained popularity in the last few years due to their impressive potential performance, higher ratios of throughput to consumed power and improved performance versus system cost when compared to multi-core architectures. Therefore, they have become essential in several areas of computer science such as High-Performance Computing (HPC), artificial intelligence (AI), data science, etc. The main examples of these recent hardware accelerators include Graphics Processing Units (GPU) from NVIDIA, AMD and ARM [1], the Intel Xeon Phi co-processors [2] and PEZY-SC many-core accelerator [3] or FPGAs [4].

The current Top500 [5] list, which ranks and details the 500 most powerful non-distributed computer systems in the world, reflects the popularity of accelerator usage in supercomputing area as four of the top 5 machines have either GPUs or Xeon Phi coprocessors as shown in Table 1.1. In fact, although the second ranked computer seems like it does

Rank	Name	Year	Processor	Accelerator/Co-Processor
1	Summit	2018	IBM POWER-9	**NVIDIA Volta GV100**
2	Sunway TaihuLight	2016	Sunway SW26010	None
3	Tianhe-2 (MilkyWay-2)	2013	Intel Xeon E5-2692v2	**Intel Xeon Phi 31S1P**
4	Sierra	2018	IBM POWER-9	**NVIDIA Volta GV100**
5	ABCI	2017	Xeon Gold 6148	**NVIDIA Tesla V100**

Table 1.1: Top 5 supercomputers in the current TOP500 list (June 2018)

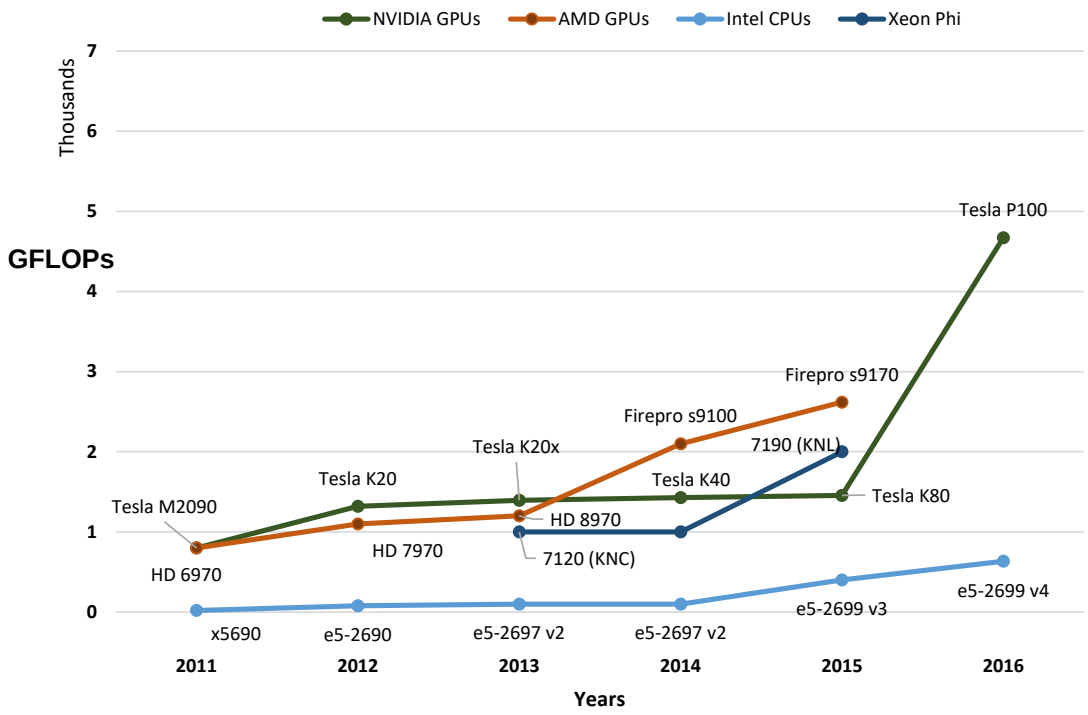


Figure 1.1: Theoretical peak floating point operation per clock cycle, double precision.

not have an accelerator, its CPU incorporates processor with many cores acting like accelerators. Based on this list, we can assume that in future many computers will include accelerators.

GPUs are one of the most important and commonly used accelerators today. They play a crucial role in the context of energy-aware high-performance accelerators. They have been already widely adopted to accelerate the execution of HPC workloads due to their large computational throughput, being able to execute a large number of threads inside SIMD groups in parallel and use of hardware multithreading to hide long pipelining and memory access latencies.

Figure 1.1 shows performances of different accelerators as well as CPUs. The performance is shown as GFLOPs, in the order of the tens for CPUs and the hundreds for GPUs

and Xeon Phi. It is worth noting the need to use parallelization or vectorization to leverage the full potential. Even though Intel CPUs involve multiple cores (for instance *e5-2699-v4* incorporates 22 cores, this number of cores is very low compared with GPUs. The impact of this is clear when you look at the low scores for CPUs in Figure 1.1. Clearly accelerators are doing a great job and obtaining very good performance compared with CPUs. However, relating a GPU processing unit with each CPU core is inappropriate: GPU multiprocessors operate batches of their processing units in lock-step, so if there is diverging code, the code is divided into paths, and each path cannot be processed independently. We explain GPU architecture and work-flow in detail in Section 2.1. Apart from accelerators, the other significant advance in computer architecture research is vectorization. Since massive parallel architectures provide very good speed, CPU vendors have put a lot of effort into vectorization. Especially Intel stepped forward vectorization and introduced the first version of AVX, then AVX2 with 256-bit vector units[6]. More recently, they moved to 512-bit vector extensions with the Initial Many-Core Instructions (IMCI) included in the Intel Xeon Phi coprocessor (Knights Corner) and later in the Skylake family of CPUs. With the introduction of AVX in CPUs, in particular AVX2 with its many masked operations, one can achieve similar execution behavior to GPUs.

Accelerator programming has gradually added more advanced and flexible instructions that bring bulk thread parallelism to a broader range of applications and domains. For instance in newer GPUs, we can find *tensor cores* designed to provide support to neural networks, support for nested parallelism with dynamic parallelism feature, support for independent thread scheduling to improve the performance of diverging code and support for NVLink [7] which is a wire-based serial multi-lane near-range communication protocol.

As accelerator architectures have become more sophisticated and advanced, programming accelerators has remained a big challenge. Using accelerators requires a parallel programming approach which is drastically different to traditional sequential programming. Therefore, several programming ideas have been proposed in recent years to facilitate programming of accelerators. One of the first attempts was Brook [8], which was the forerunner to generalized computing on GPUs. More recently CUDA [9] has become the main programming API for NVIDIA GPUs, and OpenCL [10] provides a general framework which works with Intel Xeon Phi cards and GPUs as well. However programming accelerators is still challenging. Some accelerators like GPUs may not be able run legacy CPU code. On the other hand, Xeon Phi can run the code, but may not achieve the best performance. Thus, code must be re-written for each accelerator type. In other words, accelerators have raised the programming barrier to an unaffordable level.

So to summarise, the history above shows us future computers will incorporate more and more accelerators due to the to their higher peak performance and performance

per Watt ratio when compared to multicores. Therefore, accelerator programming must become sophisticated enough to exploit the best performance from these architectures. In this thesis, we introduce, implement and optimize high-level programming model for GPUs accelerators.

1.1 The Problem Statement

While GPUs are becoming more and more sophisticated and offering promising performance with reasonable power consumption, programming accelerators in an efficient manner is still a challenge. This is because the parallel programming needed is fundamentally unlike the traditional sequential approach to programming CPUs, also the existing frameworks are still extremely low-level, further raising the barrier to programmability. To address this challenge, this thesis proposes several mechanisms for parallelization and optimizations for compilers and runtime systems targetting GPUs. From the software contributions perspective, this thesis includes two open-source contributions for the **Mercurium compiler** and **CLANG C/C++ front-end for LLVM compiler** and one commercial contribution for the **NVIDIA PGI Compiler**.

1.1.1 High-level Programming Model

Currently there are two APIs commonly used for native GPU programming: CUDA which targets NVIDIA GPUs and OpenCL which targets all GPUs as well as other accelerators such as FPGAs. Native GPU programming gives all responsibility to the programmer who should take care of transforming computationally intensive pieces of code into kernels to be executed on the accelerator devices as well as write the host code to orchestrate data allocations, data transfers and kernel invocations with the appropriate allocation of GPU resources.

To achieve maximum performance out of GPUs, programmers must know GPU architecture as well as have parallel programming knowledge. For instance, it is essential to know that the GPU hardware has an additional bunching of threads which in NVIDIA is called a `warp` with 32 threads, in AMD hardware is called a `wavefront` with 64 threads. A `warp` (or `wavefront`) is the most fundamental computing unit in GPUs. All the threads in a warp share the same program counter and therefore execute the same instruction at the same time, the only difference being the data that they operate on in that instruction. This is called lock-step execution. Therefore this question comes to mind: **Are we programming a thread or a warp?**. The short answer is we program a thread in the native programming language. However, we must also be aware of how a warp works in order to achieve the best performance. Therefore, exploiting the best performance from GPUs is not a trivial task in native GPU programming languages.

1.1.2 GPU Specific Code Transformations and Optimizations

Native GPU programming is a hard programming task, therefore several programming ideas have proposed in recent years to ease the burden of programming accelerators [11, 12, 13, 14, 15, 16, 17]. Currently there are two main standards, OpenMP [12] and OpenACC [13].

The native models like CUDA or OpenCL are language-based models, therefore these are flexible enough for the user to apply any optimization they want. Directive-based programming models like OpenMP are higher-level, therefore they handle code transformation and optimization. However, without enough information, the compiler is not able to do the optimizations as well as an expert programmer would. The reason that can stop the compiler parallelizing code can vary: unknown trip count of loops, nested and irregular loops, pointer aliasing, complex loop cross-iteration dependencies or function calls. They all affect the flow of execution, hindering the analysis of the code. Another reason is that specialised GPU compilers are still relatively new, so they still do not have comprehensive knowledge about **How should the compiler transform code for GPUs?** and **What should the compiler optimize for GPUs?**.

Code transformation for irregular applications has always been an issue for bulk parallel GPUs. Moreover, according to the recent DoE report on exascale computing [18], sparse matrix applications and graph applications are highlighted as particular challenges for exascale computing due to their load balance problem. In this thesis we propose a code transformation algorithm that targets irregular applications which have unknown trip count.

In addition, code optimization is a very important topic for high-level programming model compilers. Since GPUs are relatively new devices, there is not much compiler research for them. We investigate the loop optimization method which is one of the most important optimization approaches in compilers in a long time. However, we have seen that loop scheduling research is missing when we look at studies about GPU compilers.

Lastly, we propose several extensions to OpenACC and OpenMP models to allow their compiler to enable more optimizations. These are the two primary standards which HPC scientists and engineers have come up with to facilitate GPU utilization. However, there is still a lack of compiler support for these standards for GPUs. Compiler-generated code is not always able to exploit parallelism successfully in real applications due to insufficient directives in many cases. Another critical factor is that the native languages (C, C++ and Fortran) that the standards are based on are gradually improving; therefore the standards should keep up with the language, and in turn the compilers should keep up with the standards. However, adapting new language features to the device model is a long process. Lastly, heterogeneous systems are generally incorporating more than one accelerator and a host processor in a single node. In order to exploit from entire

machine, the workload should be balanced among all the accelerators on the system, which is a non-trivial task.

1.1.3 Heterogeneous Systems

The accelerator computing era introduces a new term which is *heterogeneous computing*. Unfortunately, heterogeneous computing makes programming a difficult task even for expert programmers, especially for programmers who want to exploit machine resources fully. Firstly, accelerators may not able to run legacy CPU code. Therefore, applications targeting traditional CPU architectures may have to be redesigned and rewritten. Another challenge with heterogeneous system is that accelerators most likely may have their separate memory space which may be designed in a different way than main memory. It breaks conventional programming approach as programmers always tend to think that CPUs use single memory space with transparent mechanisms. To use accelerators, it becomes necessary to manage separate memory spaces by adding additional code to maintain data movement and keep coherency.

1.2 Thesis Contributions

In this thesis, we present several contributions in the field of directive-based programming models, compilers and runtime algorithms aimed to leveraging the exploration of GPUs. The main contributions are the following:

A new dialect programming model: We present a new dialect model, which is a combination of OpenMP and OmpSs, along with several extensions. We also introduce a MACC infrastructure[1], which is a source-to-source compiler targeting CUDA and developed on top of OmpSs and able to support the new dialect model. Our model aims to reduce the burden of programming GPUs and deliver the best performance out of them. It is developed on top of the Mercurium compiler. This model is a result of our initial design, implementation, integration, research, and evaluation of compiler algorithms related to a different aspect of GPU code generation infrastructure.

Code transformation for GPU compilers: We propose a code transformation technique that covers all types of irregular applications such as sparse matrix, graphs, graphics, etc. We call our method Lazy Nested Parallelism (LazyNP) since it targets nested parallelism. The compiler generates code to dynamically pack kernel invocations and to postpone their execution until a bunch of them are available. Our method is highly efficient and elegant and can even outperform optimized libraries which are provided by vendors.

GPU specific optimization techniques: We explore using dynamic scheduling for

[1] MACC is an abbreviation of Mercurium ACCelerator Model

mapping parallel loop iterations to GPU threads in an NVIDIA PGI OpenACC compiler. Our method reaches maximum performance with a small grid size. In addition, it yields better performance for reduction operations. We explain the shortcomings of static scheduling and why mapping one thread per iteration is undesirable. We then show how to generate dynamically scheduled loops for GPUs safely, and describe the implementation in PGI OpenACC compiler.

Extensions for OpenMP and OpenACC standards: We introduce several different extension ideas to OpenMP and OpenACC standards which includes new directives and clause to allow programmers to direct the compiler in the code generation process in GPU device model. We also extended the meaning of some directives and clauses to increase coverage of code scenarios. These directives and clauses provide the compiler with information valuable to generate GPU code while utilizing better GPU hardware. Also we propose extensions able to adapt Modern C++ features such as *lambda* and *variadics templates* and their combinations. Some of these proposals have been included in the new version of the OpenMP and OpenACC standards.

Multiple device management: We propose a model that can automatically manage different device types such as GPUs and CPUs while generating code for them. We extended our MACC infrastructure on OmpSs task model. Our model takes the burden from programmers and moves it to runtime while minimizing data movement as task data information is provided in the code. Additionally we also propose a multiple target task sharing approach which is able to utilize all the system devices such as CPU and GPU for the same task.

1.3 Thesis Organization

The rest of this thesis is structured as follows. Chapter 2 gives some background to the problem along with overview of the OmpSs, OpenMP and OpenACC programming models. It also includes details of the Mercurium, CLANG and PGI compilers which are their respective development environments. Chapter 3 shows our MACC infrastructure which is a source-to-source generation model for GPU. Chapter 4 introduces our proposal on device model extensions for OpenMP and OpenACC standards. Chapter 5 describes our highly efficient code transformation proposal for nested parallelism for GPUs. Chapter 6 shows our dynamic loop scheduling proposal and its associated code transformation algorithm for GPUs. Finally in Chapter 7, we conclude this thesis, discuss the impact of the research and present proposals for future work in this field.

2 Background and Development Environment

In this chapter, we give the background of our research work. First, we give an overview of the GPU architecture (Section 2.1) which is the platform our research based on. Then we explain how to program GPUs with the native CUDA API (Section 2.1.2).

The next sections explain the three high-level programming models for GPUs along with their respective compilers. First in (Section 2.2) we explain OmpSs [19] which is a state-of-art task-based parallel programming model based on **(1)** Mercurium [20]. Then in (Section 2.3) we explain about OpenMP and its device model for GPUs. OpenMP is the de facto standard for shared-memory system programming model. The OpenMP we used in this thesis is based on **(2)** Clang/LLVM [21, 22]. Finally in (Section 2.4) we mention OpenACC which is the standard for the high-level directive-based programming model for accelerators. We used the **(3)** PGI [15] compiler for OpenACC.

2.1 Overview of GPU Architecture

GPU architectures differ quite a lot from traditional processors. The main differences are in the number of cores that they involve and in the size of such cores. The main characteristic of CPUs is that they consist a few cores which are highly optimized for sequential processing. By contrast GPUs involves a huge number of tiny cores which are highly optimised for efficient parallel processing tasks, but quite slow for sequential processing when compared to CPUs. For instance, Intel Xeon Processor E7-8894 v4 contains 24 cores while NVIDIA P100 GPU contains 3584 cores. Figure 2.1 illustrates the difference.

Essentially, GPUs have evolved from video cards to extremely powerful and flexible processors. Their architecture involves fast bandwidth memory and computational power, with fully programmable processing units that support vector operations. Architecturally, GPUs are highly parallel streaming processors optimized for vector operations. Researchers have found that exploiting the GPU can accelerate some

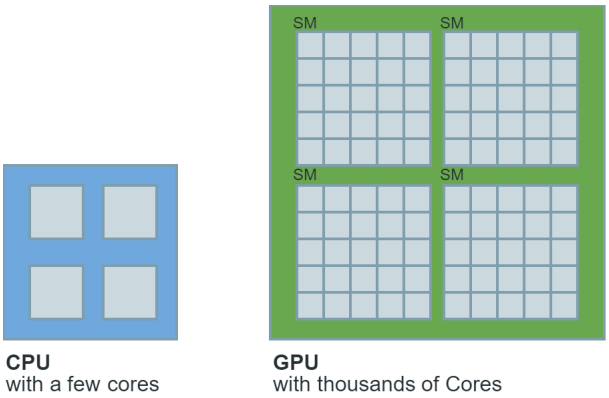


Figure 2.1: High-level architecture comparison between CPU and GPU

problems by over an order of magnitude over the CPU [8].

In NVIDIA terminology, a GPU involves a number of Streaming Multiprocessor (SM) units, where an SM roughly corresponds to a CPU core. Each SM has a large register file, a number of integer and floating point ALUs, instruction fetch units, branch processing units, load-store units, control units, and a small level 1 data cache. For example each SM on Fermi and Kepler GPUs has a 48KB configurable, 128B cache-line, write-evict L1 data cache for general off-chip DRAM access [9]. It shares the same chip storage with the shared memory. However, recent Maxwell and Pascal GPUs devote this storage entirely for shared memory, while relying on the texture cache to offer L1 caching capability. As far as we know, all SMs in a GPU are connected via a NoC (Network on Chip) to a shared L2 cache. The L2 cache is banked and is writable.

Using GPUs for general purpose computing became popular after about 2005. Since then GPUs have been evolving in the direction of general purpose computing. We show the basic architectural specifications of five NVIDIA generations in the Table 2.1.

GPUs	Arch.	Comp Cap	SMs	Warp Slots	CTA Slots	Max Dimensionality of grid of CTAs	Max x-dimension of a grid of CTAs	Max y,z-dimension of a grid of CTAs
GT200	Fermi	2.0	16	48	8	3	65535	65535
Tesla K80	Kepler	3.x	13	64	16	3	$2^{31}-1$	65535
Titan X	Maxwell	5.x	16	64	16	3	$2^{31}-1$	65535
P100	Pascal	6.x	56	64	16	3	$2^{31}-1$	65535
V100	Volta	7.x	80	64	16	3	$2^{31}-1$	65535

Table 2.1: The details of NVIDIA GPU architectures.

2.1.1 GPU Execution Model

A GPU executes a kernel grid: each SM executes one or more thread blocks and the CUDA cores and other execution units in the SM execute the individual threads. A kernel grid might consists of multiple thread blocks whose shape could be 1-3 dimensional. And each thread block consists of multiple threads whose shape could also be 1-3

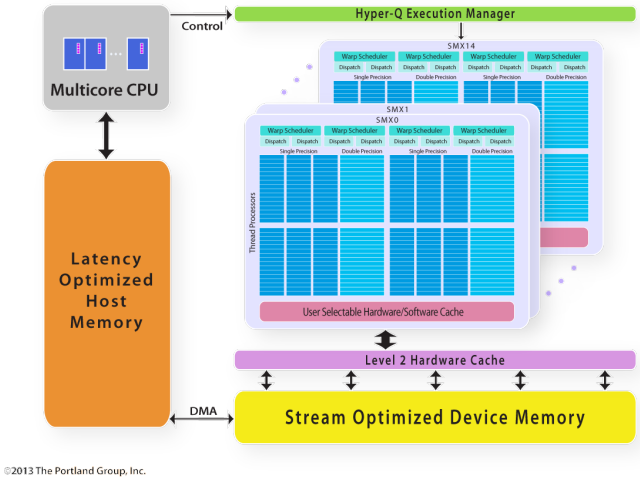


Figure 2.2: High Level NVIDIA GPU Architecture
1

dimensional. In addition, a kernel grid might involve more threads than the GPU can handle at once. Underlying hardware maps these multi-dimensional thread blocks and threads to SM and warps respectively.

The SMs combine a number of threads and run them in a SIMT (Single-Instruction Multiple Thread) manner. Figure 2.2 illustrates the SM and high-level architecture of an NVIDIA Kepler GPU. The smallest execution unit of a GPU is called a *warp*; all enabled threads in the warp are clocked so they will execute the same instruction at the same time. This is called *lock-step* execution. If some threads take a branch and others do not, the control processor will select one branch target and disable the other set of threads, continuing until all threads synchronize or otherwise *converge*.

The execution model increases efficiency by reducing the quantity of resources required to track thread state and by aggressively reconverging threads to maximize parallelism. In NVIDA's new Volta architecture [23], an independent thread scheduler is introduced to improve the *SIMT* execution pattern. It allows the GPU to yield execution of any thread, either to make better use of execution resources or to allow one thread to wait for data to be produced by another. In this thesis, we have evaluated Fermi, Kepler, Maxwell and Pascal architectures. However we have not done any evaluation on Volta architecture as they were not yet available at the time of writing.

Warps are grouped in a thread block also known as a cooperative thread array (CTA). A CTA is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. It has an ID within its grid. Each SM has a number of slots for CTAs and warps. The number of slots differs between GPU architectures as does the number of threads per SM, as is shown in Table 2.1.

2.1.2 Native GPU Programming: CUDA

NVIDIA GPUs are one of the most popular accelerators and are extensively integrated in HPC clusters. Compute Unified Device Architecture (CUDA) [9] is the de-facto standard for programming NVIDIA GPUs. The programmer has to write specialized pieces of code called CUDA kernels that are executed concurrently by many threads on the GPU. A CUDA program is a unified source code including both host and device code. The host code is pure ANSI C code and the device code is the extension of ANSI C which provides keywords for labeling data parallel functions or kernels. The host code and device code are compiled separately. The host code is compiled by the host's standard C compiler and the device code is compiled by the NVIDIA compiler or an open-source compiler. The host code is executed on the host and offloads the device code to be executed on the device. CUDA provides both a low level driver API and a high level runtime API. The programmer can use these APIs to manage the execution context environment, the device memory allocation and deallocation, the data movement between CPU and GPU, the asynchronous data movement and kernel execution, etc. Since CUDA is proprietary to NVIDIA, the performance of CUDA program is optimized for NVIDIA GPUs. However this also means that the program cannot be migrated to any other vendor's GPUs.

2.2 OmpSs

The OmpSs programming model [19] provides a task-based programming model for homogeneous and heterogeneous architectures and is designed to be able to support new architectures that may appear in the future.

2.2.1 OmpSs Programming Model

The OmpSs combines the OpenMP [12] and StarSs [24] programming models, offering a task-based programming models with extended directives and clauses. In particular, its objective is to extend OpenMP with new directives to support asynchronous parallelism and heterogeneity. Thus it significantly enhances the asynchronous parallelism support in OpenMP. It also takes the task dependence support from StarSs to allow the runtime to automatically manage and move data and perform different kinds of optimizations. The OmpSs proposal has been evolving during the last decade and aims to lower the programmability wall raised by multi/manycores, demonstrating a task based data approach in which offloading tasks to different devices, as well as managing the coherence of data in multiple address spaces, is delegated to the runtime system. Several efforts for the IBM Cell (CellSs [25]), NVIDIA GPU (GPUSs [26]) and homogeneous multicores (SMPSs [27]) were investigated before arriving at the current unified OmpSs specification and implementation.

The main goal of OmpSs is to orchestrate different types of task on heterogeneous systems. It completely enables asynchronous parallelism by the of use data-dependencies between the different tasks of the program. OmpSs is currently able to orchestrate applications on clusters of nodes that combine shared memory processors (SMPs) and other external devices, for example, GPUs, FPGAs etc. However, it has never aimed to generate optimized code for the target devices. Therefore, it has no facility to generate code specifically optimized for GPUs. One of the main contribution of this thesis is the development of GPU code generation and optimization for OmpSs.

Execution Model

OmpSs is based on a thread-pool execution model while OpenMP uses a fork-join model. The master thread starts the execution of the program. It orchestrates execution with the other threads when there is worksharing or *task* constructs. The idea behind using a thread-pool is to remove the overhead of creating new threads for each parallel region. Thus it does not use the *parallel* construct of OpenMP as there is no need to have it. It is also worth mentioning that it supports nesting constructs, which allows other threads to generate work.

The *task* construct allows expressing parallelism for OmpSs applications. It comes with data directionality clauses *in, out* and *inout* which are same as the *depend* clause dependence-type lists in OpenMP. Moreover, it offers two additional clauses to *task* constructs: *concurrent* and *commutative*. The associated runtime of OmpSs, *Nanos++*, constructs a data-dependency graph which is dynamically built with the information extracted at compiler time from the clauses. We will discuss nanos++ in detail later. This task graph construction is essential to ensure the application's data coherence and correctness. Consequently only ready tasks (i.e., tasks where dependencies have been satisfied) can be run in parallel.

Memory Model

One of the most powerful parts of OmpSs is that it offers a single address space to ease the burden on the programmer when dealing with heterogeneous systems. The underlying runtime mechanism automatically manages multiple address spaces and moves the data as necessary. Therefore, the data can be truly shared between multiple address spaces and resides in the correct memory locations. All parallel code can only safely access the private and shared data that has been marked explicitly with the OmpSs extended syntax.

Programming Model

Regarding task support, OmpSs is superset of OpenMP. It defines several extensions to the OpenMP model as follows.

Task Model

OmpSs supports OpenMP task constructs, plus it offers new clauses.

```
#pragma omp task [clauses]

 { code block | function }
```

where clauses specify:

- `in, out, inout` - input, output and input/output directionality respectively.
  - list - scalar, array section, subregion or l-values.
  - Multiple dependencies - allow expressions that determine dependencies dynamically during runtime.
- `concurrent` - express relaxed inout.
- `commutative` - relaxed inout.
- `reduction` - reduction support for tasks.
- `resources` - resource consumption of the task
- `shared, private, firsprivate` - data sharing.
- `if` - if expression.
- `final` - allows an expression to finalize tasks.
- `priority` - assigns priority to tasks.
- `untied` - allows any thread to execute this task.

```
#pragma omp taskwait [clauses]
```

- `in, out, inout` - requires to wait if and only if the data are specified with the clauses.

It has support for *taskloop*.

`#pragma omp taskloop [clauses] for-loop` where clauses specify:

- `in` - input directionality.
- `reduction` - reduction support.
- `num_tasks` - number of tasks.
- `nogroup` - no grouping.
- `grainsize` - size of granularity.

Accelerator Orchestration Model

The accelerator support in the OmpSs programming model leverages the tasking model with data directionality annotations already available in the model (which influenced the new depend clause in OpenMP 4.5). These annotations are used by the OmpSs runtime system to compute task dependencies and build a dependence task graph dynamically. This graph is used to dynamically schedule tasks in a data–flow while being conscious of the resources available at any given time. OmpSs offers a *target* directive with the following syntax:

```
#pragma omp target [clauses]

 task construct | function definition | function header
```

where clauses specify:

- `device` - the kind of devices that can execute the construct (smp, cuda or opencl).
- `copy_in` - shared data that needs to be available in the device before the construct can be executed.
- `copy_out` - shared data that will be available after the construct is executed.
- `copy_inout` - a combination of copy_in and copy_out above.
- `copy_deps` - copy semantics for the directionality clauses in the associated *task* construct (i.e., in will also be considered copy_in, output will also be considered copy_out and inout as copy_inout).
- `implements` - an alternative implementation of the function whose name is specified in the clause for a specific kind of device.

In order to allow hybrid code with native CUDA and/or OpenCL kernels, the directive includes two additional clauses:

- `ndrange` - specification of the dimensionality, iteration space and blocking size to replicate the execution of the CUDA or OpenCL kernel.
- `shmem` - data that should be mapped into shared–memory among teams in the device.

The `copy_in, copy_out` and `copy_inout` clauses, together with the lookahead provided by the availability of the task graph, are used by the runtime system to schedule data copying actions between address spaces (movements between host and accelerator or between two accelerator devices if needed). The `copydeps` is a simple shorthand to reuse the directionality annotations in the task directive.

Figure 2.3 shows a simple example based on vector addition operation. In this example, the task which has the compute loop is written as a CUDA kernel and offloaded to a device with CUDA architecture; the task checking the results is defined to be executed in the host. Observe that the output of the CUDA task instances is the input of the host task instances. The dependencies computed at runtime will honor these dependencies, and the runtime system will take care of doing the data copying operations based on the information contained in the task graph (dynamically generated at runtime). The *ndrange* clause is used to replicate the execution of the CUDA kernel in the device block/thread hierarchy (one dimension with na*na iterations in total to distribute among teams of na iterations in this example).

With the device clause, the programmer informs the compiler and runtime system about the kind of device that can execute the task, not an integer number that explicitly maps the offloading to a certain device as is done in OpenMP 4.5. This is a big difference that greatly improves programming productivity when targeting systems with different numbers and types of accelerators and regular cores.

```
using T = double;
const int N = 1024;

#pragma omp target device(cuda) ndrange(1, N*N, N)
#pragma omp task in(a[:N], b[:N]) out(c[:N])
__global__ void vecadd(T *a, T *b, T* c, int N) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < N)
    c[i] += b[i] * a[i];
}

#pragma omp target device(smp) implements(vecadd)
#pragma omp task in(a[:N], b[:N]) out(c[:N])
void vecadd_smp(T *a, T *b, T* c, int N) {
  for (int i = 0; i < N; ++i)
    c[i] += b[i] * a[i];
}

#pragma omp target device(smp)
#pragma omp task in(c[:N)
bool check_results(T *c, T checksum, int N) {
 return valid = std::all_of(c, c + N, [] (T const &v) {
    return v == T(checksum);
 });
}

int main(int argc, char **argv) {
  T *a, *b, *c;
  a = new T[N];
  b = new T[N];
  c = new T[N];

  std::fill(a, a + N, T(1));
  std::fill(b, b + N, T(2));

  vecadd(a, b, c, N);
  if( !check_results(c, 3) )
    printf("Results are not valid\n");

  #pragma omp taskwait
}
```

Figure 2.3: Heterogeneous task example with OmpSs

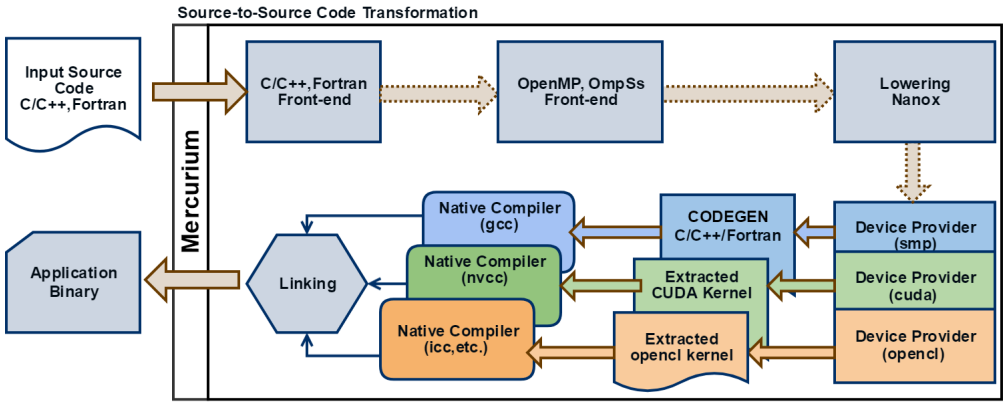


Figure 2.4: Compilation flow of Mercurium Compiler

2.2.2 OmpSs Infrastructure

Mercurium Compiler

Mercurium [20] is a research-oriented source-to-source compiler with support for C, C++ and Fortran languages. It has been developed by Barcelona Supercomputing Center with the main goal of enabling fast prototyping of new programming models. It has support for OmpSs and OpenMP 3.1 programming models.

Mercurium can be divided into three main parts; *front-end* with full support of C, C++ and Fortran, *transformation* part which does all the source-to-source transformation and the last part is *codegen* that is responsible for regenerating the final source code. Figure 2.4 shows a high-level overview of the three parts of the Mercurium compiler.

Mercurium aims to be an agile compiler without generating object code. It has been mainly used with the Nanos++ and OpenMP RTL runtimes. It has no goal of optimizing code or providing deeper analysis. Thus it does not involve any conventional compiler optimization methods. In the place of this it can enable parallelization and parallelism related optimizations.

Intermediate Representation

Mercurium maintains the input code as an intermediate representation (IR) within the compiler. The structure of the IR is based on an abstract syntax tree (AST). The nodes of the AST represent high-level code structures of the programming language. The IR is not low-level because one of the main aims of Mercurium is to generate output source code as similar as possible to the input code.

The AST is built in the Mercurium front-end with information about the explicit code and data of the compilation unit. However, information regarding declarations is

minimal in this AST as the code generator of Mercurium is able to deduce it from the code representation. The type system and other symbolic information are represented separately from the AST. Mercurium uses independent data structures for these purposes that are accessible from fields of the AST nodes.

Nanos++ Runtime

The Nanos++ runtime library is the component responsible for executing OmpSs applications that have been compiled using the Mercurium C/C++ source-to-source compiler. As we mentioned above, the Mercurium compiler transforms most of the directives into calls to Nanos++ and it restructures the code in order to be able to be handled by tasks. Nanos++ offers subsystems that handle the task creation, dependence analysis, and task execution.

The main responsibility of Nanos++ is to execute the created tasks as fast as possible. To do so, it uses all the available hardware and software resources of the system. The hardware resources that Nanos++ manages are the CPUs, GPUs, and system memory, while the software resources are the threads, provided by the operating system, and the program tasks, provided by the user application.

The internal design of Nanos++ can be divided into three parts, the representation of the program tasks, the architecture support, and the behavior subsystems. The program tasks are represented internally by Work Descriptors, which are entities that represent the code that has to be executed by Nanos++. The architecture support is composed of a set of generic concepts that model the hardware and software resources managed by Nanos++. To support a specific architecture, a concrete implementation of these concepts must be provided. The behavior subsystems are in charge of executing the program tasks. Each subsystem fulfills a specific role in the process. However, globally they provide the logic and the intelligence of the library to manage the system resources to execute the program tasks as efficiently as possible while ensuring a correct execution order. Nanos++ provides different implementations of the subsystems that can be selected by the user when running OmpSs programs in order to achieve a more optimal performance.

2.3 OpenMP

OpenMP [12] is the de facto standard shared-memory parallel programming model. It supports C, C++, and Fortran and is based on adding some compiler directives to the source code, which are then translated into calls to the OpenMP runtime library routines.

Execution Model It offers parallelization with fork-join model, a method of parallelization whereby a *master thread forks* a certain amount of *slave threads* and the runtime

```
1 void saxpy_openmp(float* x, float* y, float a, int N) {
2  #pragma omp parallel for
3  for (int i = 0; i < N; ++i)
4   y[i] += a*x[i];
5 }
```

Figure 2.5: SAXPY example with OpenMP's parallel construct

divides the workload among them. The runtime takes care of thread allocation and privatization. It also runs all threads concurrently. After the execution of the code, the *slave threads join* back to the master thread. The Figure 2.5 shows an implementation of SAXPY using OpenMP directives. Here, *fork* starts beginning of parallel construct, at the end of parallel constructs every threads *join* to the master thread.

Although OpenMP started with the simple idea of *fork-join*, it has now evolved in different directions such as task parallelism, SIMD, thread affinity and accelerator support. In this thesis, we implement and improve the OpenMP accelerator model.

2.3.1 OpenMP Accelerator Model

OpenMP announced support for accelerators starting from version 4.0. The model has been improved continuously since then. The current stable version is 4.5 which offers a programming interface that provides a set of directives to offload the execution of code regions onto accelerators, to map loops inside those regions onto the resources available in the device architecture, and to map and move data between address spaces.

The main directives are `target data` and `target`, which create the data environment and offload the execution of a code region on an accelerator device respectively. They offer a `map` clause to assign data with the region, and the directionality of data can be expressed as `to, from` or `tofrom` within the map clause. As an example, the `target` directive of the example in Figure 2.6 creates a data device environment including the arrays x and y, as specified by the map clauses. The to and tofrom clauses specify the directions of copy between host and device data environments for the two variables.

The specification also contains the `teams` directive to create thread teams. The programmer can control the number of teams and the maximum number of threads in each team by specifying the `num_teams` and `thread_limit` clauses along with the teams directive, respectively. In each team the threads other than the master thread do not begin execution until the master thread encounters a `parallel` region. The `distribute` directive specifies how the iterations of one or more loops are distributed across the master threads of all teams that execute the teams region. For instance, the for-loop in Figure 2.6 is parallelized firstly by the master thread of each teams and then by the threads within the teams.

```
1 void saxpy_openmp_gpu(float* x, float* y, float a, int N) {
2  #pragma omp target teams distribute parallel for \
3                   map(to: x[:N]) map(tofrom: y[:N])
4  for (int i = 0; i < N; ++i)
5   y[i] += a*x[i];
6 }
```

Figure 2.6: SAXPY Example with OpenMP's accelerator model

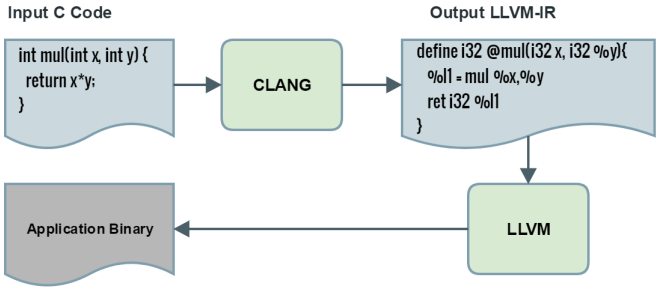


Figure 2.7: Compilation flow of Clang and LLVM

2.3.2 LLVM Compiler Infrastructure

The Low Level Virtual Machine (LLVM) began as research project with the goal of providing a modern, SSA-based compilation technology. The LLVM [21] is the main project of LLVM Compiler Infrastructure. It provides a source and target independent optimizer, along with code generation support for many target processors. It uses intermediate representations inside the compiler, which are called LLVM-IR. The input source code of LLVM is also LLVM-IR.

Clang Frontend

The Clang [22] has been developed as side project using the LLVM library and therefore follows most of its implementation guidelines. It is a front-end for C and C++. One of the key efforts is to ensure each (new) feature is structured into logical modules in order to ease the integration and reduce disruption caused by interdependencies. Following this modular design, each action accomplished by the consumers of each basic element (token, AST node) tends to be self-contained, either by extending a default action applied on top of a class of elements or by creating a new class.

Figure 2.7 illustrates the compilation flow of Clang and LLVM. Input C code is given to the Clang compiler; then Clang compiler generates LLVM-IR. The driver of Clang compiler passes the generated LLVM-IR to the LLVM compiler which does all optimizations and generates the actual binary code for the current architecture.

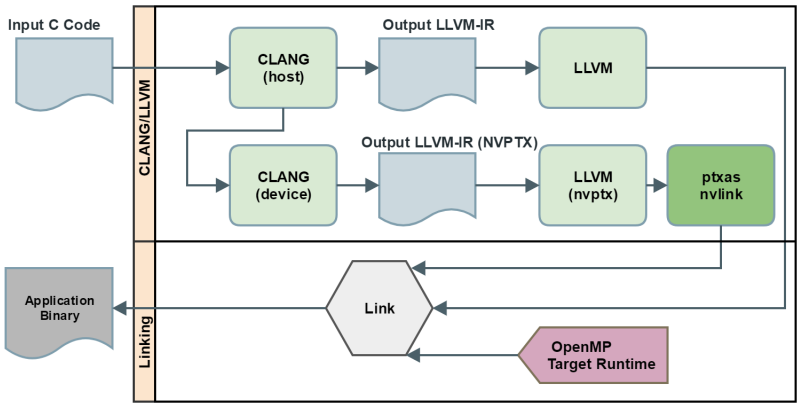


Figure 2.8: Compilation flow for OpenMP accelerator model with CLANG/LLVM

OpenMP Accelerator Model support in Clang

The Clang compiler includes the full provisions of the OpenMP 4.5 specification with offload support for NVIDIA GPUs. The initial work is explained in [28] and final work is presented in [11] where we have contribution in this thesis.

The compilation flow is shown in Figure 2.8. Input C/C++ code with OpenMP accelerator model directives is transformed into LLVM-IR by Clang. Clang generates LLVM IR code from its internal Abstract Syntax Tree representation. It is also responsible for implementing OpenMP constructs and transform code regions that are specified with OpenMP directives, and replaces them with the calls to the OpenMP runtime library. LLVM then transforms the output of Clang into the target assembly language.

For NVIDIA GPU targets, Clang does one additional compiler pass as shown in Figure 2.8. In this pass, LLVM generates PTX language, which is passed to ptxas and nvlink. These are tools that are provided by NVIDIA and transform PTX programs into the low-level native GPU assembly language called SASS, which is packed with CUDA binary ELF sections (also called cubin files).

2.4 OpenACC

OpenACC [13] provides directives that allow programmers to specify code regions to be offloaded to accelerator devices and to control many features of these devices explicitly. It was initially designed with accelerators like GPUs in mind, although it is also used to write parallel programs for multicore CPUs. It is an emerging GPU-based programming model that is working towards establishing a standard for directive-based accelerator programming. It has been started in collaboration between CAPS, CRAY, PGI, and NVIDIA. Using OpenACC allows users to maintain a single code base that is compatible

with various compilers, while the code is also portable across different possible types of platforms.

2.4.1 OpenACC Programming Model

The OpenACC API set includes directives and clauses which can be used in conjunction with C, C++, and Fortran code to program accelerator boards. It is compatible with OpenMP to provide a portable programming interface that addresses the parallelism in a shared memory multicore system as well as accelerators. However, there is no previous work combining the OpenMP device model with OpenACC.

OpenACC offers two types of compute directives *parallel* and *kernels*. The main construct is `kernels`, which instructs the compiler to transform the annotated code region to exploit the available parallelism in the device. With the *parallel* directive however, if there is any loop inside the following code block and the user does not specify any loop scheduling technique, all the threads will execute the full loop. OpenACC supports three levels of parallelism: *gang*, *worker* and *vector*. The *parallel* construct that launches *gangs* that will execute in parallel. Each of the gangs may support multiple workers that execute vector or SIMD constructs. A variety of clauses are provided to enable conditional execution, to control the number of threads, to specify the scope of the data accessed in the accelerator parallel region, and to determine if the host CPU should wait for the region to complete before proceeding with other work. It also allows asynchronous execution through an *async* clause that executes the host computation asynchronously and allows the user to synchronize using *wait*.

OpenACC offers the `data` and `update` constructs to manage data movement, and `parallel` and `loop` constructs for detailed control of kernel offloading and the parallel execution of loops.

The loop in Figure 2.9 uses OpenACC directives to identify the parallel loop, and includes copyin and copy data clauses that declare what data needs to be copied to device memory and what results need to be copied back if the loop is compiled for an accelerator device.

One of the biggest difference with OpenACC and OpenMP is that OpenACC standard gives great flexibility to the compiler implementation. It eases the burden for programmers while increasing portability.

```
1 void saxpy_openacc(float* x, float* y, float a, int N) {
2  #pragma acc parallel loop copyin(x[:N]) copy(y[:N])
3  for (int i = 0; i < N; ++i)
4   y[i] += a*x[i];
5 }
```

Figure 2.9: SAXPY Example with OpenACC

OpenACC	OpenMP
parallel	target
parallel	target teams
kernels	
loop gang	distribute
loop worker/vector	for/simd
data	target data
cache	
update	target update

OpenACC	OpenMP
async wait	wait
async	nowait
enter data	target enter data
exit data	target exit data
tile	
serial	
routine	declare target
declare	declare target

Table 2.2: Comparision of OpenACC and OpenMP device model in terms of directives

2.4.2 PGI Compiler Infrastructure

PGI provides commercial compilers for C++, C and Fortran (with CUDA Fortran) which include the OpenACC 2.6 directives, OpenMP 4.5 directives, and many other features [15]. The OpenACC directives grew out of the PGI Accelerator directives which were first introduced in 2008.

2.5 Conclusion

We have given an overview of the three main directive-based prgramming models for GPUs. OmpSs is state-of-art task-based parallel programming model which supports homogeneous and heterogeneous platform without having code generation support for them. It has been developed by Barcelona Supercomputing Center. It is forerunner of OpenMP in terms of task model. On the other hand, OpenMP and OpenACC are widely used APIs. The major directives among these models are summarized in Table 2.2. The OpenMP API covers only user-directed parallelization, where the programmer explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel. By contrast the OpenACC programming model allows the programmer to augment information available to the compilers, including specification of data local to an accelerator, guidance on mapping of loops onto an accelerator, and similar performance-related details. To summarise OpenMP is more prescriptive while OpenACC is more descriptive.

In the following chapter we first introduce device mode extensions for OpenMP. Then we focus on code transformations and optimizations to improve the overall performance.

3 MACC Infrastructure

3.1 Introduction

In this chapter, we present our MACC[1] infrastructure, which is a source-to-source code generation module for NVIDIA GPUs using OpenMP directives, and the associated code transformation algorithms. It is developed on top of the Mercurium C/C++ source-to-source compiler. Our aim is to let the compiler offload and parallelize code regions automatically under the control of user directives. This MACC is one of the main software contributions of this thesis and was developed during the early years of this Ph.D work.

From the perspective of a programming language, the MACC infrastructure combines the task model of OmpSs with the advantages of the OpenMP device model. At that time, OpenMP, which is the de facto standard for shared-memory programming, did not have support for GPU, however its draft version included their initial device model. Widely used programming models including OpenACC were relatively new in GPU area. Their models involved drawbacks in terms of asynchronous GPU kernel execution and data management for runtime; also there were intricate directives to get the best performance from a GPUs. On the other hand, the OmpSs programming model had great support for asynchronous kernel execution and data management, however OmpSs did not support GPU code generation. As a consequence, we came up with the idea of combining these two models to find the most efficient programming model for GPUs.

We have developed a device model based on OmpSs while aiming to discover novel code transformation methods that deliver the best performance. To do this we have extended the OpenMP specification in many different ways. At the same time we have streamlined it by removing the constructs which are not suitable for massive parallel

[1] MACC is an abbreviation of Mercurium ACCelerator Model

GPUs. We have researched several extensions to the OpenMP specification which are presented in Section 4 and Section 5. In this chapter, we introduce a new dialect that provides an effective solution for GPU programming. This chapter also present code transformation algorithms for for-loops based on a set of directives. Additionally, we propose several extensions for the device model that we later use for OpenMP and OpenACC. These are presented in detail in Chapter 4.

It is important to note that this chapter does not present a research contribution about compiler algorithms. However, in order to start researching code generation for GPUs, it was necessary for us to design a base infrastructure for it in a compiler. In this chapter, we outline the GPU infrastructure model that we designed. This chapter is intended to offer a high-level outline and skips over some low-level compiler-specific details. As a result it could be a good start point for the concept of GPU code generation in a compiler for those readers without expertise in the field.

3.2 Motivation

In recent years, the OpenACC standard has appeared with the aim of providing a higher-level directive-based approach for programming accelerator devices. In 2013 OpenMP also announced version 4.0, adding support for device models with the same objective of OpenACC. They further improved this model in version 4.5. The proposed directive based model allows the programmer to augment information available to the compilers, including specification of data local to an accelerator, guidance on mapping of loops onto an accelerator, and similar performance-related details. However, their solution still requires knowledge of GPUs from users in order to achieve best performance. They rely on the programmer for the specification of data regions, transfers between address spaces and for the specification of the computation to be offloaded in the devices; these solutions also put a lot of pressure on the compiler which has the responsibility of generating efficient code based on the information provided by the programmer. Thus, our first goal is to find out necessary functionalities and adapt them into our model to specify computation kernels in a more productive way.

The OmpSs proposal has been evolving during the last decade to lower the programmability wall raised by many-core CPUs/GPUs, using a task-based data flow approach in which the job of assigning tasks to different number devices, as well as managing the coherence of data in multiple address spaces, is delegated to the runtime system. Multiple implementations were investigated for the IBM Cell (CellSs [25]), NVIDIA GPU (GPUSs [26]) and homogeneous multicores (SMPSs [27]) before arriving at the current unified OmpSs specification and implementation. Initially, OmpSs relied on the use of existing CUDA and OpenCL to specify the computational kernels. Since OmpSs is quite promising for heterogeneous task orchestration, we decided to implement our MACC infrastructure on top of it. So our goal was to investigate a new dialect which is

combination between OpenMP device model and OmpSs.

Nowadays, open-source compiler technologies led by GCC (the GNU Compiler Collection) [29] and LLVM [21, 22] dominate in research and industry. Thanks to this, their maturities are pretty high. Both infrastructures cover all the compilation stages from the parsing of different programming languages to assembly code generation for several target architectures. However, researching these infrastructures would be non-trivial for a Ph.D. The learning curve is high because of the complexity of their components and the compiler topic itself. Also, evaluating new architectures often depends on the will of vendors to release a back-end for that particular compilation infrastructure as they are not commercial compilers which might lead to a long wait. Therefore, we chose to work on the Mercurium compiler which provides very agile prototyping while at the same time it taking advantage of any back-end compiler. More recently we started working on the LLVM and PGI compilers once they become mature for GPU architectures.

3.3 Objectives

The main goal of our MACC proposal is to boost the exploitation of GPUs by enabling the offloading and parallelization of code that is currently generated sequentially by the compiler. The key concept to achieve this goal is giving the programmer some of the responsibility for the offloading, parallelization process and data management that is currently assumed by the compiler. This allows the programmer to better guide the compiler parallelization process, indicating to the compiler which code regions are safe and should therefore be parallelized.

All the factors presented above motivated us to create a source-to-source compiler with code generation support for GPUs targeting loops and aimed at the fast prototyping of new compiler proposals. This infrastructure must have the following characteristics:

- **Discovering directives and clauses to extend specification:** Investigate new directives and clauses which can help the compiler to generate better code.
- **Code transformation:** Support essential code transformation for GPUs with the based on directives. Additionally find out new transformation methods that enhance performance.
- **New optimization techniques:** Discover GPU specific optimization methods, as this is a new area for compilers.

3.4 MACC Infrastructure

Our GPU code generation has three main phases: the *MACC lowering*, the *CUDA code generator*, and the *MACC device provider*. The primary objective of the *MACC lowerer* is to transform the input scalar IR into a parallelized CUDA-IR which is compatible. It passes

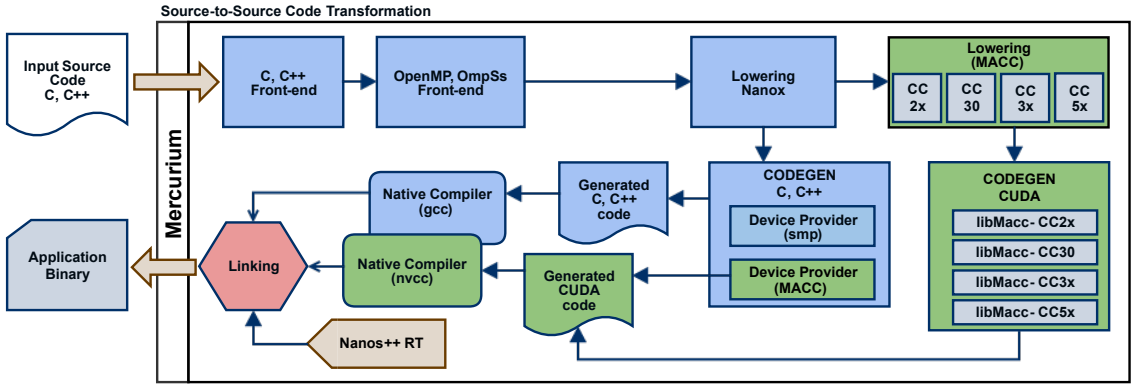


Figure 3.1: Compilation phases of MACC Compiler. Unlike Mercuirum compiler, it involves a brand new lowering phase, which is colored light green.

this CUDA-IR to the *CUDA code generator* phase, which generates the correct CUDA code for this CUDA-IR. Then, at some later point in the compiler pipeline, the *MACC device provider* generates kernel launch codes that invoke the generated CUDA kernel through *codegen CUDA*.

Figure 3.1 illustrates the new phases in the MACC Compiler, which are colored green. The *lowering Nanos* phase recognizes OpenMP and OmpSs constructs and replaces them with Nanos++ library calls. After that it passes the IR to the *codegen C/C++* phase to generate output C/C++ source code. As a result, it produces an output source file which is also called host code. Besides that, *lowering Nanos* also passes the IR to the *MACC lowering* phase. In this phase, the scalar IR is transformed into CUDA-IR according to the computation capability of the target NVIDIA GPU. Since each computation capability has different features, it might affect the code transformation pattern. We support cc2x for Fermi, cc30 and cc35 for Kepler, 5x for Maxwell and 6x for Pascal architectures. After that, the CUDA-IR is passed to the *CUDA code generator* to generate an output CUDA kernel source file which is also called a device file. In this phase, we inline a library for the device code that involves device specific functions such as reduction, atomic, etc. On the other side, the *macc device provider* inside the *codegen C/C++* phase generates CUDA kernel launch codes and adds them into device file. Both of the files are then compiled by native compilers such as gcc and nvcc. Finally, our compiler driver links all compiled files and the Nanos++ runtime library and produces an application binary.

3.5 Programming Model

In this section, we introduce constructs of our model, which is combination of OpenMP device model and OmpSs. These constructs allow programmers to specify and parallelize regions to offload and guide the compiler in the GPU code transformation.

3.5.1 Language Terminology

- **processor** Implementation defined hardware unit.
- **device** An implementation defined logical execution engine.
- **host device** The device on which the program begins execution.
- **target device** A device onto which code and data may be offloaded from the host.
- **acc target device** A GPU device onto which code and data may be offloaded from the host.

- **task** A specific instance of executable code and its data environment, generated when a thread encounters a `task` construct.
- **task region** a code region consisting of all code encountered during the execution of a *task*.
- **target task** A *task* generated when a `task` construct is used immediately after a `target` construct.
- **acc target task** A *target task* that is generated when `task` construct is used immediately after a `target` construct with `device` clause and `acc` device type. In this way, it is guaranteed that the *task region* will be offloaded on any *target device* in the system. It is the *target task* of MACC infrastructure.

- **host memory** The memory space of the `processor`.
- **device memory** The memory space of the `device`.
- **team memory** The memory space of the `team`.

3.5.2 Execution and Data Model

The MACC execution model is inherited from the OmpSs execution model, which we explained in Section 2.2.1. This allows it to take advantage of powerful asynchronous task execution support and seamless data management. The design of the OmpSs runtime is highly biassed to delegate most of the decisions to the runtime system, which is based on the task graph built at runtime (task data dependency clauses) and can schedule tasks in a data flow way to the available processors and accelerator devices and orchestrate data transfers and reuse among multiple address spaces. Our MACC assumes the OmpSs runtime dependency based parallel execution of (offloaded and non-offloaded) task instances, task scheduling and transparent management of (coherent or non-coherent) physically distributed address spaces. In our case, multiple address spaces of different GPUs are managed by the underlying runtime while using using task dependency clauses.

We propose an *acc target task* approach to replace the OmpSs execution model, which is one of the significant difference between MACC and OpenMP. In this approach, the `target` construct is always associated to a `task` construct to make it asynchronous. It does not have to be used with a `nowait` clause unlike OpenMP, because every *task*,

target task and *acc target task* is fully asynchronous in our model. Moreover, our runtime is capable of distributing the created *acc target tasks* among multiple *acc target devices* automatically without any hint from user. To conclude, we support only asynchronous task while aiming to reduce the complexity of managing multiple GPUs. This is the most distinct difference between our MACC execution model and OpenMP.

We provide automatic multiple/single *target device* management with a `device` clause within the `target` construct. Unlike OpenMP, we do not force programmers to set an integer value which creates a direct mapping between the `target region` with the physical GPU devices. This direct mapping makes it difficult to write applications that dynamically offload work to GPUs to achieve load balancing or adapt to device variability since it forces the programmer to embed in the application logic code to manage resources. Instead, we use an `acc` device-type to specify the type of accelerator that the region can run on and make use of `device types` to specify what to execute on the GPU. We then rely on the compiler to generate the kernel code to be executed on the device. We call *acc target device* when a `device` clause is used with an `acc` device type. Additionally, any other device types which OmpSs already supports (Section 2.2.1) can be used together with our model. This approach gives responsibility to the runtime to choose any physical GPU devices in the system automatically without user interaction.

The task's data environment can be specified using the `in, out` and `inout` clauses of the `task` construct (like OpenMP's `depends` clause). Our runtime guarantees the task's data environment to make ready on the *acc target device* when the task is executed. The data environment can be copied from the *host device* or another *acc target device*. Lastly, these clauses enforce dependency constraints on the scheduling of tasks. These constraints establish dependencies only among tasks. When additional data is required to be copied apart from a task dependency, it can be specified by the `copy_in, copy_out` and `copy_inout` clause of the `target` construct. The `target data` and `target update` constructs are not used as the actual data movement is automatically implemented by the data-flow. Ultimately, our model greatly facilitates data management among the *host device* and single or multiple *acc target device*s.

Figure 3.2 shows an example code using our MACC infrastructure on the top and its associated execution diagram on the bottom. There are five *task*s specified; the device type of 1st task is `cuda`, the 2nd, 3rd and 4th ones are `acc`, and the last one is an `smp` type. In addition, there is a `taskwait` construct at the end of program to make sure that all the tasks finish before the program is over. For the 1st task, MACC infrastructure seamlessly launches an optimized cuda kernel in line 10. Then, it makes use of the GPU code generation facility for the 2nd, 3rd and 4th tasks. Lastly, for the 5th task it creates a new *task* on the *host device*. Now let us explain our execution model which is shown on the bottom of the Figure. For convenience, let us assume there are two GPUs in this system. Based on this information, our runtime will firstly execute the 1st task and make its data environment ready on any available GPU in the system. In our example, the

first GPU is chosen; the runtime copies array **C** as it is specified as input dependency and array **D** is allocated on the GPU. Then, it executes the 2nd task on the second GPU concurrently since there is no dependency between the 1st and 2nd tasks. Right after that it schedules the 3rd task on second GPU due to dependency between the 2nd and 3rd tasks. As it would be inefficient to move data to another GPU, it uses the second GPU; the OmpSs runtime is locality-aware among *target devices*. Since there is a for-loop with a size 2 in the input code, it executes the 1st, 2nd and 3rd tasks twice. When the program encounters the 4th task, the runtime chooses one of the available GPUs to run the task. It is important to note that the 4th task has dependencies on the 1st, 2nd and 3rd tasks. In our case, the runtime chooses the first GPU; therefore the necessary data environment, which is array **B**, is copied to the first GPU from the second GPU. When the program reaches the 5th task, the runtime makes ready its data environment by copying the arras from first and second GPUs; then it executes the 5th task on the *host device*.

3.5.3 Memory Model

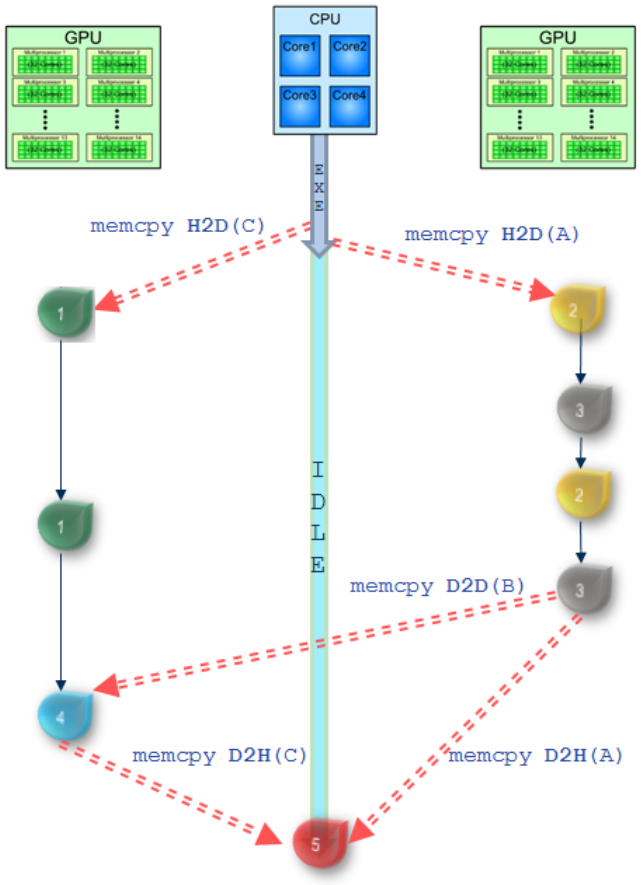
OmpSs provides a relaxed-consistency model among *task*s. All *task*s have access to a place to store and to retrieve variables, called the *host memory*. Additionally, each of them can have a private memory area. Any access by one task to the private memory area of another task results in unspecified behavior.

To implement the device model, we need to expand the current memory model of OmpSs as we include GPU devices. In our memory model, all *acc target task* have their own memory that is called *device memory*, which is accessible by *host threads* via dependency clauses of the `task` construct or by copy clauses of the `target` construct. Furthermore, the *acc target task* incorporates one or more `teams` with a *team memory* for each. The *team memory* is a team private memory, which is accessible only for the threads within a given team; there is no direct access for *host threads* and threads within other teams to a *team memory*. Figure 3.3 illustrates the memory model of our MACC infrastructure with a system that has two GPU device. In this example for clarity we only show four *host thread*s and three `teams`.

In our MACC infrastructure we map *device memory* with the global memory of the GPUs and *team memory* with shared memory which is on-chip memory. It is important to note that *team memory* becomes the fastest memory area after register file of the model. Using shared memory efficiently is a well-known challenge for the native and user-directed compiler. We introduce a couple of extensions such as complete and partial array privatization to better exploit shared memory in Section 4.

Input Code

```
1  #pragma omp target device(cuda)
2  #pragma omp task inout(C) out(D)        TASK 1
3  __global__ void cuda-kernel(double C, double D) {
4    cuda-codes
5  }
6  int main(int argc char **argv) {
7   double A[N], B[N], C[N] , D[N];
8
9   for(int i = 0; i < 2; ++i)  {
10   cuda-kernel(C, D);
11
12   #pragma omp target device(acc)
13   #pragma omp task in(A) out(B)          TASK 2
14     structured-block
15
16   #pragma omp target device(acc)
17   #pragma omp task inout(A, B)           TASK 3
18     structured-block
19  }
20  #pragma omp target device(acc)
21  #pragma omp task inout(C, B) in(D)      TASK 4
22    structured-block
23
24  #pragma omp target device(smp)
25  #pragma omp task in(A, C)               TASK 5
26    structured-block
27
28  #pragma omp taskwait
29 }
```



Execution Diagram

Figure 3.2: Execution model example of MACC infrastructure.

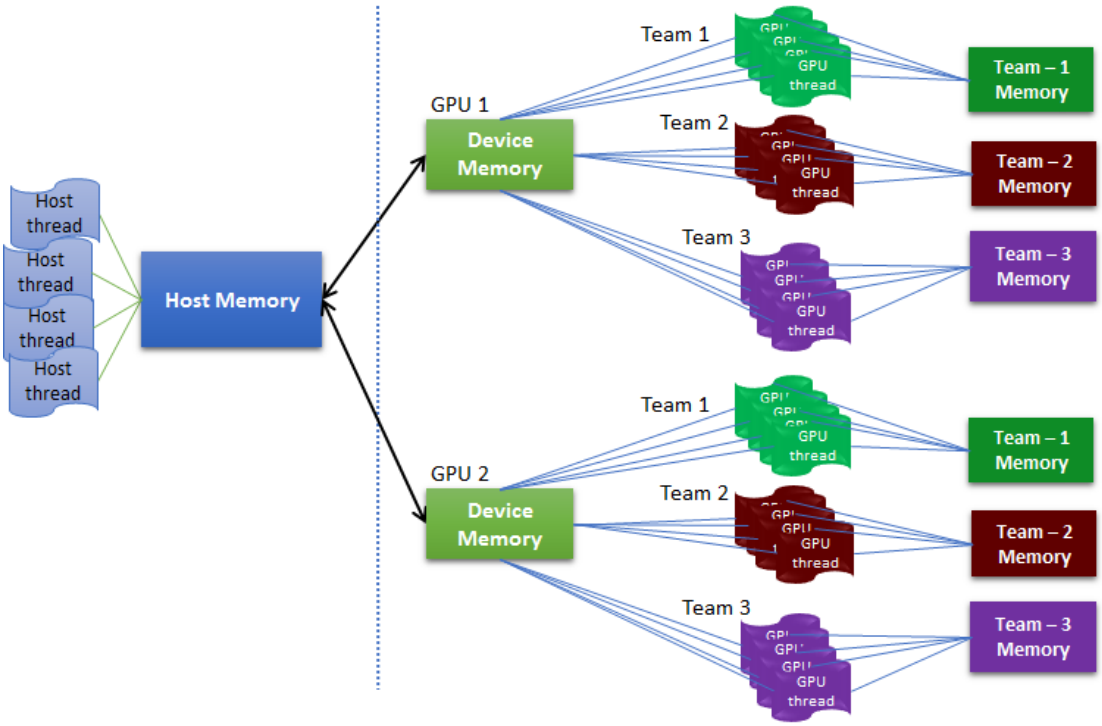


Figure 3.3: Memory model of MACC programming Model

3.5.4 Threading Model

OmpSs uses a thread pool mechanism; several threads are created to run the tasks on the different execution units. Threads that run tasks on CPUs are called worker threads and the threads used to manage and offload tasks to accelerators are called helper threads and run on the CPU as well. In our MACC infrastructure, we use *acc target task*s which are run on GPUs. In order to manage this, the runtime creates one helper thread for each GPU devices. As the OmpSs application binary is executed, task creation codes will be reached and the runtime's dependency layer will create a new node in the task dependency graph for each created task. When the task becomes ready, the runtime scheduler component decides which execution unit will run the task. Then the architecture support for that unit and the coherence layer coordinate the necessary data transfers and run the task.

3.5.5 Device Constructs

In this section, we introduce the constructs of our MACC infrastructure that make use of the code generation facility for GPUs. As we already mentioned, it is combination and extension of the OmpSs and OpenMP programming models.

```
1 #pragma omp target device(acc) [clause [clause], ...] new-line
2 #pragma omp task [clause [claues], ...] new-line
3   structured-block
```

Figure 3.4: C/C++ syntax of the *acc target task*

`target` Construct

The `target` is the main construct in our device model. The construct is used to offload a code region; these regions can be any structured block except for function declarations and definitions. To take advantage of the GPU code generation facility, it must be in the form of a *acc target task*, which uses both the `target device (acc)` and `task` constructs.

Figure 3.4 shows the syntax of the *acc target task* for C/C++ which offloads the following structured-block while enabling GPU code generation. In addition, the data needed for the task is guaranteed to be ready on the offloaded physical device when runtime runs it.

`teams` Construct

The `teams` construct creates a league of thread teams and a master thread for each thread team executes the code region within the construct. It is used to specify a hierarchy of resources in the GPUs: a league of `num_teams` clause teams, each with `thread_limit` threads. It is important to note that teams are not connected directly, which is to say threads in different teams are not allowed to communicate in any native way. It is also not possible to express a barrier between threads in different teams, and there is no implicit barrier at end of the construct. Lastly, **teams** are not allowed to be nested. In our model, the `teams` construct must be contained in a *acc target task*; there must not be any statements among `target`, `task` and `teams`.

In addition to this, the `teams` construct involves data sharing attribute clauses which are: `private`, `firstprivate`, and `shared`. As we discussed in Section 3.5.3, being able to use data sharing clauses for a team means that there is a special memory space for the team, which we called the *team memory*. We explain how we exploit from *team memory* in Section 4 as an optimization proposal.

The code in the Figure 3.5 (a) shows usage of the `teams` construct within a *acc target task*. In this example, the structured block is executed by each master thread of the thread team sequentially. The compiler is free to create up to 16 teams and 32 threads within each team as is indicated in the `num_teams` and `thread_limit` clauses respectively.

Input code | **Generated code**

a. Example with `teams` construct

```
#pragma omp target device(acc)
#pragma omp task
#pragma omp teams \
   num_teams(16) thread_limit(32)
{
  //team sequential region
}
```

```
__global__ void kernel1() {
 if(threadIdx.x == 0 ) {
   //sequential region
 }
}
// kernel launch
kernel1 <<< 16,32 >>> (a, b, c);
```

b. Example with `distribute` construct

```
#pragma omp target device(acc)
#pragma omp task \
   in(a[:N], b[:N]) out(c[:N])
#pragma omp teams \
   num_teams(16) thread_limit(32)
#pragma omp distribute parallel for
for(int i = 0; i < N; ++i) {
   c[i] = a[i] + b[i];
}
```

```
__global__ void
kernel2(T* a,T *b,T* c){
 if(threadIdx.x == 0)
  for(int i = blockIdx.x;
      i < N; i+= gridDim.x)
       c[i] = a[i] + b[i];
}
//kernel launch
kernel2<<<16,32>>>(a,b,c,N);
```

c. Example with `parallel for` construct

```
#pragma omp target device(acc)
#pragma omp task \
   in(a[:N], b[:N]) out(c[:N])
#pragma omp parallel for
for(int i = 0; i < N; ++i) {
   c[i] = a[i] + b[i];

}
```

```
__global__ void
kernel3(T* a,T *b,T* c){
  for(int i = threadIdx.x;
   i < N; i+= blockDim.x)
       c[i] = a[i] + b[i];
}
// kernel launch
kernel3<<<1, tdim(N)>>>(a,b,c,N);
```

d. Example with several constructs

```
#pragma omp target device(acc)
#pragma omp task in(Anew) out(A)
#pragma omp teams
#pragma omp distribute
for(int j = 0; j < N; ++j) {
  #pragma omp parallel for \
                collapse(2)
  for(int i = 0; i < M; ++i) {
   for(int k = 0; k < L; ++k) {
     A[j][i][k] = Anew[j][i][k];
   }
  }
}
```

```
__global__ void
kernel4(T* A, T* Anew) {
  for(int j = blockIdx.x;
   j < N; j += gridDim.x)
   for(int i = threadIdx.x;
    i < M; i+= blockDim.x)
    for(int k = threadIdx.y;
     k < L; k+= blockDim.y)
     A[j][i][k] = Anew[j][i][k];
}
//kernel launch
kernel4 << bdim(N), tdim(M, K) >>>
             (a, b, c, N, M, L);
```

Figure 3.5: Examples of device constructs and their generated codes by MACC infrastructure.

`distribute` Construct

The `distribute` construct is associated with a loop and can be used to partition it into chunks which are assigned to teams, which is done by the team master of each team. It must be enclosed within a `teams` construct.

The code in Figure 3.5 (b) shows usage of a `distribute` construct. First of all, the *acc target task* requires a data environment; when the runtime runs the task, the data must be ready on the physical device. When compiler reaches the `distribute` construct, it will distribute the iterations on the master thread of each team.

`parallel` Construct

Parallel is the fundamental construct of the OpenMP model that starts parallel execution. We adapted a similar approach for our model. Essentially, when the program encounters a `parallel` construct, it creates a team for that thread. There is an implicit barrier at the end of the parallel construct. Only the master thread resumes execution beyond the end of the parallel construct, resuming the task region that was suspended upon encountering the parallel construct. Any number of parallel constructs can be specified in a single program.

`for` Construct

The `for` directive, which is also know worksharing construct in OpenMP, distributes the execution of the region among the other threads of the team that encounters it. Threads execute portions of the region in the context of the implicit tasks each one is executing. In case there is only one thread in a team, the region is not executed in parallel.

The code in Figure 3.5 (c) shows an example that makes use of **parallel for**. This example is similar to the example in Figure 3.5 (b). The only difference is that here the iterations are distributed across the threads.

`atomic` Construct

We also integrated the `atomic` construct to allow user to specify if a storage location is accessed atomically, rather than exposing it to the possibility of simultaneous read or write by threads which may result in wrong values.

Combined Constructs

teams distribute The `teams distribute` construct is a shortcut for specifying a teams construct containing a distribute and no other statements.

teams distribute parallel for The `teams distribute parallel for` construct is a shortcut for specifying a teams construct containing a distribute parallel loop construct and no other statements.

parallel for The `parallel for` construct is a shortcut for specifying a parallel construct containing one loop construct with one or more associated loops and no other statements.

3.6 Code Transformations

In this section, we explain code transformation algorithms with examples.

3.6.1 Kernel Configuration

When generating kernel code, MACC needs to decide: 1) the dimensionality of the resources hierarchy (one-, two- or three-dimension teams and threads) and 2) the size in each dimension (number of teams and threads). In order to support the organization of the threads in multiple dimensions, MACC supports to use of a `collapse` clause which includes an integer to specify the number of nested loops (dimensionality equals the nesting degree). Since GPUs offer maximum three dimensional thread blocks, the compiler can collapse maximum three nested loops and assign them to each dimension of a thread block.

The MACC currently generates one-dimensional teams (the current implementation does not support nesting of distribute directives). Thread dimensions are initially assigned in loop nesting order.

3.6.2 Loop Transformation

The MACC takes into account the restrictions of the device (for example a maximum number of blocks and threads for each CUDA computing capability) and the information provided by the programmer in the `num_teams` and `thread_limit` clauses; if not specified, the MACC first tries to simply assign one iteration per block and one iteration per thread. If the iteration count is bigger than allowed by the maximum value of iteration thread or block, the MACC generates a loop that iterates the loop until it finishes in the output CUDA kernel. We discuss loop scheduling techniques and a novel contribution to this thesis, the dynamic loop scheduling method in Section 6.

Our MACC's code generation does not break the coalesced memory access pattern to read or write the global memory efficiently by warps. To that end, MACC performs a cyclic mapping of loop iterations and tries to eliminate redundant "one iteration" loops and simplifies increment expressions for induction variables to improve kernel execution time.

The code in Figure 3.5 (d) shows an example that makes use of several constructs from our device model. The compiler starts by distributing the first loop across thread blocks. Then it distributes the second and third loops across threads using the 1st (threadIdx.x) and 2nd (threadIdx.y) dimensions of the thread block respectively.

3.6.3 Reduction Transformation

The `teams, parallel` and `for` constructs allow us to create a `reduction(reduction-identifier: list)` clause. Therefore we have implemented reduction as well for the device model. When the MACC compiler encounters this clause, it generates reduction code. We can divide reduction in two parts; 1) reduction within a thread team, 2) reducing item with thread teams. The next section explains the code generation algorithms for the `reduction` clause in detail.

Reduction within a thread team

Figure 3.6(a) and (b) shows examples of reduction occurring only within a thread team. There are different strategies to parallelize in this case, as shown in the generated CUDA codes on the right side of the same figure. Figure 3.6(a) has a single thread team in the kernel; Figure 3.6(b) incorporates multiple thread teams. However, reduction is specified in a `parallel for` construct, which tells the compiler to do reduction by threads of a thread team.

We take advantage from *shfl* intrinsic to implement our reduction algorithm within a block for Kepler and later GPU architectures. The full reduction algorithm is explained in [30]. In brief, *shfl* enables a thread to directly read a register from another thread in the same warp. It allows threads in a warp to collectively reduce the reduction value. The following code example shows reduction within a warp. In our algorithm, we first reduce within warps. Then the first thread of each warp writes its partial reduction item to shared memory. Finally, after synchronizing, the first warp reads from shared memory and reduces again. The algorithm is implemented in the **blockReduceSum** function. It is important note that *shfl* is not available for the Fermi architecture. Thus, when the compiler needs to generate code for the Fermi architecture, it generates codes using reduction using shared memory. As we mention in Section 3.4, our compiler uses different code generators to generate suitable code for each GPU architecture.

```
1 for (int i = warpSize/2; i > 0; i /= 2)
2   reductionItem += __shfl_down(reductionItem, i);
```

Reduction with thread teams

The example of `reduction` with thread teams is shown in the code on the bottom of the Figure 3.6(b). Here `reduction` applies on the entire grid, which should be handled by threads and thread teams.

In our device model, we map each `thread teams` to each thread block in CUDA and there is no synchronization mechanism to synchronize all thread blocks. Our strategy is to take advantage of the atomic operations of CUDA. In this way, the compiler generates code to reduce the reduction item by thread block using **blockReduceSum**. Then, the first thread of each thread block reduces the partial reduction using atomic operations as shown in Line 9 on the right side of the same figure.

3.7 GPU Device Model Challenges

In the OpenMP device model, execution in the teams region initially starts in the master thread of each team. As we integrate this `teams` construct into our device model, we inherited its execution model as well. However in the GPU programming model, all threads are immediately and actively available when the kernel is launched. Because of this we have to control the `teams` region with the compiler. Our idea is to coordinate thread execution: the master thread of each team is used to execute each `team`'s specific region, which is guarded by an if statement. When code reaches a `distribute` or `parallel` construct later on, it activates the rest of the threads in the team. However, this can be tricky sometimes, especially when data needs to be created or if a statement must be executed by the master thread of thread team.

Figure 3.7 (a) shows the first challenge with the `teams` construct. In this example, `alpha` is created and assigned with a value inside the team region. Once the code reaches the for loop in Line 10, `alpha` is used by each thread of the thread team. In other words, alpha must be broadcast to all the threads by the master thread of the thread team. The generated kernel is showed in the same figure. We solve this issue by increasing the complexity of the CUDA code. We get the master thread of thread team to create `alpha` in shared memory. When the master thread is done with the memory operation, we enforce a CUDA ordering constraint on memory operations issued before and after the barrier instruction by using a **threadfence_block**. Afterwards, each thread reads `alpha` from shared memory safely. Unfortunately this is an implementation challenge

a. Reduction within thread team

```
#pragma omp target device(acc)
#pragma omp task in(a[:N])
#pragma omp parallel for \
            reduction(+:sum)
for(int i = 0; i < N; ++i)
{
  sum += a[i];

}
```

Generated Code

```
__global__ void
kernel1(T *a, int N, T* sum) {
 int m_sum = 0;
 for(int i = threadIdx.x;
      i < N; i+= blockDim.x)
        m_sum += a[i];
 int res = blockReduceSum(m_sum);
 if(threadIdx.x == 0) *sum += res;
}
```

b. Reduction within thread team

```
#pragma omp target device(acc)
#pragma omp task in(a[:N])
#pragma omp teams distribute
for(int i = 0; i < N; ++i) {
  int sum = 0;
  #pragma omp parallel for \
              reduction(+:sum)
  for(int i = 0; i < N; ++i) {
    sum += a[i];
  }

  b[i] = sum;
}
```

Generated Code

```
__global__ void
kernel2(T *a, int N, T* sum) {
 for(int i = blockIdx.x;
      i < N; i+= gridDim.x) {
    int sum = 0, m_sum = 0;
    for(int i = threadIdx.x;
      i < N; i+= blockDim.x) {
        m_sum += a[i];
    }
    int res = blockReduceSum(m_sum);
    if(threadIdx.x == 0) b[i] = res;
 }
}
```

Reduction with thread blocks

```
#pragma omp target device(acc)
#pragma omp task in(a[:N])
#pragma omp teams distribute \
            parallel for \
            reduction(+:sum)
for(int i = 0; i < N; ++i)
{

  sum += a[i];

}
```

Generated Code

```
__global__ void
kernel3(T *a, int N, T* sum) {
 int m_sum = 0;
 for(int i=threadIdx.x+blockDim.x*
   blockIdx.x;
   i < N;i+= blockDim.x*gridDim.x)
        m_sum += a[i];
 int res = blockReduceSum(m_sum);
 if(threadIdx.x == 0)
   atomicAdd(sum, res);
}
```

Figure 3.6: Examples with `reduction` clause

for compilers since there is team sequential region.

To understand in more detail, we introduce a further example in Figure 3.7 (b). A team sequential region takes decisions on what control-flow path is to be followed, and ultimately which parallel region is to be executed. In this scheme, the compiler generates code from a single code section that is split in two parts: 1) team sequential region for if-cond; 2) the execution phase for `distribute parallel for` regions. The generated code is shown on the right side of the same figure. The compiler generates lines 2 - 8 for the team sequential region. A variable shared in Line 2 by a thread in the same team is allocated into shared memory and used by team masters to guide the execution of all threads later on. This part is executed by the master of the thread teams, which is the first thread in our case. After this phase, a memory fence and synchronization guarantees that non-master threads wait for the masters to take decisions on their behalf. Once the master threads of each team finish this phase, all threads can proceed to the next phase. Here, based on the decisions taken in the previous phase, the threads execute one of the parallel regions. As seen, the team sequential region increases the complexity of the generated CUDA code. Even though the input code seems very neutral, the compiler has to generate inefficient code as there is no pattern for GPUs.

A recent study investigated another solution for these challenges in the OpenMP device model of the Clang front-end. This is called *control-loop* [11, 28]. Here the compiler generates a `while` loop to mimic a state machine that handles codes regions such as team, thread, sequential etc. The aim of doing that is that the control-loop scheme does not affect the core code generation scheme of Clang. Their approach solves the problem perfectly, however it results in poorer performance than our approach.

3.8 Conclusion

In this chapter, we introduced our MACC infrastructure, source-to-source code generation scheme for NVIDIA GPUs infrastructure designed and implemented in the Mercurium C/C++ source-to-source compiler. It is one of the the largest piece of software developed during this Ph.D. It is also used as the basis for the two following contributions of the thesis (presented in Chapter 4 and Chapter 5).

a. Input Code

```
#pragma omp target device(acc)
#pragma omp task in(a) inout(c)
#pragma omp teams
{

 // team sequential region
 float alpha = 2.0;

 #pragma omp parallel for
 for(int i = 0; i < N; ++i)
 {
   y[i] += alpha * x[i];
 }

}
```

a. Generated Kernel

```
__global__ kernel_1(...) {
  __shared__ float macc_sh_1;
  float macc_local_1;
  if(threadIdx.x == 0) {
   macc_sh_1 = 2.0;
   __threadfence_block();
  }
  __syncthreads();
  macc_local_1 = macc_sh_1;
  for(int i = threadIdx.x;
   i < N;
   i += blockDim.x ) {
    y[i] += macc_local_1 * x[i];
  }
}
```

b. Input Code

```
#pragma omp target device(acc)
#pragma omp task \
            in(a, x, d) inout(b, y)
#pragma omp teams
{
 // if-cond in team sequential region
 if(++d[0] > 0)
 {
  #pragma omp distribute parallel for
  for(int i = 0; i < N; ++i)
  {
    y[i] += alpha * x[i];
  }
 }
 else {
  #pragma omp distribute parallel for
  for(int i = 0; i < N; ++i)
  {
   b[i] += alpha * a[i];
  }
 }
}
```

b. Generated Kernel

```
__global__ kernel_2(...) {
 __shared__ int macc_sh_1;
 int  macc_local_1;
 if(threadIdx.x == 0) {
  macc_sh_1 = ++d[0] > 0;
  __threadfence_block();
 }
 __syncthreads();
 macc_local_1 = macc_sh_1;
 int tid =
   threadIdx.x+blockDim.x*BlockIdx.x;
 if(macc_sh_1) {
  for(int i = tid;
  i < N; i += gridDim.x*blockDim.x)
   y[i] += alpha * x[i];
 }
 else {
  for(int i = tid;
  i < N;
  i += gridDim.x*blockDim.x)
   b[i] += alpha * a[i];
} }
```

Figure 3.7: Device model challenging examples for GPUs.

4 Device Model Extensions for OpenMP and OpenACC

4.1 Multiple Target Code Generation

In this chapter we propose an extension to the directive-based programming models to support multiple-target task sharing, i.e. the possibility of sharing the execution (of multiple instances) of a task on different devices. We also analyze its implementation in the compiler and runtime system and evaluate its performance in a prototype implementation in the OmpSs programming model. The proposed extension eases the use of multiple accelerators in conjunction with the vector and heavily multithreaded capabilities in multicore processors without any code modification. The compiler is responsible for the generation of device-specific code for each device kind, delegating to the runtime system the dynamic scheduling of tasks to each available device. The new proposed clause conveys useful insight to guide the scheduler while keeping a clean, abstract and machine independent programmer interface.

The proposal is evaluated using three kernels (N-Body, tiled matrix multiply and Stream benchmark) on ARM, Intel x86 and IBM Power-8 based systems, resulting in speed–ups in the 8-20% range compared to versions in which only the GPU is used, reaching 96% of the additional peak performance thanks to the reduction of data transfers and the benefits introduced by the OmpSs NUMA-aware scheduler.

4.1.1 The Proposal of Multi Target Approach

In this section we propose an extension to the `target` directive to provide support for multiple target task sharing and comment on its implications for the implementation of the compiler and runtime system.

```
1 #pragma omp target device( list[ device-name ] ) \
2                     resources(list[ device-name :  percentage-integer-expression])
3 #pragma omp task
4  structured-block
```

Figure 4.1: Usage of Multi Targeting support

Target Directive Syntax Extension

We propose to extend the `target` construct in two complementary ways: (i) by allowing the specification of multiple device kinds in the `device` clause; and (ii) with a new `resources` clause to give hints to the runtime system to appropriately balance the scheduling of tasks to the different devices in the system. Figure 4.1 shows the proposed syntax extension.

When more than one device type is listed in the `device` clause, the compiler will have to generate a different code version for each of them. This requires mapping the generic thread hierarchy in the OpenMP 4.0 accelerator model to the actual thread hierarchy in the device. At runtime, the information provided in the `resources` clause will be used to decide where to schedule the execution of the next task instance. For each device the programmer specifies a value (or expression) over 100 which indicates the amount of "tokens" consumed every time a task is scheduled on that device; once a task finishes its execution, that number of tokens is restored. If at any time the number of tokens available is not sufficient, the runtime will not be able to schedule the task to that device, choosing a different device that requires less resources if possible.

As an example, the upper part in Figure 4.2 shows the main loop in an N-Body simulation kernel. In this example, the programmer specifies in Line 4 that the target region can be executed both on the host (`smp`) and the accelerator (`acc`). In addition, the programmer specifies the number of tokens consumed/released every time a task is scheduled to execute or finishes its execution on every possible device: 1 token (over 100) when task is offloaded to the accelerator or 40 tokens (over 100) when executed in the host. In this case the programmer expresses that no more than 2 tasks should be scheduled to be executed in the host at any time. The combinations can be estimated considering separate performance of multiple targets after some initial experiments.

Compiler Support to the device Clause

Figure 4.3 shows the compilation pipeline in the MACC compiler once the Mercurium Intermediate Representation (IR) has been generated. Different device-specific IR lowering phases can be implemented, each one either transforming the IR (e.g. for the `smp` device by inserting the appropriate calls to the OmpSs runtime system) or generating an output file to be compiled by a device-specific native compiler (e.g. CUDA for the

All-Pairs N-Body Simulation with $\mathcal{O}(n^2)$ Complexity

```
1  //N-Body Computation
2  for (int ii = 0; ii < n; ii+=BS)
3  {
4   #pragma omp target device(acc,smp) resources(acc:1, smp:40)
5   #pragma omp task in([n]px,[n]py,[n]pz) inout(pvx[ii;BS],pvy[ii;BS],pvz[ii;BS
      ])
6   #pragma omp teams
7   #pragma omp distribute parallel for
8   for (int i = ii; i < ii+BS; ++i) {
9    float Fx = 0.0f; float Fy = 0.0f; float Fz = 0.0f;
10
11   #pragma omp simd reduction(+:Fx,Fy,Fz)
12   for (int j = 0; j < n; j++) {
13     float dy = py[j] - py[i];
14     float dz = pz[j] - pz[i];
15     float dx = px[j] - px[i];
16     float distSqr = dx*dx + dy*dy + dz*dz+CONST;
17     float invDist = 1.0f / sqrtf(distSqr);
18     float invDist3 = invDist * invDist * invDist;
19     Fx += dx * invDist3; Fy += dy * invDist3; Fz += dz * invDist3;
20   }
21   pvx[i] += dt*Fx;  pvy[i] += dt*Fy;  pvz[i] += dt*Fz;
22  }
23 }
```

Transformed ACC-Task Code

```
1  #pragma omp teams
2  #pragma omp distribute parallel
     for
3  for (int i ...) {
4     for (int j ...) {
5        ...
6     }
7  }
```

Transformed SMP-Task Code

```
1  #pragma omp parallel for schedule(
     runtime)
2  for (int i ...) {
3     #pragma omp simd [ clause ...]
4     for (int j ...) {
5        ...
6     }
7  }
```

Figure 4.2: N-Body example of MACC IR Code Transformation

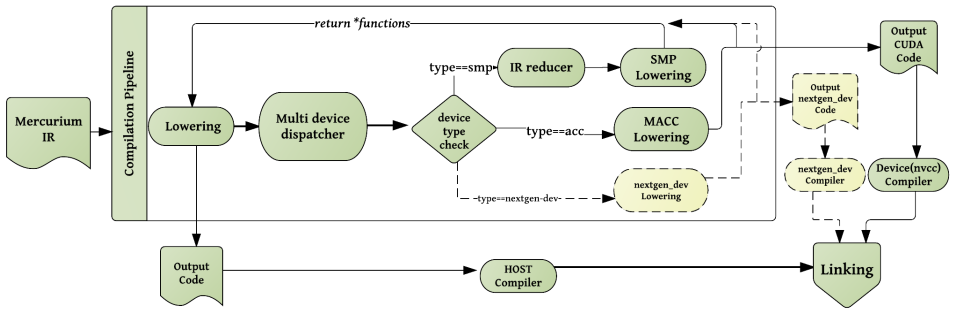


Figure 4.3: Overview of device dispatcher and IR lowering units.

`acc` device). The *multiple device dispatcher* unit is in charge of forwarding a new copy of the IR for each device type listed in the `device` clause to the appropriate lowering phase. The implementation is extensible as shown by dotted lines and the *nextgen_device* lowering phase. At the end of the compilation pipeline, the compiler driver compiles each output file with appropriate back-end compiler and link object files to generate the final executable file.

Before the execution of the *SMP lowering* phase it is necessary to execute an `IR reducer` phase. This phase is in charge of adapting the thread hierarchy supported by the OpenMP 4.5 accelerator model (teams and threads) to the flat thread model in the host, as shown in Figure 4.4. This step basically selects the outermost loop affected with a `distribute` or `parallel for` directive, and transforms it into a `parallel for` directive with `runtime` schedule type, which is set to *(dynamic, iteration_count / omp_get_thread_num())*. Other directives in the `target` region are ignored, except for `simd` constructs which are then lowered to specific SIMD operations in the host.

The lower part in Figure 4.2 shows how the directives are interpreted for each device type in order to adapt the generic thread hierarchy to each specific device: `acc` on the left and `smp` on the right.

Compiler and Runtime Support for the `resources` Clause

For the `resources` clause, the compiler just parses the two fields for each device kind and passes this information to the runtime system though an internal runtime call. This information is used by the runtime system to account for the total number of resource "tokens" available at any time. When a task is ready for execution, the runtime checks if enough tokens are available for any of the possible target devices; if so, then the runtime subtracts the specified resource tokens for the selected device from the currently available tokens. When the task finishes its execution, the runtime adds the same amount of tokens to the total count. Both operations are done using *atomic* operations. The OmpSs runtime also offers a call to initialize the total number of tokens to a certain value.

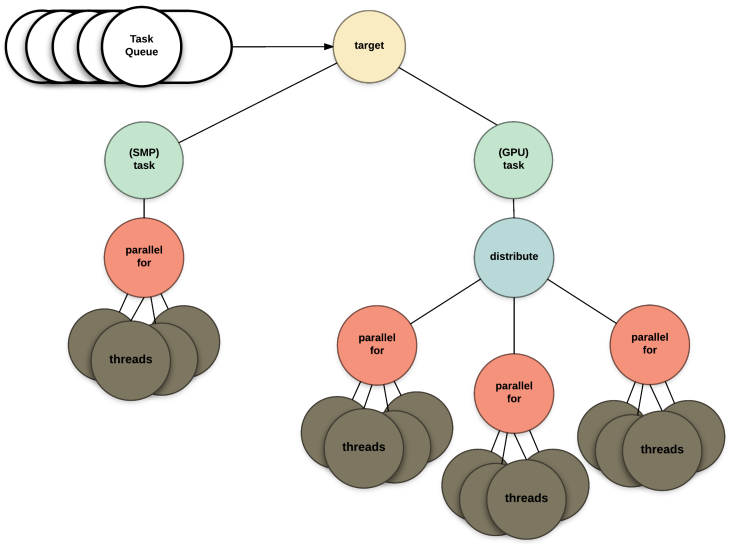


Figure 4.4: Overview of Automatically Transformed Thread Hierarchy

4.1.2 Experimental Evaluation

In this section we present the performance evaluation of the multiple targeted task-sharing proposal and its implementation in the MACC compiler and OmpSs runtime system. To that end we use a variety of system configurations and three small kernel applications: N-Body, tiled matrix multiply and the Stream benchmark.

System Configurations

Table 4.1 shows the main characteristics of the four systems that have been used for the experimental evaluation of the proposal. The different system configurations offer different ratios between the performance of the host and the performance of the accelerator devices.

The 1st system is based on an old generation of Nvidia GPUs (Fermi architecture) while the 2nd and 3rd systems are based on a more recent Nvidia GPU (Tesla K40). The first two systems are based on Intel hosts while the 3rd system is based on the emergent IBM Power8 architecture with high memory bandwidth and increased hardware thread counts. Finally the 4th system is based on ARM SoC with a tiny GPU which includes just one Streaming Multiprocessor Architecture (SMX).

All CUDA codes in this section, have been automatically generated by the MACC compiler and compiled with *nvcc* v7.0, except for the 4th system which makes use of v6.0. GCC 4.9 is used to compile host codes on all systems with -O3 optimization level. The `simd` construct is in OpenMP and auto-vectorization is performed by the back-end GCC compiler.

System	Processor	Memory	Nvidia GPU
1	*2 x Intel Xeon(TM) E5649 sockets 6-core/socket at 2.53GHz*	*24 GB*	*2 x Tesla M2090 (Fermi, 512 cores)*
2	*1 x Intel Core(TM) i7-4820K socket 4-core/socket, 2-hw threads/core at 3.70GHz*	*64 GB*	*2 x Tesla K40c (Kepler, 2880 cores)*
3	*2 x IBM Power S824L sockets 12-core/socket, 8-hw threads/core at 3.52 GHz*	*1 TB*	*2 x Tesla K40m (Kepler, 2880 cores)*
4	*Nvidia Jetson TK1 SoC 4-core Cortex-A15 up to 2.5GHz*	*2 GB*	*1 x GK20A (Kepler, 192 cores)*

Table 4.1: System Configurations

OmpSs Runtime Configurations and Thread Binding

The OmpSs runtime is used to support the execution of work-sharing and tasking constructs. In addition, the OmpSs runtime manages host/GPU data transfers and concurrent kernel execution and CUDA streams. To that end OmpSs reserves a helper thread in the socket for each GPU device attached to it; the rest of threads are used to execute `smp` tasks. The execution of `smp` target regions is assigned to sockets in a round-robin way, and work-sharing constructs inside an `smp` target region are bound to the threads in a single socket,

For the 3rd system, based on IBM Power8 processors, we have activated the NUMA-aware scheduler feature in the OmpSs runtime. The runtime detects the socket architecture of the system and binds threads properly, distributing tasks according to the memory layout. Besides that, in order to investigate the effect of multithreading inside a core, we adjust the OmpSs thread binding (using an environment variable) to use 1, 2, 4 or 8 threads per core.

Performance Results

N-Body. This kernel computes the motion of a set of bodies based on the forces between them. For this simulation, an all-pairs algorithm is used with $\mathcal{O}(n^2)$ complexity, as shown in the upper code in Figure 4.2. The `resources` values have been set to maximize load balancing between tasks executed on the host processors and the GPU devices.

Figure 4.5 shows the performance results obtained for the N-Body kernel. The performance plot of the top-right corner shows the performace achieved when using the cores in the host for the three first system configurations. The main performance plot in shows how that performance is improved when using one and two GPUs, reaching performance increases in the 8%-14% and 4%-10% ranges, respectively. This contributed performance is very close to the ideal performance which could be obtained by just adding the performance of the CPU to the GPU.

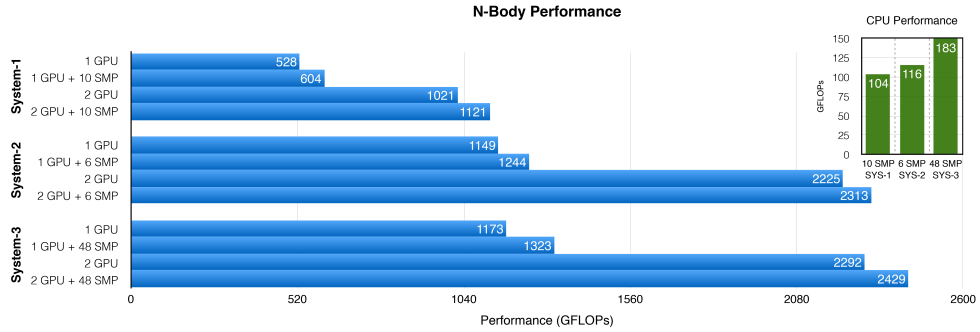


Figure 4.5: N-Body Simulation Performance Results

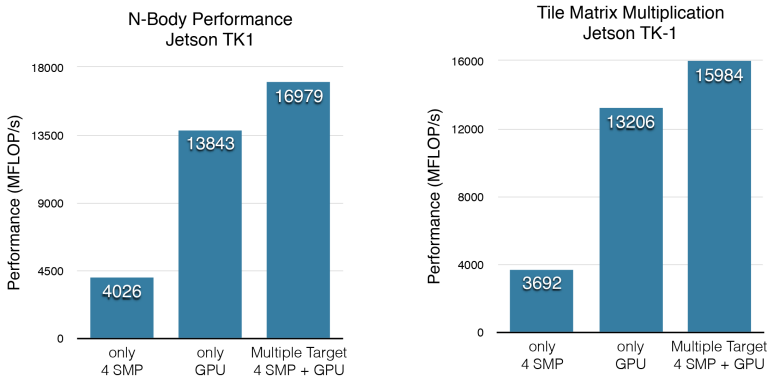


Figure 4.6: N-Body and tiled-gemm Performance on Jetson TK1.

Finally, the performance of the N-Body kernel has also been evaluated on the 4th system based on the Jetson TK1. The left plot in the Figure 4.6 shows three different results: CPU only, GPU only and combined CPU/GPU. In this case, the performance benefit is up to 20% due to the relatively close performance of the ARM cores and the small GPU in the SoC.

Tiled Matrix Multiply.

The kernel performs a dense matrix multiplication of two square matrices $A \times B$ C. Matrices are divided in blocks and each task is responsible for the computation of one of such blocks of the output matrix C. The matrix size is used 8192x8192 double-precision floating-point elements with 512x512 block size.

The matrix multiply kernel is written using six nested loops: the three innermost ones are annotated with MACC directives for multiple target devices. The MACC compiler transformed these into non-optimized CUDA code (the current implementation lacks many optimization phases that would be necessary to generate an optimized kernel) and they were then highly optimized using expensive optimization features of back-end

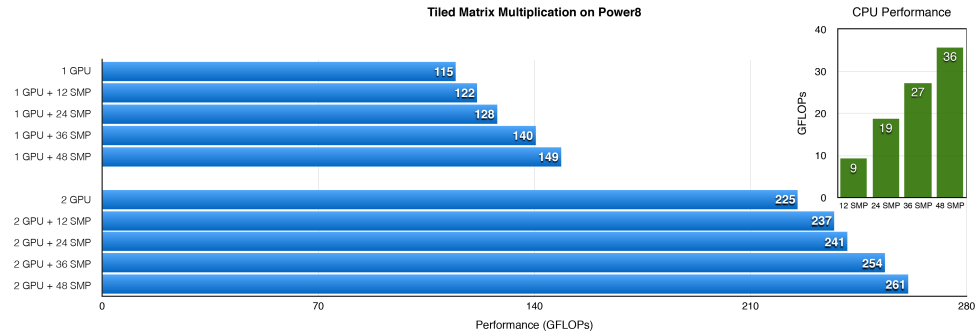


Figure 4.7: Matrix Multiplication Computation Performance Results

compiler for the host.

The performance for this kernel has been evaluated on the NVIDIA Jetson TK1 (right plot in Figure 4.6) and IBM Power8 (Figure 4.7) platforms. For the TK1 system we can conclude that the work was shared among all the SoC elements, with all the cores being able to contribute to the performance of the GPU.

For the Power8 system, the plot on the top-right corner shows the performance that is obtained when using different numbers of SMT threads. The main plot in that figure then shows how this performance is improved using the hybrid system, observing performance increases of 30% and 16% when one and two GPUs are used, respectively. Observe that the best result is obtained when two SMT threads per core are activated, since each Power8 core includes two vector units and load/store units.

STREAM. This benchmark [31] is commonly used to benchmark memory bandwidth. It consists of four micro-benchmarks accessing three vectors `a`, `b` and `c` and a scalar variable, inside an iterative loop that repeats their execution a number of times. Loop tiling has been applied to the outermost loop in these four operations in order to divide the iterations into multiple tasks and to run them in parallel. Task dependencies are specified between the tasks computing the four different operations.

This benchmark is evaluated using the 2nd and 3rd system. Both use the same Nvidia GPU (with theoretical memory bandwidth of 288 GB/s). However, the processors in the 2nd system report a memory bandwidth of 59.7 GB/s while the processors in the 3rd system report an average 192 GB/s (with a maximum of 275GB/s on the individual micro benchmarks [32]). Therefore, comparing these two systems provides a good opportunity to see how the runtime is able to fully exploit the additional bandwidth in the Power8 system.

Figure 4.8 shows the average bandwidth reported by the Stream benchmark when different numbers of SMP workers are used, for both systems evaluated. For the Power8 system (right plot), a speed up to 84% over GPU baseline is achieved when using all the

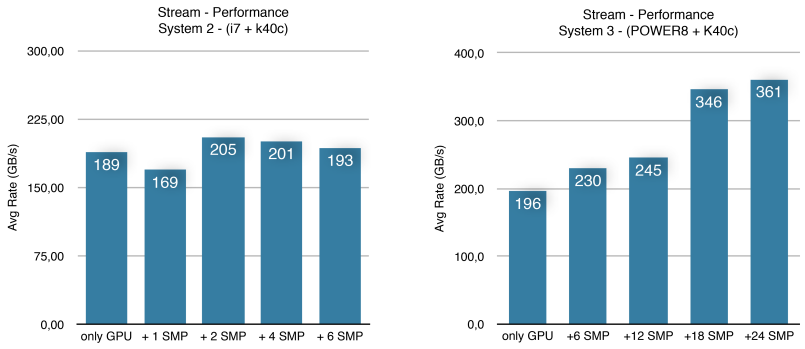


Figure 4.8: Stream Bandwidth Performance Avg Rate(GB/s)

cores in the entire system. For the i7-based system (left plot), the best performance is achieved when only two cores are used, showing the memory bandwidth bottleneck of the socket. When GPU tasks are finished, the runtime steals tasks which were initially assigned to the CPU, forcing the runtime to copy data from host to device.

4.2 Exploiting On-Chip Memory

We introduce a new data sharing clauses for the `distribute` construct of OpenMP which exploit the on chip memory of GPUs. We have implemented our idea on the MACC compiler, introduced in the Section 3.

The on chip memory, which is also known as shared memory in CUDA, is the fastest memory memory area after registers. It is allocated per block, so each thread in the block shares the same shared memory. It has a number of uses, such as reduction to make possible global memory coalescing. However, there are a few difficulties with shared memory; 1) shared memory is a fairly small area, typically 48KB or 96KB, 2) it requires size configuration before launching the CUDA kernel, 3) there is no direct access to shared memory from host memory, threads in the same block must load data from the global memory to shared memory.

From the viewpoint of the compiler, the difficulties of shared memory make it impossible for compilers to generate code automatically using shared memory. First of all, compilers can not make a decision about where and how much of global memory is going to load to shared memory. Secondly, they can not decide how much shared memory to use, because the types of variables are not clear at compile time. Because of these reasons, we introduce three new clauses to give hints to the compiler. We also extended the current specification.

4.2.1 Array Privatization

MACC makes use of shared memory for threads in a team based on the specification of `private` and `firstprivate` data structures in the distribute directive, so that each team allocates a private copy in its own shared memory. MACC analyzes the size of the data structure to be privatized and generates code for its allocation and copying from global memory to shared memory in each team.

However, for very large private arrays this is not possible to apply. For these cases we have implemented three new clauses. With these clauses, and the chunk size provided in the `dist_schedule(static,chunk size)` clause in the distribute directive or near by array variable, the compiler just allocates a portion of the arrays to each team and performs the necessary copies according to the `firstprivate` and `lastprivate` semantics.

- `dist_private(list)` : shared memory is only allocated up to indicated chunk size.
- `dist_firstprivate(list)` : shared memory is allocated up to indicated *chunk size* and it is filled with own part of array at global memory.
- `dist_lastprivate(list)` : shared memory is allocated up to indicated *chunk size*. End of the distribute scope, allocated area from shared memory is recopied to own location at the global memory.
- `dist_firstlastprivate(list)` : it is a short-cut for specifying dist_firstprivate(list) and dist_lastprivate(list) at the same time.

4.2.2 Experimental Evaluation

For the experimental evaluation we have used a node with 2 Intel Xeon E5649 sockets (6 cores each) running at 2.53 GHz and with 24 GB of main memory, and two Nvidia Tesla M2090 GPU devices (512 CUDA cores each, compute capability 2.0) running at 1.3GHZ and with 6GB of memory per device. For the compilation of OpenACC codes we have used the HMPP (version 3.2.3) compiler from CAPS [33]. For the compilation of OmpSs codes, we have used the Mercurium and Nanos environment. GCC 4.6.1 has been used as the back-end compiler for CPU code generation and the CUDA 5.0 toolkit for device code generation. Performance is reported in terms of execution time for the kernels generated and the speed-up relative to sequential execution on a single core for the complete application.

We used **DG** which is a kernel version of a climate benchmark developed by National Center for Atmospheric Research [34]. The code consists of a single target region that

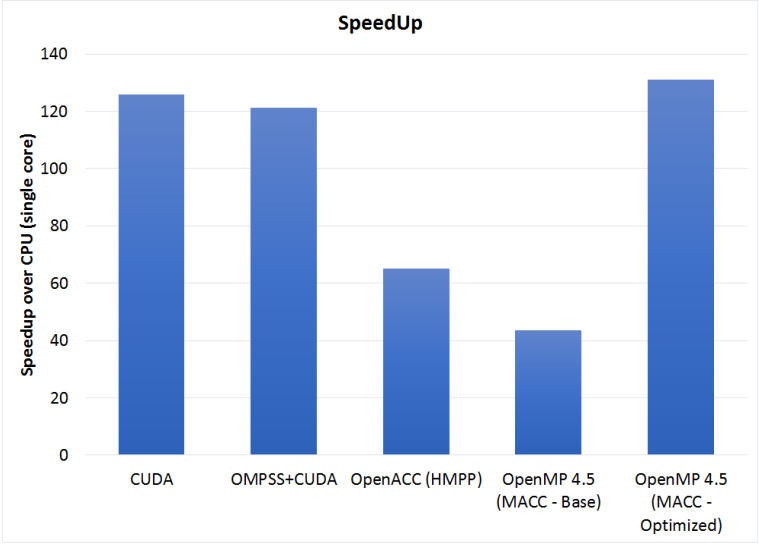


Figure 4.9: Performance evaluation for DG kernel

is executed inside an iterative time step loop that is repeated for a fixed number of iterations. Inside the target region the iterations of two nested loops are mapped to the teams/thread hierarchy as specified by the programmer. Figure 4.10 shows the code for the program.

Figure 4.9 shows the performance that is achieved by different versions of the code, described in the following bullet points:

- **CUDA**: hand-optimized CUDA version of the application (with host and kernel code written in CUDA) available from NCAR.
- **OmpSs+CUDA**: OmpSs version of the application leveraging (only) the computational kernels written in CUDA.
- **OpenACC**: OpenACC version available from NCAR compiled by HMPP compiler.
- **OpenMP 4.5**: Extended versions of our OpenMP accelerator model implementation compiled by MACC compiler, including additional clauses to influence kernel code generation.

Comparing bars labelled CUDA and OmpSs/CUDA in Figure 4.9 one can extract a first conclusion: OmpSs is able to leverage existing CUDA kernels with similar performance as full host/device CUDA codes. In this case we observe a small performance degradation probably due to overheads of the runtime in generating tasks in each iteration of the time step loop that are unnecessary.

```
#define NELEM 90000
#define SIZE (NELEM*nX*nX)
#define CHUNK 256
#define NT 5625
double delta[nX*nX], der[nX*nX], grad[nX*nX], flx[SIZE], fly[SIZE];
//
for (it=0; it<nit; it++)
{
 #pragma omp target device(acc)
 #pragma omp task inout(flx[0:SIZE],fly[0:SIZE])
 #pragma omp teams num_teams(NT) private(grad) firstprivate(delta,der)
 #pragma omp distribute parallel for \
             dist_first_lastprivate([CHUNK]flx, [CHUNK]fly)
 for (ie=0; ie < nelem; ie++) {
   #pragma omp parallel for private(j)
   for (ii=0; ii < nX*nX; ii++)  {
     int k = ii % nXa, l = ii / nX;
     double s2 = 0.0, s1 = 0.0;
     for (j=0; j < nX; j++) {
         s1 = 0;
         for (int i=0; i < nX; i++)
             s1 = s1 + (delta[j*nX+l]* flx[ie*nX*nX+i+j*nX]* der[k*nX+i]+
               delta[k*nX+i]* fly[ie*nX*nX+i+j*nX]* der[l*nX+j])* gw[i];

         s2 = s2 + s1*gw[j];
     }
     grad[ii] = s2;
   }
   #pragma omp parallel for
   for (ii=0; ii < nX*nX; ii++) {
     flx[ie*nX*nX+ii] = flx[ie*nX*nX+ii]+ dt*grad[ii];
     fly[ie*nX*nX+ii] = fly[ie*nX*nX+ii]+ dt*grad[ii];
   }
 }
}
```

Figure 4.10: Example to explain MACC implementation of shared memory - DG Kernel

4.3 Conclusion

In this chapter, we firstly have proposed an extension to the directive-based programming models to support the possibility of sharing the execution (of multiple instances) of a task on different devices in a heterogenous architecture. We have analyzed its implementation in the compiler and runtime system and evaluated its performance in a prototype implementation using the OmpSs programming model. For this we used a variety of system configurations and three different intensive kernel applications. The proposed extension eases the use of multiple accelerators in conjunction with the vector and heavily multithreaded capabilities in multicore processors without any code modification. The new proposed clauses convey useful insight to guide the scheduler while keeping a clean, abstract and machine independent programmer interface. The performance evaluation shows that with the new resources-based scheduler the runtime is able to take advantage of all devices available in the heterogeneous system.

Programming models are playing an important role in the "slow" widespread adoption of heterogeneous systems. In this section we comment on three proposals closely related to the work presented in this section: versioning scheduler in OmpSs [35], CoreTSAR [36] and the resource-aware scheduling in [37].

Regarding the original OmpSs, the programmer has to provide the kernels to be executed on the devices using CUDA or OpenCL. The MACC compiler used in this section is able to target the thread hierarchies in different devices from a single OpenMP 4.0 target region. The versioning scheduler policy schedules a task to the fastest device available at that time; prior to that, in a learning phase the runtime measures the execution of the task on the different devices in order to train the system. This greedy policy may easily result in load imbalance. CoreTSAR [36] also proposes an adaptive scheduler with different techniques, based on an initial learning phase for a subset of statically assigned iterations of the target loop region.

Resource-aware task scheduling [37] investigates the benefits of adding information about the usage of resources (such as memory and bandwidth) for each task. This information, provided by the programmer thorugh a new clause in the `task` construct, is then used by the runtime to decide where to schedule that task. In our proposal we extend this model by considering the heterogeneity of the different devices available in the system.

Secondly, in this section, we have proposed an extension to the directive-based programming models to improve on-chip usage. The method is also knows as array privatization which makes different threads access distinct memory addresses, so that different threads do not access the same memory address. It is a technique where data that is common or shared among parallel tasks is duplicated so that different parallel tasks can have a private copy to operate on. Our results shows that automatically generated code can

reach the performance of hand-written CUDA codes.

5 Code Transformation of Nested Parallelism for GPUs

As we explained in Chapter 2, Graphics Processing Units (GPU) have been widely adopted to accelerate the execution of HPC workloads due to their vast computational throughput, being able to execute a large number of threads inside SIMD groups in parallel and using hardware multithreading to hide long pipelining and memory access latencies. Early GPU programs were based on a flat, bulk parallel programming model, in which programs had to perform a sequence of kernel launches. In the latest releases of these devices, dynamic parallelism is supported, adding the ability to launch kernels from threads running on the device, without host intervention. Unfortunately, the overhead of launching kernels from the device is higher than launching them from the host. In order to reduce these overheads, this thesis proposes and evaluates a user-directed code transformation technique (LazyNP which stands for Lazy Nested Parallelism) that targets nested parallelism. The compiler generates code to dynamically pack kernel invocations and to postpone their execution until a bunch of them are available. The thesis shows that for seven benchmark programs used in the evaluation, LazyNP can effectively exploit nested parallelism, significantly increasing performance when compared to eager implementations using dynamic parallelism or other code versions not making use of nested parallelism and well-tuned libraries.

5.1 Introduction

Early GPU programs were based on a flat, bulk parallel programming model, in which programs had to perform a sequence of kernel launches, each kernel trying to expose enough data parallelism to efficiently use the available resources in the GPU. In order to give support to more irregular codes (e.g., graph algorithms with irregular data access patterns and unpredictable control flows), the latest releases of GPUs include dynamic parallelism [38]. As explained above, dynamic parallelism makes it possible to launch kernels from threads running on the device, without CPU intervention. The nested parallelism that is available in a wide range of applications is thus supported,

dispatching a coarse-grained kernel which in turn dispatches finer-grained kernels to do work where needed, improving load balancing.

Unfortunately, the overhead of launching kernels from the device is much higher compared to launching from the CPU. Recent studies [39, 40] show that dynamic parallelism introduces noticeable overhead during kernel launch, precluding in most cases the exploitation of nested parallelism to improve application performance. In order to mitigate this overhead, this thesis proposes LazyNP, a code transformation technique for directive-based programming model in order to efficiently exploit the potential performance of nested parallelism.

This thesis makes the following contributions in this area:

- LazyNP is proposed as a compiler technique for directive based programming languages that collects nested kernel invocations until a reasonable number of them are available, postponing their execution as a single kernel or multi-thread code.
- Three different code transformations are proposed to implement LazyNP for GPU accelerator devices (CTA-based, warp-based and host-based) together with some additional compiler optimizations.
- LazyNP also proposes two different code transformation techniques (CPU Managed and Cross) to implement nested parallelism using hybrid GPU+CPU.
- Performance evaluation of LazyNP with real applications and data sets, comparing results with other code generation alternatives and/or optimized libraries.

The techniques proposed in this chapter target CUDA code, although they could be directly re-targeted to OpenCL or to an intermediate assembly such as PTX [41] or SPIR [42]. However, the proposed code transformation techniques can be applied onto CUDA compilers as a replacement algorithm for dynamic parallelism launch.

5.2 Motivation

Dynamic parallelism was introduced in an attempt to improve programming productivity and to broaden the applicability of GPU programming for irregular algorithms. However, it comes at the expense of a noticeable kernel dispatch overhead which precludes the exploitation of nested parallelism if used in a naïve way. This section motivates the proposal in this chapter by analyzing these aspects.

Kernel dispatch overhead

The overhead of dispatching a new kernel from the device incurs a noticeable slowdown due to the cost of dispatching device-side kernels, as already shown in previous studies [40, 39]. This section tries to quantify these overheads in a more precise way, exploring

three different ways to dispatch kernels from the device. This initial evaluation will be useful to understand the decisions taken in the code generation strategies that are proposed in this chapter. The three alternatives are shown in Figure 5.1 and explained below:

- Thread-based kernel dispatch: each thread executing the parent kernel dispatches new child kernels. For example, for a device with 24 SM and 128 threads per SM, the number of child kernels simultaneously launched is 128×24.
- Warp-based kernel dispatch: only the master thread in each warp executing the parent kernel dispatches new child kernels. For example, assuming 32 threads per warp, the number of child kernels simultaneously launched is $(128 \div 32) \times 24$.
- CTA-based kernel dispatching: only the master thread in each thread block (CTA) executing the parent kernel dispatches new child kernels. For example, 24 simultaneously launched kernels for a device with 24 SM units.

In all three cases the parent grid is configured so that the complete device is occupied (launching as many thread blocks as the number of SM in the device and as many threads per thread block as possible). Child kernels dispatched from the parent kernel are configured with NG 64 blocks and BS 128 threads per block.

```
1  __global__ void child( ){ /* Dummy Child */}
2
3  /* Thread-based kernel dispatch */
4  __global__ void kernel_THREADbased(..., int N) {
5  while (N > 0) {
6      if (threadIdx.x < N) child<<<NG, BS>>>();
7      N -= blockDim.x*gridDim.x;
8  } }
9
10 /* Warp-based kernel dispatch */
11 __global__ void kernel_WARPbased(..., int N) {
12 while (N > 0){
13     if (warpId < N && laneId == 0)
14         child<<<NG, BS>>>();
15     N -= numWARPs;
16 } }
17
18 /* CTA-based kernel dispatch */
19 __global__ void kernel_CTAbased(..., int N) {
20   while (N > 0) {
21     if (threadIdx.x == 0 && blockIdx.x < N)
22         child<<<NG, BS>>>();
23     N -= gridDim.x;
24 } }
```

Figure 5.1: Three alternatives for device-side kernel dispatch.

Figure 5.2 shows the overall overhead of dispatching a certain number of threads for the three different schemes mentioned above for two different NVIDIA devices (Tesla K80 and Titan-X). The main observations from this plot are: **(1)** the serialization imposed and conflicts in the Grid Management Unit (GMU) lead to an exponential growth in the overhead experienced; and **(2)**, the CTA-based approach seems to be the most competitive approach for dispatching child kernels for every number of threads tested.

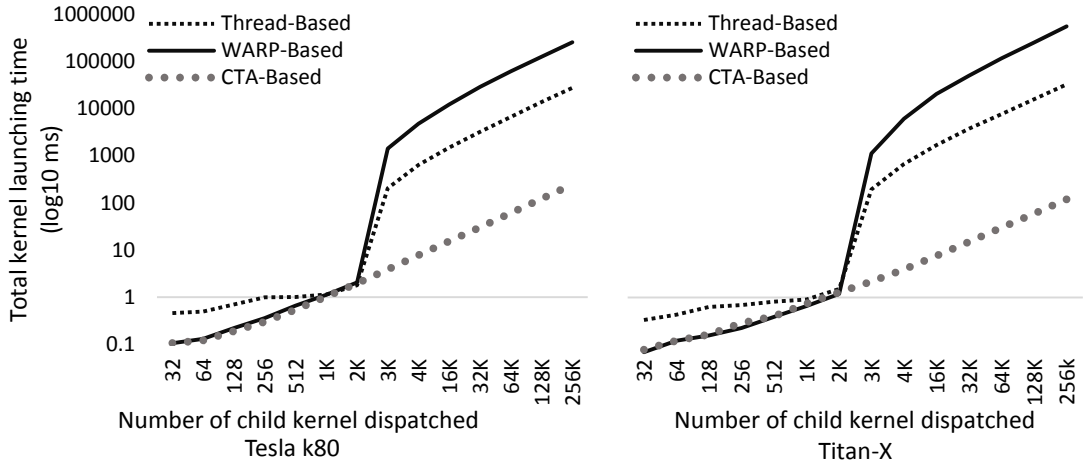


Figure 5.2: Overall overhead for Thread-, Warp- and CTA-based dynamic parallelism.

5.2.1 Naïve approaches to make use of dynamic parallelism

This subsection discusses the effect of the observed kernel dispatch overhead in the implementation of a BFS (breadth-first search) application kernel, showing that a naïve use of dynamic parallelism may not improve performance as expected when it should be useful to catch the appropriate granularity in this irregular application.

Figure 5.3 shows the skeleton of a graph traversal kernel that follows the usual edges-oriented approach to parallelize its execution [43, 44, 45, 1, 46]. Each node of the graph is assigned to a thread that is responsible for processing a variable number of edges in the loop at line 8. The variable `A[i].Edges` (trip count) in this loop introduces thread divergence which may be very relevant when the variability is high. For illustrative purposes in this section, Table 5.1 shows this variability for the lanes within a warp.

Figure 5.4 illustrates the execution timeline for a single warp of the kernel in Figure 5.3 assuming the number of iterations shown in Table 5.1 (for warp size equals to 8). In the timeline, each horizontal row shows the threads (lanes) that actively participate in the lock-step execution and the number of times it is repeated. In the beginning, all threads in the warp first execute `BB.init`; after that, they all start executing the `BB.process` loop body. After 10 iterations, some lanes (2,4,6) stop doing useful work, followed by lanes 3 and 5 at iteration 30 and during 100 iterations. At the end of the kernel, only lane 1 is active during 200 iterations while all other lanes wait idle.

```
1  __global__ void graphTraversal (...) {
2   int tid = threadIdx.x + blockIdx.x * blockDim.x;
3   int nt = grimdDim.x * blockDim.x;
4   for (int i = tid; i < N; i += nt) {
5         BB.Init
6
7         /* Imbalanced workload issue */
8         for (int j = 0; j < A[i].Edges; j++){
9             BB.Process
10        }
11        BB.AfterProcess
12 } }
```

Figure 5.3: CUDA code skeleton for graph traversal without using dynamic parallelism

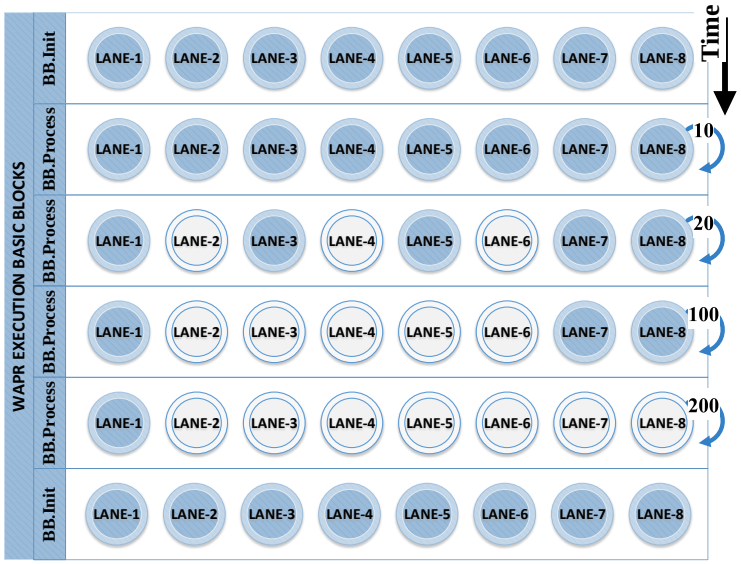


Figure 5.4: Execution timeline for a single warp (warp size 8) of the kernel in Figure 5.3

LANES	1	2	3	4	5	6	7	8
trip count	330	10	30	10	30	10	130	130

Table 5.1: Inner iteration trip count for the loop in line 9 in Figure 5.3.

Since lanes 1, 7 and 8 have much more work than the others, dispatching their work as a child kernel would be the solution offered by dynamic parallelism to cope with the load imbalance problem, executing them with additional resources. Figure 5.5 shows the CUDA code skeleton that would naïvely exploit dynamic parallelism, only using it when the number of iterations is larger than a certain threshold. This approached is named *EagerDP* in contrast to the *LazyNP* proposal in this thesis. The lower part in the same figure shows the timeline for its execution, assuming a threshold of 100 iterations. In order to show the performance of the *EagerDP* approach, the original CUDA imple-

```
1  __global__ void graphTraversal_eagerDP (...) {
2    int tid = threadIdx.x + blockIdx.x * blockDim.x;
3    int nt = grimdDim.x * blockDim.x;
4    for (int i = tid; i < N; i += nt) {
5         BB.Init
6         if(A[i].Edges > THRESHOLD)
7           BB.EagerDP //launch child_kernel<<<...>>>(...);
8         else
9          for (int j = 0; j < A[i].Edges; j++){
10             BB.Process
11         }
12        BB.AfterProcess
13 } }
```

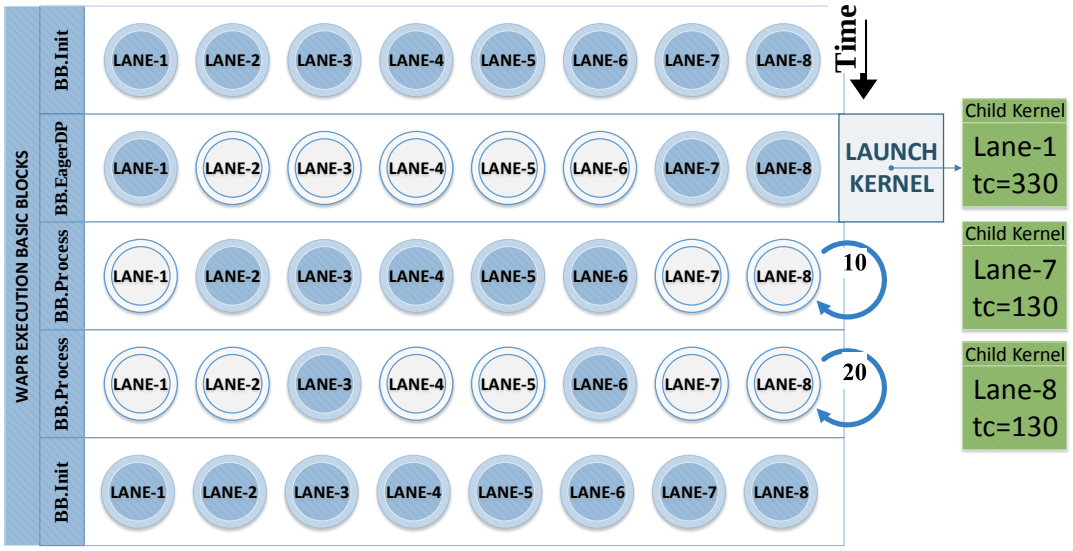


Figure 5.5: EagerDP: (top) CUDA skeleton code for graph traversal application making a naïve use of dynamic parallelism; (bottom) warp execution timeline, assuming warp size equals to 8.

mentation of BFS in the Rodinia Benchmark [44] was translated to the OpenMP 4.5 accelerator model, as shown Figure 5.6. It is important to note that in the innermost loop, nested parallelism is only enabled when the number of edges for the current node is higher than `THRESHOLD`, making use of the `if` clause for the `parallel for` pragma. This version has was compiled using the open-source MACC compiler (implementing [47]) and executed with two graphs with different distributions of number of edges per node (linear and exponential distributions). The left vertical axis in Figure 5.7 shows the speed–up that was obtained when using *EagerDP* for these two graphs when compared to the original CUDA version in Rodinia, for different values of the threshold that decides when to dispatch child kernels (horizontal axis). Notice that bars that are not visible in fact have a slowdown. The same plot includes two lines (referenced to the right vertical

```
1  #pragma omp target teams distribute parallel for
2  for( i = 0; i < N; i++ ){
3       BB.Init
4
5       #pragma omp parallel for if(A[i].Edges > THRESHOLD)
6       for(int j = 0; j < A[i].Edges; j++){
7          BB.Process
8       }
9       BB.AfterProcess
10 }
```

Figure 5.6: Version of the edges-oriented graph application in the Figure 5.3 making use of the OpenMP accelerator directives.

axis) showing the number of child kernels that have been dispatched. As can be seen, for the linear distribution graph the use of dynamic parallelism results in a slowdown since an excessive number of child kernels are invoked; for the exponential distribution graph, a maximum 1.24x speed-up is obtained, although for most values of threshold the execution is slower than the original CUDA version.

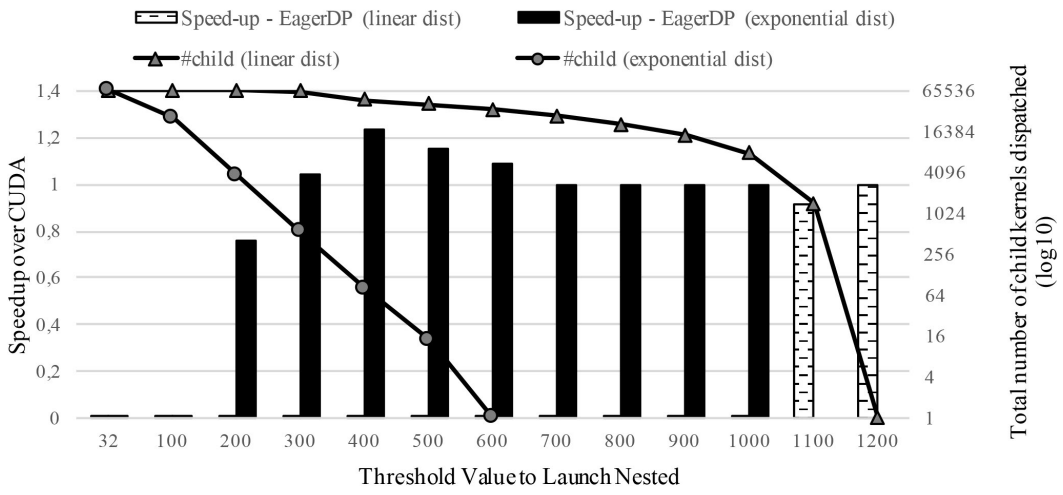


Figure 5.7: Speedup for the EagerDP Implementation of BFS.

The main conclusion from these experiments is that the programmer should be selective when deciding on the use of dynamic parallelism, carefully validating when it is worthwhile to use it in the application. In the naïve (EagerDP) implementation of dynamic parallelism, the value of the threshold that is used to make this decision has a critical impact, resulting in catastrophic effects on performance. In order to handle this in an appropriate way, this thesis contributes a new code generation strategy and execution approach that will make dynamic parallelism useful and perform better by reducing the number of child kernel dispatches without the need of carefully tuning the threshold value.

5.3 Background

This section briefly describes dynamic parallelism which has been introduced in Nvidia GPUs and its current support in user-directed accelerator programming models (in particular OpenACC and OpenMP).

5.3.1 Dynamic parallelism in Nvidia GPU

Nvidia GPU architectures are based on a number of Streaming Multiprocessor (SM) units, which are named SMX in Kepler [48] and SMM in Maxwell [49]. The devices include several SM units, with 192 and 128 single-precision cores per SM unit, respectively. To exploit all cores available, CUDA has two levels of parallelism: threads in a Cooperative Thread Array (CTA, also called thread block) and then CTAs in a Grid (also known as a kernel). Threads in a kernel are bundled into 32-thread warps, a warp being the basic lock-step execution unit in the GPU: threads within each warp share the same program counter (PC) and execute the same instruction at each cycle.

With dynamic parallelism, threads executing a kernel dispatched from the host (parent kernel) can launch new kernels (child kernels) without any host intervention. Nvidia GPUs include the so-called Grid Management Unit (GMU) which manages and prioritizes Grids to be executed. The GMU can pause the dispatch of new Grids and queue pending and suspended Grids until they are ready to run. The GMU also has a direct connection to the SM units in the device to permit Grids that launch additional work on the GPU via dynamic parallelism to send the new work back to the GMU to be prioritized and dispatched. As when launching kernels from a host, parameters for the child kernel to be dispatch are stored in constant memory, and the address is passed to the GMU with its associated configurations. The kernel that dispatched the additional workload is paused, and the GMU can dispatch the device-side kernels or the suspended parent kernel. The GMU will hold the parent kernel inactive until the dependent work has completed.

5.3.2 Support for dynamic parallelism in directive-based accelerator models

OpenACC [13] provides directives that allow programmers to specify code regions to be offloaded to accelerator devices and to control many features of these devices explicitly. The main construct is `kernels`, instructing the compiler to transform the annotated code region to exploit the available parallelism in the device. OpenACC also offers the `data` and `update` constructs to manage data movement, and `parallel` and `loop` constructs for detailed control of kernel offloading and the parallel execution of loops. Version 2.5 provides support for nested parallelism, allowing the programmer to nest `parallel` and `kernels` constructs. Nevertheless, the currently available versions of OpenACC 2.5 compilers from PGI, CRAY and GCC do not support nested parallelism.

The accelerator model in the **OpenMP** 4.5 programming interface also provides a set of directives to offload the execution of code regions onto accelerators.The main directives are `target data` and `target`, which creates the data environment and offload the execution of a code region on an accelerator device, respectively. The specification also contains the `teams` directive to create thread teams. In each team, the threads other than the master thread do not begin execution until the master thread encounters a `parallel` region. The `distribute` directive specifies how the iterations of one or more loops are distributed across the master threads of all teams that execute the region.

5.4 Lazy Nested Parallelism code transformation

This section presents the basic idea behind the proposed *Lazy Nested Parallelism (LazyNP)* code transformation, which could be used to compile nested parallelism in compilers for directive–based accelerator programming models such as OpenMP or OpenACC. As we mentioned in the introduction, LazyNP offers efficient nested parallelism for GPUs and hybrid CPU/GPU systems. Sections 5.5 and 5.6 present the details for the implementation of *LazyNP* for GPUs and hybrid systems, respectively.

Instead of directly launching the execution of child kernels, *LazyNP* dynamically bundles kernel instantiations and postpones their execution until a sufficiently large number of them is available. To do that, *LazyNP* saves in a buffer the necessary variables in the context of the parent thread to execute the kernel instantiation in a deferred way. Finally, when all threads executing the parent kernel finish, *LazyNP* launches the execution of a single kernel or multi-thread parallel code, named Next-Level parallel Kernel (NLK), to actually execute the finer-grained invocations in parallel to saturate the GPU. Hence, *LazyNP* thoroughly minimizes the kernel launching count and makes use of nested parallelism to exploit the right granularity for irregular applications.

Figure 5.8 shows how the original code in Figure 5.3 could be transformed in order to implement the *LazyNP* idea and a possible execution timeline for a warp. The outer iteration is mapped to all threads of the parent kernel as done by the MACC compiler. The basic blocks starting with `BB.LazyNP` are injected by the compiler to implement the fundamental *LazyNP* approach. After initializing the data structures necessary to support *LazyNP* in `BB.LazyNP.Init`, the parent kernel may iterate several times (line 7) depending on the size of the problem `N`. Later, each thread executing the parent kernel decides (conditional statement in line 9 checking if enough iterations need to be executed) whether to immediately execute the loop (lines 13–14) or to save the context for postponed execution of the loop as a child kernel (line 10). Threads that save the context are directed to `LazyNP_HandleCases_1` label with the aid of `goto` in line 11. The reason for that is we support *LazyNP* even in case of the idle lane within a warp due to the control-flow dependency. Finally, in `BB.LazyNP.HandleCases`, all postponed kernel invocations are executed through a single NLK dispatch. Section 5.5 and 5.6 provide

three different implementations for GPU and two implementations for hybrid CPU/GPU systems.

```
1  struct buffer_type{ int i; };
2
3  __global__ void graphTraversal_LazyNP (...,
4                 buffer_type *global_buffer) {
5    BB.LazyNP.Init
6    int tid = threadIdx.x + blockIdx.x * blockDim.x;
7    int nt = grimdDim.x * blockDim.x;
8    for (int i = tid; i < N; i += nt) {
9        BB.Init
10       if(A[i].Edges > warpSize){
11           BB.LazyNP.Save /* Save into global_buffer */
12           goto LazyNP_HandleCases_1;
13       }
14       for (int j = 0; j < A[i].Edges; j++)
15             BB.Process
16       BB.AfterProcess
17
18       LazyNP_HandleCases_1:
19         BB.LazyNP.HandleCases
20 } }
```

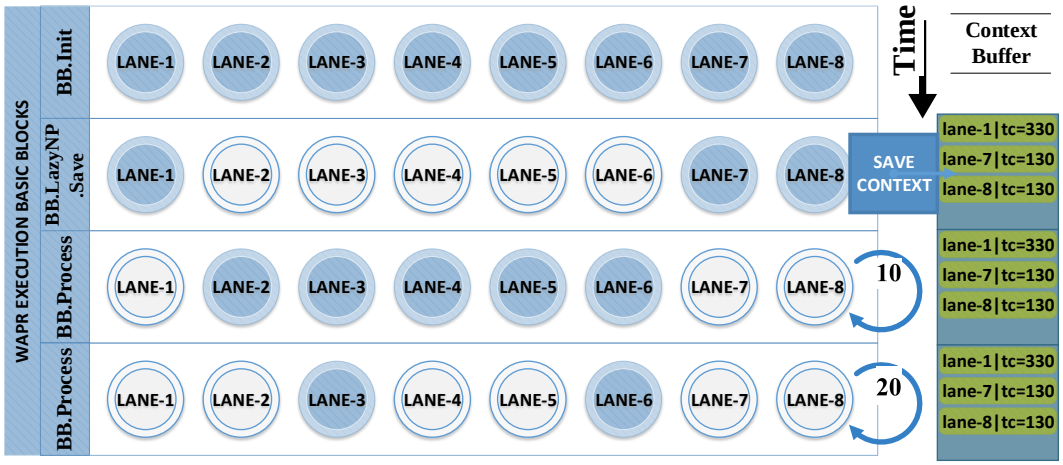


Figure 5.8: After applying *LazyNP* code transformation to graph traversal in the Figure 5.6 (top) and warp execution timeline (warp size 8)

Figures 5.9 and 5.17 show the codes for the *NLK*, in the form of a GPU kernel and a multi-thread code, respectively, that process the contexts that were saved in the global context buffer (`global_buffer`) for postponed kernel executions. First, the *NLK* pops a context of the parent thread according to its block identifier (line 3 and line 5, respectively); after that the *NLK* simply executes the code in the original kernel with the context popped from that buffer in parallel. In this example, the context only consists of

the value of variable `i` in the code on the top part of Figure 5.8.

LazyNP is also applicable when there are opportunities for multiple nested-parallel regions; in this case, the compiler generates multiple NLK kernels and global context buffers to handle their execution independently.

```
1 __global__ void LazyNP_child_NLK(..., buffer_type *global_buffer) {
2   int i = global_buffer[blockIdx.x].i;
3   /* Parallelised inner-loop of outer-loop*/
4   for(int j = threadIdx.x ; j < A[i].Edges; j += blockDim.x)
5         BB.Process
6
7   if(macc_if_master_cta())
8         BB.AfterProcess
9 }
```

Figure 5.9: NLK generated by the compiler for the postponed execution of iterations in the graph traversal applications.

5.4.1 LazyNP condition

LazyNP does not require programmers to choose a threshold value. The compiler automatically injects a condition in line 10 of the code transformed in the Figure 5.8 to decide between context saving and in-place execution. The threshold value is based on the warp size of the GPU actually used (which can be obtained from the read-only PTX `%warpsize` register): if the trip count is smaller than the threshold, the NLK will not be able to fully utilize warp SIMD lanes. However the programmer can always override this threshold by using the `if` clause in the inner OpenMP `parallel` or `target` directive.

5.4.2 Packaging kernel invocations

As mentioned before, *LazyNP* needs to save the context of the parent thread in order to execute the NLK. The context could include the value of variables in the thread's stack or in shared memory, while the rest of variables are accessed as global variables (function parameters or global memory variables). The global context buffer resides in global memory since this is the only memory in the device that is accessible to both kernels and host. It is also coherent when dynamic parallelism is used between child and parent kernels. The size of each element in the context buffer is determined by the compiler taking into account the size of necessary context variables of the parent thread. During parent kernel configuration, the buffer is allocated, and its pointer is passed (line 3 in code in Figure 5.8). The number of elements in the buffer is decided by the compiler based on the trip count of the thread mapped outermost loop.

5.4.3 Code execution order

LazyNP should preserve the execution order of the code that follows postponed kernel invocations. To maintain this order, *LazyNP* moves the code blocks after the inner-iteration into the NLK, executing them right after parallel execution of the inner-iteration. In the code generated in Figure 5.8, `BB.AfterProcess` is not executed by the parent kernel anymore, but moved into lines 8–9 inside the NLK in Figure 5.9 and only executed by a single thread of the NLK as expected. In the case when the execution order does not need to be maintained, a `nowait` clause on the nested construct of inner iteration should be used.

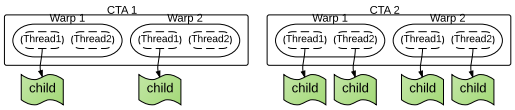


Figure 5.10: EagerDP

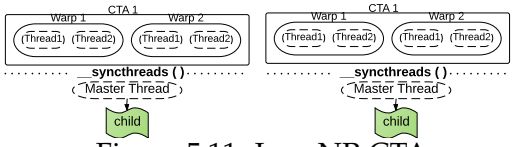


Figure 5.11: LazyNP-CTA

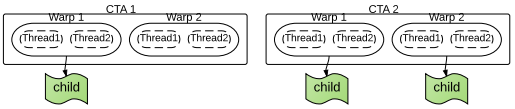


Figure 5.12: LazyNP-WARP

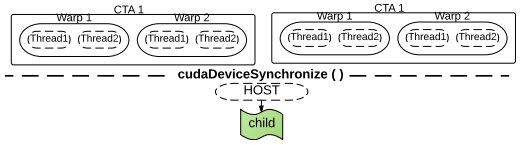


Figure 5.13: LazyNP-HOST

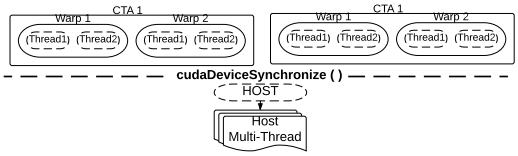


Figure 5.14: LazyNP-CPU Managed

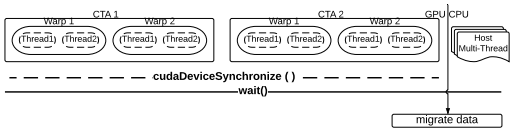


Figure 5.15: LazyNP-Cross

Figure 5.16: Illustration of EagerDP approach of dynamic parallelism (a), three LazyNP techniques for GPUs (b, c and d) and two LazyNP techniques for CPU+GPU hybrid systems (e and f).

5.5 LazyNP code transformations for GPUs

Three different strategies are proposed to dispatch the NLK: *CTA-based*, *Warp-based* and *Host-based LazyNP*. The main difference is which threads in the device actually dispatch the NLK in `BB.LazyNP.HandleCases` in Figure 5.8: the main thread for each CTA, the main thread of each warp or the thread running on the host, respectively. The three approaches also differ in the actual support for dynamic parallelism in the device; *Host-based LazyNP* allows nested parallelism in devices that do not provide support for

dynamic parallelism, while the other two require support in the device. The following subsection examines the three approaches in detail.

5.5.1 CTA-based LazyNP

In this approach, all threads in each CTA collectively contribute to kernel bundling; however, only the master thread in each CTA is responsible for dispatching the NLK. Figure 5.16(b) illustrates the NLK dispatching and synchronization of this code transformation technique.

In Figure 5.8 for `BB.LazyNP.Init` block, the compiler needs to generate code to 1) initialize the per–CTA local buffer that will store kernel invocations, 2) initialize the CUDA stream that is needed to allow the concurrent execution of postponed kernel invocations and 3) initialize three counters which help to manage memory offset and to count the total number of iterations bundled. In addition to the global context buffer, this approach makes use of a per–CTA local buffer allocated in GPU shared–memory for fast access the fastest memory after CUDA thread registers); its size is calculated together with the global context buffer and passed as a dynamic shared memory argument to the parent configuration. In `BB.LazyNP.Save` the compiler injects code to save the context for the postponed child kernel in the local buffer, incrementing the counters one by one. The counters are increased using atomic operations of the GPU API. Finally, in `BB.LazyNP.HandleCases` the compiler first injects code to synchronize threads in same CTA (`__syncthreads()`), making sure that all instances are already bundled. Then the memory portion of global memory is determined, and the local buffer is collectively copied to this area. After that, the master thread of each CTA configures and invokes the NLK. Also, the thread number of the NLK is found by determining the mean value of the total trip count of bundled inner iterations.

5.5.2 Warp-based LazyNP

In this approach, only the master thread in each warp dispatches the NLK as illustrated in Figure 5.16(c). Each warp has a buffer that is located in the shared–memory and an individual set of counters that need to be injected by the compiler for the `BB.LazyNP.Init` basic block in Figure 5.8. The per–warp local buffer is a portion of shared memory that is calculated using the warp and lane ID. To find the warp and lane ID, *inline* device functions are generated that return read-only registers `%warpid` and `%laneid` and only can be accessed via PTX.

In `BB.LazyNP.Save` the compiler injects code to insert contexts in each per–warp buffer using the corresponding per–warp counters. To increment the warp specific counter in the fastest way, we make use of warp aggregated functions following the same implementation as in [50].

Finally, in the code injected by the compiler for `BB.LazyNP.HandleCases` there is no need for synchronization because threads belonging to the same warp share the same instruction counter and thus execute the same instruction in "lock step" fashion. The per–warp local buffer is copied to global memory and the master thread of each warp dispatches the NLK using dynamic parallelism.

5.5.3 Host-based LazyNP

In this approach, the NLK is dispatched from the host thread in the CPU without relying on dynamic parallelism in the device as shown in the Figure 5.16(d). Therefore this approach allows the exploitation of nested parallelism in devices prior to the Kepler architecture. The basic idea is to bundle child kernel invocations into a global buffer during the execution of the parent kernel; once the parent kernel is finished, the postponed invocations in the NLK are then dispatched from the host.

The host is responsible for the initialization of the counters and global buffer in global memory. Once the parent kernel is finished, the thread in the host continues execution with all bundled contexts in the global buffer, which is copied to host memory. Finally, the host configures and dispatches the execution of the NLK if there are postponed instances available.

5.6 LazyNP code transformations for hybrid systems

Two different code transformation strategies are proposed to implement LazyNP to exploit hybrid CPU/GPU systems: *CPU Managed* and *Cross*. The main difference between them, as shown in Figure 5.16, is the parallel (Cross) or serialized (CPU Managed) execution in the CPU and GPU devices. The two approaches also differ in how the merging of data computed in the two devices occurs: *CPU Managed* relies on the automatic data migration feature of CUDA Unified Memory Access (UMA) while *Cross* relies on code injected by the compiler to migrate data.

5.6.1 CPU Managed LazyNP

In this approach, CUDA Unified Memory (UMA) is used to take advantage of simple data movement. The CUDA UMA creates a pool of managed memory that is shared between the CPU and GPU, bridging the CPU-GPU divide. The data used in the parent kernel and global context buffer are allocated by `CudaMallocManaged` routine with accordance with UMA. Different from other device based *LazyNP* techniques, the host dispatches multi-thread the NLK as shown in the Figure 5.16(e). The child kernel invocations are bundled collectively by the entire kernel into global buffer during execution. Once the parent kernel is finished, the host executes multi-thread the NLK as shown in the

```
1  void LazyNP_child_NLK(..., int buffer_count,
2              DT *global_buffer) {
3    #pragma omp parallel for
4    for (int ctxid = 0; i < buffer_count; ++i){
5      int i = global_buffer[ctxid].i;
6      for(int j = 0 ; j < A[i].Edges; j++) {
7        BB.Process
8      }
9      BB.AfterProcess
10 }
```

Figure 5.17: Multi-thread Next Level Kernel (NLK) generated by the compiler for the postponed execution of iterations in the graph traversal applications

Figure 5.17. As kernel data and global buffer are accessible to both GPU and CPU side, they are passed as pointers to the NLK.

5.6.2 Cross Offloading LazyNP

The last strategy goes beyond the postponing idea of LazyNP. The main difference of *LazyNP Cross* is that the outermost iteration is run by the GPU and CPU concurrently as shown in Figure 5.16(f). According to the LazyNP condition, CPU and GPU execute a different partition of the inner iteration. When the LazyNP condition is true, the GPU kernel simply ignores execution of the inner iteration while CPU code executes it and vice versa. Unfortunately, the CUDA UMA model does not allow concurrent data access from both CPU and GPU side (as far as we know, future NVIDIA devices will provide this support). Therefore, after execution, partially computed data needs to be merged. To do that, the data is replicated into GPU memory and it is kept as original data. Once the execution is finished, first the data produced by CPU is sent to GPU, then merging is carried out by comparing the original data with the data sent by the CPU and copying in the CPU computed data if it differs from the original.

5.7 Complementary Optimizations

In this section we discuss optimization techniques to enhance the performance and applicability of LazyNP in certain situations.

5.7.1 Reducing idleness in LazyNP

This section proposes a simple optimization to reduce the “idleness” of threads during the parent kernel execution. In the execution timeline in Figure 5.8 one can observe

that those lanes that save their context for postponed execution (2nd step for lanes 1, 7 and 8) remain idle after that. However, in fact these lanes are not idle but executed the same instructions within the loop with no effect (since all lanes share the same program counter PC register). With the optimization that we are proposing, called *w-boost*, lanes 1, 7 and 8 also execute some iterations in their original trip count and save the rest for postponed execution in the NLK. With this optimization, we increase warp execution efficiency in the parent kernel and at the same time we decrease the execution time of the NLK.

Figure 5.18 shows the skeleton of the new parent kernel (top) and warp execurion timeline. Observe that now lanes 1, 7 and 8 execute iterations during parent kernel execution (steps 3^{rd} and 4^{th}). First, warp lanes reduce the largest trip-count to the value `warpTc` that is smaller than threshold using the butterfly shuffle instruction [30] (lines 9–11). At the same time, the value obtained corresponds to the largest trip count of lanes that do not need nested parallelism. Next, *w-boost* lets all lanes execute iterations up to `warpTc`. For those lanes that want to exploit nested parallelism, the associated NLK will start executing iterations starting from `warpTc`.

5.7.2 Memory Space Tracker

GPU devices have several memory address spaces; global, shared, local, constant and texture memories. Each has advantages and disadvantages. Unlike host programming, address spaces are specifiable by using device API qualifiers such as `__shared__`, `__device__` or `__constant__` in the code. Also, directive based GPU compilers decide address spaces either automatically according to their compiler algorithms or based on directives. However, address space is only declared in variable declarations. Later on the address space of variables is not clear to the compiler.

The challenge of address spaces with nested parallelism is that only global and constant memory can be used for communication. The other memory spaces are exclusive to kernels and cannot passed as a parameter to another kernel. Figure 5.19 demonstrates these challenges. It follows this transformation; firstly it copies `a` to global memory from host, afterwards it copies shared memory as it is specified as `firstprivate`. Also, pointer variable `p` are derived from the `a` variable and this is also used in the nested `teams` construct. However, the compiler doesn't know where pointer `p` resides when communication is needed. Therefore, it has to ask the device runtime even though the variable address space is quite obvious.

We propose a memory space tracker for directive based GPU compilers as an elegant solution to this problem. We have extended our MACC compiler's IR for the GPU variable symbols in order to keep its address space. When pointer arithmetic occurs, the compiler will already be able to figure out where the assigned variables reside

```
__global__ void graphTraversal_LazyNP_wboost (...,
  buffer_type *global_buffer) {
   BB.LazyNP.Init
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   int nt = grimdDim.x * blockDim.x;
   for (int i = tid; i < N; i += nt) {
     BB.Init
     int warpTc = A[i].Edges > warpSize ? 0 : A[i].Edges;
     warpTc = macc_warp_reduce_max(warpTc);

     for (int j = 0; j < A[i].Edges; j++)
       if( j < warpTc )
         BB.Process
     if(A[i].Edges > warpSize)
       BB.LazyNP.Save /* Save into global_buffer */
     else
       BB.AfterProcess

     BB.LazyNP.HandleCases
} }
```

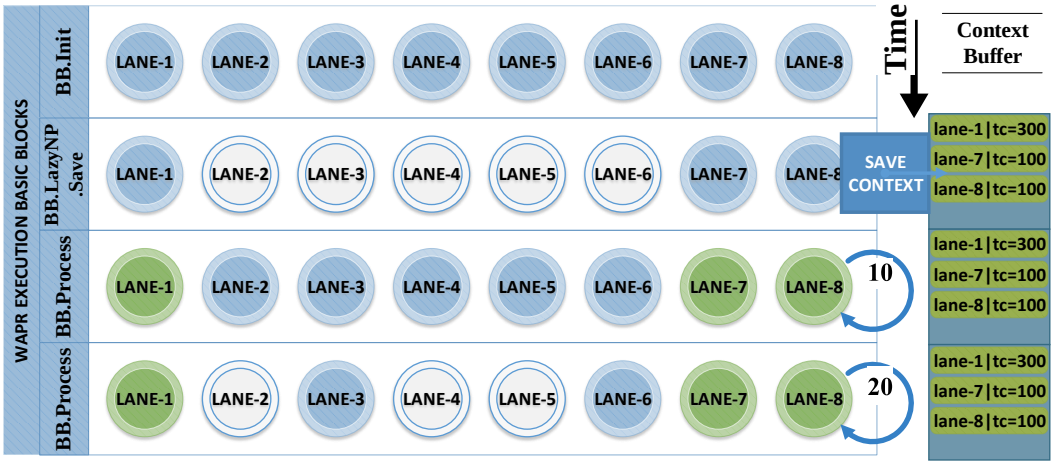


Figure 5.18: LazyNP with w-boost optimization: (top) code for parent kernel and (bottom) warp execution timeline assuming Iteration trip counts in Table 1.

by doing address space backtracking. For example, to decide the address space of `p` in the Figure 5.19, the compiler will check easily where pointer a resides. Also, if pointer arithmetic is not solvable during compilation, we use the `__isGlobal` device API function.

5.8 Experimental evaluation

In this section, we evaluate the performance that is obtained when *LazyNP* is used to exploit nested parallelism. Results are compared with the original CUDA implementation.

```
#pragma omp target map(to:a[N], b[N])
#pragma omp teams distribute firstprivate(a, b)
for (int i = 0; i < count; ++i){
  int* p = a;
  #pragma omp teams distribute parallel for
  for (int j = start; j < end; ++j)
  {
     int v1 = p[j];
     int v2 = b[j];
     // < ... Codes ... >
  }
}
```

Figure 5.19: Simple Example with Pointer Arithmetic in Directive Specified Code Block

In some cases, performance is compared with state-of-art libraries and multi-thread OpenMP. We implemented five LazyNP code transformation proposals in the source-to-source MACC compiler[14] that supports OpenMP 4.5 targeting CUDA code.

5.8.1 Experimental platform

Experiments are done on two different systems:

1. Intel(R) Xeon E5-2630 v3 (Haswell) 8-core, 128GB of main memory, executing Red Hat Enterprise Server and an Nvidia Tesla K80 GPU with 2x2449 CUDA cores, compute capability 3.7 and 2x6GB of device memory. The GPU includes 2x13 SM units, each with 192 single-precision CUDA cores;
2. Nvidia Jetson TK1 ARM SoC, 4-core Cortex-A15 up to 2.5GHz, with 2 GB of shared memory between CPU and GPU and GK20A Kepler GPU that has one SMX with 192 cores.

GCC 4.9 has been used as a back-end compiler for CPU code generation, and CUDA 7.5 and 6.5 toolkits have been used for 1st and 2nd system respectively for device code generation.

5.8.2 Benchmarks

Seven benchmarks have been selected to evaluate the performance of *LazyNP*:

- Kernels originally implemented making use of dynamic parallelism:
  1. Connected Component Labelling (CCL) is a well-known labeling algorithm that is commonly used for object detection[51].
  2. Bezier Tessellation(BT), a frequently used kernel in computer graphic packages [52].

- Graph algorithms:
  1. Breadth-First Search (BFS) iterative graph traversal algorithm [44].
  2. Betweenness Centrality (BC) to compute the centrality metric (number of shortest paths traversing a node) for all nodes in a graph [1]
  3. Graph Coloring (GC), refers to the problem of finding the minimum number of colors [45]
  4. Single Source Shortest Path (SSSP) to find the shortest path to every vertex from a single source[46].
- Sparse matrix vector multiplication (SpMV `y = Ax`), one of the most used kernels in high-performance computing programs. A nested parallel version of SpMV in OpenMP 4.5 has been (using the CSR format for matrices) and compared with state-of-the-art libraries: Nvidia cuSPARSE [53] and CUSP [54]. A variety of sparse matrices have been used from the University of Florida sparse matrix collection [55].

For the four last graph-based benchmarks, we have selected 3 of the largest social network graphs from SNAP dataset [56] which have different characteristic and that fit on the GPU device used for the experimental evaluation.

5.8.3 Overall results

Figure 5.20 shows the overall speed–up on the 1st system relative to original CUDA implementation, OpenMP CPU and the five different flavours of *LazyNP* for the 7 benchmarks in this chapter. In general, we can conclude that *Host-based LazyNP* yields the best performance with a speed–up of up to 43x and an average 5x speed–up. We observe *WARP* and *CTA-Based* methods always exhibit lower speed–up than *Host-based LazyNP*. The main reason is that they invoke a relatively greater number of child kernels than *Host-based LazyNP*, thus kernel launch overhead becomes a significant factor degrading performance. On the other hand, the two LazyNP approaches that make use of hybrid CPU+GPU obtain speed-ups against CUDA in general. *LazyNP Cross* exhibits the best performance for SpMV and CCL applications due to the effective CPU performance as is shown by the *OpenMP CPU* result. However, *LazyNP CPU Managed* is not good as *LazyNP Cross*. We observe that *LazyNP CPU Managed* suffers from performance degradation due to CUDA UMA. Our results show that managing data transfer and migration by the user-directed compiler and runtime has better performance than the current CUDA UMA for non-unified memory systems. In summary we find *LazyNP Host-based* is the best to implement nested parallelism, which is interesting since it is the only approach that does not require the dynamic parallelism support available in recent GPU devices.

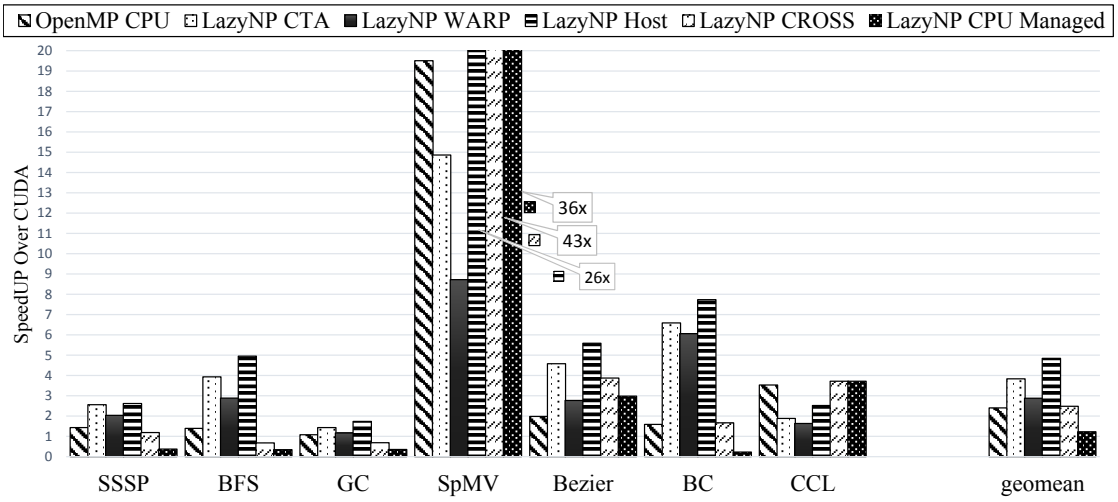


Figure 5.20: Overall speed–up of LazyNP with respect to baseline original CUDA implementation on Nvidia k80 and Intel CPU

Figure 5.21 shows the overall speed–up of the 2nd system. Since the 2nd system does not have support for dynamic parallelism, CCL and BT are excluded as they are originally implemented with dynamic parallelism, and *LazyNP CTA and WARP* are not evaluated for the same reason. In general *LazyNP CPU Managed* yields the best result with 1.6x speed–up over CUDA. However, if only the GPU is desired to be used, *LazyNP Host-based* exhibits nearly as good performance with 1.4x speed–up. On the other hand, we observed CUDA UMA efficiently solves data migration for physically unified memory systems. In general, we can conclude that *LazyNP CPU Managed* obtains the best result due to the relatively close performance of the ARM cores and the small GPU in the SoC.

5.8.4 SpMV

Figure 5.22 presents the speed–ups on the 1st system with the different *LazyNP* approaches. Speed–up is measured relative to the Intel MKL library (11.2 using Intel OpenMP) executed on the host that are obtained. Five different input matrices were used: three of them highly irregular (rajat30,ASIC_680k and trans5) and two regular (cothesissDBLP and eu-2005). For the highly irregular ones, device based *LazyNP Host* achieves a maximum speed-up of 1.5x, 1.9x and 1.2x, respectively. Compared to the well optimized Nvidia cuSPARSE and CUSP, *LazyNP* obtains an speed–up of 55x, 82x and 110x, respectively, despite the fact that the MACC compiler used to generate CUDA code does not apply a wide range of optimizations, such as coalesced memory access, CUDA vectorization, (etc.).

For the two less irregular matrices, *LazyNP CPU Managed* yields maximum speed–ups of 2.84x and 2.1x over Intel-MKL. In this case, cuSPARSE achieves much better performance than *LazyNP Host* because it benefits from compiler optimizations not available in the

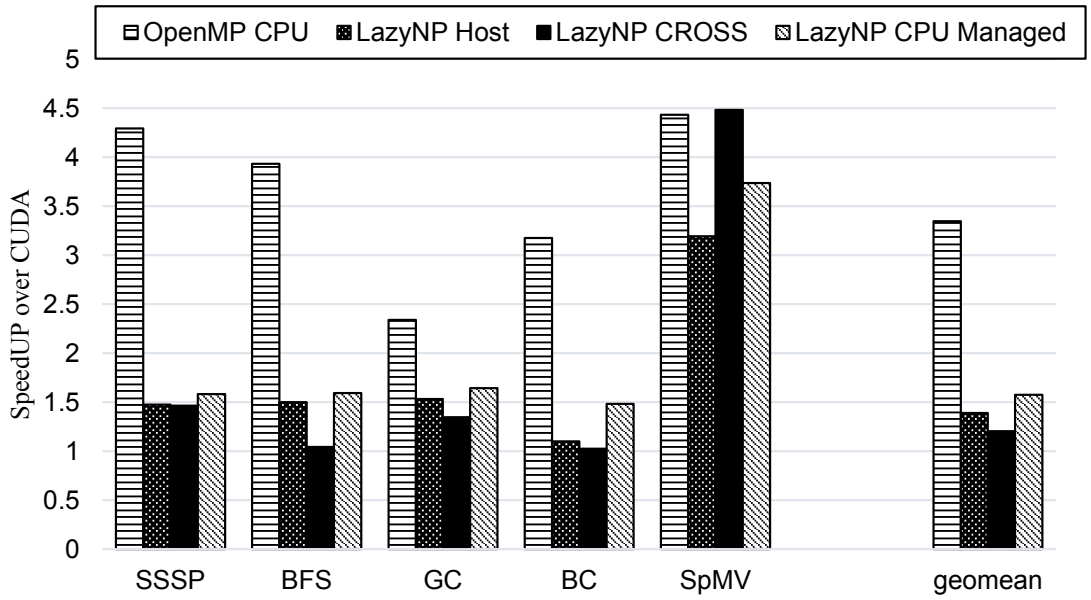


Figure 5.21: Overall speed–up of LazyNP with respect to baseline original CUDA implementation on Jetson TK1 SoC

MACC compiler. In all cases, *LazyNP* is able to achieve better results than *EagerDP* and the CUDA version without nested parallelism.

Finally we have also implemented the SpMV kernel following the Free Launch [57] proposal, as it addresses the same problem as our *LazyNP* proposal. The basic idea is that the parent kernel collects contexts for postponed inner iteration and it assigns execution of postponed iteration to parent kernel threads. To ensure collection is done properly, it uses entire grid (inter-blocks) synchronization which GPUs naturally do not support. Figure 5.22 shows that Free Launch outperforms CUDA and naïve *EagerDP*. However, it suffers several times from the synchronization cost of the entire grid and possible imbalanced inner iteration assignment, and thus it obtains slower performance than *LazyNP*.

5.8.5 Graph algorithms

The three plots in the upper part of Figure 5.23 show the speed–up on the 1st system, relative to the original CUDA version without dynamic parallelism, obtained by the five different *LazyNP* approaches, for the three different graphs used. For the web-google graph LazyNP achieves 1.65x, 2.43x, 1.6x and 1.78x for BC, GC, BFS and SSSP, respectively. The best results are obtained when using the wikitalk dataset since it is the most irregular one; in this case, the speedups are 19.2x, 2.9x, 9.1x and 3.2x, respectively.

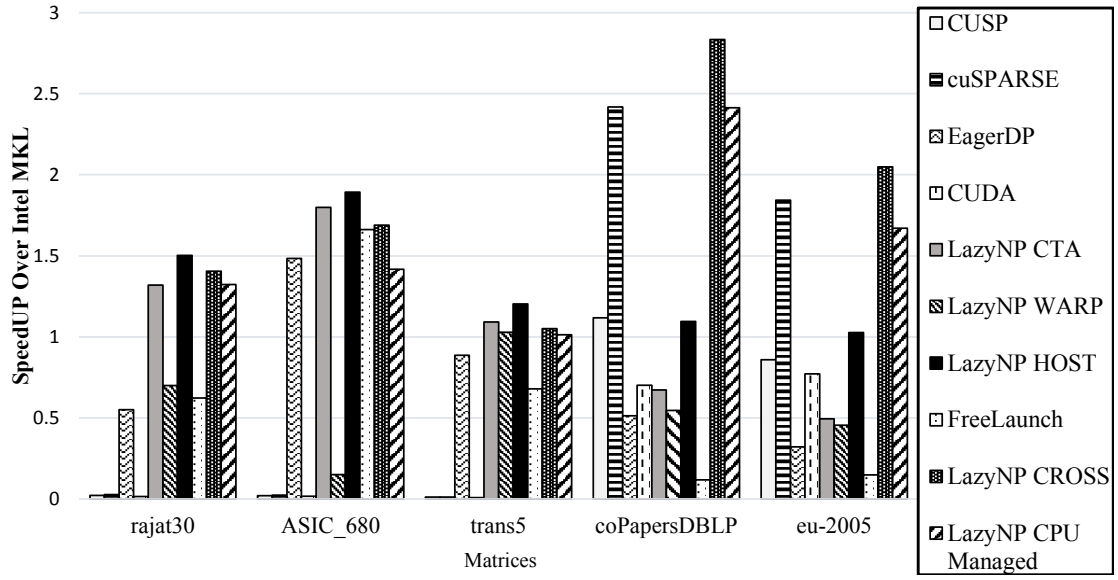


Figure 5.22: Speed–up for SpMV(*LazyNP*) over Intel MKL library(CPU)

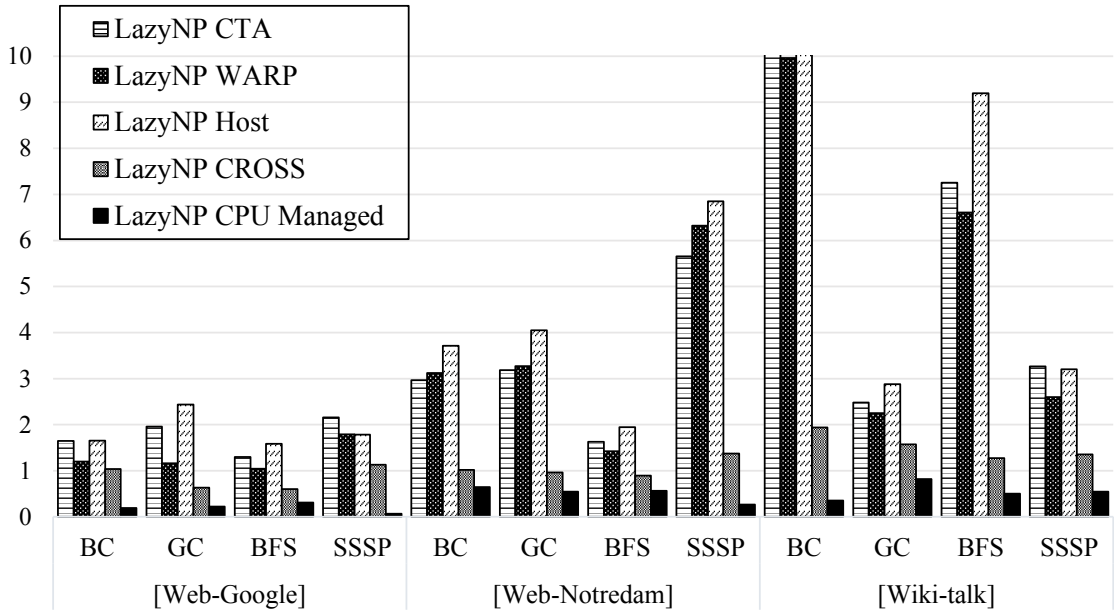


Figure 5.23: SpeedUp graph of widely used graph applications with different characteristic datasets.

5.8.6 Effect of w-boost optimization

The purpose of w-boost in *LazyNP* was to improve warp utilization during the execution of the parent kernel. We have examined the impact of w-boost on three LazyNP flavours with four graph algorithms in the Figure 5.24 using the 1st system. As can be seen from

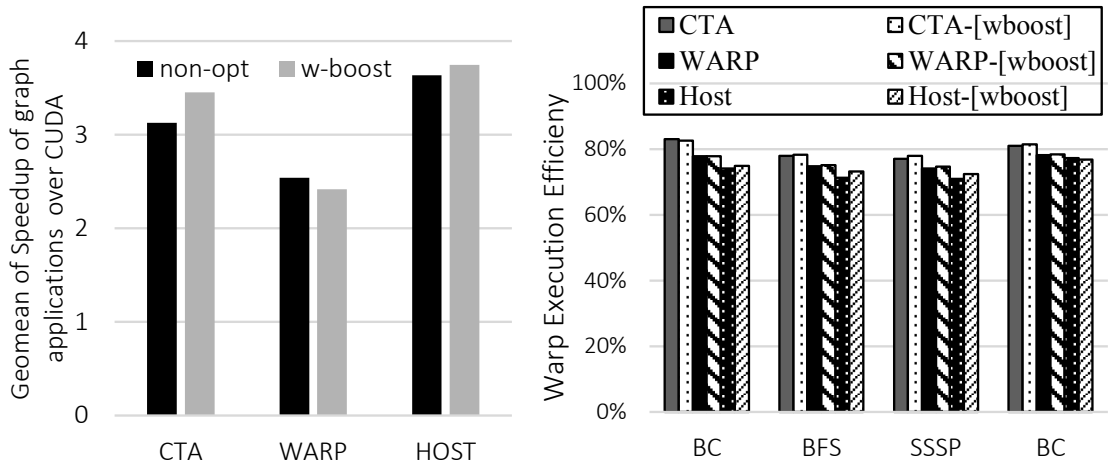


Figure 5.24: Left: Warp Execution Efficiency Graph of four graph applications with w-boost optimization. Right: Geometric mean of SpeedUP of four graph application

the results, we obtain improvements in general. However, we observe that w-boost may result in worse performance. In order to understand the reason, we have used the Nvidia *nvprof* profiler, measuring the ratio of active threads in the warp for all executed instructions (this gives a measure of control divergence and/or workload unbalance). `nvprof` does not give the metric for parent and child kernels individually in case of dynamic parallelism. Figure 5.24 shows warp efficiency, as one can see, and as expected, w-boost improves warp execution efficiency, making SIMD lanes in warps better utilized and decreasing control flow divergence. However we observe that w-boost increases the memory stall ratio since the parent kernel does not have recommended style memory access pattern among all applications that we used. Therefore, before applying w-boost optimization one should analyze the code block inside the nested parallel region in terms of its memory pattern.

5.8.7 LazyNP sensitivity analysis

Finally we analyze the effect of the `threshold` value that can be added in the `if` clause that is used to decide between postponed and direct execution (e.g. line 10 in Figure 5.8). The lower 3 plots in Figure 5.23 show the speed–up over CUDA that is obtained by *LazyNP* when different values for `threshold` are used for the Betwenness Centrality OpenMP code. The minimum threshold evaluated is 32 (minimum value chosen by default in order to ensure the complete utilization of SIMD lanes in warps). The three graphs shown correspond to the three graphs that have been used for the evaluation (Google, NotreDame and WikiTalk). The conclusions for the other benchmarks are similar. Observe that in general performance decreases when the threshold increases (although in all cases is always larger that 1x); in general the best performance is obtained when threshold is the smallest value, trying to make use of nested parallelism as much

as possible.

5.9 Related Work

Different architectural solutions have been proposed to reduce warp divergence; none of them is currently available on real hardware. DWF [58],[59] proposes a solution to collect threads in different warps according to their workload, merging them and executing under the same PC. LWM [60] proposes a large warp micro architecture that is able to create sub-warps that can fit the SIMD lane inside a large warp. Dynamic Thread Block Launch (DTBL [61]) proposes a more lightweight mechanism to support dynamic parallelism, dynamically adding blocks to the current kernel.

In this second group of related work, we consider different software solutions that also try to reduce thread divergence and load unbalance, although with some restrictions when applied to highly irregular problems and workload dependent memory accesses. In an early OpenMP compiler framework for GPU [62] the authors proposed loop collapsing for irregular applications; however their solution is only applicable for very specific code patterns. The authots [63] propose to apply data re-mapping across multiple warps to cope with thread divergence; however, the proposal may be inefficient since it requires communication between host and kernel. [64] proposes loop merging to re-order code blocks within a loop. The same effect is obtained with our warp boosting (w-boost) optimization technique.[65] offers a CUDA compiler patch to re-order codes for warp efficiency, automatically detecting divergence and transforming code; unfortunately, their proposal does not improve performance for programs with irregular memory patterns.

Finally, a few current proposals are dealing with nested parallelism in GPU programming. CUDA-NP [39] is proposed as an alternative to dynamic parallelism. CUDA-NP introduces directives to mimic nested parallelism without using dynamic parallelism; the main idea is to create more than the requested number of threads in advance, activating them when necessary to compute the iterations annotated with their directives. We believe that the solution based on dynamic parallelism is more suitable for large numbers of threads. FreeLaunch [57] has proposed a substitution for the child kernel invocation method based on reusing the parent kernel thread. It uses software-based grid syncronization, so it is based on persistence thread [66]. However the persistence thread approach results in a slowdown since it is forced to create small grid sizes. Also, grid synchronization brings overhead as it is based on CUDA atomics. [67] offers pragma directives for CUDA compilers to handle nested parallelism. [68] offers code replacing dynamic parallelism for CUDA compilers. [67] and [68] are very similar solutions to each other. They both handle dynamic parallelism using a similar solution to ours. However they don't focus on nested parallelism in directive based compilers. Unlike ours, their methods do not make use of hybrid systems. We also offer two additional

code transformation mechanisms to benefit from the performance of hybrid CPU/GPU systems.

5.10 Conclusion

This chapter proposes and evaluates a code transformation technique, *LazyNP*, which effectively supports nested parallelism in GPUs and is tailored to compilers for user-directed accelerator programming models such as OpenACC or OpenMP. *LazyNP* dynamically packs kernel invocations and postpones their execution until a bunch of them are available. The chapter proposes three different approaches for only device based code generation; for one of them, we also show how is it possible to exploit nested parallelism in GPUs that do not provide hardware support for dynamic parallelism. Also, two different code generation techniques are proposed for hybrid CPU/GPU usage.

This implementation of nested parallelism can overcome the problem of finding the right granularity for irregular nested loops and changing memory access patterns. We have proposed three different code transformation techniques and explored their pros and cons to better understand how they work. LazyNP collects the context of the irregular workload of threads at the kernel to context buffer and retrieves them in another kernel with fine granularity. LazyNP can increase performance, warp execution efficiency and memory access efficiency of real-world programs that contain the irregular pattern.

LazyNP has been evaluated using very relevant algorithms both in sparse scientific computation and graph algorithms, resulting in important speed–ups when compared to eager implementations using dynamic parallelism, other code versions not making use of nested parallelism and well-tuned libraries. For example, for nested Bezier tessellation, *LazyNP* achieves between 4x-24x speed–up with respect to non-nested version, depending on the input data. Speed–ups between 2x–20x are achieved over native CUDA implementation for a graph processing application. For SpMV, LazyNP outperforms well optimized libraries, making The GPU useful when matrices are highly irregular. *LazyNP* thus forms one of the major contributions of this thesis.

6 Dynamic Loop Scheduling

When compiling parallel loops for execution on a multi-core CPU, the programmer or compiler must decide how many threads to create and then must decide which thread will execute each loop iteration. For CPU execution, the number of threads is usually the number of cores, or the number of available thread contexts, which is a small multiple of the number of cores. Two common scheduling decisions are *static* or *dynamic* loop scheduling, and choosing cyclic or chunk iteration scheduling.

GPUs support many more threads within thread blocks of execution, typically thousands of threads. However these threads are short-lived and typically not all threads will be simultaneously resident. When developing or compiling parallel loops for execution on a GPU, the programmer or compiler must again decide how many threads to launch, and which thread will execute each iterations. The simplest mapping is to create one thread for each loop iteration and to statically map the loop iterations sequentially across the threads. However, it is undesirable for directive-based compilers due to the unknown trip count of the parallel loop. For this reason, typical compilers generate a kernel with a for-loop which processes the parallel loop of the kernels' threads in a cyclic order. In this way, compilers make sure that the generated kernel completes all the iterations of the parallel loop; also the thread and grid size of the kernel will never exceed the maximum limit. Unfortunately, this method does not yield as good performance as the simplest mapping.

To address these issues, we first conduct a thorough exploration of conventional loop scheduling methods on GPUs. Through further quantification, we find out the advantages and disadvantages of each scheduling method. By leveraging these insights and taking the hardware scheduler into account, we propose the concept of optimized dynamic loop scheduling along with an implementation in the NVIDIA PGI OpenACC compiler. Our method yields better performance than the solution used in directive-based compilers, providing the same performance as the simplest mapping. In addition, in the presence of reduction operations it delivers the better performance than all previous

scheduling methods. Moreover, it uses very small grid size to reach peak performance. We evaluate our techniques using a wide range of popular OpenACC applications on all modern generations of NVIDIA GPU architectures. We conclude with performance comparisons of dynamic scheduling vs. static scheduling by using the latest version of the PGI compiler, showing speed–ups of up to 1.24X and 1.36X for Kepler K80 and Pascal P100 GPUs respectively with mean speed–ups of 1.08X and 1.05X.

6.1 Introduction

GPU programming models offer a hierarchy of 1D-3D thread and grid sizes (number of thread blocks) in software that mimics how thread processors are grouped on the GPU. This mechanism provides a natural way to invoke computation across the elements in domains such as vector, matrix and so on. To facilitate choosing the thread and grid size, CUDA materials and sample codes from NVIDIA [52] recommend configuring kernels with a size and dimension with a domain element. A CUDA kernel can have maximum 1024 threads as the thread block size and $2^{31}-1$ thread blocks as the grid size. For a typical CUDA kernel, the grid size is meant to be able to increase if the domain size is larger than maximum allowed thread block size. In this thesis, we call this method **static one-to-one** scheduling. Unfortunately, it is undesirable for directive-based GPU compilers due to the unknown trip count and domain size at the compile time. On the other hand, this technique might still leave room for performance improvements. For instance, thread blocks, which are also known as Cooperative Thread Arrays (CTA), are processed by a streaming processing (SM) unit, so they share the resources of the SM such as on-chip memory. This might lead to idle thread blocks if they are not scheduled due to insufficient resources. In order to reduce the number of thread blocks, a widely used scheduling method is **static cyclic** scheduling. In this algorithm, the workload is partitioned into blocks with a fixed size statically. Then the partitioned blocks are processed in a sequential fashion. Unfortunately, in this case, domain elements are mapped statically; each iteration is assigned to the block id and thread id that is provided by software in sequential order. This can lead to stalling among thread blocks since a GPU is designed to be run parallel.

Various micro architectural and compiler solutions have been suggested for improving thread scheduling for GPUs. Most of the previous methods are based on warp formation [69, 59], the compiler based solutions rely on the voting feature of warps [65] to improve warp efficiency. However, loop scheduling based on CTAs seems to be viewed as not worth researching. We think that the reasons that hinder the research advancement in this domain are unclear details about hardware and unknown CTA-Scheduler. The default CTA scheduler is hardware implemented and is called the GigaThread Engine [70]. It promises to manage load balanced scheduling for CTAs on SMs. Since it claims that they tune or influence it directly, CTA scheduling is widely ignored by applications and compiler developers. Because of this, previously-studied CTA scheduling techniques

are hardware based [71, 72].

In this chapter of the thesis, we first study different scheduling techniques on GPUs and discuss advantages and disadvantages of them. This led to our design for a Optimized Dynamic Loop Scheduling compiler method for GPUs that enables efficient execution of CTAs. It can yield the best performance with a small grid size. Moreover, it performs better than the scheduling method that user-directed GPU compilers use. In our solution, the workload is assigned to CTAs dynamically by the compiler, once a CTA finishes the work it requests the next part from the remaining work. In addition to this, we provide a CTA throttling method that limits the number of CTAs on an SM to reduce the contention for execution resources. To the best of our knowledge, this is the first work that provides compiler automated loop scheduling on GPUs for performance enhancement.

We provide a code transformation algorithm for our proposed dynamic loop scheduling solution for user-directed accelerator programming models. This is implemented on the NVIDIA PGI OpenACC compiler [15]. We also discuss how our technique can be applied to other programming models.

Overall, this work makes several major contributions:

- We first demonstrate scheduling techniques and discuss their advantages and disadvantages.
- We propose an *optimized dynamic loop scheduling* method along with code transformation and complementary optimization, and describe its implementation in the NVIDIA PGI OpenACC compiler.
- We show the limitations of our approach, which are directly related to limitations of the current GPU programming models and hardware.
- We evaluate the performance improvement of *optimized dynamic loop* scheduling relative to the *static cyclic* scheduling which is currently used by the PGI compiler, both on simple kernel loops and for whole applications.

6.2 More GPU Architecture

We showed the general architecture of GPUs and its execution model in Chapter 2.2. As mentioned, a GPU processor consists of several SIMD cores, known as Streaming Multiprocessors (SMs) in CUDA terminology and Compute Units (CUs) in OpenCL. Thread block/CTA, which encapsulates several threads, is the primary unit for delivering jobs to SMs. The GPU schedules CTAs among SMs. From the hardware perspective, there is no execution dependency among CTAs — there is no execution order of CTAs.

CTA Scheduling : The default CTA scheduler, known as GigaThread Engine [70], is hardware implemented. It is not programmable at all. As no software approaches can

tune it directly, previously proposed CTA scheduling techniques are mostly hardware-based [71].

The default CTA scheduling policy on the GPU has been assumed as round-robin (RR) [73, 74, 75, 76, 77]: First, the CTA-scheduler (i.e., GigaThread Engine) assigns each SM with at least one CTA. If an SM still has sufficient resources (e.g., registers, shared memory, warpslots, etc) to sustain extra CTAs, a second round of assignment happens. However, the actual scheduling could be as simple or as complicated as NVIDIA wishes to make it (trade-off: hardware simplicity versus load balancing) [78]. With more transistors available now, scheduling is likely more sophisticated. To conclude all we can say is the hardware embedded CTA scheduler promises load balancing.

6.3 Loop Scheduling on GPUs

Loops are the primary source of parallelism for compilers. Scheduling of parallel loops has always been an important issue in parallel programming. A number of algorithms have been proposed for how to partition and schedule the loops onto available processors (which can also be called computation units).

In order to implement our software scheduling methods, our first challenge is to find out the most effective computation unit to schedule loops on. As we mentioned in Section 2.1, GPUs have several computation units such as threads, warps, SMs and CTAs. Firstly, we simply exclude SMs as we do not have direct access from software. Then, we eliminate threads since threads within a warp share the same program counter (PC) and execute the same instruction at each cycle. So they cannot be scheduled individually. That leaves CTA as the most basic unit that delivers the job to the SM and also each CTA contains warps. Moreover, CTAs are essentially individual units and they are accessible from software. For this reason, we choose CTA as our computation unit to implement scheduling methods.

In the rest of section, we first explain state-of-the-art loop scheduling methods, which are used by OpenACC compilers and CUDA programmers, and illustrate their execution diagram. We then describe our proposal for optimized dynamic loop scheduling. To be able to illustrate loop scheduling methods clearly, we explain them through the AXPY application, which stands for `A*X + Y`, which is a function from Basic Linear Algebra Subroutines (BLAS) [79] library. AXPY is a combination of scalar multiplication and vector addition, and it is very simple: it takes as input two vectors of X and Y with N elements each, and a scalar value A.

6.3.1 Conventional Loop Scheduling Methods

Static scheduling describes the approach where we have already controlled the mapping of the iteration to threads/processes are executed in our code at compile time. However, static scheduling algorithms ignore the fact that the amount of computation performed per iteration may differ, or that it cannot always be determined a priori (for example, the amount of computation could be dependent on the data). Moreover, in our case, the speed of each thread block (CTA) may also differ because of multiple interferences in the SM. Therefore static scheduling can often suffers from load imbalance, resulting in poor speed-up eve inf the workload appears perfectly balanced.

Static Scheduling (One-to-One)

This is a simple way of scheduling in native GPU programming. First the CUDA kernel is configured with same dimension as the domain element such as vector, matrix or another data type. Then the kernel size is configured with the same size as the domain element. In a nutshell, each iteration is assigned to each thread; there is a one-to-one relationship between iteration size and grid size. As we mentioned in the previous section, GPU hardware involves a scheduler that takes care of load balancing of CTAs. From the perspective of CUDA, this is the most common way of developing and configuring a kernel.

Figure 6.1 shows a simple example of AXPY using static scheduling in CUDA. Each element of the vector is assigned to a thread of the grid. In the top of the same figure the execution diagram of the code block is illustrated. For convenience we assume the size of domain element, that is *array y*, is not bigger than the size that CUDA allows. Thus the entire grid fits perfectly to the iteration size.

Static Cyclic Scheduling

An alternative method of static scheduling is static cyclic scheduling. In this algorithm, the workload is partitioned into blocks with a fixed size statically. Then, it processes partitioned blocks in a sequential fashion.

When we apply this method on the CUDA kernel, we create a strip-mining loop in the kernel. The kernel iterates the loop [N/G] times, where N is our domain size or trip count of the loop and G is the fixed block size which is the size of the kernel. It is important to note that, the kernel size [G] should smaller than domain size [N]. Otherwise, this method transforms to static one-to-one scheduling. Figure 6.1 (second from the top) shows the same AXPY example using static cyclic scheduling in CUDA. Again the figure illustrates its execution graph. As you can see, each thread block processes a chunk of the parallel loop in sequential order. When they finish execution, they increase their

offset to process the next chunk. Thread blocks continue execution until they finish all iterations of the parallel loop.

6.3.2 The Proposal of Dynamic Loop Scheduling

Dynamic scheduling, also known as self-scheduling, has always been a major topic in parallel programming. It divides the iteration space up into a fixed number of blocks, which can be also called subtasks, and processes them by computation units in a first-come-first-served basis. There are numerous previous studies in this topic such as [80, 81]. If the imbalance becomes large, then it is necessary to dynamically adjust the work assigned to each processor at run-time to balance the load. However, from the perspective of a CUDA programmer, load balancing is not a viable problem as CUDA promises a hardware scheduler. Therefore, dynamic loop scheduling has been ignored off-limits in CUDA programming research.

When we apply the idea of dynamic scheduling on CUDA, we divide the entire workload into subtasks [N/G]. In this case, N is our domain size or trip count and G is the size of the CUDA thread block. We process them using computation units which are CTAs. Ideally, the subtasks need not be of fixed granularity, and in fact, the granularity could vary dynamically. In our case, as we schedule loops by CTAs we just simply assign [G] with the CUDA thread block size.

It is important to note that in CUDA programming, programmers can create an excessive number of CTAs and performance might differ according the kernel size. For this reason, choosing the right number of CTAs becomes a challenge as controlling the subtasks could become a bottleneck in dynamic scheduling. We overcome this problem via a throttling technique that we propose in Section 6.4.2.

Dynamic Scheduling

This algorithm partitions the loops into subtasks containing one or more iterations [N/C], where N is the number of iteration and C is the number of CTAs. Each then continuously allocates and executes one subtask at a time until no subtasks are left for processing. Each CTA processes a subtask by its threads.

From the perspective of CUDA, the common method for implementing dynamic loop scheduling is to let each thread atomically increment a global *scheduling* counter. The only problem with this approach is the need to reinitialize the scheduling counter before each use. Our implementation uses this method with the available efficient atomic increment operation, but we will see the problems caused by the need for re-initialization.

Since this is used for thread block scheduling, we have thread zero in the thread block

perform the atomic increment. Thread zero then shares the updated value using CUDA shared memory. Effectively, the dynamically-scheduled code for the SAXPY loop looks like that in Figure 6.2 (top). The dynamically-scheduled loop now has an atomic increment at line 6 and one synchronization which is at line 7 to ensure that thread zero doesn't finish its work and do another increment before the other threads are done using its value. We also show its GPU execution diagram in the same figure. As you can see in the diagram, thread blocks process the parallel loop dynamically and chunk by chunk. In our implementation, we map chunk size with number of threads within the thread block to be able to map one thread with one iteration.

Optimized Dynamic Loop Scheduling

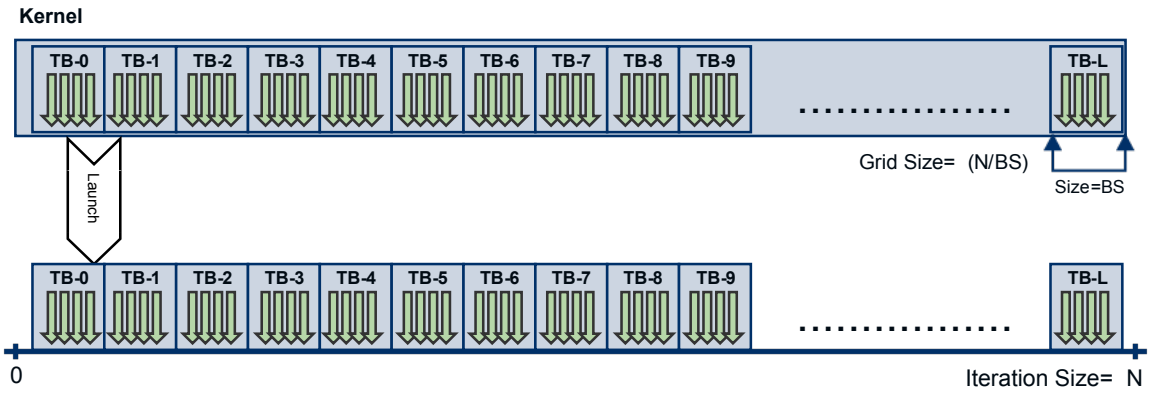
We propose an optimized dynamic loop scheduling technique which is a compromise between static cyclic scheduling and dynamic scheduling. As we mentioned above, GPUs have a hardware embedded CTA scheduler on each SM that seemingly works in a round-robin fashion. Hence, we do not need dynamic scheduling for the first iteration as the GPU has already scheduled CTAs to the SM in a balanced way. However, we do need dynamic scheduling for later on since the performance of each CTA may differ for later parts of the loop.

In the straightforward code for a dynamically-scheduled loop in Figure 6.2 (top), the first operation by every thread block is an atomic increment. If all thread blocks start at essentially the same time, this will cause immediate conflicts on the memory accesses, requiring many instruction *retry* operations. In fact, we know the grid size, so we can statically schedule the first iteration for each thread block and then use dynamic scheduling for the subsequent iterations. This is shown in Figure 6.2 (bottom), with the initial iteration statically assigned at line 6, and the atomic increment for subsequent iterations moved to line 11.

Figure 6.2 (bottom) shows the AXPY example using dynamic scheduling in CUDA. Also, the figure on the right side illustrates its execution graph.

Static scheduling (one-to-one)

```
__global__ void AXPY(T* x, T* y, T a, int N) {
  /* BlockId is statically assigned by blockIdx.a */
  int i = threadIdx.x + blockIdx.a * blockDim.x;
  if(i < N)
    y[i] = a * x[i] + y[i];
}
...
/* Kernel Launch */
 AXPY<<< N/BS, BS >>> (x, y, a, N);
```



Static Cyclic scheduling

```
__global__ void AXPY(T* x, T* y, T a, int N) {
 /* BlockId is statically assigned by blockIdx.a */
 for(int i=threadIdx.x+blockIdx.a*blockDim.x; i<N; i+=blockDim.x*gridDim.x) {
    y[i] = a * x[i] + y[i];
 }
}
...
/* Kernel Launch (GridSize < N) */
 AXPY<<< GridSize, BS >>> (x, y, a, N);
```

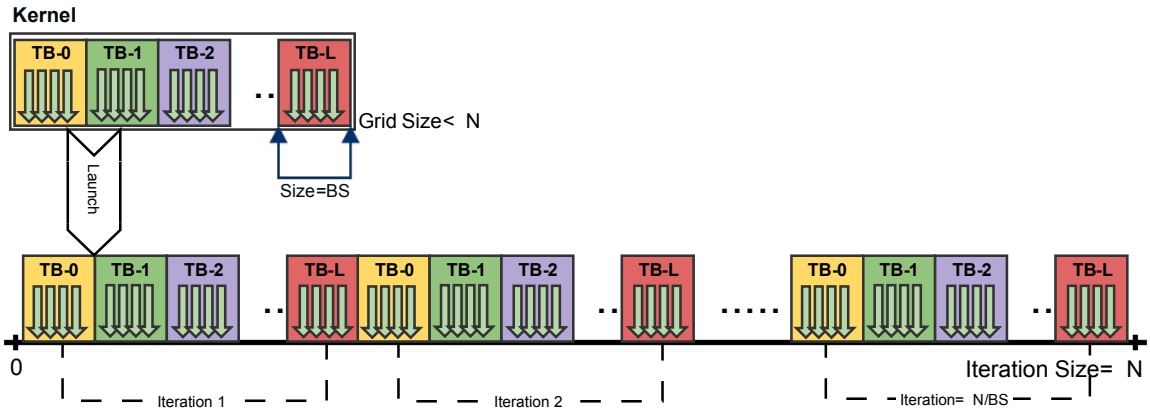
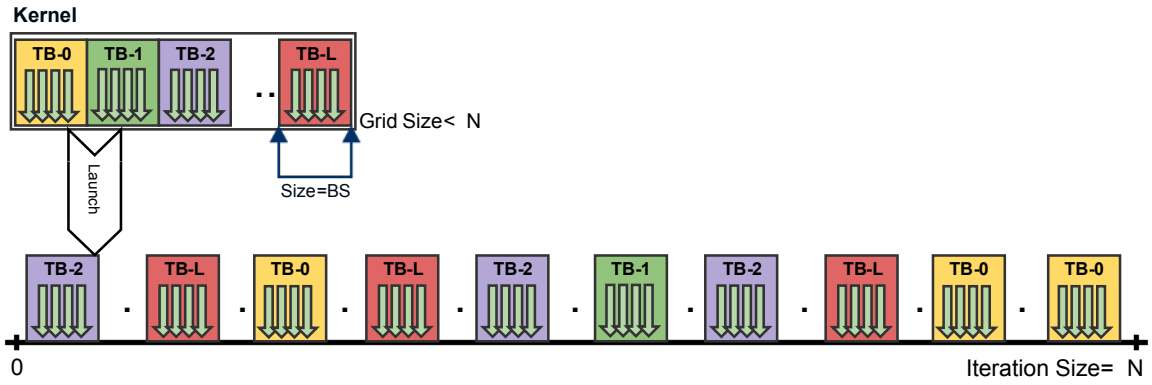


Figure 6.1: The CUDA code of AXPY example using static scheduling methods.

Dynamic scheduling

```
__device__ static int sctr = 0;
__global__ void AXPY_dynamic(T* x, T* y, T a, size_t N) {
  __shared__ int si;
  while(true) {
    if( threadIdx.x == 0 )
      si = atomicAdd(&sctr, 1);
      __syncthreads();
      int i = si * blockDim.x + threadIdx.x;
      if( i >= N ) break;
          y[i] = a * x[i] + y[i];
  }
}
/* Kernel Launch */
 GridSize = MaxActiveConcurrentThread/BS;
 AXPY_dynamic<<< GridSize, BS >>> (x, y, a, N);
```



Optimized Dynamic GPU scheduling

```
#define NT 128
__device__ int sctr = 0;
__global__ void AXPY_opt_dynamic __launch_bounds__(NT)
  (T* x, T* y, T a, size_t N) {
  __shared__ int si;
  int i = threadIdx.x+blockIdx.x*NT;
  loop:
    y[i] = a * x[i] + y[i];
    if( threadIdx.x == 0)  si = atomicAdd(&sctr, 1);
    __syncthreads();
    i = threadIdx.x + gridDim.x * NT + si * NT;
    if( i < N) goto loop;
}
/* Kernel Launch */
 GridSize = MaxActiveConcurrentThread/BS;
 AXPY_opt_dynamic<<< GridSize, BS >>> (x, y, a, N);
```

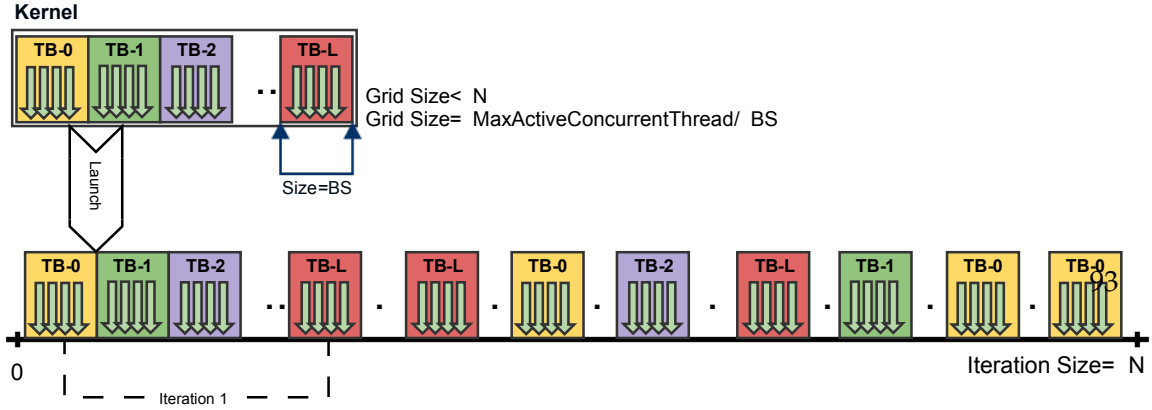


Figure 6.2: The CUDA code of AXPY example using dynamic scheduling methods.

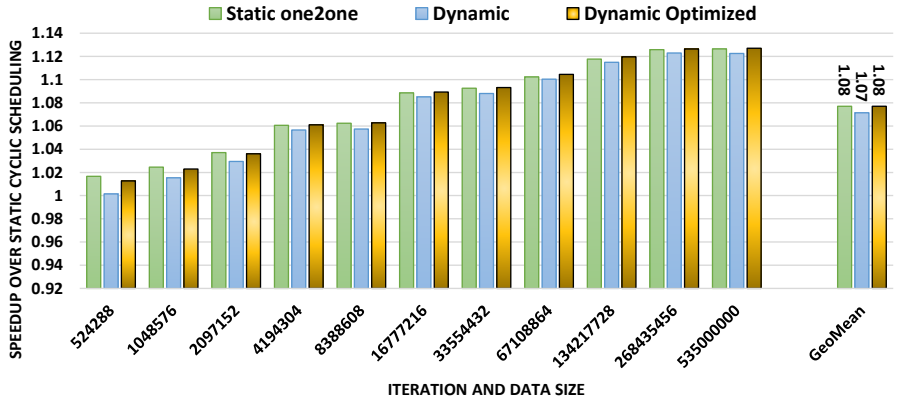


Figure 6.3: Speedup of loop scheduling methods with different trip count.

6.3.3 Preliminary Evaluation

To clarify performance, we run all the AXPY applications with different loop scheduling methods and with different data sizes. We used 1st platform from Section 6.5.1.

We analyzed the performance of each AXPY application from the following perspectives: first how does it change with different trip count, second which grid size is the best. We use static cyclic as our baseline since it is used by directive based compilers. Figure 6.3 shows the speedup graph over static cyclic scheduling. We used the kernels that we show in Figure 6.1, 6.2 with 128 block size and 208 grid size. For static one-to-one scheduling, we create a bigger kernel to map each iteration with each thread. It is worth noting that in AXPY the data size is equal to the trip count of the loop that we want to parallelize. We start evaluating AXPY from trip count 524.288, because we want to ensure that static cyclic scheduling iterates a couple times through the parallel loop. As the results show, static cyclic scheduling yields the worst performance when the kernel size is smaller than the trip count. Another conclusion of this graph is that static one-to-one and optimized dynamic scheduling show close performance with all trip counts and geometric mean. However, we can conclude that when static one-to-one cannot be used, optimized dynamic scheduling achieves the same performance with a small grid size.

We also explore how performance changes with different grid sizes. Figure 6.4 shows the speedup of dynamic scheduling methods over static cyclic scheduling with different grid size when trip count is 535.000.000. Here, the best performance comes from 208 grid size which is not a surprise for us. We explain our method of finding the best grid size in Section 6.4.2. As you can see from graph, finding grid size is tricky but important. If you do not define the right grid size, performance degrades.

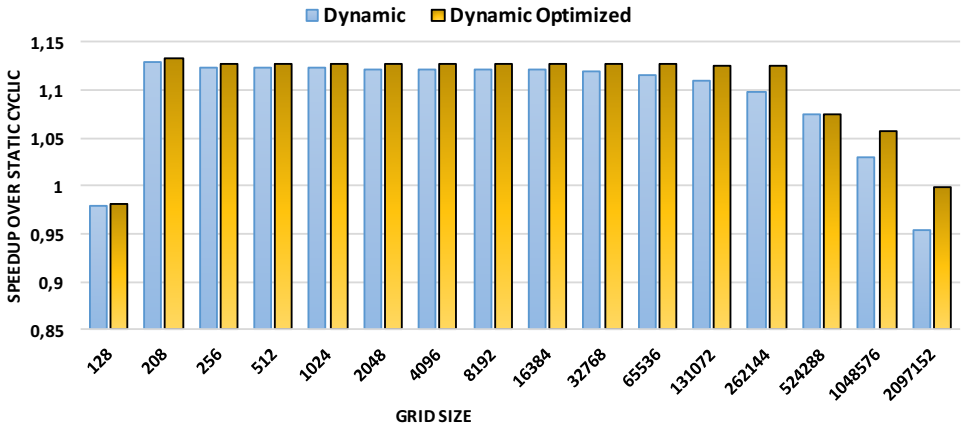


Figure 6.4: Speedup of dynamic loop scheduling methods with different grid size.

6.4 Implementation of Dynamic Scheduling on OpenACC

In this thesis, we have used the PGI OpenACC compiler to implement dynamic scheduling, so we are going to explain the challenges involved. Even though we have already showed the dynamic scheduling code pattern in CUDA, there are two challenges to implementing it in a directive based compiler. The first is that OpenACC allows asynchronous parallel loops, by putting the loop on an async queue (an OpenACC async queue maps to a CUDA stream). This means the loop can run asynchronously with the host CPU, and can run asynchronously with another parallel loop on a different async queue. Each parallel loop is compiled into a GPU kernel, and current GPUs allow a second kernel to start before the first kernel is completely finished. If we have two instances of this kernel running asynchronously, perhaps launched from two host threads, then both copies would be updating the same counter. Our implementation has a fixed limit of async queues, so we addressed this problem by having an array of scheduling counters, one for each possible async queue. The compiler adds an additional argument to the generated kernel containing the async queue number, which is used as the index to the array of counters.

The second problem is that the scheduling counter has to be reinitialized to zero. On a multicore CPU, this is typically done by having the thread that starts the parallel operation reset the counter before initiating the other threads. On a GPU, the thread blocks can't communicate and usually can't even reach a global barrier. New features in NVIDIA Pascal and Volta GPUs with the CUDA 9.0 drivers will support a global barrier in limited cases, which we discuss further below.

We explored two different methods to reinitialize the counter. One method would be to have the host either issue a memset operation or launch a trivial kernel, to reset the counter either just before or just after the computational kernel. This is safe as along

```
1  #define NT 128
2  __device__ int sctr[16] = {0,0,...,0},
3                 sexit[16] = {0,0,...,0};
4  __global__ void AXPY_opt_dynamic_v2 __launch_bounds__(NT)
5    (T *y, T a, T *x, int n, int async) {
6    __shared__ int si;
7    int i = threadIdx.x+blockIdx.x*NT;
8    loop:
9      y[i] = a * x[i] + y[i];
10   if( threadIdx.x == 0)
11     si = atomicAdd(&sctr, 1);
12   __syncthreads();
13   i = threadIdx.x + gridDim.x * NT + si * NT;
14   if( i < N)
15     goto loop;
16   if (threadIdx.x == 0) {
17     int e = atomicAdd(&sedit[async], 1);
18     if (e + 1 == gridDim.x) { // last block
19       sctr[async] = 0;
20       sexit[async] = 0;
21     }
22    }
23 }
```

Figure 6.5: Dynamically-scheduled saxpy with counter reinitialization

as the memset or kernel are on the same async queue, since only one operation can be active on any async queue. However, while the memset or small kernel are fast, the CPU overhead of initiating the memset or launching the kernel is relatively high, much higher than the actual reset operation, so we wanted to avoid this.

The other method is to have the kernel itself reset the counter when the last thread block finishes. This requires a second *exit* counter to keep track of how many thread blocks have completed, and for the last thread block to reset both the scheduling counter and the exit counter. Now, the generated code for the kernel looks like Figure 6.5, with the reset code at lines 16-22. Since this is compiler-generated code, the complexity is completely hidden from programmers.

6.4.1 Limitations

Our method for reinitializing the scheduling counter enforces limitations on which loops can benefit from dynamic scheduling. In particular, it can only be applied to outermost loops or outermost collapsed loops in a compute construct. This method of dynamic scheduling can't be applied to parallel loops contained in an outer sequential loop, or in a procedure called from the compute construct. This is because the scheduling counter is reinitialized only once, when the kernel completes.

The CUDA 9.0 driver with NVIDIA Pascal and Volta GPUs allows kernels to synchronize at a global barrier, across all threads of all thread blocks. This requires launching the kernel differently, and requires the grid to be small enough so that the entire grid can be active at the same time. With this feature, the scheduling counter could be reinitialized in a global critical section just before or just after the parallel loop, which would remove the above restrictions. Our initial experiments using the global barrier were not promising, however. The overhead of the global barrier is high enough to eliminate the benefit of dynamic scheduling, though we are continuing our experiments. Future implementations may improve this overhead.

One advantage of our reinitialization scheme compared to using a global barrier is that all thread blocks except the last can exit, freeing up their GPU resources. If there are other parallel kernels on other async queues, those kernels could start execution before all thread blocks finish.

6.4.2 Complementary Optimization

CTA Throttling CTA throttling limits the number of concurrent CTAs on an SM to reduce the contention for execution resources (e.g., caches and bandwidth). However, naïvely decreasing the total number of CTAs in the kernel to adjust the throttling degree can cause performance degradation due to the imbalance of CTAs on SMs for the hardware scheduler. We propose a software based CTA throttling optimization method. Our method always allocates the maximum number of CTA slots that the GPU has for an application at kernel grid configuration in order to occupy all the CTA slots. Our method finds the exact grid size which the GPU can host at the same time. The formula we use is shown in Figure 6.6. We give the GPU related variables for the NVIDIA Tesla K80 in the same Figure (top). For the sake of simplicity, let us assume that we would like to have a 128 block size. Then when we put all the variables into the formula, we obtain 208 grid size. It is important to note that we used 208 when evaluating the preliminary AXPY in Section 6.3.3. However, in practice calculating the right grid size would be more complicated when dynamic/static shared memory is desired. For those kind of cases, we take advantage of the CUDA Occupancy API.

Kepler K80 variables	*Pascal P100 variables*
`NumOfSM = 13`	`NumOfSM = 56`
`WarpSize = 32`	`WarpSize = 32`
`WarpSlotPerSM = 64`	`WarpSlotPerSM = 64`

Formula of Finding "The Right GridSize"

MaxActiveConcurrentThread *= NumOfSM x WarpSize x WarpSlotPerSM*
GridSize *= MaxActiveConcurrentThread / BS*

Figure 6.6: Formula of Finding Maximum Active Concurrent Threads and Grid Size of GPU

6.5 Experimental Evaluation

This section describes the experimental platform, the test programs, and the results of running those programs with static and dynamic loop scheduling.

6.5.1 Experimental Platform

We ran experiments on two different systems. These are:

1. An NVIDIA Tesla K80 GPU with two GPUs, each with 2449 CUDA cores, compute capability 3.7 and 6GB of device memory. Each GPU includes 13 SM units, each with 192 single-precision CUDA cores. The host processor is 2 x Intel Xeon E5–2630 v3 (Haswell) 8-core processors, (each core at 2.4 GHz,and with 20 MB L3 cache).
2. An NVIDIA Tesla P100 GPU with 3584 CUDA cores, compute capability 6.0 and 16GB of device memory. The GPU includes 56 SM units, each with 192 single-precision CUDA cores The host processor is 2x IBM PowerNV 8335-GTB @ 4.00GHz (10 cores and 8 threads/core, total 160 threads per node). The system is also called Power8 Minsky.

We used NIVIDA PGI compiler 17.10 version. The CUDA 8.0 driver and toolkit was used for both systems.

6.5.2 Benchmarks

Table 6.1 shows the properties of the benchmark programs used for evaluation. We have used twelve applications with different problem sizes as shown in the Problem Size row

Category	Language	Name	Abbrv	Description	Problem Size	Origin
Linear Algebra	C	axpy	SPY	y = a * x + y	[16M], [64M], [256M]	[82]
	C	gemm	MM	dense matrix multiplication	4096	[82]
	C	syrk	SK	symmetric rank-k matrix operation	4096	[82]
	C	syrk2	SK2	symmetric rank-2k matrix operation	4096	[82]
Graph Traversal	C++	bfs	BFS	breadth first search	16M	[44]
	C++	sssp	SP	single-source shortest path	16M	[46]
Iterative Solvers	C	Jacobi 1D	JB1	Jacobi iterative solver 1D array	[16M], [64M], [256M]	[82]
	C	Seidel 2d	SD2	Gauss-seidel iterative solver, 2D array	4096	[82]
Reduction	C	montecarlo pi	PI	pi calculation	[20M], [64M]	[84]
	C	352.ep	EP	embarrassingly parallell benchmark (NAS)	CLASS=D	[83]
	Fortran	Laplace 2d	LP2	Jacobi iterative solver 2D array	4096	[52]
	C	dotproduct	DP	dot product	[16M], [64M], [256M]	[82]

Table 6.1: Benchmarks, descriptions, problem sizes tested, and references.

in the same table. These include seven stnad alone applications and the PolyBench/GPU benchmark suite [82]. Problem sizes for these applications correspond to the vector size for the first two linear algebra programs, and the size of one dimension of the matrix for the other linear algebra and solver programs.

For the bfs and sssp graph traversals from the Rodinia benchmark suite [44], we used a 64K node graph. We used the PI calculation application for calculating monte carlo algorithms, we evaluated it with different problem sizes. The 352.EP application is from the SPEC ACCEL benchmark suite [83], which is each run with class=D datasets. The SPEC ACCEL runs are *estimates*, as they were not run in the SPEC performance harness. We evaluated laplace equations as Jacobi 2d using the Fortran programming language.

Methodology

The PGI 17.10 compiler imposes a maximum limits of 65536 on grid size because creating a bigger grid size is not desirable for many cases as mentioned. Also, it configures thread block size with 128 unless user specifies thread size by using OpenACC clauses. As we would like to compare performance between static cyclic scheduling and our optimized scheduling approach, we should enable static cyclic scheduling. To do this, the trip count of the loop we want to parallelize must exceed *thread block size x grid size*. Otherwise, the compiler can use static one-to-one scheduling as the trip count does not exceed the limit.

In Table 6.1 the **Problem size** column shows the input data set size. We choose problem sizes which always enable static cyclic scheduling, thus we ensure a bigger trip count than the limit. In case of SPY, JB1, PI, DP, problem size equals to trip count of the outer loop. In case of MM, SK, SK2, JB2, SD2, LP2, we use a *collapse(2)* clause on the outermost iteration to fuse the trip counts of two loops. For the EP application, we used the biggest class which the SPEC benchmark provides [83]. For the graph traversal application, we used graph generator application, which comes with Rodinia benchmark, to generate a bigger graph.

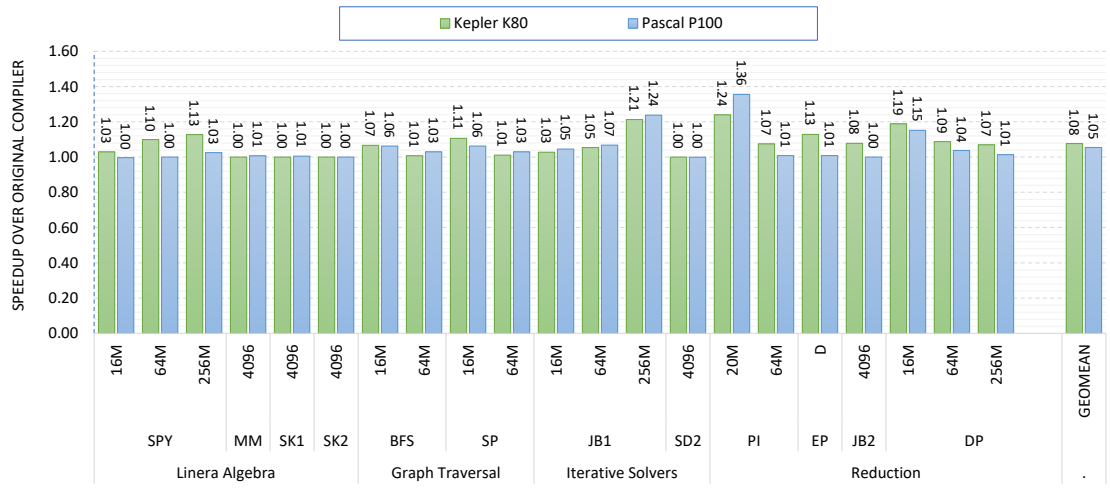


Figure 6.7: Application based speed-up of optized dynamic scheduling over static cyclic scheduling.

6.5.3 Experimental Results

The performance runs were made on the systems described above. For all the benchmarks, the times to compute kernels were measured and compared. We run each kernel multiple times; the results are then the average of multiple runs.

Figure 6.7 shows the speed up of each application with different problem sizes as we show in Table 6.1. The x-axis shows the category of benchmark, the final column shows the geometric mean of all benchmarks. All the results are normalized to the baseline and measured by the average of multiple runs. We took the original PGI compiler, which uses static cyclic scheduling as default, as a basis. Then we compared it with our optimized dynamic loop scheduling. The implementation includes CTA throttling optimization. Enabling CTA throttling improves the performance in almost every case. We intentionally did not measure performance without CTA throttling optimization since we know performance would not be better as we explained in the AXPY example above. Our method yields up to 1.24X and 1.36X speed-up for Kepler K80 and Pascal P100 GPUs respectively with mean speed-up of 1.08X and 1.05X. From these results, we believe that dynamic loop scheduling is effective, giving better performance even with smaller grids, thus enabling the additional benefits of smaller grids.

The mean speed-up for each application is positive and applications which make use of reduction see a better speed-up. The reason for the better reduction operation is that dynamic scheduling reduces significantly the grid size which indirectly reduces the cost of reducing the reduction item. Another important thing is that we have not measured speed up relative to static one-to-one scheduling. As we explained in the experimental methodology section, we forced the compiler to enabled static cyclic scheduling. However, we believe that if we used static one-to-one scheduling, we would

not obtain speed-up except with applications that use reduction.

6.6 Related Work

To the best of our knowledge, all other compilers that generate CUDA-like code from parallel loops use a static schedule. This is the first study to explore alternative compiler-generated loop scheduling methods.

Different architectural solutions have been proposed to schedule warps or thread blocks; none are currently available on real hardware. Kayiran et al [76] propose a thread block scheduling mechanism that estimates the amount of thread-level parallelism to improve GPU performance by reducing cache and DRAM contention. Jog et al [73] propose OWL, a series of thread block-aware scheduling techniques to enhance the performance of DRAM. Fung et al [69] and Mao et al [71] focused on warp scheduling to increase off-chip memory performance. Lee et al [72] proposed alternative thread block scheduling in hardware, restricting the maximum number of thread blocks in each SM to reduce contention. Two other publications [85, 77] propose other hardware-based thread block schedulers. All the hardware proposals were validated using simulators.

Li et al [75] studies scheduling in GPU programming. They propose a method to schedule thread blocks where there is potential data reuse together on the same SM. Although their work is different from static cyclic scheduling and can increase performance, it is not a dynamic loop scheduling technique.

6.7 Conclusions

In this chapter, we described the design, implementation, and evaluation of an optimized dynamic loop scheduling technique for GPUs. This is designed for use by with high-level parallel programming models, such as OpenACC and OpenMP. Our experiments show that dynamic loop scheduling significantly improves performance compared to the common static cyclic schedule, and allows use of smaller grids without performance loss, which then enables other benefits. This is the final significant contribution of this thesis.

7 Conclusions and Future Work

7.1 Conclusions of the Thesis

As we get approach the Exascale era, GPUs have gained great traction due to their performance / power ratio. However, programming GPUs remains a challenging task. To overcome this, several high-level directive-based programming models have emerged that simplify the GPU programming while maintaining the high performance. OpenACC and OpenMP have done a great job by standardizing the programming model for GPUs. Nevertheless, due to lack of code generation algorithms, optimization techniques and unprepared runtimes, these approaches can not become de facto standards for GPU programming. This dissertation tries to address these issues by showing how different parallelization techniques and optimizations can be applied both manually by the user and automatically in the compiler and at runtime.

In this thesis, we presented the following contributions that target GPU compilation within the scope of the OpenMP, OpenACC and OmpSs programming models:

- **A New Dialect Programming Model:** We presented the device model approach for GPU developed in the Mercurium source-to-source compiler as a combination of OpenMP and OmpSs in the Chapter 3. This model is a result of our initial design, implementation, integration, research, and evaluation of compiler algorithms related to a different aspect of GPU code generation infrastructure.

- **Extensions for OpenMP and OpenACC standards:** In Chapter 4, we introduced several different extension ideas to OpenMP and OpenACC standards including new directives and clauses that allow programmers to direct the compiler in the code generation process in GPU device model. We extended the meaning of some of directives and clauses to increase coverage of code scenarios. These directives and clauses provide the compiler with information valuable to generate GPU code while utilizing better GPU hardware. Besides, we propose extensions set to adapt

Modern C++ features such as lambda, variadics templates and usage of their combination. Some of this proposals are included in the new version of OpenMP and OpenACC standards.

- **Multiple Device Management:** We proposed multiple target task sharing model which provides a mechanism to exploit GPUs, CPUs and another accelerators in Chapter 4.1. can automatically manage different device types such as GPUs and CPUs while generating code for them.

- **Code Transformation for GPU compilers:** In Chapter 5, we proposed a code transformation technique that covers all the irregular applications such as sparse matrix, graphs, graphics, etc. We proposed and evaluated a code transformation technique, LazyNP that effectively supports nested parallelism in GPUs, mainly tailored to compilers for user-directed accelerator programming models such as OpenACC or OpenMP. LazyNP dynamically packs kernel invocations and postpones their execution until a bunch of them are available. We propose three different approaches for GPU device based code generation; for one of them we also show how is it possible to exploit nested parallelism in GPUs that do not provide hardware support for dynamic parallelism. Also, two different code generation techniques are proposed for hybrid CPU/GPU usage. LazyNP is evaluated for very relevant algorithms both in sparse scientific computations and graph algorithms, resulting in important speed–ups when compared to eager implementations using dynamic parallelism, other code versions that don't use nested parallelism and well-tuned standard libraries.

- **Optimization Techniques for GPU compilers:** We explore using dynamic scheduling for mapping parallel loop iterations to GPU threads in an NVIDIA PGI OpenACC compiler in Chapter 6. We described the design, implementation, and evaluation of an optimized dynamic loop scheduling technique for GPUs. This is designed for use with high-level parallel programming models, such as OpenACC and OpenMP. Our experiments show that dynamic loop scheduling improves performance compared to the common static cyclic schedule, and allows use of smaller grids without performance loss, which then enables other benefits.

7.2 Impact

This thesis has had significant impact on the fields of programming models and compilers for GPU applications. In addition, it has driven contributions to the OpenMP and OpenACC programming standard. These two standards are widely adopted and used by many people to run and parallelize applications. We presented our proposal to the OpenMP Architecture Review Board (ARB) Committee, OpenACC Language Committee and participated in its refinement. Our proposals on extending OpenMP and OpenACC are still under discussion.

To the best of our knowledge, our LazyNP code transformation is first successful compiler method for transformation of nested directives for GPUs. Although nested directive usage is an allowed pattern in OpenMP and OpenACC, there is no compiler to support the use of it since it is not a successful solution. Our solution is very influential and it has piqued the interest of some members of the IBM compiler team. We are going to move on by implementing it for commonly used compilers such as Clang or PGI.

Our dynamic loop scheduling method is the first research we know of into loop scheduling for GPUs. It is a result of collaboration with the NVIDIA PGI OpenACC compiler team. It is implemented in the NVIDIA PGI OpenACC compiler which is one of the best OpenACC compilers and delivers the best performance also it is a commercial product.

Finally, we have introduced some complementary extensions which we explain in the following section; these have had a direct impact on the OpenMP and OpenACC standards. Our extension proposals influenced the official extensions (OpenMP 5.0 and the next version of OpenACC). In terms of software contribution they were already implemented in the open-source CLANG compiler and are already pushed to the master branch of the Clang compiler during collaboration with the IBM T.J. Watson Research Center. We also integrated support for the PGI NVIDIA OpenACC compiler by collaborating with NVIDIA Corporation. The following list summarize the impact of thesis. The following section explains these extensions in more detail.

- Accepted features by OpenMP or OpenACC Standards
  - Implicitly Function Offload Feature
  - Lambda Expression Support (C++11/14/17)
- Software Impact
  - Mercurium Compiler
  - Clang Frontend
  - PGI Compiler

Implicit declare target and routine

When a function needs to be invoked in an OpenMP *target* or an OpenACC *parallel* or *kernels* region, its definition and declaration must be marked by *declare target* or *routine* in OpenMP and OpenACC respectively. However, it can be very tedious if there is code with many many lines like in Figure 7.1 (a). Programmers must includes required directives as in Figure 7.1 (b).

To ease burden for programmers, we have proposed a compiler based solution that automatically figures out which functions are called in *target*, *kernels* or *parallel* regions.

```
1
2  int bar();
3  int foo() { return bar(); }
4
5  void main () {
6    #pragma omp target
7      foo();
8  }
9
10 int baz() { return 1; }
11 int bar() { return baz(); }
12 ;
```

```
1  #pragma omp declare target
2  int bar();
3  int foo() { return bar(); }
4  #pragma omp end declare target
5  void main () {
6    #pragma omp target
7      foo();
8  }
9  #pragma omp declare target
10 int baz() { return 1; }
11 int bar() { return baz(); }
12 #pragma omp end declare target
```

Figure 7.1: Example of implicit declare target in OpenMP 4.5

After that, the compiler generates device codes for each function automatically. The extension is already in the OpenMP 5.0 document, we have also proposed the same thing to the OpenACC language committee.

Lambda expression of C++

In recent years, the C++ programming language moved into new era. C++11 introduced several major new features, C++14 added even more features on top of C++11 and C++17 is coming with a lot of new features. One of the most important feature of modern C++ is *lambda* which is introduced to provide a convenient way of defining an anonymous function object right at the location where it is invoked or passed as an argument to a function. In general, lambdas are used to encapsulate a few lines of code that are passed to algorithms or asynchronous methods. OpenMP and OpenACC intend to support C++. However their C++ support is out of date and doesn't include the new features in C++11/14.

Using lambda with OpenACC or OpenMP seems like it should be easy as shown in Figure 7.2. However, it is not possible using it with the current standard. Lambda makes it very easy to define anonymous function in C++, however it hides the real function which is generated. This leads a problem for the device model of OpenMP and OpenACC as each function must be marked. When the compiler comes across a lambda, it deduces a struct that involves an `operator()` function and captured data. The code on the bottom in the same Figure shows the deduced middle-level code for lambda expression by compiler. Here, the compiler generates a struct that has `*a, *b` private variables inside and `operator()` function for lambda expression. Also, it replaces the lambda with a struct creation. Now let's imagine this from the perspective of the device model of the OpenMP/OpenACC compiler This expects each function to be marked if the function is used on the device. In this example, `operator()` function is hidden by

C++ language and user has no right to mark it with directives. Therefore, our solution is that compiler must support an implicit declare target and routine feature.

The problem does not end here. As you can see Figure 7.2, lambda's *capture-by-value* feature is used which captures automatically the variables used in the lambda section. In our example **a* and **b* are captured. The compiler creates a lambda struct as mentioned with two member for **a* and **b*. Unfortunately, these data and the struct are hidden by the compiler. If the pointer fields (`*a` and `*b`) of this structure is not translated; it still refer to an address in the host's address space. This leads to invalid references when the lambda runs on a GPU.

To successfully map a structure of arrays, as in this case, the runtime must map not just the structure itself but traverse deeper to also map the fields (pointers) within it. We modified our compiler and runtime to add limited deep copy support for the special case of a lambda function. After mapping the lambda structure, the runtime checks for the presence of each member pointer within the device data environment. If present, the corresponding device address replaces the host pointer value in the field of the mapped lambda structure.

7.3 Publications

The work presented in this thesis resulted in five main publications plus a further pending publication.

The first paper is titled *On the Roles of the Programmer, the Compiler and the Runtime System When Programming Accelerators in OpenMP* [14]. It was published in the *10th International Workshop on OpenMP*, in 2014. This paper introduces our device model infrastructure with OpenMP in Mercurium Compiler.

The second paper is titled *Multiple Target Task Sharing Support for the OpenMP Accelerator Model* [86]. It was published in the *12th International Workshop on OpenMP*, in 2016. This paper was written in collaboration with NVIDIA Corporation and it proposes an extension to the OpenMP directive-based programming model to support the specification and execution of multiple instances of task regions on different devices.

The third paper is called *Exploring Dynamic Parallelism in OpenMP* [47] which is published in *Second Workshop on Accelerator Programming using Directives at Supercomputing 2015 conference*. In this paper, we performed a preliminary evaluation of our nested parallelism in OpenMP.

The fourth article is a short paper titled *Collective Dynamic Parallelism for Directive Based GPU Programming Languages and Compilers* [87] which is published in *International Conference on Parallel Architectures and Compilation, PACT 2016*. It introduces efficient

```
template <typename EXE, typename BODY>
double bench_forall(int s, int e, BODY body)
{
  StartTimer();
  if (is_same<EXE, Serial>::value) {
    for (int i = s; i < e; ++i)
      body(i);
  } else if (is_same<EXE, OpenMP>::value)
  {
    #pragma omp parallel for
    for (int i = s; i < e; ++i)
      body(i);
  } else if (is_same<EXE, OpenMP_device>::value)
  {
    #pragma omp target distribute parallel for
    for (int i = s; i < e; ++i)
      body(i);
  } else if (is_same<EXE, OpenACC>::value)
  {
    #pragma acc parallel loop
    for (int i = s; i < e; ++i)
      body(i);
  }
  return  EndTimer();
}

template<typename T>
void do_bench_saxpy(int N, T *a, T* b, T x)
{
 double stime, time;
 auto saxpy = [=](int i)
         { b[i] += a[i] * x; };

 stime = bench_forall<Serial>(0, N, saxpy);
 time = bench_forall<OpenMP>(0,N, saxpy);
 printf("OpenMP Multicore Speedup %.2lf\n",stime/time);

 #pragma omp target enter data map(in:a[:N], b[:N])
 time = bench_forall<OpenMP_device>(0,N, saxpy);
 printf("OpenMP GPU Speedup %.2lf\n",stime/time);

 #pragma acc enter data copyin(in:a[:N], b[:N])
 time = bench_forall<OpenACC>(0,N, saxpy);
 printf("OpenACC Speedup %.2lf\n",stime/time);
}
```

Figure 7.2: The for_all implementer with lambda that supports serial, OpenMP for multicore, OpenMP_device for GPUs and OpenACC version

nested parallelism code transformations techniques for GPU compilers.

The fifth paper is *Offloading Support for OpenMP in Clang and LLVM* [11]. It was published in the *Third Workshop on the LLVM Compiler Infrastructure in HPC at Supercomputing Conference* in 2016. This paper was written in collaboration with IBM T.J. Watson Research Center and it describes the code generation support on Clang compiler.

The latest paper is still pending. It is called *LazyNP: Effective Code Transformation for Nested Parallelism in User-Directed Accelerator Programming* and is submitted for publication in *IEEE Transactions on Parallel and Distributed Systems*. It improves efficient nested parallelism code transformations techniques for GPU compilers and heterogeneous systems.

In addition to the above publications we plan to produce a further paper about dynamic loop scheduling as described in Section 6.

7.4 Future Work

Our device model infrastructure in Mercurium compiler lays the basis for further research. This research can include new compiler algorithms and approaches that take advantage of instructions of new GPUs architectures to bring.

In the context of the device model extensions for OpenMP and OpenACC there is still room for improvement. More extensions in the programming model's language may be in user-directed optimizations that allow the compiler to use more sophisticated approaches.

Bibliography

[1] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. Betweenness centrality on gpus and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, GPGPU-6, Houston, Texas, USA, March 16, 2013*, pages 76–85, 2013.

[2] G. Chrysos. Intel xeon phi coprocessor - the architecture, intel whitepaper, 2014.

[3] PEZY. Pezy computing - http://pezy.jp/.

[4] K. Underwood. Fpgas vs. cpus: trends in peak floating-point performance,. in Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, ser. FPGA '04. New York, NY, USA: ACM, 2004, pp. 171–180.

[5] TOP500. Supercomputer list. https://www.top500.org/, 2017.

[6] Chris Lomont. Introduction to intel advanced vector extensions. intel white paper, 2011.

[7] NVIDIA. Nvidia nvlink high-speed interconnect, http://www.nvidia.com/object/nvlink.html. *NVIDIA Corporations*, 2013.

[8] Ian Buck, Tim Foley, Daniel Reiter Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[9] NVIDIA. Cuda c programming guide version 7.0. *NVIDIA Corporations*, 2013.

[10] Khronos OpenCL Working Group. The opencl specification, version 2.0, 2014.

[11] Samuel F. Antão, Alexey Bataev, Arpith C. Jacob, Gheorghe-Teodor Bercea, Alexandre E. Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, Tong Chen, Hyojin Sung, Carlo Bertolli, and Kevin O'Brien. Offloading support for openmp in clang and LLVM. In *Third Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2016, Salt Lake City, UT, USA, November 14, 2016*, pages 1–11, 2016.

[12] OpenMP ARB. OpenMP application program interface, v. 4.5. http://www.openmp.org, 2015.

[13] OpenACC. The OpenACC application programming interface, version 2.5. https://www.openacc.org/.

[14] Guray Ozen, Eduard Ayguadé, and Jesús Labarta. On the roles of the programmer, the compiler and the runtime system when programming accelerators in OpenMP. In *Using and Improving OpenMP for Devices, Tasks, and More - 10th International Workshop on OpenMP, IWOMP 2014, Salvador, Brazil, September 28-30, 2014. Proceedings*, pages 215–229, 2014.

[15] PGI Compilers NVIDIA Corporation and Tools. Pgi openacc compiler. http://www.pgroup.com/.

[16] Tianyi David Han and Tarek S. Abdelrahman. hicuda: High-level GPGPU programming. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):78–90, 2011.

[17] Seyong Lee and Rudolf Eigenmann. Openmpc: Extended openmp programming and tuning for gpus. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*, pages 1–11, 2010.

[18] Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee U.S. Department of Energy. Report on exascale computing. https://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf.

[19] Eduard Ayguadé, Rosa M. Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc González, Francisco D. Igual, Daniel Jiménez-González, and Jesús Labarta. Extending openmp to survive the heterogeneous multi-core era. *International Journal of Parallel Programming*, 38(5-6):440–459, 2010.

[20] OmpSs Group. The mercurium at nanos ompss site. https://pm.bsc.es/mcxx.

[21] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[22] Clang. Clang compiler. http://clang.llvm.org/.

[23] NVIDIA. Volta gpu architecture. https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/, 2017.

[24] Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*, pages 142–151, 2008.

[25] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *ACM/IEEE CONFERENCE ON SUPERCOMPUTING*, page 86. ACM, 2006.

[26] J. Bueno, J. Planas, A. Duran, R.M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of gpu clusters with ompss. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 557–568, May 2012.

[27] Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*, pages 142–151, 2008.

[28] Carlo Bertolli, Samuel Antão, Alexandre E. Eichenberger, Kevin O'Brien, Zehra Sura, Arpith C. Jacob, Tong Chen, and Olivier Sallenave. Coordinating GPU threads for openmp 4.0 in LLVM. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, LLVM 2014, New Orleans, LA, USA, November 17, 2014*, pages 12–21, 2014.

[29] Free Software Foundation. Gcc, the gnu compiler collection, offload support. https://gcc.gnu.org/wiki/Offloading.

[30] NVIDIA. Faster parallel reductions on kepler, https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/.

[31] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[32] Balaram Sinharoy, James Van Norstrand, Richard J. Eickemeyer, Hung Q. Le, Jens Leenstra, Dung Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, José E. Moreira, D. Levitan, S. Tung, David Hrusecky, James W. Bishop, Michael Gschwind, Maarten Boersma, Michael Kroener, Markus Kaltenbach, Tejas Karkhanis, and K. M. Fernsler. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1), 2015.

[33] CAPS entreprise. Caps compilers-3.3 hmppcg directives reference-manual, 2012.

[34] S. Vadlamani, Youngsung Kim, and J. Dennis. Dg-kernel: A climate benchmark on accelerated and conventional architectures. In *Extreme Scaling Workshop (XSW), 2013*, pages 51–57, Aug 2013.

[35] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Self-adaptive ompss tasks in heterogeneous environments. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*, pages 138–149, 2013.

[36] Thomas R. W. Scogland, Wu-chun Feng, Barry Rountree, and Bronis R. de Supinski. Coretsar: Core task-size adapting runtime. *IEEE Trans. Parallel Distrib. Syst.*, 26(11):2970–2983, 2015.

[37] Martin Tillenius, Elisabeth Larsson, Rosa M. Badia, and Xavier Martorell. Resource-aware task scheduling. *ACM Trans. Embedded Comput. Syst.*, 14(1):5:1–5:25, 2015.

[38] NVIDIA. Cuda dynamic parallelism programming guide, 2013.

[39] Yi Yang and Huiyang Zhou. CUDA-NP: realizing nested thread-level parallelism in GPGPU applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 93–106, 2014.

[40] Jin Wang and Sudhakar Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*, pages 51–60, 2014.

[41] NVIDIA. Parallel thread execution isa version 4.3. http://docs.nvidia.com/cuda/parallel-thread-execution/.

[42] KHRONOS. The first open standard intermediate language for parallel compute and graphics. https://www.khronos.org/spir.

[43] Duane Merrill, Michael Garland, and Andrew S. Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 117–128, 2012.

[44] 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 *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*, pages 44–54, 2009.

[45] Andre Vincent Pascal Grosset, Peihong Zhu, Shusen Liu, Suresh Venkatasubramanian, and Mary W. Hall. Evaluating graph coloring on gpus. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 297–298, 2011.

[46] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *High Performance Computing - HiPC 2007, 14th International Conference, Goa, India, December 18-21, 2007, Proceedings*, pages 197–208, 2007.

[47] Guray Ozen, Eduard Ayguadé, and Jesús Labarta. Exploring dynamic parallelism in openmp. In *Proceedings of the Second Workshop on Accelerator Programming using Directives, WACCPD 2015, Austin, Texas, USA, November 15, 2015*, pages 5:1–5:8, 2015.

[48] NVIDIA. Next generation cuda compute architecture: Kepler tm gk110 http://www.nvidia.es/content/pdf/kepler/nvidia-kepler-gk110-architecture-whitepaper.pdf.

[49] NVIDIA. Next generation cuda compute architecture: Maxwell http://www.nvidia.es/content/pdf/kepler/nvidia-kepler-gk110-architecture-whitepaper.pdf.

[50] Andrew Adinetz NVIDIA. Cuda pro tip: Optimized filtering with warp-aggregated atomics, https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/.

[51] Yash Ukidave, Fanny Nina Paravecino, Leiming Yu, Charu Kalra, Amir Momeni, Zhongliang Chen, Nick Materise, Brett Daley, Perhaad Mistry, and David Kaeli. Nupar: A benchmark suite for modern gpu architectures. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 253–264, New York, NY, USA, 2015. ACM.

[52] NVIDIA. Cuda samples. http://docs.nvidia.com/cuda/cuda-samples/, 2016.

[53] NVIDIA. Nvidia: The nvidia cuda sparse matrix library (cusparse), v 7.5. 2015.

[54] CUSP. Generic parallel algorithms for sparse matrix and graph computations. http://cusplibrary.github.io/. http://cusplibrary.github.io/, June 2016.

[55] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1, 2011.

[56] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[57] Guoyang Chen and Xipeng Shen. Free launch: optimizing GPU dynamic kernel launches through thread reuse. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, pages 407–419, 2015.

[58] Wilson W. L. Fung, Ivan Sham, George L. Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40 2007), 1-5 December 2007, Chicago, Illinois, USA*, pages 407–420, 2007.

[59] Wilson W. L. Fung, Ivan Sham, George L. Yuan, and Tor M. Aamodt. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *TACO*, 6(2), 2009.

[60] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving GPU performance via large warps and

two-level warp scheduling. In *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, 3-7 December 2011, Porto Alegre, Brazil*, pages 308–317, 2011.

[61] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. Dynamic thread block launch: a lightweight execution mechanism to support irregular applications on gpus. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 528–540, 2015.

[62] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, pages 101–110, 2009.

[63] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. Streamlining gpu applications on the fly: Thread divergence elimination through runtime thread-data remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 115–126, New York, NY, USA, 2010. ACM.

[64] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8, New York, NY, USA, 2011. ACM.

[65] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Efficient warp execution in presence of divergence with collaborative context collection. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, pages 204–215, 2015.

[66] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14, May 2012.

[67] Hancheng Wu, Da Li, and Michela Becchi. Compiler-assisted workload consolidation for efficient dynamic parallelism on GPU. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 534–543, 2016.

[68] Izzat El Hajj, Juan Gómez-Luna, Cheng Li, Li-Wen Chang, Dejan S. Milojicic, and Wen-mei W. Hwu. KLAP: kernel launch aggregation and promotion for optimizing dynamic parallelism. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 1–12, 2016.

[69] Wilson W. L. Fung, Ivan Sham, George L. Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40 2007), 1-5 December 2007, Chicago, Illinois, USA*, pages 407–420, 2007.

[70] NVIDIA Corporationn. Nvidia's next generation cuda compute architecture: Fermi. comput. syst, 26:63–72, 2009.

[71] Mengjie Mao, Wujie Wen, Xiaoxiao Liu, Jingtong Hu, Danghui Wang, Yiran Chen, and Hai Li. TEMP: thread batch enabled memory partitioning for GPU. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, pages 65:1–65:6, 2016.

[72] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeon-Gon Cho, and Soojung Ryu. Improving GPGPU resource utilization through alternative thread block scheduling. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 260–271, 2014.

[73] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 395–406, 2013.

[74] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeon-Gon Cho, and Soojung Ryu. Improving GPGPU resource utilization through alternative thread block scheduling. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 260–271, 2014.

[75] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. Locality-aware CTA clustering for modern GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 297–311, 2017.

[76] Onur Kayiran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, pages 157–166, 2013.

[77] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Orchestrated scheduling and prefetching for GPGPUs. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pages 332–343, 2013.

[78] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, 28-30 March 2010, White Plains, NY, USA*, pages 235–246, 2010.

[79] An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002.

[80] Peiyi Tang and Pen-Chung Yew. Processor self-scheduling for multiple-nested parallel loops. In *International Conference on Parallel Processing, ICPP'86, University Park, PA, USA, August 1986.*, pages 528–535, 1986.

[81] Clyde P. Kruskal and Alan Weiss. Allocating independent subtasks on parallel processors. *IEEE Trans. Software Eng.*, 11(10):1001–1016, 1985.

[82] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10. IEEE, 2012.

[83] Guido Juckeland, William C. Brantley, Sunita Chandrasekaran, Barbara M. Chapman, Shuai Che, Mathew E. Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John A. Stratton, Alexey Titov, Ke Wang, G. Matthijs van Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. SPEC ACCEL: A standard application suite for measuring hardware accelerator performance. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation - 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers*, pages 46–67, 2014.

[84] OLCF User Assistance Center. Accelerating serial code for gpus. https://www.olcf.ornl.gov/tutorials/cuda-monte-carlo-pi/.

[85] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. Laperm: Locality aware scheduler for dynamic parallelism on GPUs. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 583–595, 2016.

[86] Guray Ozen, Sergi Mateo, Eduard Ayguadé, Jesús Labarta, and James Beyer. Multiple target task sharing support for the openmp accelerator model. In *OpenMP: Memory, Devices, and Tasks - 12th International Workshop on OpenMP, IWOMP 2016, Nara, Japan, October 5-7, 2016, Proceedings*, pages 268–280, 2016.

[87] Guray Ozen, Eduard Ayguadé, and Jesús Labarta. POSTER: collective dynamic parallelism for directive based GPU programming languages and compilers. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*, pages 423–424, 2016.