Performance Prediction of Applications Executed on GPUs using a Simple Analytical Model and Machine Learning Techniques

Marcos Tulio Amarís González

Thesis presented
to the
Institute of Mathematics and Statistics
of the
University of São Paulo
to
Obtain the title
of
Doctor in Computer Science

Supervisor: Prof. Dr. Alfredo Goldman
Co-supervisor: Prof. Dr. Raphael Yokoingawa de Camargo

During the development of this work the author received financial grants from FAPESP, Proccesses No. 2012/23300-7 and 2015/19399-6.

São Paulo, June 2018

Performance Prediction of Applications Executed on GPUs using a Simple Analytical Model and Machine Learning Techniques

This version of the thesis has the corrections and modifications suggested for the Thesis committee during the defense of the original version of the work, held on 25/06/2018 in São Paulo, Brazil. A copy of the original version is available in the Institute of Mathematics and Statistics of the University of São Paulo.

Examining Committee:

- Dr. Alfredo Goldman (Advisor - President)
  Professor, University of São Paulo (IME - Brazil)
- Dr. Arnaud Legrand
  Researcher Scientist, Grenoble Informatics Laboratory (LIG - France)
- Dr. Hermes Senger
  Professor, University Federal of São Carlos (UFSCAR - Brazil)
- Dr. Liria Matsumoto Sato
  Professor, University of São Paulo (EP - Brazil)
- Dr. Philipe Navaux
  University Federal of Rio Grande do Sul (UFRGS - Brazil)

For my Family; my mother, my brother and my sister.

Acknowledgment

"***Heart has its reasons of which reason knows nothing***"
— **Blaise Pascal**

First to all, thanks to that vital energy, which us, human beings have named of **God**. Because, It has put in my way many good people which has helped me in anyway to finish this Ph.D.

Thanks to my advisor, Prof. Dr. Alfredo Goldman for his counsels as advisor and friend, because he did it possible. Thanks to my Co-advisor, Prof. Dr. Raphael Yokoingawa de Camargo for all his technical advices and guide during this cycle. Special thankful to Prof. Dr. Denis Trystram, who was my tutor during one year in a doctoral internship at the University of Grenoble Alps. Thanks to Prof. Dr. Jean-Luc Gaudiot, who received me for three months as a PhD Visitor Student at the University of California Irvine. Thanks to Prof. Dr. Fabio Kon at the IME for having trust in me and opened the doors to my Doctorate. Thanks to Professor Dr. Daniel Cordeiro who gave me important counsels in the beginning of this academical cycle.

Thanks to all and each one of the members of the Software System Laboratory in the CCSL at IME. Special thankful to Nelson Lago, Emilio Francesquini, Vinicius Pinheiros, Pedro Bruel, Graziela Tonin and Higor Amario. Thanks to the member of the teams POLARIS and DATAMOVE at the Grenoble Institute of Technology, because they made my internship in France much more pleasant. Specially thanks to Bruno Raffin, Arnaud Legrand, Josu Doncel, Vinicius Garcia, Annie Simon, Raphaël Bleuse, Pierre Neyron, Julio Toss, Clement Mommessin, Giorgio Lucarelli, Ezequel Tristan and Théophile Terraz. Thanks to the member of the PASCAL group at the University of California Irvine. Thanks to Tongsheng and Nazanin.

Thanks to my Cousins Tito and Rafita, because 12 years ago, they exhorted me to believe that this Doctorate may be possible. Thanks to my girlfriend and partner Antônia Fernanda because her kindness and caring support helped to do it possible. Thanks to my roommates, Juan Manuel, Pedro Pablo and Maria Joana, for the nice moments that we have shared.

Resumo

Amarís, M. **Predição de Desempenho de Aplicações Executadas em GPUs usando um Modelo Analítico Simples e Técnicas de Aprendizado de Máquina.** Tese (Doutorado) - Instituto de Matemática e Estatística, Universidade de São Paulo, SP-Brasil, Junho de 2018. 95 páginas.

As plataformas paralelas e distribuídas de computação de alto desempenho disponíveis hoje se tornaram mais e mais *heterogêneas* (CPUs, GPUs, FPGAs, etc). As Unidades de processamento gráfico são co-processadores especializados para acelerar operações vetoriais em paralelo. As GPUs tem um alto grau de paralelismo e conseguem executar milhares ou milhões de *threads* concorrentemente e ocultar a latência do escalonador. Elas tem uma profunda hierarquia de memória de diferentes tipos e também uma profunda configuração da memória hierárquica. A predição de desempenho de aplicações executadas nesses dispositivos é um grande desafio e é essencial para o uso eficiente dos recursos computacionais de máquinas com esses co-processadores. Existem diferentes abordagens para fazer essa predição, como técnicas de modelagem analática e aprendizado de máquina.

Nesta tese, nós apresentamos uma análise e caracterização do desempenho de aplicações executadas em Unidades de Processamento Gráfico de propósito geral. Nós propomos um modelo simples e intuitivo fundamentado no modelo BSP para predizer a execução de funções kernels de CUDA sobre diferentes GPUs. O modelo está baseado no número de computações e acessos à memória da GPU, com informação adicional do uso das memórias cachês obtidas do processo de *profiling*. Nós também comparamos três diferentes enfoques de aprendizado de máquina (ML): Regressão Linear, Máquinas de Vetores de Suporte e Florestas Aleatórias com o nosso modelo analítico proposto. Esta comparação é feita em dois diferentes contextos, primeiro, dados de entrada ou *features* para as técnicas de aprendizado de máquinas eram as mesmas que no modelo analítico, e, segundo, usando um processo de extração de *features*, usando análise de correlação e *clustering* hierarquizado.

Nós mostramos que aplicações executadas em GPUs que escalam regularmente podem ser preditas com modelos analíticos simples e um parâmetro de ajuste. Esse parâmetro pode ser usado para predizer essas aplicações em outras GPUs. Nós também demonstramos que abordagens de ML proveem predições aceitáveis para diferentes casos e essas abordagens não exigem um conhecimento detalhado do código da aplicação, características de hardware ou modelagens explícita. Consequentemente, sempre e quando um banco de dados com informação de *profiling* esteja disponível ou possa ser gerado, técnicas de ML podem ser úteis para aplicar uma predição automatizada de desempenho para escalonadores de aplicações em arquiteturas heterogêneas contendo GPUs.

Palavras-chaves: Predição de Desempenho, Máquinas de Aprendizado, Modelo BSP, Unidades de Processamento Gráfico, CUDA.

Abstract

Amarís, M. **Performance Prediction of Application Executed on GPUs Using a Simple Analytical Model and Machine Learning Techniques**. Thesis (Doctorate) - Institute of Mathematics and Statistics, University of São Paulo, SP-Brazil, June 2018. 95 pages.
The parallel and distributed platforms of High Performance Computing available today have became more and more *heterogeneous* (CPUs, GPUs, FPGAs, etc). Graphics Processing Units (GPU) are specialized co-processor to accelerate and improve the performance of parallel vector operations. GPUs have a high degree of parallelism and can execute thousands or millions of threads concurrently and hide the latency of the scheduler. GPUs have a deep hierarchical memory of different types as well as different configurations of these memories.

Performance prediction of applications executed on these devices is a great challenge and is essential for the efficient use of resources in machines with these co-processors. There are different approaches for these predictions, such as analytical modeling and machine learning techniques.

In this thesis, we present an analysis and characterization of the performance of applications executed on GPUs. We propose a simple and intuitive BSP-based model for predicting the CUDA application execution times on different GPUs. The model is based on the number of computations and memory accesses of the GPU, with additional information on cache usage obtained from profiling. We also compare three different Machine Learning (ML) approaches: Linear Regression, Support Vector Machines and Random Forests with BSP-based analytical model. This comparison is made in two contexts, first, data input or features for ML techniques were the same than analytical model, and, second, using a process of feature extraction, using correlation analysis and hierarchical clustering.

We show that GPU applications that scale regularly can be predicted with simple analytical models, and an adjusting parameter. This parameter can be used to predict these applications in other GPUs. We also demonstrate that ML approaches provide reasonable predictions for different cases and ML techniques required no detailed knowledge of application code, hardware characteristics or explicit modeling. Consequently, whenever a large data set with information about similar applications are available or it can be created, ML techniques can be useful for deploying automated on-line performance prediction for scheduling applications on heterogeneous architectures with GPUs.
Keywords: Performance Prediction, Machine Learning, BSP model, GPU Architectures, CUDA.

Contents

List of Abbreviations

API	Application Programming Interface
BS	Block Size
BSP	Bulk Synchronous Parallel Model
CGM	Coarse Grain Multi-computer Model
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
ECC	Error-Correcting Code
GPC	Graphic Processing Cluster
GPGPU	General Purpose Computing on GPUs
GPU	Graphic Processing Unit
GS	Grid Size
JMS	Job Management System
LD/ST	Load/Store
NVCC	NVIDIA's CUDA Compiler
NVPROF	NVIDIA's CUDA Profiler
OpenCL	Open Computing Language
PRAM	Parallel Random Access Machine Model
PTX	Parallel Thread eXecution
RF	Random Forest
SFU	Special Function Unit
SP	Streaming Processor
SM	Streaming Multi-processor
SVM	Support Vector Machine
TPC	Texture Processing Clusters

Lists of Symbols

$comp$	Computation done by a single thread
$comm_{GM}$	Communication done by a single thread on the global memory
$comm_{SM}$	Communication done by a single thread over the shared memory
g_{GM}	Communication latency of the global memory
g_{SM}	Communication latency of the shared memory
g_{L1}	Communication latency of the L1 cache
g_{L2}	Communication latency of the L2 cache
$L1$	Hits over L1 cache memory
$L2$	Hits over L2 cache memory
ld_0	Load instructions done over shared memory by a single thread
ld_1	Load instructions done on global memory by a single thread
st_0	Store instructions done over shared memory by a single thread
st_1	Store instructions done on global memory by a single thread
t_k	predicted execution time of a kernel
λ	Parameter of adjust of the simple BSP-based GPU model

List of Figures

List of Tables

Chapter 1.

Introduction and Motivation

Parallel and distributed platforms available today are becoming more and more *heterogeneous*. Such heterogeneous architectures, composed of several kinds of computing units, have had a growing impact on performance in high-performance computing. Hardware accelerators, such as General Purpose Graphical Processing Units (in short GPUs), are often used in conjunction with multiple computing units (CPUs) on the same motherboard sharing the same common memory. For instance, the number of platforms of the TOP-500-Supercomputer (2017) equipped with accelerators has significantly increased during the last years. In the future it is expected that the nodes of these platforms will be even more diverse than today: they will be composed of fast computing nodes, hybrid computing nodes mixing general purpose units with accelerators, I/O nodes, nodes specialized in data analytics, etc.

The GPUs were initially conceived with the purpose to accelerate the 2D and 3D graphics tasks. However, the evolution of the GPU architectures has gone from a single core to a set of dense arrays of highly parallel cores for more general purpose computation. Not surprisingly, the use of this hardware for parallel computing is a very attractive and it is used in several scientific areas. Moreover, nowadays GPUs can be found in a wide number of electronic devices, just to name a few: servers, supercomputers and cars (Verber, 2014, chap. 10).

GPUs are integrated with multi-core CPUs with the goal to boost more computational processing power. In this way, the regular users have access to a massively parallel environment when acquiring any kind of these devices. Figure 1.1 shows the theoretical performance and bandwidth reached for those devices. This figure shows a comparison between NVIDIA GPUs and current x86 CPUs. The predominant manufacturer of CPUs are Intel and AMD, and of GPUs are NVIDIA and AMD.

GPUs are composed of hundreds and even thousands of simple cores. Figure 1.2 shows a generic scheme of a heterogeneous system composed of a CPU and a GPU. In this figure we can see that CPUs have much more transistors for control and cache; on other hand, GPUs have much more ALUs and few transistors for cache and control. This is observed by the different color in Figure 1.2, where, the orange color is used to paint memory transistors, green is used to represent Arithmetic Logic Units and yellow is used to paint control transistors.

In the last decade with the development of application programming interfaces for GPUs emerged the concept of General Purpose Computing on GPU or GPGPU. GPGPU has evidenced

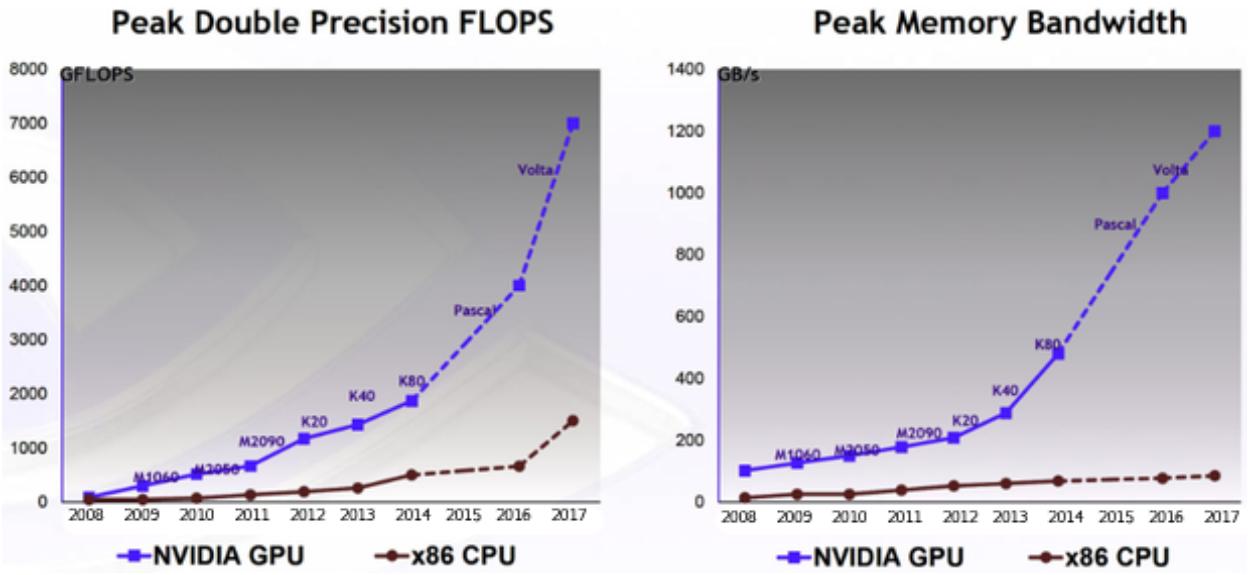

Figure 1.1: *Theoretical performance and bandwidth reached for recent GPUs. Copied from NVIDIA (2018)*

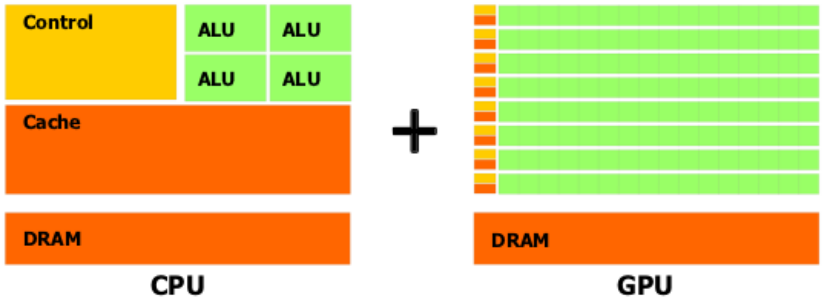

Figure 1.2: *Multiples Cores CPU + Hundreds or Thousands Cores GPU. Copied from NVIDIA (2018)*

outstanding results in many scientific areas, however, tools to further capitalize these parallel devices and more refined implementations are still emerging (Hou *et al.*, 2008; NVIDIA, 2018). In GPU applications the computation is done with a large amount of data, which expose restrictions on the storage space available in the device and the latency of data transfers (Gregg and Hazelwood, 2011).

Researchers and developers in GPGPU area can create applications with a concept of parallelism that are able to run on both CPU and GPU architectures. Nevertheless, the implementation of an algorithm, for parallel execution on CPU or GPU, does not guarantee that will execute with the same efficiency in the two architectures. In particular, algorithms with parallelism in large blocks of data, i.e. applications with intensity in array arithmetic and linear data structures, can obtain great benefits when they are implemented for execution exclusively on a GPU (Gaster and Howes, 2012; Zhong *et al.*, 2012).

Nowadays, GPUs are general purpose parallel processing units with accessible programming interfaces, including standard languages such as C, Java, and Python. In particular, the Compute Unified Device Architecture (CUDA) is a parallel computing platform that facilitates the development of applications which execute in GPU (NVIDIA, 2018). CUDA was introduced by NVIDIA in 2006 for their GPU hardware line. CUDA is a SIMT programming model, it is an extension of SIMD classification according to Flynn architecture. SIMD classification describes computers with

multiple processing elements that perform the same operation on multiple data points simultaneously. In CUDA, each thread accesses to different memory addresses. CUDA is an extension that developers of different languages can use.

CUDA applications are organized in kernels, which are functions executed in GPUs. Kernels are compounded of thread blocks of execution and each one of these blocks can have hundreds or even thousands of threads. Each kernel is executed asynchronously, and various kernels can execute concurrently. The programming model allows directive of synchronization in different levels of the memory hierarchy. In order to use all the computational power available, applications should be composed of multiple tasks that allow the usage of the available resources as efficiently as possible. The Job Management System (JMS) is the middleware responsible for distributing computing power to applications.

A JMS must allocate resources to tasks in order to optimize the use of the available resources while guaranteeing good performance for all applications running concurrently. A promising way to achieve this is by using performance prediction. Performance prediction would improve the scheduling of these applications and circumstantially the usage of the available resources. However, in a scenario with millions of processors and a large number of tasks it is very difficult to predict the performance of applications.

Job management systems require that users provide an upper bound of the execution times of their jobs (wall time). Usually, if the execution goes beyond this upper bound, the job is killed. According to Gaj *et al.* (2002), this leads to very bad estimations, with an obvious bias that tends to overestimate this duration.

The software development process and performance modeling of GPU applications require a high degree of understanding and knowledge (Baghsorkhi *et al.*, 2010). Performance prediction is often just ignored in a development process, this is due to the in-depth analysis that must be done. Models with specific properties to these problems have been created (Skillicorn and Talia, 1998) to facilitate programming on parallel machines. Properties such as synchronization and division of work between computing and communication improve the way developers can solve a problem and how the performance of applications running on parallel machines can be modeled.

The accuracy of a GPU performance model is subject to low-level elements such as instruction pipeline usage and cache hierarchy. GPU performance approaches its peak when the instruction pipeline is saturated but becomes unpredictable when the pipeline is under-utilized (Baghsorkhi *et al.* , 2010). Effects of cache hierarchy and memory-access divergence are also critical to GPU performance models. Some parallel programs can be efficiently executed on some architectures, but not on others.

Performance prediction over these devices is a great challenge because hardware characteristics can impact their performance in different ways. There are different approaches to do this, such as analytical modeling and machine learning techniques. Analytic predictive models are useful, but require the manual inclusion of interactions between architecture and software, and may not capture the complex interactions in GPU architectures. Machine learning techniques can learn to capture these interactions without manual intervention but can require large training sets (Bukh, 1992).

There are different models of parallel machines that help to deal with these problems. Among these models, we have the Parallel Random Access Memory (PRAM), Bulk Synchronous Parallel

(BSP) and Coarse Grained Multicomputer (CGM). The main objective of a parallel computing model is to provide a set of parameters that have an impact on the performance of applications in parallel architectures, and thus to be able to simulate the behavior of these applications in their respective architectures.

The Bulk Synchronous Parallel is a computing model for parallel computing introduced for Valiant (1990), BSP model provides a simple but accurate characterization of all parallel machines-past, present, and future. There are other parallel models parameterized, almost all of them using or extending the focus of the BSP model. Dehne *et al.* (2002) studied the problem to design scalable parallel geometric algorithms for the Coarse Grained Multicomputer model which are optimal or at least efficient for a wide range of the ratio $\frac{n}{p}$, with n the size of the problem and p the number of processors.

The BSP model allows developers to design algorithms and software that can run on any standard parallel architecture with very high performance (Goldchleger *et al.*, 2005a). This model may potentially serve as a unified programming model for both coarse-grained and fine-grained. The model BSP has been implemented as API libraries and programming languages (Holl *et al.*, 1998) and API libraries have been developed to use BSP on GPU architectures (Hou *et al.*, 2008). These developments aim to create scientific applications in massively parallel environments computing in an easy and better way. Recently, Valiant (2011) has implemented the BSP model over multicore architectures. The implementation of BSP model for multicore architecture incorporates the memory size shared as a further parameter in the computer. The BSP model is a good alternative to tackle parallel problems in massively parallel architectures.

The developed methods in the area of machine learning can be also used to tackle the prediction of performance of GPU applications. Information from JMS can be used as data input for different machine learning techniques. This will be useful for JMS in new computing platforms to large scale. Information about profiling and traces of heterogeneous applications can be used to improve current JMS, which require a better knowledge of the applications (Emeras *et al.*, 2014).

This thesis is related to the modeling and performance prediction of applications executed in GPUs using two approaches: an analytical model and machine learning techniques. Considering that the market has been taken by NVIDIA graphics cards, our models and experimentation are done on GPUs manufactured by NVIDIA. We evaluated our model using 9 matrix/vector operations kernels and other kernels from Rodinia Benchmark suite (Che *et al.*, 2009), Rodinia has been used and accepted by the GPGPU community (Che *et al.*, 2010; Nagasaka *et al.*, 2010), and is used in various GPU simulators (Power *et al.*, 2014).

These CUDA kernels were executed over GPUs with different architectures among these architectures are Kepler, Maxwell, and Pascal. We showed by using profile information for a single board, that the models are general enough to predict the execution time of an application with different input sizes and on different boards.

1.1 Objectives and Thesis Contributions

The main propose of this study is to present an analysis and design of techniques to predict the performance of applications executed on General-purpose Graphic Processing Units. For this, we present an analysis and design of a simple and intuitive BSP-based model to predict the performance of GPU applications (Amaris *et al.*, 2015). The model is based on the number of computations and memory accesses of the GPU, with additional information on cache usage obtained from profiling.

Scalability, divergence, the effect of optimizations and differences of architectures are adjusted by a single parameter.

We also evaluate the use of different machine learning techniques to reach the same objective. We compare three different machine learning approaches: Linear Regression, Support Vector Machines and Random Forests with our BSP-based analytical model (Amaris *et al.*, 2016). Two approaches were tested with machine learning techniques. First, machine learning techniques used similar features than the analytical model and, second, a process of feature extraction was performed. For the second approach, we proposed a process implementing a correlation analysis and hierarchical clustering to reduce the number of features or dimensionality.

Our main contribution was to show that machine learning techniques provided acceptable predictions for all the applications over all the GPUs. Although the analytical model provided better predictions for the applications which scale regularly, it requires sometime a deep analysis and knowledge of the applications and hardware structure. Consequently, whenever a database with profile information is available or can be generated, ML techniques can be useful for deploying automated on-line performance prediction for scheduling applications on heterogeneous architectures containing GPUs. This contribution was presented in the Doctoral Showcase of the International Conference for High Performance Computing, Networking, Storage and Analysis of 2017. In this International conference, a summary of the thesis was presented in a short presentation and a poster session.

During my Doctorate, I have done two different international internships, which I worked with high-quality researchers. First, I did an internship of 12 months at the Grenoble Informatics Laboratory in France, and after, I was 3 months at the University of California Irvine. In the first internship I worked with the team DATAMOVE (Data Aware Large Scale Computing) and in the second I was a visitor in the PASCAL (PArallel Systems & Computer Architecture Lab). In both internships, I worked on topics related to this thesis. I will mention the main works related to this thesis in the end of this document. A summary of all the publications in conference proceedings and journals are mentioned in the section below.

1.2 List of Publications

We list the publications directly related to this thesis. They are ordered chronologically, first the more recent.

Bibliography

Marcos Amaris, Raphael Y. de Camargo and Alfredo Goldman. Performance Prediction Modeling of GPU Applications: Analytical Modeling and Machine Learning In *2017 Doctoral Showcase SC17*, poster session. International Conference for High Performance Computing, Networking, Storage and Analysis. Denver-CO, USA.

Marcos Amaris, Raphael Y. de Camargo, Mohammed Dyab, Alfredo Goldman and Denis Trystram. A comparison of GPU execution time prediction using machine learning and analytical modeling. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pages 326–333. Boston-MA, USA.

Marcos Amaris, Daniel Cordeiro, Alfredo Goldman and Raphael Y.de Camargo. A simple bsp-based model to predict execution time in gpu applications. In *High Performance Computing (HiPC), 2015 IEEE 22nd International Conference on*, páginas 285–294. Bangalore, India.

There are other works which were published during the Doctorate of the candidate and are indirectly related to this thesis, they are presented above in chronological order, first the more recent.

Marcos Amaris, Clement Mommessin, Giorgio Lucarelli and Denis Trystram. Generic algorithms for scheduling applications on heterogeneous multi-core platforms *arXiv preprint arXiv:1711.06433* in revision. Submitted in Dec. 2017. *Concurrency and Computation: Practice and Experience*

Pedro Bruel, **Marcos Amarís** and Alfredo Goldman. Autotuning cuda compiler parameters for heterogeneous applications using the opentuner framework. *Concurrency and Computation: Practice and Experience* 29(22). 2017.

Marcos Amaris, Clement Mommessin, Giorgio Lucarelli and Denis Trystram. Generic algorithms for scheduling applications on hybrid multi-core machines. In *2017 Euro-Par: International Conference on Parallel and Distributed Computing*, pages 220–231. Santiago de Compostela, Spain.

Thiago Kenji Okada, **Marcos Amarís** e Alfredo Goldman vel Lejbman. Scheduling moldable bsp tasks on clouds. In *WSCAD'15 - XVI Simpósio em Sistemas Computacionais de Alto Desempenho*. Florianópolis-SC. Brazil.

Pedro Bruel, **Marcos Amarís** and Alfredo Goldman. Autotuning gpu compiler parameters using opentuner. In *WSCAD'15 - XVI Simpósio em Sistemas Computacionais de Alto Desempenho*, pageséginas 13–23. Florianópolis-SC. Brazil.

Rogério Gonçalves, **Marcos Amarís**, Thiago Okada, Pedro Bruel and Alfredo Goldman. OpenMP is not as easy as it appears. In *49th Hawaii International Conference on System Sciences*, January-2016, pages 5742–5751.

1.3 Thesis Outline

This document is organized into 6 chapters. In Chapter 2, we present the roadmap of Nvidia GPU architectures and describe the CUDA platform; besides the GPU testbed and the selected CUDA kernels used in this research are presented. Section 2.3 shows a brief characterization of GPU applications and different optimization which impact the performance of GPU applications. Chapter 3 shows our proposed BSP-based model used to predict execution times of GPU applications. Chapter 4 presents a machine learning approach for predicting execution times and compare the predictions with those of the analytical model using a percentage error. Finally, conclusions and future works are presented in Chapter 5. Appendix A describes two international collaborations. First shows a prediction of running times and workload of hybrid CPU-GPU applications and the second proposes a benchmark of task-based applications executed in different resources (CPUs and GPUs).

Chapter 2.

Characterization of Applications Executed on GPUs

GPUs initially had only graphical functions, but their potential as co-processors and accelerators were quickly perceived and new computing platforms evolved to support and take advance of these devices. Different industries, e.g, video games, design 3D, image rendering and computer simulation required more computational features and, as a result of this need, new GPU architectures became more flexible and powerful. Nowadays, we can find on the market, GPUs with thousands of processors for general purpose computing that are used in numerous and different fields, among them, Deep Learning, Autonomous Vehicle (cars, airplanes (Cetin and Yilmaz, 2016), helicopters), Cryptocurrency, Virtual Reality, bioinformatics, fluid dynamic, image rendering, etc.

The two major GPU manufacturers today are NVIDIA and AMD, which produce GPUs that operate in conjunction with standard CPUs. The main programming environments for GPUs are CUDA (NVIDIA, 2018) and OpenCL (Khronos, 2013). OpenCL is considerably more recent than CUDA. One of the main dilemmas of OpenCL is to maintain performance while preserving portability between distinct devices (Du *et al.*, 2012).

This chapter is organized in three sections, Section 2.1 discuss relevant advances in NVIDIA GPU architectures and their development platform CUDA, Section 2.2 shows the GPU testbed and the CUDA Kernels that were selected in order to perform our experiments. Finally, Section 2.3 described the main optimizations of applications executed on GPUs.

2.1 GPU Architectures and CUDA

The most widely GPUs used for High Performance Computing are those produced by NVIDIA. The architecture of these GPUs is based on a set of Streaming Multiprocessors (SMs), each containing Streaming Processors (SPs), a set of Special Function Units (SFUs) and a number of load/store (LD/ST) units. The multiprocessors execute threads asynchronously, in parallel. The SM schedules threads in groups of 32 parallel threads called warps, which can use LD/ST units concurrently, allowing simultaneous reads and writes to memory. Above the main parts of a GPU is explained:

- Graphic Processing Clusters (GPC): TPC is a chip which grouped the streaming multiprocessor in a GPU. Single GPC contains a raster engine. It is a way to encapsulate all key graphics processing units. Before to Fermi, SMs and Texture Units were grouped together in hardware blocks called Texture Processing Clusters (TPCs). But after Fermi, SMs have dedicated Texture Units.

- Streaming Multiprocessors (SM): A SM is designed to execute thousands of threads concurrently, up to 2048 threads on recent architectures. To manage such a large amount of threads, the instructions are pipelined through simultaneous hardware multithreading. These hardware units are SP, SFU, LD/ST, among others. The smallest executable unit of parallelism on a NVIDIA device is 32 threads or a warp. Normally different architectures have a different number of SMs and the number of hardware units inside it.

- Hardware units: Streaming multiprocessor are commonly organized by FP32 and FP64 cores, SFU and LD/ST units. FP32 and FP64 cores are shaders processors which are designed to do streaming floating point calculations. SFUs are special function unit for "fast approximate transcendental operations". They execute transcendental instructions such as *sin*, *cosine* and *square root*, with each SFU executing one instruction per clock. Load/Store units are used to read and write in the global memory. One instruction can read/write up to 128 bytes, as a consequence, if each thread in a warp reads 4 bytes and they are coalesced, then whole warp would require a single load/store instruction. If accesses are uncoalesced, then more transaction should be issued.

Until now, the GPU architectures manufactured by NVIDIA are Tesla, Fermi, Kepler, Maxwell, Pascal and recently Volta. Generally, the hierarchical memory of a NVIDIA GPU contains global and shared portions. Global memory is large, is off-chip, has a high latency and can be accessed by all threads in the GPU. Shared memory is small, is on-chip on each SM, has a low-latency and can be accessed only by threads in the same SM. Each SM has its own shared L1 cache and an off-chip coherent global L2 cache.

All NVIDIA architectures or models vary in many different features, such as the number of cores, registers, SFUs, load/store (LD/ST) units, cache memory sizes and configuration, processor clock frequency and boost frequency, memory bandwidth, unified virtual memory, and dynamic parallelism. One main challenge in the designing of new generations of massively parallel architectures is the ratio of improvement between energy consumption and performance (Mittal and Vetter, 2014). Some differences are summarized in the Compute Capability (C.C.) of NVIDIA GPUs, main characteristics of NIVIDIA GPUs for HPC are shown in Table 2.1. Many of these differences will be exposed in Subsection 2.1.1, where we mention some technical specifications of each NVIDIA GPU architecture.

Features/Tesla GPUs	**Fermi**	**Kepler**	**Maxwell**	**Pascal**	**Volta**
Compute Capability	2.0	3.5	5.3	6.0	7.0
SMs	13	15	24	56	84
SPs per SM	32	192	128	64	64
FP32 Units	512	2880	3072	3840	5376
FP64 Units	-	512	960	1792	2560
Max Warps per SM	48	64	64	64	64
Max Thread per SM	1536	2048	2048	2048	2048
Shared memory size (KB)	48	48	96	64	up to 96
Manufacturing process (nm)	40	28	28	16FinFET	12 FinFET
Hyper-Q	No	Yes	Yes	Yes	Yes
Dynamic Parallelism	No	Yes	Yes	Yes	Yes
Unified Memory	No	No	No	Yes	Yes
Preemption	No	No	No	Yes	Yes

Table 2.1: *Main characteristics of the C.C. of NVIDIA GPUs for HPC (Tesla models)*

2.1.1 NVIDIA GPU Roadmap

Tesla Architecture: This architecture was one of the first GPU to support C or CUDA, allowing programmers to use these devices without having to learn a new programming language. Currently, Tesla is also the name of the GPUs designed for High Performance Computing.

The model GeForce 8800 was one of the first model designed with this architecture, see Figure 2.1. This GPU has 112 SP, these SPs were grouped in seven independent chips called Texture/Processor Clusters (TPCs) (Lindholm *et al.*, 2008). Each TPC has 2 SM and each SM consists of eight streaming processors, 16KB of on-chip shared memory, two SFUs, a constant cache, a multithreaded instruction fetch and issue unit and a read-only constant cache. Second versions of Tesla architectures implemented fused multiply-add (FMA) for double precision.

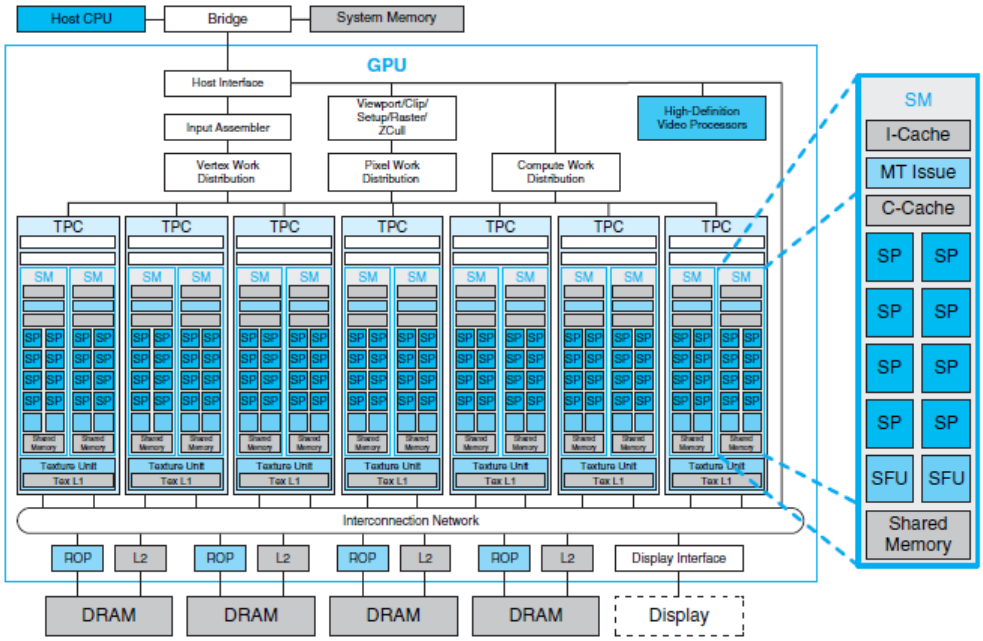

Figure 2.1: *GPU GeForce GTX 8800, the first GPU to support CUDA platform, extracted from NVIDIA Documentation*

Fermi Architecture: Fermi architectures implemented the IEEE 754-2008 floating-point standard (IEEE, 2008) for both single and double precision arithmetic which performs multiplication and addition with a single rounding step. In this architecture is possible to have a second-level cache L2 shared by all SMs and each SM has a memory cache L1.

A NVIDIA's Fermi GF-100 has 4 GPC, each GPC has 4 SMs for a total of 16 Streaming Multiprocessors. Each SM is composed of 32 cores, resulting in a total of 512 cores. It also has 4 SFUs, 2 warp schedulers, and 16 LD/ST units. Now it has configurable cache L1 or shared memory of 64KB, these configurations can be of 48 KB of shared memory and 16 KB of L1 cache or 16 KB of shared memory and 48KB of L1 cache.

Among the most important characteristics of programmability of this architecture were to allow concurrent kernel execution, out of order thread block executions and dual overlapped memory transfer engines.

Kepler Architecture: In this architecture, the abbreviation for SM changed to SMX. Each SMX has 4 warp schedulers and eight instruction dispatch units, allowing four warps to be issued

and executed concurrently. Unlike Fermi, Kepler architectures allow double precision instructions to be paired with other instructions. The shared memory continues sharing the same chip than L1 cache and the same size, 64KB, but now this architecture allowed the configuration of the shared memory and cache memory L1 in different 3 sizes. The 3 different configurations in KB for shared memory and L1 cache are 48/16 32/32 or 16/48. Kepler also introduced a 48KB of cache for read-only data for SM.

The GeForce GTX 680 GPU consists of four GPCs, each one of 4 SM for a total of 16 Streaming Multiprocessors. Each SM is composed of 192 cores, resulting in a total of 1536 cores. It has also 32 SFU, 4 warp scheduler and 32 LD/ST units.

Among the most important characteristics of programmability of this architecture were the addition of dynamic parallelism and Hyper-Q. Dynamic Parallelism allowed the GPU to create new work for itself, synchronize on results, and control the scheduling without involving the CPU. Hyper-Q enables different host threads to trigger execution of kernels simultaneously, in the Kepler architecture 32 concurrent connections are possible with the CPU.

Maxwell Architecture: The L1 cache is shared now with the texture cache with a size of 24 Kb. Unlike in Kepler and Fermi, the Maxwell architecture has a dedicated shared memory and does not share level with the level 1 cache. The shared memory now is a dedicated chip of 96 KB. Scheduling algorithms have been rewritten to avoid stalls[1], thus reducing the amount of energy spent per instruction.

GPU Maxwell GM204 consists of four GPCs, each one of 4 SM for a total of 16 Streaming Processors. Each SM is composed of 128 cores, resulting in a total of 2048 CUDA cores. This time, the SMs are partitioned into 4 blocks, each with its instruction buffer, its scheduler. Each block also has 8 LD/ST units, 8 SFU. The main features introduced by the Maxwell architecture are related to power consumption and Unified Virtual Addressing (UVA). UVA allows direct access between the host and the devices, without requiring a buffer in the host to perform data transfer.

Pascal Architecture: Pascal architecture brings big changes in NVIDIA GPUs, it introduces the new technology of stacked DRAM memory and it brings a new technology of interconnection between GPUs named NVlink. Another important characteristic of this architecture is the addition of preemption, now GPU applications are available to do preemption during their executions.

Pascal continues with the groups of multiprocessors in GPC and a dedicated chip for shared memory per Streaming Multiprocessors. Cache L1 continues sharing the same chip than Texture cache, but the size increased to 48KB.

Volta Architecture: This architecture has came optimized for Deep Learning applications. Save energy has been an important parameter in the design of this architecture, in comparison with Pascal GPUs the new Volta SM is 50% more energy efficient than the previous Pascal architecture. This architecture introduces a new characteristic, this characteristic is named **Tensor Cores**. Shared memory back to be combined with a L1 cache memory. This chip-on memory increased its size and it is now of 124 MB and shared memory is configurable up to 96 KB.

A Volta GV-100 has 6 GPC, each GPC inside has 14 SMs and each SM has 64 SP for a total of 5376 cores. The GV100 SM is partitioned into four processing blocks, each with 16 FP32 units, 8 FP64 units, two tensor cores for deep learning matrix arithmetic, one warp scheduler and one

[1] stalls is a delay before the processor can continue to execute a statement

dispatch unit.

One of the new characteristics in the programmability is the concept of Cooperative Groups, this is a programming model for organizing groups of communicating threads, it allows developers to express the granularity at which threads are communicating. This is done using a namespace which defines the thread level which performs the synchronization.

In the left side of Figure 2.2 is shown the memory hierarchy of a thread that runs on a GPUs with Fermi architecture. In this architecture, the L1 cache is in the same chip than shared memory. In the middle is shown the memory hierarchy of a thread that runs on a GPU with Kepler or Volta architecture, in this architecture a cache read-only is added and the L1 cache is in the same chip than Shared memory. In the right side is shown a memory hierarchy of a thread which runs on a GPU with Maxwell or Pascal architecture, in these architectures the shared memory is dedicated and the L1 cache is in the same chip than a texture cache. Figures were based on the NVIDIA documentation.

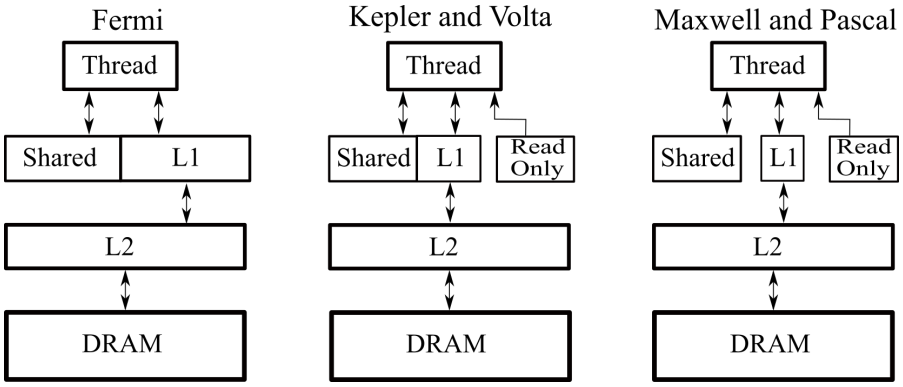

Figure 2.2: *Memory hierarchy accessed from a thread which run on a GPU with architecture (Left) Fermi. (Middle) Kepler and Volta. (Right) Maxwell and Pascal*

2.1.2 *Compute Unified Device Architecture* (CUDA)

The parallelism reached by the GPUs of NVIDIA is of the type Single Instruction Multiple Data (SIMD), but commonly NVIDIA denominates this type of parallelism *Single Instruction Multiple Threads* (SIMT), where thousands or millions of threads are executed concurrently into the GPU. In the type of parallelism used by the SIMT, an application launches a function with many threads which will execute the same instructions, and threads are programmed dynamically in a SIMD-like parallelism to access data. By itself, SIMD only describes how the instructions are executed.

The SIMT programming model has an approach that is different from the traditional *multicore* processor model. In particular, the Open Computing Language (OpenCL) is a low-level API for writing programs that can execute across heterogeneous computing systems (i.e., consisting of GPUs and CPUs) and even other kinds of processing units. CUDA and others tools provide ways to parallel computing using data-based or task-based parallelism on different architectures. CUDA works in a single-instruction multiple-data parallelism, where data is accessed by the index of each thread in a CUDA function.

The CUDA platform enables the use of NVIDIA GPUs for scientific and general purpose computation. A single *master* thread runs in the CPU, launching and managing computations on the GPU. GPUs have their own memory and data must be transferred through a PCI Express bus. Data for the computations have to be transferred from the host memory to the device memory.

Actually, the execution flow of a GPU application can be divided into three key stages. First, data is transferred to the memory of the GPU; in a second stage, the main program executed on the CPU (called host) is responsible for starting threads in the GPU (called device), launching a function (called kernel). Finally, results are sent back to the host. Figure 2.3 shows a classical execution workflow of a CUDA application.

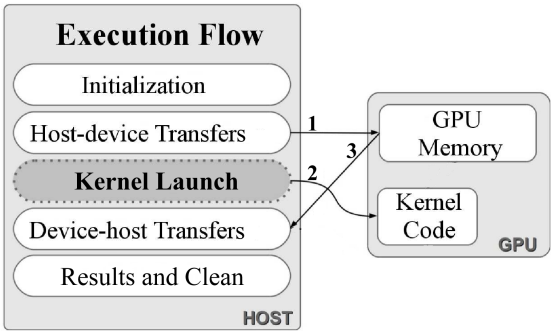

Figure 2.3: *Classical Execution of a GPU application*

The CUDA language extends C and provides a multi-step compiler, called `NVCC`, that translates CUDA code to Parallel Thread Execution code or *PTX*. `NVCC` uses the host's C++ compiler in several compilation steps, and also to generate code to be executed in the host. The final binary generated by `NVCC` contains code for the GPU and the host. When `PTX` code is loaded by an application at run-time, it is compiled to binary code by the host's device driver. This binary code can be executed in the device's processing cores and is architecture-specific. The targeted architecture can be specified using `NVCC` parameters (NVIDIA, 2018).

When a kernel is launched, t parallel threads are executed into the CUDA cores of a GPU. All threads execute a copy of the same code defined in the body declaration of the kernel function. The number of threads in a kernel is defined in two tri-dimensional parameters. The number of threads in a block and the number of blocks in a grid, the special words in CUDA to define those parameters are `blockDim` and `gridDim`, respectively. Each kernel create a grid of threads into the GPU, each grid is organized by blocks and these blocks are divided by threads, see Figure 2.4. It is normal to associate the dimension of the block to the dimension of the problem to solve.

Threads in a grid have access to different memory types. Threads in the same block can access with a high bandwidth to registers, shared memory and local memory, i.e. on-chip memory in the SM. All threads in the grid have access to load and write in the global, but only to load from the constant and texture memory, see Figure 2.4. Threads can be synchronized on two different levels, global memory, and shared memory. Thread into the same block can be synchronized across the shared memory with a high bandwidth. All threads in a kernel are synchronized across the global memory with a bandwidth lesser than shared memory, resulting in an expensive instruction for GPU applications.

The maximum number of threads in a block or blocks in a grid is specific for each Compute Capability, these values are shown in Table 2.1. something to notice In this table is the number of SPs per SM which increase in the Kepler architecture and after decreases in the next architectures; and the number maximum of warp per SM increase in Kepler and it keeps the same (64) in posterior architectures. This ensures a better occupancy of the SM.

If threads of the same warp execute different instructions, i.e. warps executing a conditional

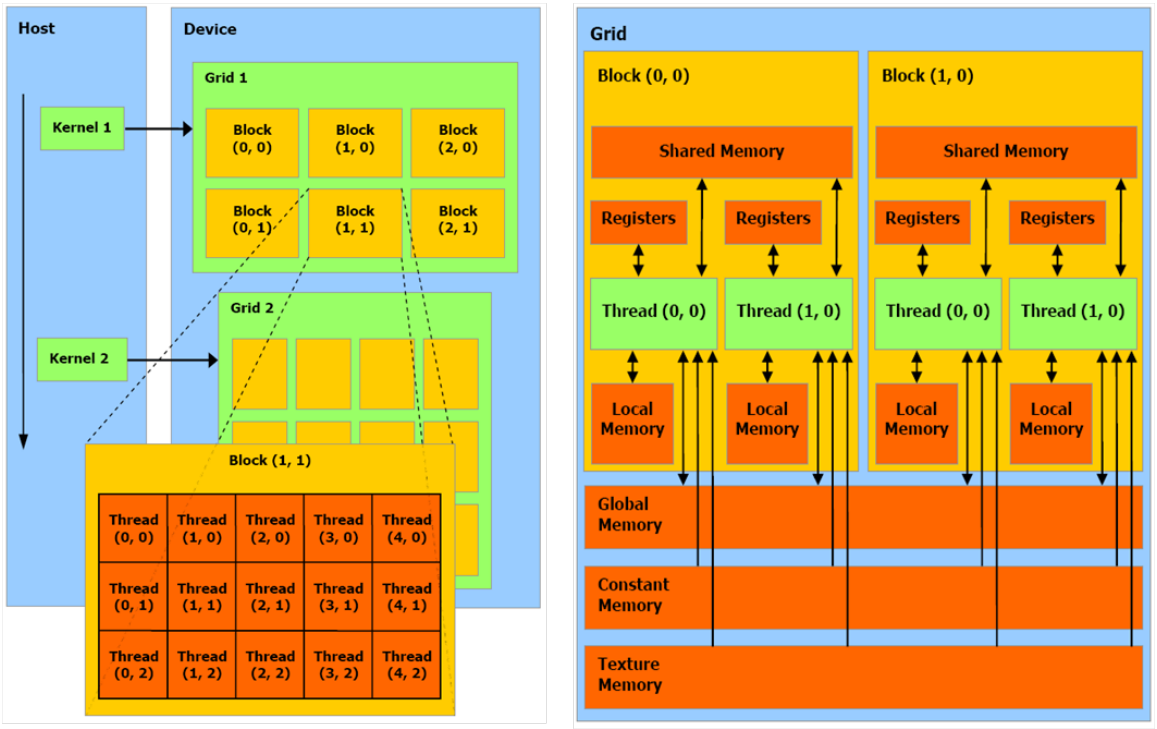

Figure 2.4: *Thread hierarchy and memory visibility of a kernel, copied from NVIDIA Documentation*

statement, the CUDA compiler must generate code to branch the execution correctly, making the program lose performance due to this *warp divergence*. Code divergence greatly degrades the performance of an application running on such architectures.

The bandwidth of the global memory can be largely improved by combining the LD/ST requests from different threads of a single warp, in a process called coalescing (Ha *et al.*, 2008). The coalescing occurs when the threads access contiguous global memory addresses, which permits usage of the multiple LD/ST units available per SM.

Multiple computations launched by the master thread, or *kernels*, can run asynchronously and concurrently in a current GPU. In CUDA, all kernel executions are asynchronous, threads of the same block can be synchronized with the function `__syncthreads()` and the function `cudaThreadSynchonize()` can synchronize all the threads of a kernel or grid.

Finally, all threads of a grid are mapped, and receive identifiers corresponding to the declared dimensions. These identifiers are available and accessible in CUDA code through variables such as `threadIdx` and `blockIdx`. This allows the execution, identification and control of threads in a GPU. The identifiers `threadIdx.x`, `threadIdx.y` and `threadIdx.z` are associated with the threads within a block and `blockIdx.x`, `blockIdx.y` and `blockIdx.z` are associated with the index of a block within a grid.

In next section, we describe each one of the kernels used as benchmark in this research and discuss main characteristics of communication and computation of each one.

2.2 GPU Testbed and Selected CUDA kernels

Several different benchmarks have been proposed in the literature to measure and assess existing GPGPU heterogeneous architectures, such as Rodinia (Che *et al.*, 2009), Parboil (Stratton *et al.*, 2012) and SHOC (Danalis *et al.*, 2010). Rodinia benchmark was devised for heterogeneous parallel computing research, it has had a high level of acceptance in the community and its applications

represent different high-level domains or behaviours, called the Berkeley dwarfs (Asanovic *et al.* , 2009). In this work, we have selected a set of GPU applications from the Rodinia Benchmark suite and other classical algorithms of linear algebra. First to all, we present in next subsection the different GPUs that we used for our experiments. After in Subsection 2.2.2, we characterize each one of the kernel that we used for our experiments.

2.2.1 GPU Testbed

For our experiments in the Chapter 3 and Chapter 4 we used 9 different GPUs, described in Table 2.2, with 5 belonging to Kepler architecture (Compute Capability 3.X), 3 to Maxwell (C.C. 5.X) and 1 to Pascal (C.C. 6.x). More information about these GPUs are presented in Table 2.1 and we have described the main changes of hardware and software between architectures in Section 2.1.1.

Model	C.C.	Memory	Bus	Bandwidth	L2	Cores/SM	Clock
GTX-680	3.0	2 GB	256-bit	192.2 GB/s	0.5 M	1536/8	1058 Mhz
Tesla-K40	3.5	12 GB	384-bit	276.5 GB/s	1.5 MB	2880/15	745 Mhz
Tesla-K20	3.5	4 GB	320-bit	200 GB/s	1 MB	2496/13	706 MHz
Titan	3.5	6 GB	384-bit	288.4 GB/s	1.5 MB	2688/14	876 Mhz
Quadro K5200	3.5	8 GB	256-bit	192.2 Gb/s	1 MB	2304/12	771 Mhz
Titan X	5.2	12 GB	384-bit	336.5 GB/s	3 MB	3072/24	1076 Mhz
GTX-970	5.2	4 GB	256-bit	224.3 GB/s	1.75 MB	1664/13	1279 Mhz
GTX-980	5.2	4 GB	256-bit	224.3 GB/s	2 MB	2048/16	1216 Mhz
Pascal-P100	6.0 GB	16 GB	4096-bit	732 GB/s	4 MB	3584/56	1328 Mhz

Table 2.2: *Hardware specifications of the GPUs in the testbed*

2.2.2 Selected CUDA kernels

The source code for all the use cases, experiments and results are available[2] under Creative Commons Public License for the sake of reproducibility.

Our benchmark contains 4 different strategies for *matrix multiplication* (NVIDIA, 2018), 2 algorithms for *matrix addition*, 1 dot product algorithm, 1 vector addition algorithm and 1 *maximum sub-array problem* algorithm (Silva *et al.*, 2014), and 11 CUDA kernel functions belonging to 6 applications from Rodinia benchmarking suite (see Table 2.4). The remainder of this section discusses some details of these algorithms, and introduces a code of letters for each application, used in whole the thesis.

Matrix Multiplication

Matrix multiplication is the core of many scientific areas. This operation is highly used in areas like deep learning, visual computing and digital images processing, among others. The analysis and modeling of these algorithms brings a better understanding and help to researcher and developer of Job Management Systems to deal with this mathematical operations.

In this research, we used four different versions of matrix multiplication, these versions differ in memory access optimizations: global memory with non-coalesced accesses (MMGU); global memory with coalesced accesses (MMGC); shared memory with non-coalesced accesses (MMSU); and shared memory with coalesced accesses (MMSC). The matrix multiplication algorithm has a high utilization of the Streaming Processors and it obtains a high throughput of communication in the different

[2]Hosted at GitHub: `https://github.com/marcosamaris/gpuperfpredict` [Accessed on March 2018]

levels of memory, L2 cache and L1 cache for small matrices. We have adopted `block_size`^{2} threads per block and defined the number of blocks to be square of `(N + block_size)/block_size`, dynamically devised from the size of the problem (N) and the block size. Aiming to take advantage of coalesced accesses the value of `block_size` is equal to 16 in our experiments. In Section 2.3 is presented an communication analysis of this application with different sizes of threads per block.

The asymptotic computational complexity of a square matrix multiplication of size N is $O(N^3)$ in a sequential algorithm, in a CUDA algorithm this complexity is $O(N)$ using $N \times N$ threads. In this algorithm each thread requests N elements from both matrices and perform a dot product with these arrays. This CUDA kernel performs N reads from global memory for each matrix and N arithmetic operations. A single write instruction is performed. shared memory is not used in two versions, MMGU and MMGC. In Figure 2.5 is shown the source code of the kernel MMGU, we only changed the data access pattern, to permit coalesced accesses to data in global memory. In Figure 2.5 line 7 is changed to `Pvalue += A_d[i * N + k] * B_d[k * N + j];` and line 10 is changed to `C_d[i * Width + j] = Pvalue;`.

```
1  __global__ void matMul(float* C_d, float* A_d, float* B_d, int N) {
2    float Pvalue = 0.0;
3    int j = blockIdx.x * blockDim.x + threadIdx.x;
4    int i = blockIdx.y * blockDim.y + threadIdx.y;
5
6    for (int k = 0; k < N; ++k)
7      Pvalue += A_d[j * N + k] * B_d[k * N + i];
8
9    C_d[j * N + i] = Pvalue;
10 }
```

Figure 2.5: *Matrix multiplication CUDA kernel using only global memory and uncoalesced accesses (MMGU).*

The version MMSU and MMSC use shared memory to load data from global memory and to process them with a lower latency of communication. As shared memory is limited in GPU architectures, the implementations of matrix multiplication with shared memory must be tiled. The concept of tiling in shared memory is graphically described with Figure 2.6, with the matrix multiplication. Tiling is a common strategy to partition data into subset called tiles such that each tile fits into the shared memory. This technique splits our problem domain into phases. The tiled process is executed to guarantee that each thread can access data in shared memory to perform its part of the matrix multiplication.

In process shown in Figure 2.6, a tile process charges a subset from matrix A and Matrix B in the shared memory of the GPU. A barrier synchronization is used to guarantee that all data was loaded in the shared memory, after the calculations are performed, the barrier synchronization guarantees that the shared memory can be safely overwritten. In this application `tile_size=block_size`, consequently the sizes in bytes of both subsets of the matrices will be `(tile_size)`$^2 \times$`FP_Bytes`, where `FP_Bytes` is the size of the single precision used for the application.

Matrix Addition

For the Matrix addition algorithm, we used two different memory access optimizations: global memory with non-coalesced accesses (MAU); and global memory with coalesced accesses (MAC); The run-time complexity for a sequential matrix addition algorithm using two matrices of size $N \times N$ is $O(N^2)$. In a CUDA implementation, the run-time complexity of the matrix addition

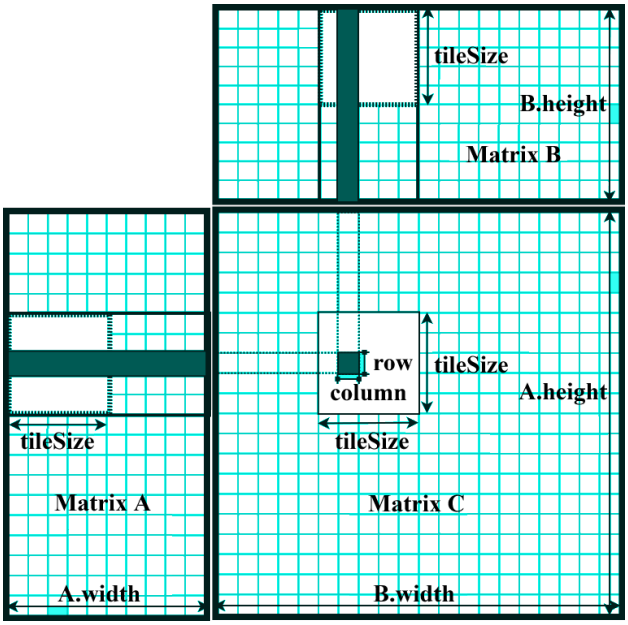

Figure 2.6: *Tiling technique of the Matrix multiplication using shared memory*

is $O(1)$ using N^2 threads. In this algorithm each thread request 1 element from each one of the matrix elements and perform a single addition. The implementation of this kernel is similar than matrix multiplication. In Figure 2.5, the loop of the line 6-7 is deleted and the statement with the addition is added `C[tid] = A[tid] + B[tid];`, where `tid` = `int tid = i*N + j;`. To get coalesced accesses in the global memory, `tid` is changed to `int tid = j*N + i;`.

Matrix addition has the same threads hierarchy than matrix multiplication. The blocks are bi-dimensional with `block_size`^{2} threads per block. The size of the grid is square of `N/block_size`, dynamically devised from the size of the problem `N` and `block_size`.

Vector Addition Algorithm (vAdd)

For two vectors A and B, the Vector Addition $C = A + B$ is obtained by adding the corresponding components. In a CUDA implementation, the run-time complexity of the vector addition algorithm is $O(1)$ using N threads, so each threads perform an addition of a position of the vectors A and B and stores the result in the vector C. This algorithm is a simplified version of the matrix addition. This application has an uni-dimensional block, the block size used for the experiments is 256 and the size of the grid is also uni-dimensional and dynamically devised from the size of the problem `N` and `block_size`. The source code of this kernel is similar than matrix addition, only change the dimension indexes of each threads, i.e. `int tid = blockDim.x * blockIdx.x + threadIdx.x;`, using only the dimension x of `blockDim`.

Dot Product Algorithm (dotP)

For two vectors A and B, the dot product $C = A \cdot B$ is obtained by adding the multiplication of corresponding components of the input, the result of this operation is a scalar. Unlike vector addition, dot product is a reduction from vectors to a scalar. In a GPU algorithm, each thread performs a multiplication of a position of the vectors A and B and stores the result in a shared variable. Then a reduction using the shared memory is performed and finally a vector with size equal to `N/block_size` is transferred to the CPU memory for later processing. The block size used for the experiments is 256 and the size of the grid is also uni-dimensional and dynamically devised

from the size of the problem `N` and `block_size`.

Maximum Sub-Array Problem (MSA)

Let X be a sequence of N integer numbers $(x_1, ..., x_N)$. The Maximum Sub-Array Problem (MSA) consists of finding the contiguous sub-array within X which has the largest sum of elements. The solution for this problem is frequently used in computational biology for gene identification, analysis of sequence of protein and DNAs, identification of hydrophobic regions, among others. The implementation used in this paper creates a kernel with 4096 threads, divided in 32 blocks with 128 threads (Silva *et al.*, 2014). The N elements are divided in intervals of N/t, and each block receives a portion of the array. The blocks use the shared memory for storing segments, which are read from the global memory using coalesced accesses. Each interval is reduced to a set of 5 integer variables, which are stored in vector of size $5 \times t$ in global memory. This vector is then transferred to the CPU memory for later processing.

A summary of each one of these applications is shown in Table 2.3, in this table is shown the thread hierarchy and the request shared memory per block in each kernel. Columns `dimGrid` and `dimBlock` show the thread hierarchy solution of each kernel. BS is the block size and GS is the grid size. Column **Shared Mem** shows the size of shared memory that each kernel needs per block during its execution.

Application	Param	Kernel	dimGrid	dimBlock	Shared Mem
Matrix Mul	1	MMGU MMGC MMSU MMSC	(GS, GS, 1)	(BS, BS, 1)	0 0 $(BS^2 \times 2 \times 4B)$
Matrix Add	1	MAU MAC	(GS, GS, 1)	(BS, BS, 1)	0
Vector Add	1	VAdd	(GS, 1, 1)	(BS, 1, 1)	0
Dot Product	1	dotP	(GS, 1, 1)	(BS, 1, 1)	$(BS \times 4B)$
Max. Sub Array	1	MSA	(48, 1, 1)	(128, 1, 1)	$(4096 \times 4B)$

Table 2.3: *Key linear algebra applications used in the experiments*

Rodinia Benchmark Suite

We selected 11 CUDA kernel functions belonging to 6 applications from Rodinia for the benchmarks (Table 2.4). Some applications invoke the same kernel multiple times on each execution, resulting in the number of collected samples shown in the last column, i.e. some kernels iterate multiple times in a single execution. These applications are Gaussian, Heartwall, Hotspot, LU Factorization and Needleman-Wunsch. Only both kernels in Back Propagation execute one time in each execution of the whole application. For each application we iterated over the values of one or two parameters, with the number of iterations indicated inside the brackets. For instance, the Hot Spot application has 2 parameters, with the first iterated among 5 values and the second 4 values, for a total of 20 executions in each machine.

Back Propagation (BCK): BCK trains a layered neural network. The application is comprised of two kernels: Forward Phase (BCK-K1), in which the activation are propagated from the input

Application	Berkeley Dwarf	Domain	Param.	Kernels	Samples
Back Propagation (BCK)	Unstructured Grid	Pattern Recognition	1 - [57]	layerforward adjust-weights	57
Gaussian Elimination (GAU)	Dense Linear Algebra	Linear Algebra	1 - [32]	Fan1 Fan2	34800
Heart Wall (HWL)	Structured Grid	Medical Imaging	1 - [84]	heartWall	5270
Hot Spot (HOT)	Structured Grid	Physics Simulation	2 - [5,4]	calculate-temp	396288
LU Decomposition (LUD)	Dense Linear Algebra	Linear Algebra	1 - [32]	diagonal perimeter internal	8448 8416 8416
Needleman-Wunsch (NDL)	Dynamic Programming	Bioinformatics	2 - [16,10]	needle-1 needle-2	21760 21600

Table 2.4: *Rodinia applications used in the experiments*

to the output layer, and Backward Phase (BCK-K2), in which the error between the observed and requested values in the output layer is propagated backwards to adjust the weights and bias values. The time complexity of a back-propagation neural network algorithm on a single processor is of $O(W^3)$; where W is the count of weights in the network. In this CUDA implementation first kernel has a complexity $O(log(BS))$ where $BS = 16$ and the second kernel has a complexity of $O(1)$. Each block has `BLOCK_SIZE`^{2} number of threads. The number of blocks in the grid is `(N/block_size)` and it is dynamically calculated from the layer size (N) and the block size.

In kernel (BCK-K1), only one thread (with id 0) in the each block loads an element of the input layer on the shared memory, after each thread load an element of the weight matrix on the shared memory. Then, the weight matrix is updated with the values of the input layer. Finally a loop reduction is done with a loop of size log2(block_size). Inside this loop, log2(block_size) power instructions are done and log2(block_size) additions over data in the shared memory. Each interval is reduced to a set of `grid_size`×`block_size` integer variables. This vector is then transferred to the main memory of the host for later processing. In kernel (BCK-K2), shared memory is not used. According to the source code of (BCK-K2), each thread performs O(1) reads and write in the global memory and does different computations over this data.

Gaussian Elimination (GAU): GAU solves a linear system $Ax = b$, the application analyzes an $n \times n$ matrix and an associated $1 \times n$ vector to solve a set of equations with n variables and n unknowns. Gaussian Elimination algorithms has a complexity $O(n^3)$. This application compute the results row by row. The difficulty in this parallel Gaussian elimination is that calculating a new value in the upper triangle regions requires that all previous values of the upper triangle matrix be known. The algorithm synchronizes between iterations, but the values calculated in each iteration is computed in parallel, see Figure 2.7. The application has two different kernels Fan1 and Fan2, which we call (GAU-K1) and (GAU-K2) respectively. (GAU-1) calculate multiplier matrix and (GAU-2) modify the matrix A into LUD. In the experiments, we varied the size of the matrix. In this implementation, First kernel GAU-K1, has a uni-dimensionmal block, each block `block_size = 512` threads and the number of block is dynamic and computed with the next expression `(N/block_size)`. Second kernel has a bi-dimensional block, its dimension is `block_size`^{2} and `block_size = 4` in this implementation. The size of the grid also is dynamic depending to size of the problem N and the BS (Block size), and it is computed as `(N/block_size)`. Both kernel

are iterative, it means that the same kernels are invoked multiple times on a single execution of the whole application, see Figure 2.7. Any of these kernel use the shared memory of the streaming processors.

```
1 for (t=0; t<(Size-1); t++) {
2  Fan1<<<dimGrid1,dimBlock1>>>(m_cuda,a_cuda,Size,t);
3  cudaThreadSynchronize();
4  Fan2<<<dimGrid2,dimBlock2>>>(m_cuda,a_cuda,b_cuda,Size,t);
5  cudaThreadSynchronize();
6 }
```

Figure 2.7: *Main loop of the application Gaussian, which calls both CUDA Kernels (Fan1 and Fan2)*

Gaussian Elimination as well as LU Decomposition present kernels which the number of threads decrease during whole the execution of the applications. In the loop below, we can see that both kernels in Gaussian Elimination iterate according to the variable `Size`, which is the size of the problem. The variable `t` is among the input variables of each kernels. This variable `t` is considered in each kernel to decrease the number of threads which realize the main operations. This is done during the execution of both kernels (Fan1 and Fan2). This makes that predictions of the running times were very difficult to obtain.

Heart Wall (HWL): HWL tracks the movement of a mouse heart on a sequence of 104 ultrasound images with 609x590 of resolution to record response to the stimulus (Szafaryn *et al.*, 2009). In this application, we varied the number of frames to process for execution. This kernel has two different stages, in the first stage the kernel performs operations on the first frame to detect initial, partial shapes of inner and outer heart walls. In the second stage the kernel presents multiple nested loops that process batches of 10 frames and 51 points in each image. This kernel is very complicated to analyze, i.e. this kernel has 1300 codes lines more or less, it uses 8 variables in the shared memory and aroud 44 variables to store all its computations in each streaming multiprocessor. This application was developed following a CGM model, in this sense, the number of threads per block and the number of blocks per grid are uni-dimensionals. For all the experiments, this kernels has a `block_size`= 256 and a `grid_size`= 51. For a total of 13056 threads in each one of the executions. This application models a set of ordinary differential equations (ODEs) that are determined by more of 200 parameters. HWL requires the inclusion of some non-parallel computation into the kernel, leading to a slight warp under-utilization.

HotSpot (HOT): HOT is an algorithm to estimate processor temperature based on an architectural floor plan (Huang *et al.*, 2006). This kernel computes the final state of a grid of cells when given the initial conditions (temperature and power dissipation per cell). This solution iteratively updates the temperature values in all cells in parallel, and usually stops after a given number of iteration. This application includes the 2D transient thermal simulation kernel which iteratively solves a series of differential equations to determine block temperatures. The Kernel in this applications has a bi-dimensional grid and block. The dimension of the block is square of `block_size` and `block_size`= 16. The size of the grid is the square of `N/block_size` and it is dynamic depending the size of the problem and the size of the thread block. In the experiments, we varied two parameters: size of the problem and number of iterations. This application has a single kernel. It uses the shared memory. Figure 2.8 shows the main loop, which executes the main and single kernels *calculate_temp*. This kernels is launched *total_itarations/num_iterations* times.

```
1  for (t = 0; t < total_iterations; t+=num_iterations) {
2      int temp = src;
3      src = dst;
4      dst = temp;
5      calculate_temp<<<dimGrid, dimBlock>>>(MIN(num_iterations, total_iterations-t), \
6      MatrixPower,MatrixTemp[src],MatrixTemp[dst],col,row,borderCols, borderRows, \
7    Cap,Rx,Ry,Rz,step,time_elapsed);
8  }
```

Figure 2.8: *Main loop of application HotSpot, which launches a single kernel* ***calculate_temp*** *total_itarations/num_iterations times*

LU Decomposition (LUD): LUD is a factorization algoritm, where "LU" means *lower upper*, LUD is an algorithm to calculate the solutions of a set of linear equations. The LUD kernel decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix. This benchmark present tree different kernels (LUD-K1, LUD-K2 and LUD-K3). Similarly to the Gaussian Elimination, the matrix size of this experiment was also iterated. First kernel LUD-K1 (named diagonal) has a static size of threads. The total threads of this kernels always is 16. The second kernel LUD-K2 (named perimeter) has a uni-dimensional block the size of thread per block always is `block_size`$*2$, the number of block in the grid also is uni-dimensional and it is computed dynamically with the next expression `(matrix_dim-i)/block_size-1`. The third kernel LUD-K3 (named internal) has a block bi-dimensional, this size is the square of `block_size` and `block_size`$= 16$, the grid also is bi-dimensional and it is computed with the same expression than the second kernel, `(matrix_dim-i)/block_size-1`. Main loop of this function if shown in Figure 2.9. This figure shows how each kernel is executed with different thread configurations. `lud_diagonal` and `lud_perimeter` is always executed with a uni-dimensional thread configuration while the other two has a bi-dimensional configuration.

```
1  void lud_cuda(float *m, int matrix_dim)
2  {
3    int i=0;
4    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
5    float *m_debug = (float*)malloc(matrix_dim*matrix_dim*sizeof(float));
6
7    for (i=0; i < matrix_dim-BLOCK_SIZE; i += BLOCK_SIZE) {
8        lud_diagonal<<<1, BLOCK_SIZE>>>(m, matrix_dim, i);
9        lud_perimeter<<<(matrix_dim-i)/BLOCK_SIZE-1, BLOCK_SIZE*2>>>(m, matrix_dim, i);
10       dim3 dimGrid((matrix_dim-i)/BLOCK_SIZE-1, (matrix_dim-i)/BLOCK_SIZE-1);
11       lud_internal<<<dimGrid, dimBlock>>>(m, matrix_dim, i);
12   }
13   lud_diagonal<<<1,BLOCK_SIZE>>>(m, matrix_dim, i);
14 }
```

Figure 2.9: *Main loop of application LUD, which launches three different kernels with different thread configuration*

Needleman-Wunsch (NDL): is a nonlinear global optimization method for DNA sequence alignments. The potential pairs of sequences are organized in a 2D matrix. In the first step, the algorithm fills the matrix from top left to bottom right, step-by-step. In the second step, the maximum path is traced backward to deduce the optimal alignment. Needleman-Wunsch has two CUDA kernels, NDL-K1 and NDL-K2. We iterated over the matrix dimension and penalty positive integer parameters. In both kernels each block use 2180 KB of shared memory. This implementation use a static number of threads per block of 16 and the grid is dynamically computed with this

expression $(N-1)/16$. Where N is the size of the input matrix. Figure 2.10 shows two different loops. Inside each loop is executed each one of the kernels of this application. In both loops the variable `dimGrid.x` receives the value of the iterated variable i. Variable i in the first loop grow incrementally; while in the second loop, the value of i starts in a maximum size (block_width) and decreases until 1.

```
1  for( int i = 1 ; i <= block_width ; i++){
2   dimGrid.x = i;
3   dimGrid.y = 1;
4   needle_cuda_shared_1<<<dimGrid, dimBlock>>>(referrence_cuda, matrix_cuda, \
   max_cols, penalty, i, block_width);
5  }
6  for( int i = block_width - 1  ; i >= 1 ; i--){
7   dimGrid.x = i;
8   dimGrid.y = 1;
9   needle_cuda_shared_2<<<dimGrid, dimBlock>>>(referrence_cuda, matrix_cuda,\
   max_cols, penalty, i, block_width);
10 }
```

Figure 2.10: *Main loops of application Needleman-Wunsch, which launches two kernels (NDL-K1 and NDL-K2)*

2.3 Characterization of CUDA Kernels

Current GPU architectures have a deep hierarchical memory management, where cores make data requests in their registers, these requests pass through caches that in some cases may be shared caches, and so on until to arrive to its global memory. When a requested data from a thread increases of level in the memory hierarchy, it increases the communication latency. On the contrary, while higher memory level, higher capacity of storage, see Figure 2.11.

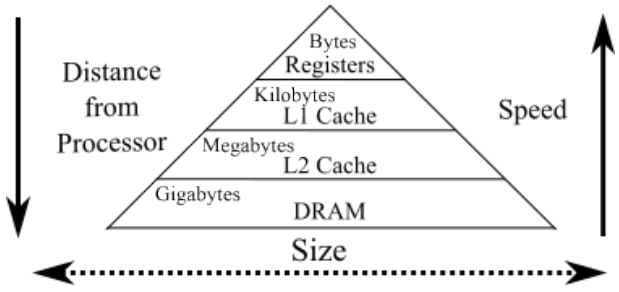

Figure 2.11: *Variance of the latency and capacity of storage in the memory systems*

The performance of a function executed over a GPU depends greatly on the optimizations made in the accesses to data in the memory hierarchy. The bandwidth of the memory is optimized by grouping a set of threads, so, the threads in the group are benefited in the communication. This effect is called coalesced accesses (Che *et al.*, 2011; Wu *et al.*, 2013). In Tesla architectures, the first GPGPU architecture, these coalesced accesses were by threads of a half warp, i.e. 16 threads can be coalesced to one transaction for word of size 8-bit, 16-bit, 32 bit, 64-bit or 128-bit. On Fermi, Kepler and newer architectures coalesced accesses can be done by all threads of a warp.

Since the early generations of GPUs for general purpose, GPUs have had different types of memories, which are differentiated by the type and visibility in the data. In Table 2.5 are presented the different types of memories existing in current GPUs. This table shows the type of operation that each memory can do. The constant memory is an off-chip and it can be read by all threads

in a kernel, however the CPU is the only which can write on it. The On Chip memories are those that are inside each multiprocessors and consequently the communication latency is lower. Global memory is the main memory of the GPU, local and constant memory are just different addressing modes of the global memory. Global Memory is DRAM, on the contrary, all on-chip memory (shared memory, registers, and caches) are SRAM.

Type	***On Chip***	**Cacheable**	**Operations**	**Visibility**
Registers	Yes	No	Read/Write	*Thread*
Local	Not	Yes	Read/Write	*Thread*
Shared	Yes	No	Read/Write	Block
Global	No	Yes	Read/Write	*Kernels*
Constant	No	Yes	Read	*Kernels*
Texture	No	Yes	Read/Write	*Kernels*

Table 2.5: *Memory types of the GPUs manufactured by Nvidia*

We wanted to show in this thesis different optimizations which impact the performance of a GPU application. Figure 2.12 shows the running times of the kernel of matrix multiplication using only global memory without coalesced accesses, i.e. the version $MMGU$. The experiments are done varying the number of threads per block, with dimensions of 8×8, 16×16, or 32×32. Figure 2.12A shows the mean of 10 running times of each dimension, Figure 2.12B presents the number of load transactions per request in the global memory and Figure 2.12C the number of store transactions per request in the global memory. This kernel has a bad access in the global memory. The coalesced accesses are not used and the communication complexity of this algorithm is $2N$, with N the size of the problem.

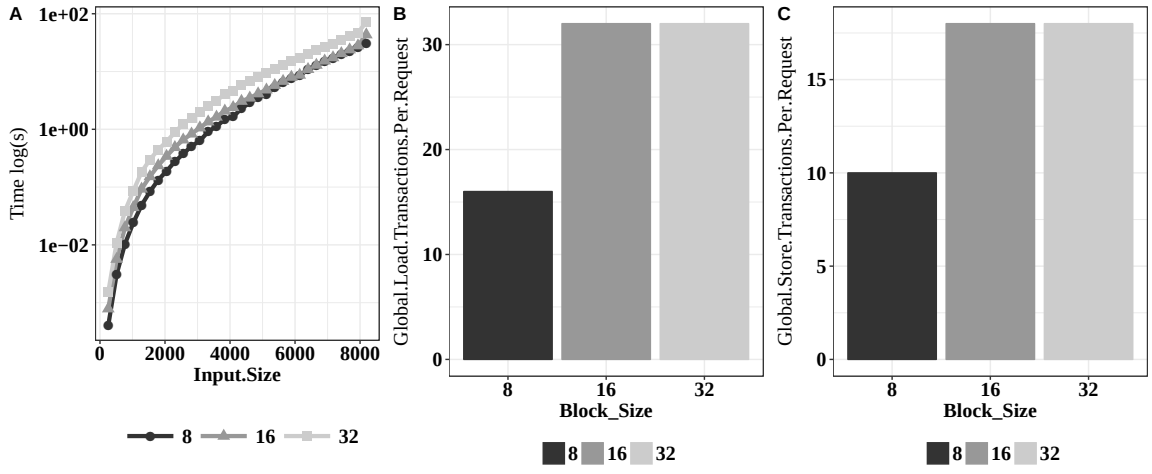

Figure 2.12: *Tuning of threads per Block in MMGU on the GPU GTX-970*

Figure 2.13 shows the performance of two applications in two different versions each one, the applications are matrix multiplication and matrix addition, the versions of these applications are $MMGC$ and $MMGU$; and, MAC and MAU; respectively. The selected dimensions of threads per block in each kernel was 16×16. Each kernel change communication pattern in the global memory. Figure 2.13A shows the running time of the kernels $MMGC$ and $MMGU$ and Figure 2.13B the number of load transactions per request in the global memory. Figure 2.13C shows the running time of the kernels MAC and MAU and Figure 2.13D the number of load transactions per request in

the global memory of two different version of the matrix addition. When the number of transaction per request in the global memory is smaller the running times of the application improve.

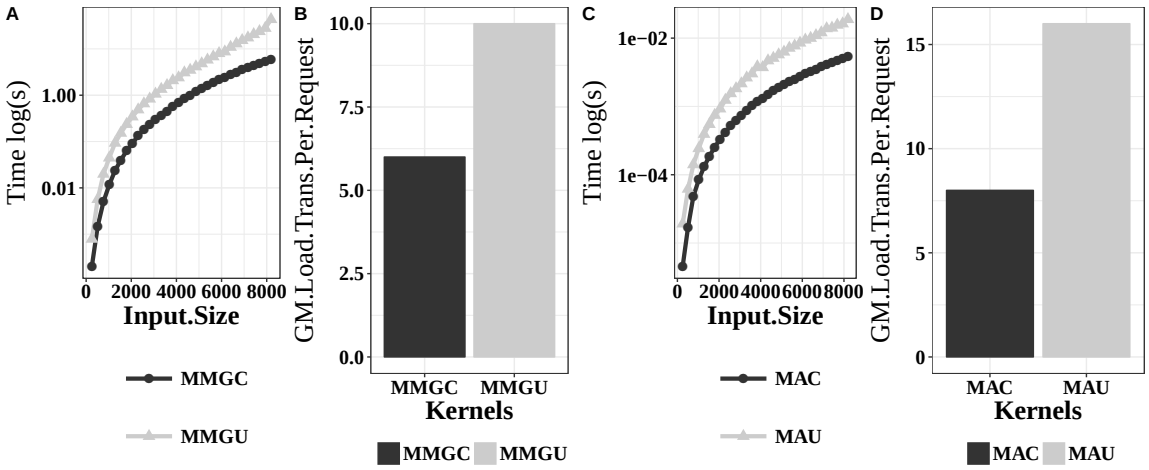

Figure 2.13: *Coalesced accesses impact in 2 different kernels of Matrix Multiplication and Matrix Addition on the GPU GTX-970*

Figure 2.14 shows the impact of the shared memory and coalesced accesses in the 4 versions of matrix multiplication. Figure 2.14A shows the running time of the kernels $MMGU$, $MMGC$, $MMSU$ and $MMSC$; Figure 2.14B shows the throughput in the global memory and Figure 2.14C shows the number of load transactions per request in the global memory of the 4 kernels. The worst throughput in the load memory is done for the kernel $MMSU$, it means that using shared memory does not improve the throughput in the load memory, coalesced accesses are necessary for this goal.

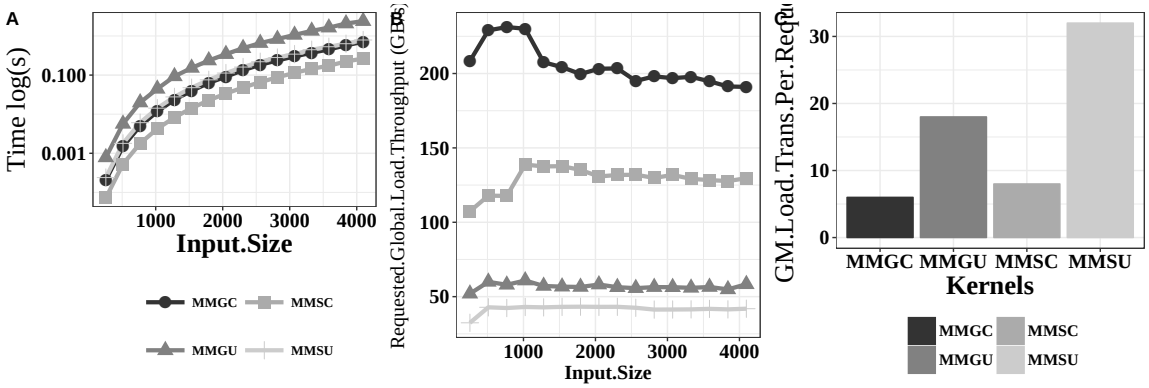

Figure 2.14: *Shared memory optimizations in kernels of Matrix Multiplication on the GPU GTX-970*

In one of the work indirectly relate with this thesis, we implemented an autotuner for the CUDA compiler (Bruel *et al.*, 2017). This was made using the OpenTuner framework (Ansel *et al.*, 2014). Compilation parameters and their values were used to build the search space. This resulted in performance improvement of different applications over different GPUs. Follow the list of compiler flags used as search space in the autotuning process.

The Search Space Table 2.6 details the subset of the CUDA configuration parameters used in the experiments[3]. The parameters target different compilation steps: the *PTX* optimizing assem-

[3]Adapted from: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc [Accessed on 20 February 2018]

Flag	Description
`no-align-double`	Specifies that `malign-double` should not be passed as a compiler argument on 32-bit platforms. **Step**: NVCC
`use_fast_math`	Uses the fast math library, implies `ftz=true`, `prec-div=false`, `prec-sqrt=false` and `fmad=true`. **Step**: NVCC
`gpu-architecture`	Specifies the NVIDIA virtual GPU architecture for which the CUDA input files must be compiled. **Step**: NVCC **Values**: `sm_20`, `sm_21`, `sm_30`, `sm_32`, `sm_35`, `sm_50`, `sm_52`
`relocatable-device-code`	Enables the generation of relocatable device code. If disabled, executable device code is generated. Relocatable device code must be linked before it can be executed. **Step**: NVCC
`ftz`	Controls single-precision denormals support. `ftz=true` flushes denormal values to zero and `ftz=false` preserves denormal values. **Step**: NVCC
`prec-div`	Controls single-precision floating-point division and reciprocals. `prec-div=true` enables the IEEE round-to-nearest mode and `prec-div=false` enables the fast approximation mode. **Step**: NVCC
`prec-sqrt`	Controls single-precision floating-point squre root. `prec-sqrt=true` enables the IEEE round-to-nearest mode and `prec-sqrt=false` enables the fast approximation mode. **Step**: NVCC
`def-load-cache`	Default cache modifier on global/generic load. **Step**: PTX **Values**: `ca`, `cg`, `cv`, `cs`
`opt-level`	Specifies high-level optimizations. **Step**: PTX **Values**: `0 - 3`
`fmad`	Enables the contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add operations (FMAD, FFMA, or DFMA). **Step**: PTX
`allow-expensive-optimizations`	Enables the compiler to perform expensive optimizations using maximum available resources (memory and compile-time). If unspecified, default behavior is to enable this feature for optimization level $\geqslant$O2. **Step**: PTX
`maxrregcount`	Specifies the maximum number of registers that GPU functions can use. **Step**: PTX **Values**: `16 - 64`
`preserve-relocs`	Makes the `PTX` assembler generate relocatable references for variables and preserve relocations generated for them in the linked executable. **Step**: NVLINK

Table 2.6: *Description of flags in the search space*

bler; the *NVLINK* linker; and the *NVCC* compiler. We compared the performance of programs generated by tuned parameters with the standard compiler optimizations, namely `-opt-level=0,1,2,3`. Different `-opt-level`s could also be selected during tuning. We did not use compiler options that target the host linker or the library manager since they do not affect performance. The size of the search space defined by all possible combinations of the flags in Table 2.6 is in the order of 10^6 making hand-optimization or exhaustive searches very time consuming.

Figure 2.15 shows the results of the autotuner for the CUDA compiler. It shows the results of the 4 kernel versions of the matrix multiplication used in this work. We can see that optimizations can be done by compiler parameters, and these compiler parameters can impact the speedup of GPU application in up $4x$, this work has been done by Bruel *et al.* (2017).

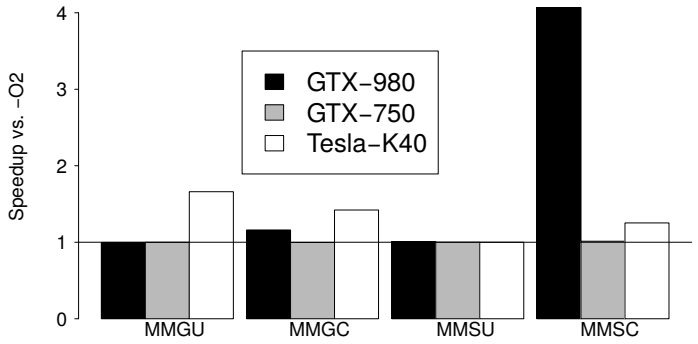

Figure 2.15: *Summary of the speedups achieved versus* -O2 *in matrix multiplication versions*

Chapter 3.

Simple BSP-based Model to Predict CUDA Kernels

Computational models are useful to represent abstractions of software and hardware processes and/or interaction among them. The Bulk Synchronous Parallel (BSP) is a bridging model for parallel computation that allows algorithmic analysis of programs on parallel computers using performance modeling. The main idea of the BSP model is the treatment of communication and computation as abstractions of a parallel system.

In this chapter, we present a simple and intuitive BSP-based model for predicting execution times of CUDA applications. The model is based on the number of computations and memory accesses of an application, with additional information from profiling. Scalability, divergence, the effect of optimizations and differences of architectures are adjusted by a single parameter.

The structure of this chapter is organized as follow, Section 3.1 presents the most important parallel models of the literature. Section 3.2 shows the proposed parallel BSP-based model to predict execution time of GPU applications. The methodology and the use cases are described in Section 3.3. Section 3.4 presents some experimental results and finally Section 3.5 exposes the main related works of this model.

3.1 Review of Parallel Computational Models

Mathematical models are simplified abstraction of a real situation. A important area of the computer science is related with the analysis, design and development of algoritmh that are implemented in real machines. The basic computer architecture is known as von Neummann architecture or von Neumman model. This model was created in 1945 and it has the following components: a memory; an arithmetic-logic unit (ALU); a central processing unit (CPU), composed of several registers; and a control unit. New technologies and computational models began to be developed simultaneously with the evolution of the von Neumann model. Another example is the RAM model (Random-Access Memory). This model is an abstraction for data storage. Nowadays, it has taken a form of integrated circuits that allows the stored data to be accessed randomly.

In 1972, Michael Flynn proposed a classification of parallel computing architectures. This classification distinguishes the number of instructions and the number of data that can be computed in parallel (Flynn and Rudd, 1996). This classification is presented in Table 3.1.

Table above is illustrated as follow, a machine with the von Newman model and only one processing core fits in the SISD (Single Instruction; Single Data) classification; a machine with multiple processing cores can be classified as MIMD; GPUs are classified on the SIMD classification,

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 3.1: *Classification of parallel architectures proposed by Michael Flynn (1972).*

where each thread takes index to perform vector computations.

Parallel computing models have been an active research topic since the development of modern computers. They provide a standard way of describing and evaluating the performance of parallel applications. For the success of a parallel computing model, it is paramount to also consider the characteristics of the underlying architecture of the hardware being used.

The main objective of a parallel computing model is to provide a set of parameters to be considered in the implementation of a parallel algorithm. These parameters can be used to simulate the behavior of these algorithms over different parallel platforms. To facilitate the programming and simulation of these applications, models with specific properties to parallel programming problems have been created (Skillicorn and Talia, 1998).

In computing, granularity is associated with the amount of computation in relation to communication, that is, the ratio of computation to the amount of communication. Parallelism of fine granularity means relatively small amounts of computational work are done between communication events Low computation to communication ratio. Coarse and Bulk granularity is the opposite: data transfers are less frequent, and present large amounts of computation. Parallelism of coarse or Bulk granularity means relatively large amounts of computational work are done between communication events, high computation to communication ratio. The finer granularity have greater potential for parallelism and consequently the increase in speed, but the overhead costs of synchronization and communication are expensive in terms of latency.

The most important parallel models in the litearature are the PRAM (Parallel Random Access Memory), LogP, BSP (Bulk Synchonous Parallel) and CGM (Coarse Grained Multicomputer). They are explained below.

3.1.1 Parallel Random Access Machine Model (PRAM)

The PRAM model was created by Fortune and Wyllie (1978). This model is a simple extension of the RAM model. It consists of an infinite set of processors and a centralized memory, which is shared by all processors. Figure 3.1 shows graphically the PRAM model

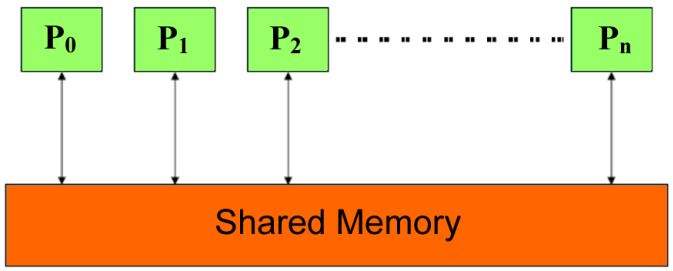

Figure 3.1: *PRAM model (Parallel Random Access Machine.)*

The advantage of the PRAM model is its simplicity and its similarity to the sequential model of von Neumann. The processor can only read or write a memory address in one cycle. The cost of

writing is equal to the cost of reading, and is also equal to the cost of any operation performed by the processor. However, in spite of its simplicity, this model has become more and more unrealistic, because of the increasing of the latency in different operations of computation and communication.

Different submodels were created from the PRAM model. Researchers have made adaptations varying the way of access to memory, trying to avoid the maximum of conflicts in the communication. The different adaptations have arisen to propose concurrent or exclusive communications in the memory access (Gibbons *et al.*, 1998).

3.1.2 Bulk Synchronous Parallel Model (BSP)

The BSP model was introduced by Valiant (1990). This model offers a simple abstraction of parallel architectures, see Figure 3.2. In this figure, a set of processors are running local computations in a superstep and before the synchronization, all the messages are delivered and ready to be used in the next superstep.

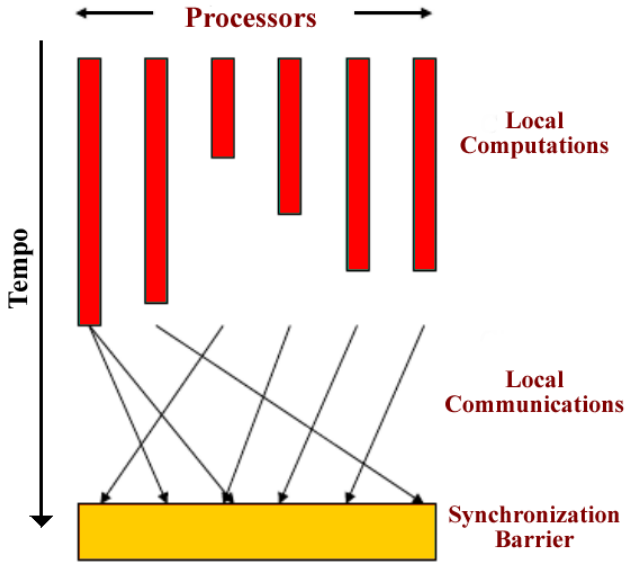

Figure 3.2: *Superstep in a Bulk Synchronous Parallel Model.*

The BSP model bridges the essential characteristics of different kinds of machines as a combination of three attributes:

- a set of virtual processors, each associated to a local memory;
- a router, that delivers the messages in a point-to-point manner;
- a synchronization mechanism.

The execution of a parallel application is organized in a sequence of *supersteps*, each one divided into three successive—logically disjointed—phases. On the first phase, all processors use their local data to perform local sequential computations in parallel (i.e., there is no communication among the processors). The second phase is a communication phase, where all nodes exchange data performing personalized all-to-all communication. The last phase consists of a global synchronization barrier, that guarantees that all messages were delivered and all processors are ready to start the next superstep.

Figure 3.2 depicts the phases of a BSP application. In this figure, a processor distributes tasks to a set of processors that execute local computations and communicate in a global form, if necessary. All processors wait for the others finish their tasks in a synchronization barrier, to be able to execute the next task. Sending and receiving messages between processors is only allowed at the end of each super-step.

On the BSP model there is no restriction on sending messages, but all of them should be received before the synchronization barrier. According to the execution model, the first and second phase may occur simultaneously. A BSP algorithm consists of an arbitrary number of super-steps. The BSP model has been widely used on different applications contexts. HPC practitioners have been using the BSP model to design algorithms and software that can run on any standard architecture with guaranteed performance (de Camargo *et al.*, 2006; Goldchleger *et al.*, 2005b; Kirtzic, 2012). Consider a BSP program that runs on S supersteps. Let g be the bandwidth of the network and L the latency—i.e., the minimum duration of a superstep—which reflects not only the latency of the network, but also the overhead of the synchronization step. The cost to execute the i-th superstep is then given by:

$$w_i + gh_i + L \tag{3.1}$$

where w_i is the maximum amount of local computations executed, and h_i is the largest number of packets sent or received by any processor during the superstep. If $W = \sum_{i=1}^{S} w_i$ is the sum of the maximum work executed on all supersteps and $H = \sum_{i=1}^{S} h_i$ the sum of the maximum number of messages exchanged in each superstep, then the total execution time of the parallel application is given by:

$$T = W + gH + LS \tag{3.2}$$

A BSP algorithm, consequently, can be completely modeled by the parameters (w, h, g, l). Using these parameters, the approximate execution time of a BSP algorithm can be characterized. One of the great advantages of the BSP model is that it facilitates to develop parallel programs on different systems and architectures, serving as a bridge between the programmer who develop parallel applications in massively parallel architectures.

To ease the development and analysis of parallel programs, the BSP programming model has been implemented as API libraries (Holl *et al.*, 1998), and recently enhanced to simplify programming on GPU architectures (Hou *et al.*, 2008). These developments help to create scientific applications in massively parallel environments computing in an easier and better way.

Multi-BSP model

The Multi-BSP model is an adaptation of the BSP model, the BSP model is commonly used in a distributed memory parallel environment, and multi-BSP is a BSP extension for multi-core processors. Valiant (2011) created the Multi-BSP model which is used in a parallel shared memory environment. Computational models, such as multi-BSP, allow abstraction of the complexity of the problem in a simplification that is not significantly away from the reality of current computational architectures.

Multi-BSP is a multi-level model that has explicit parameters at each level: number of processors p, memory/cache sizes m, communication latency costs g and synchronization costs L. The multi-BSP model of an architecture with depth d will be determined by $4d$ numeric parameters,

$(p1, g1, L1, m1), (p2, g2, L2, m2), (P3, g3, L3, m3), ..., (pd, gd, Ld, md)$. At each level the four parameters quantify, respectively, the number of subcomponents, processors, communication bandwidth, synchronization cost, and memory/cache size (Savadi and Deldari, 2014).

Multi-BSP and BSP are important models that allow bridging the analysis, design and development of parallel algorithms, but they are not useful to design algorithms that are executed in massively parallel architectures. New models of performance prediction of applications that run on GPUs have arisen from adaptations of the models in this literature review.

3.1.3 Coarse Grained Multicomputer Model (CGM)

Dehne *et al.* (2002) studied the problem of designing scalable parallel geometric algorithms for coarse grained cases. They called this model as Coarse Grained Multicomputer model (CGM), which is very efficient for a large set of problems of the ratio $\frac{n}{p}$, with n the size of the problem and p the number of processors. In other words, the CGM model is very good for addressing problems where the problem of size n can be divided between an determined number of processors p. A CGM algorithm is a special case of a BSP algorithm where all communication operations of a super-step are done in h relations. A fundamental difference between the BSP model and CGM is that the first captures real machine parameters while the CGM is an abstraction that allows to develop efficient algorithms in parallel machines.

An algorithm on a CGM machine can be modeled using only two parameters, N and p. This algorithm consists of an alternating sequence of computing and communication rounds also separated by a synchronization barrier. A computational/communication round of the CGM model corresponds to a super-step of the BSP model with communication cost $g(N/p)$. A good performance of algorithms with the CGM model is achieved by minimizing the number of super-steps, the total time of local computations and the total size of the messages.

3.1.4 LogP Model

Synchronization of a large group of processes or threads is expensive in terms of latency, especially on architectures with classification MIMD. For these cases, parallel computing models without synchronization were created. The most popular of these models is the logP model.

LogP model was proposed by Culler *et al.* (1993) and received its name exactly for the variables of the model. Culler et. al. perceived that the PRAM model was not realistic due to the lack of parameters to represent communication costs in parallel applications, especially for distributed applications. The parameters used to describe a parallel system according to LogP model are:

L: *Latency* - caused by communicating a message from a source to a destiny processor.

o: *Overhead* - time during which a processor is busy sending or receiving a message, during that time it can not do computations.

g: *Gap* - minimum time between consecutive message transmissions or between receiving consecutive messages; the reciprocal of g corresponds to the bandwidth of the system.

P: *Processors.*

3.2 A Simple BSP-based Model to Predict Execution Time of CUDA Kernels

In this section, we present a novel simple BSP-model to predict the execution time of CUDA kernels executed over GPUs. Similarly to the BSP model, our model is mainly based on the number

of computational and communication steps used by each thread in a GPU application. These values are multiplied by parameters that describe the number of threads, number of cores, and the processor's clock rate. Differently, from the BSP model, we did not include the synchronization steps of the BSP model, since global synchronizations occur only at the end of the kernels. This model takes into account the main physical properties and optimizations of GPU architectures. Our performance prediction model is based on the cost of communication and computation, which are determined independently. The execution time is split between computation and data transfers to and from global and shared memories.

$$T_k = \frac{t \cdot (comp + comm_{GM} + comm_{SM})}{R \cdot P \cdot \lambda}, \tag{3.3}$$

in Equation 3.3, T_k is the approximated execution time of a kernel function with t threads. It sums the computational cost ($comp$) with the communication cost of global memory ($comm_{GM}$) and shared memory ($comm_{SM}$) accesses, performed by each thread. This cost is multiplied by the total number of threads t and divided by the clock rate R times the number of cores P in the GPU. The parameter λ is used to model the effects of application optimizations, such as divergence, coalesced global memory accesses and shared memory conflicts. When the other parameters of the model are found, the kernel function is executed and the value of λ is estimated as the ratio between the predicted execution time of the kernel with the actual measured execution time. Below, each parameter will be explained in more detail.

The computational time used by each thread in a kernel is denoted by $comp$. It is determined by the number of cycles that each thread spends in its computation. FMA operations can be included in $comp$ by reading the source code of the kernel and verifying this possibility with profiling tools.

Communication is evaluated at two levels: global and shared memory. The execution times for communication in global and shared memory per thread are given by $comm_{GM}$ and $comm_{SM}$, respectively. These are defined as the sum of load and write transactions over the global memory and shared memory. This information can be extracted directly from the source code.

Additionally, to account the effects of cache memories on recent GPU architectures, the number of L1 and L2 cache hits are subtracted from the number of loads over the global memory. We have used metrics and events to confirm information about the number of L1 and L2 cache hits. Their contribution to the execution time is calculated separately, multiplying them by their latency times (Mei *et al.*, 2014; Wong *et al.*, 2010). This model allows an easy parametrization, well-suited for any GPU applications in practice. For simplification, we do not consider constant and texture memories nor differences between the latency of load and store transactions. $comm_{SM}$ and $comm_{GM}$ are defined as:

$$comm_{SM} = (ld_0 + st_0) \cdot g_{SM} \tag{3.4}$$

$$comm_{GM} = (ld_1 + st_1 - L1 - L2) \cdot g_{GM} + L1 \cdot g_{L1} + L2 \cdot g_{L2} \tag{3.5}$$

g_{GM}, g_{SM}, g_{L1} and g_{L2} represent the latency in communication over global, shared, L1 cache and L2 cache memory, respectively. Some typical values are 5 cycles for g_{SM} and g_{L1}, 500 cycles for g_{GM} (NVIDIA Corporation, 2014), and 250 cycles for g_{L2}. ld_0 and st_0 represent the total number of load and stores performed by all threads in the shared memory, and ld_1 and st_1 represent the

loads and stores for global memory. The number of loads and stores to global and shared memory are determined by analyzing the CUDA source code. $L1$ and $L2$ are determined by executing an application execution profile and taking the number of hits over each one of these memory levels.

When an application reaches a stable access on the L2 cache and the L1 cache is disabled for caching, the Equation 3.5 can be expressed for the Equation 3.6.

$$Comm_{GM} = (ld_1 + st_1) \cdot g_{GM} \tag{3.6}$$

Aspects about optimization of CUDA kernels, such as coalesced accesses, shared bank conflicts, and divergence are important to define the performance of a kernel (Wu *et al.*, 2013). We consider the effects of those optimizations using the λ factor. It is estimated as the ratio between the predicted execution time of the kernel with the actual measured execution time. The λ factor is important since it permits the adjustment of application performance with the implemented CUDA optimizations and GPU architectures. Finally, intra-block synchronization is not computed, since it does not affect processing time (Feng and Xiao, 2010; NVIDIA Corporation, 2014). Intra-block synchronization is very fast, and it did not need to be included. Nevertheless, we maintained the inspiration on the BSP-model because the extended version of the model considering host memory needs global synchronizations.

Consequently, except for the value of λ and effects on caches L1 and L2, all other parameters are constants. The effect of usage of caches L1 and L2 must be confirmed by profiling. λ performs the adjustment of application performance with the implemented CUDA kernel function. Once defined for the application, the same relative value should work for other similar GPUs and input sizes of the application.

On the next section, we will show how each one of the parameters of the model was obtained. This process was done for a set of selected CUDA kernels. All the CUDA kernels in Table 2.3 were used and we also used 6 CUDA kernels from 4 Rodinia applications.

3.3 Methodology

We have tested the model with all applications presented in Table 2.3, all of them in CUDA using the single-precision format and running a single kernel. These are: *Matrix Multiplication*, *Matrix Addition*, *Dot Product*, *Vector addition* and *Maximum Subarray Problem* (Silva *et al.*, 2014). The last three applications have a single version and only one kernel. For *matrix multiplication*, we have used 4 different kernel strategies; and for *matrix addition* we have used 2 kernel strategies. We have also used 6 kernels of the Rodinia applications, these kernels belong to 4 different GPU applications. These applications are: Back-propagation, Gaussian Elimination, Heartwall, and Hotspot. In total, we have used 15 different CUDA kernels to test our simple analytical model. Each execution of Back-propagation application resulted in a single sample (kernels executions) of their 2 kernels. Every single executions of the other Rodinia applications generated multiple samples (kernels executions) for our experiments, since these kernels are executed inside loops.

The number of computation (*comp*) and communication (ld_0, st_0, ld_1 and st_1) steps were extracted from the application source codes, and information about cache hits in cache L1 and L2 were extracted from profiling. We also confirmed the usage of FMA and SFU using profiling. During our evaluations, all applications were executed using the CUDA profile tool *nvprof*. Each experiment is presented as the average of ten executions, with a confidence interval of 95%. Only

Rodinia Applications were executed on GPU Pascal. Because the machine with this board was not freely available. The process to get each one of the parameter values of the analytical model is explained below.

Vector-Matrix Applications

For problems of one dimension (i.e. vAdd, dotP, MSA), 69 samples were taken of each application. 6 Samples were collected with vector sizes from 2^{17} to 2^{22} increasing the vector size in a power two pattern; and 63 samples from 2^{23} to 2^{28} with a step of 2^{22}. 32 samples were collected of each bi-dimensional application. The matrix sizes of these applications were from 2^8 until 2^{13}, increasing the matrix size in a step of 2^8. Vector-Matrix applications were well characterized in Section 2.3. According to this information the values of the equations 3.3, 3.4 and 3.6 can be computed.

For matrix multiplication versions, *comp* is determined by the number of multiplications and/or operations computed by a thread. In this case, each thread performs N FMA single precision operations. IEEE 754-2008 floating-point standard (IEEE, 2008) states that those operations need a single rounding step. The value of *comp* is the same for the all four optimizations modes, since they differ only in the memory access patterns. With those values, we can compute —using equations 3.4 and 3.6— the values of *comp*, $comm_{GM}$, and $comm_{SM}$. These variables are then multiplied by the number of threads t in the kernel execution and divided by the number of processor p times the clock rate R of the GPU.

The optimizations actually affect only the performance of the communication between threads. As explained above, $\lambda = 1$ in the first execution and it is obtained by the ratio of the predicted execution time of the application with the actual measured execution time. The parameter λ captures the effects of thread divergence, global memory access optimizations, and shared memory bank conflicts. It needs to be measured only once, for a single input size and a single board. The same lambda should work for all input sizes and boards of the same architecture. These parameters are the same for all the simulations and are presented in Table 3.2.

As it was explained in Section 2.2.2, each thread in both versions of matrix addition request 1 element from each matrix elements and compute a single addition. The values of the parameters of the model for (MAU) and (MAC) is $comp = 1add$, $ld_1 = 2$, $st_1 = 1$, $ld_0 = 0$ and $st_0 = 0$. In the same way the parameter values of the model for the kernel (vAdd) were determined.

Application dot Product performs a reduction sum operation after a product. This reduction is performed according to the size of the thread per blocks. The number of steps for this reduction is $log(block\_size)$. This kernel performs a first multiplication with data from global memory, thus each position from each vector is required to global memory. After, a reduction process is performed using a shared variables.

Anyone of the before mentioned applications present branch divergence. The kernel of the Maximum Sub-Array Problem (MSA) is implemented using the CGM model, however, it can be modeled with the proposed model. The scalability of this application is also very regular. Despite the branch code divergence in the source code of this kernel, its scalability is regular. This kernel is computed with 4096 threads, divided into 32 thread blocks with 128 threads on each. The N elements are divided into intervals of N/t elements, one per block and each block receive a portion of the array. The blocks use the shared memory for storing segments of its interval, which are read from the global memory using coalesced accesses. The values of the parameters of the model for (MSA) are shown in Table 3.2.

Par.	Matrix Multiplication				Matrix Addition		vAdd	dotP	MSA
	MMGU	MMGC	MMSU	MMSC	MAU	MAC			
comp	$N\cdot$ FMA				$1 \cdot 24$			$1 \cdot 96$	$(N/t) \cdot 100$
$\mathbf{ld_1}$	$2 \cdot N$				2			2	N/t
$\mathbf{st_1}$	1				1			$1/GS$	5
$\mathbf{ld_0}$	0		$2 \cdot N$		0			$log(BS)$	N/t
$\mathbf{st_0}$	0		1		0			$log(BS)$	N/BS

Table 3.2: *Values of the model parameters over 9 different vector/matrix applications*

Different published micro-benchmarks were used to consider the number of cycles per computation operation in GPUs (Mei *et al.*, 2014), with FMAs, additions, multiplications, divisions, taking approximately 2, 24, 32 and up to 96 cycles of clock. For all simulations, we also considered 5 cycles for latency in the communication for shared memory and 500 cycles for global memory (NVIDIA, 2018). Finally, when the models were complete, we executed a single average instance of each application on each GPU to determine the λ values. As explained above, $\lambda = 1$ in the first execution and it is obtained by the ratio of the predicted execution time of the application with the actual measured execution time. Lastly, for the parameter λ, which captures the effects of thread divergence, global memory access optimizations, and shared memory bank conflicts, we used the values described in Table 3.3.

Kernels / GPUs	Matrix Multiplication				Matrix Addition		vAdd	dProd	MSA
	MMGU	MMGC	MMSU	MMSC	MAU	MAC			
GTX-680	4.50	19.00	20.00	68.00	1.50	9.25	14.00	11.00	0.68
Tesla-K40	4.30	20.00	19.00	65.00	2.50	9.50	5.50	10.00	0.48
Tesla-K20	4.50	21.00	18.00	52.00	2.50	9.00	6.00	10.00	0.55
Titan	4.25	21.00	17.00	50.00	2.50	10.00	5.50	12.00	0.48
Quadro	4.75	20.00	20.00	64.00	1.50	8.25	11.00	9.50	0.55
TitanX	9.50	36.00	36.00	110.00	3.00	9.50	7.00	9.75	0.95
GTX-970	13.00	44.00	46.00	120.00	3.75	9.50	8.00	10.50	1.95
GTX-980	13.00	44.00	46.00	120.00	3.25	9.50	7.00	9.50	1.50

Table 3.3: *Values of the parameter λ for each vector/matrix CUDA kernel in the GPUs used*

The Values of λ for each application and each GPU is shown in Table 3.3 and graphically in Figure 3.3. Gray cells in Table 3.3 group the values of λ by Kepler architectures and those without color belong to Maxwell architecture. The same λ should work for all input sizes and boards of the same architecture. The box plots of Figure 3.3 the statistical information of all the λ values, thus this figure shows the median for each application in all the GPUs, the upper and lower first quartiles and outliers are marked as individual points.

Rodinia Benchmark Applications

Our analytical model was tested with 4 Rodinia algorithms, Back-propagation, Gaussian Elimination, Heartwall, and Hotspot. The number of samples of each application is shown in Table 2.4. Similarly to the vector/matrix applications mentioned before, the variables of computation (*comp*) and communication (ld_0, st_0, ld_1 and ld_1) were extracted from the application source codes. For the kernel (HWL), it was not possible to know the values of the parameters of Equation 3.3, due to a large number of code lines. For this reason, the parameter values of the analytical model were extracted from profile information. These values are presented in Table 3.4. The kernel (HWL) uses constant memory to store large numbers of parameters which cannot be readily fit into shared

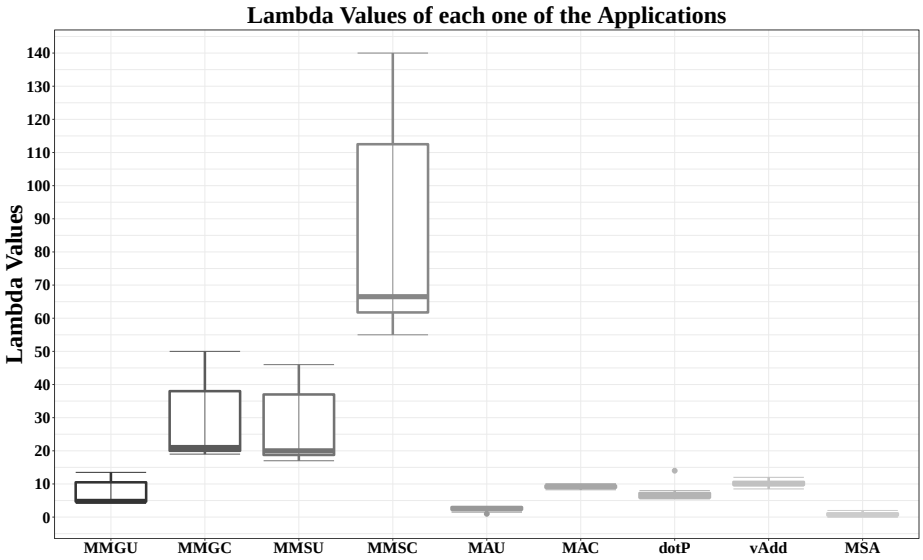

Figure 3.3: *Boxplot of λ values, see table 3.3*

memory, resulting in a high number of instructions over the global memory. Finally, when values of the parameters of the models were completed, we executed a single instance of each application on each GPU to determine the λ values. As explained above, $\lambda = 1$ in the first execution and it is obtained by the ratio of the predicted execution time of the application with the actual measured execution time.

Par.	**Back propagation**		**Gaussian**		**HWL**	**HOT**
	BCK-1	**BCK-2**	**GAU-1**	**GAU-2**		
comp	404	116	36	96	760	500
$\mathbf{ld_1}$	2	4	1	3	7000	2
$\mathbf{st_1}$	1/GS	1	1	1	2000	1
$\mathbf{ld_0}$	BS	0	0	0	2800	2
$\mathbf{st_0}$	BS	0	0	0	1	1

Table 3.4: *Values of the model parameters over 6 CUDA kernels of Rodinia Benchmark Suite*

Finally, for the parameter λ of the Rodinia CUDA kernel, which captures the different optimizations are presented in Table 3.5. These values are also shown graphically in Figure 3.5. This table presents three different groups, they are soft gray, hard gray and without color. These color group the λ values by GPU architectures, thus soft gray is Kepler GPUs, hard gray is Maxwell GPUs and without color is Pascal architectures. It can be possible to notice that λ values are closer when they belong to the same architecture.

In all experiments, we modeled only communication over global memory and shared memory. We did not include the values of the cache L2 for these experiments because they did not impact the variance of the predictions. $L1$ cache in Kepler, Maxwell, and Pascal architectures is reserved for register spills in local memory. Local memory is used for some variables, large structures or arrays. When this information does not fit in the registers and/or when the kernels use more registers than are available, L1 cache helps to spill registers.

GPUs \ Kernels	BCK-1	BCK-2	GAU-1	GAU-2	HWL	HOT
GTX-680	13.20	7.50	0.13	0.65	1.07	20.00
Tesla-K40	13.50	8.50	0.17	1.25	1.12	35.00
Tesla-K20	13.80	8.50	0.17	1.25	1.25	36.25
Titan	13.20	8.50	0.17	1.25	1.12	35.00
Quadro	12.00	7.00	0.12	0.70	0.83	20.00
TitanX	9.00	5.50	0.20	2.00	1.62	17.50
GTX-970	12.00	6.50	0.35	3.50	2.50	20.00
GTX-980	9.60	5.75	0.23	2.50	1.88	17.50
Tesla-P100	18.00	10.50	0.15	2.00	2.75	62.50

Table 3.5: *Values of the parameter λ for each Kernels of the Rodinia Benchmark in the GPUs used*

In a previous work Amaris *et al.* (2015), we performed experiments on a GPU with Fermi architecture. This GPU was the GeForce GT-630. GT-630 has only two Streaming Multiprocessor, each one with 48 cores, resulting in 96 cores. This is a low-level GPU. The L1 cache is used as default configuration in all kernel executions in those Fermi GPUs. The proposed model was used to predict the running times of the 4 kernel versions of Matrix multiplication over this GPU. The version (MMGU) was the only version that had a high rate of L1 cache utilization. Figure 3.4 shows the accuracy of the proposed BSP-model tested in the 4 kernel versions of matrix multiplication over the GT-630. Left-side shows the accuracy of our analytical model using a constant value L1 cache parameter. Right side presents the accuracy of the model of our analytical model using the adaptive values of the L1 cache hits, these values came from the profile information.

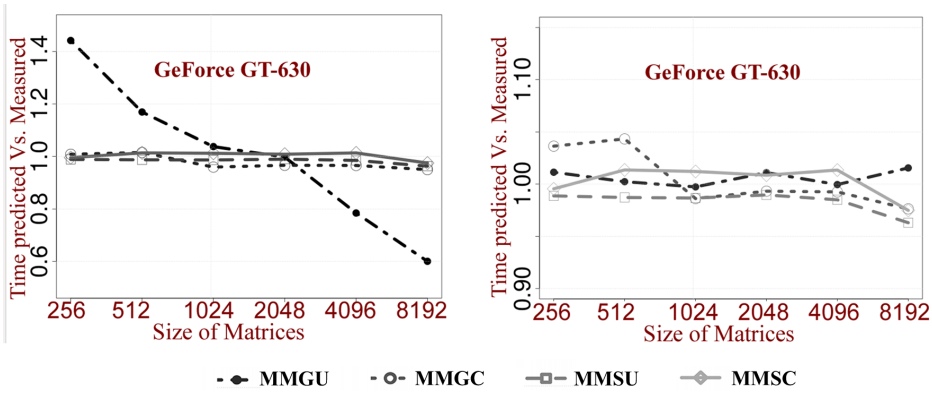

Figure 3.4: *Performance prediction of 4 kernel versions of Matrix Multiplication over a Fermi GPU. Left: Constant value of cache hits. Right: Adaptive value of cache hits based on profile information*

For this reason, $L1$ and $L2$ are always 0 for all our experiments. Cache effects are hard to predict and its impact is higher for specific applications and problem sizes. For example, small vector/-matrix problems without divergence can profit more from the L1 cache memory than applications with branch divergence. This happens in applications which do not use shared memory and their accesses to the global memory are uncoalesced, since they can exploit the cache line size and load multiple elements in the same transaction (Wu *et al.*, 2013).

3.4 Experimental Results

We used T_k as the values computed for our model, such as it was described in Section 3.2. We have performed experiments to evaluate the predictions of our model, by comparing these predic-

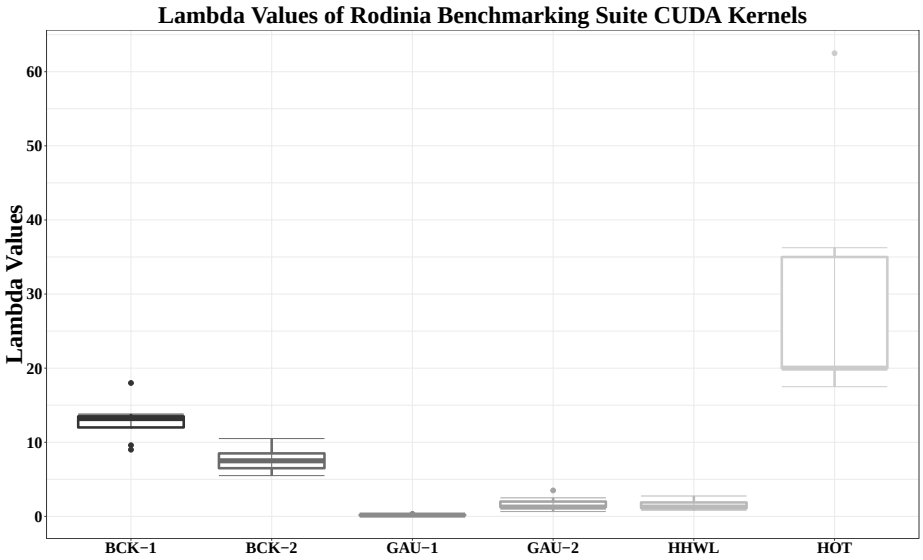

Figure 3.5: *Boxplot of λ values of Rodinia CUDA Kernels, see table 3.5*

tions with measurements of executions of CUDA kernels over different GPUs showed in Table 2.2. For all simulations, we considered 5 cycles for latency in the communication in shared memory and 500 cycles are considered for latency communication in global memory (NVIDIA Corporation , 2014). Finally, for the parameter λ, which captures the effects of thread divergence, global memory access optimizations, and shared memory bank conflicts, we used the values described in the previous section. We compared the measured times (T_m) with the predicted time by the proposed model (T_k), and used the ratio T_k/T_m to define the accuracy of the predictions.

Figure 3.6 and 3.7 show the accuracy of the predictions for the vector/matrix applications and for the selected Rodinia Kernels. These figures show the box plots of the accuracy of the BSP-based analytical model over the different selected CUDA kernels. The box plots show the median for the predictions and the upper and lower first quartiles, with whiskers representing the 95% confidence interval. Outliers are marked as individual points. For vector/matrix applications in the 3.6, the predicted execution time was within 10% of the measured time (T_k/T_m between 0.9 and 1.1). We consider this a reasonable result, considering that all the complexity in the memory and thread hierarchy was adjusted using a single parameter λ. Moreover, this ratio remained nearly constant for almost all input sizes, which shows that the prediction accuracy is dependent on the problem size.

Figure 3.7 shows the rate between the predicted and measured times of each selected Rodinia CUDA kernels over the selected GPUs. This figure shows that predictions of the kernels BCK-1, BCK-2, HTW and HOT were between 0.9 and 1.1, showing good predictions capability of the model. It was not possible to predict correctly the execution times of the kernels (GAU-1) and (GAU-2) because in both kernels the number of threads decreases in a loop during the execution of whole the application. In both kernels execution with few threads are launched. GAU-K1 and GAU-K2 decrease the number of threads in each iteration during their executions and consequently the number of instructions. These instruction variations degraded the throughput significantly and the model required calibration of the parameter λ or another adjustable parameter. Many samples of the kernels GAU-K1 and GAU-K2 resulted in big outliers. These bad predictions can be fix them using machine learning techniques or adding parameters based on throughput.

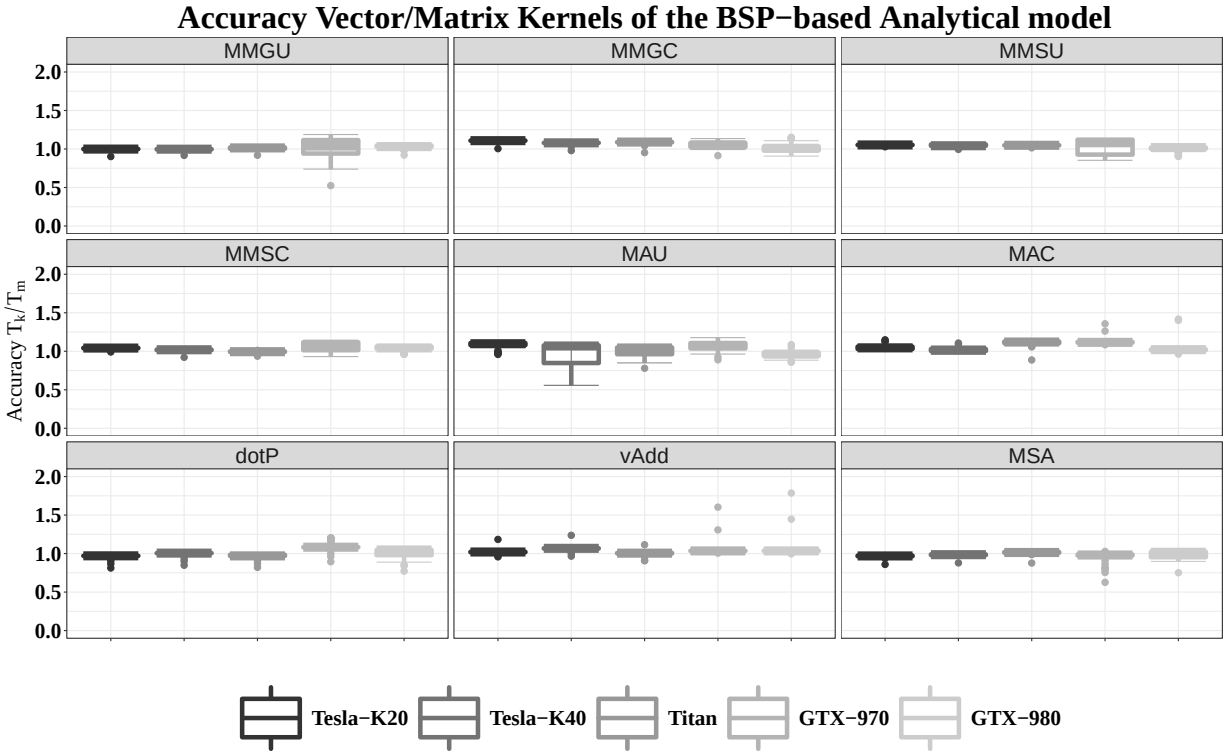

Figure 3.6: *T_k/T_m of vector and matrix algorithms with different values of λ, see Table 3.3*

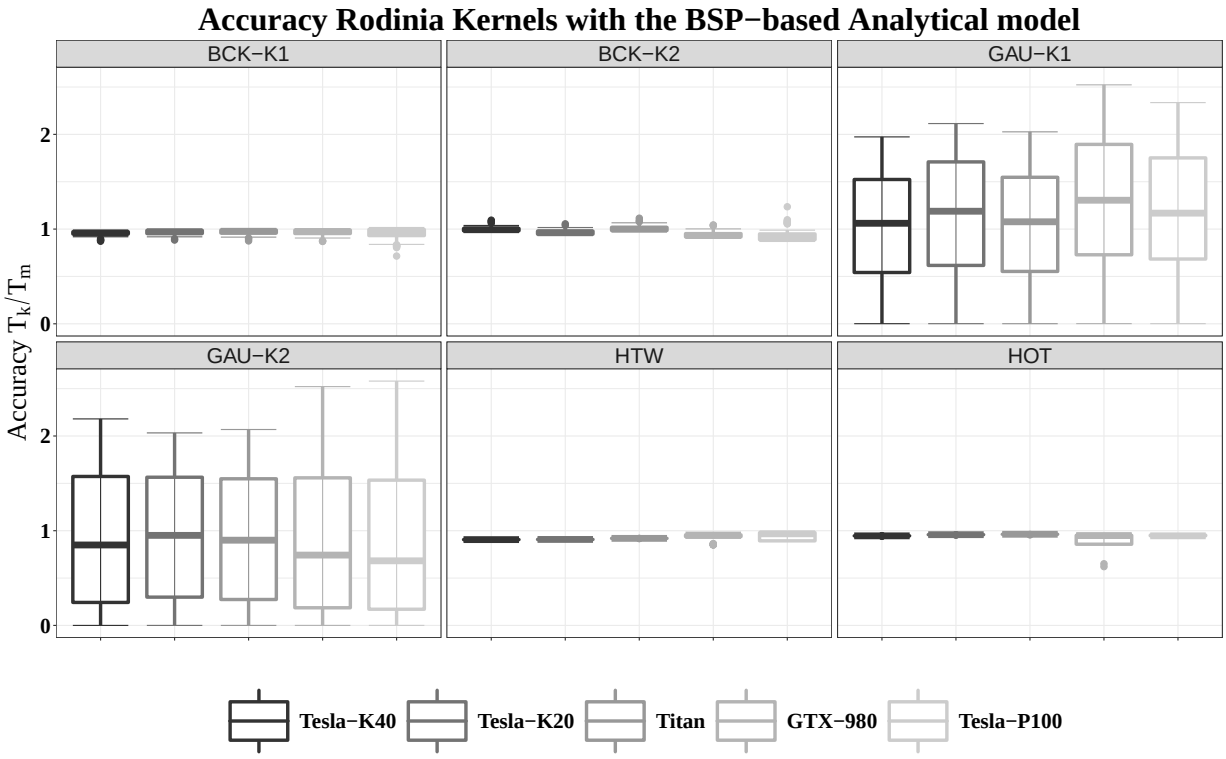

Figure 3.7: *T_k/T_m of Rodinia CUDA kernels with different values of λ, see table 3.5*

Experimental results show that we can use the simple analytical models in scenarios with GPUs of the same architecture using an adjusting parameter. In both cases the model can predict applications execution time from measurements on a single board with a single input size.

By considering two levels of memory, shared and global memories, we could accurately model the performance of these applications using several GPU models and problem sizes. The usage of one adaptable parameter λ was enough to model the effect of data coalescing, divergence, bank conflict over shared memory, and other optimizations. A similar set of parameters also model the effects of cache hits, computation and communication process of any GPU application. In the majority of the scenarios, the time measured were around 0.8 to 1.2 times the model predicted execution time.

The proposed BSP-based analytical model is useful for those applications where scalability is regular according to their input parameters. When kernels iterate in an application and the number of threads changes on each iteration, it makes difficult to predict the running time of any CUDA kernels. Machine learning can be a solution for this type of kernels, next Chapter 4 describes an implementation of machine learning techniques in two different scenarios. Another option would be to use other analytical models which are explained in Section 3.5 of this chapter. Next section will present some main related works using analytical models to predict GPU applications.

3.5 Related Works

In recent years, studies on GPU performance using analytical modeling have appeared, most of them have also used NVIDIA cards in their experiments. Hong and Kim (2009) have proposed and evaluated a memory and parallelism-aware analytic model to estimate execution time of massively parallel application in GPUs. The key idea is to find a metric which they have called MWP (Memory Warp Parallelism) and CWP (Compute Warp Parallelism). The analytic model provides good performance predictions, however, this model requires a deep analysis and understanding by third-party developers of CUDA applications. They have introduced the metrics MWP and CWP, MWP is related to how much memory parallelism in the application and CWP is related to the program characteristics. CWP and MWP are used to decide whether performance is dominated by computation or communication. To obtain the parameter values for this analytical model, authors used microbenchmarks.

Kothapalli *et al.* (2009) have presented a combination of known models with small extensions. The models they have used are: BSP model, PRAM model by (Fortune and Wyllie, 1978) and the QRQW model by Gibbons *et al.* (1998). The authors abstract the GPU computational model by considering the pipeline characteristic of the application in GPU architectures. But they do not describe how divergence can impact their model and the efficiency of GPU applications.

Zhang and Owens (2011) have presented a quantitative performance analysis model, based on micro-benchmarks for NVIDIA GeForce 200-series GPUs. They have developed a throughput model for three components of GPU execution time: the instruction pipeline, shared memory access, and global memory access. The model is based on a native GPU instruction set instead of the intermediate PTX assembly language or a high-level language. Our model uses a high-level bridging model for parallel computation and is focused on computation and communication processes for any GPU application. This encourages developers to use better optimizations in communication and computation.

Meng *et al.* (2011) created a framework named GROPHECY. GROPHECY can estimate the

performance benefit of GPU acceleration from CPU code skeletons. They used internally the analytical model proposed by Hong and Kim (2009). They extracted information from the CPU code skeletons and performed an automated process comparing with different synthetic kernels, another GPU performance model and compare both, the user CPU code skeleton and the best proposed GPU layout.

Kerr *et al.* (2012) developed a methodology for the systematic construction of performance models of heterogeneous processors. This methodology is comprised of experimental data acquisition and database construction, a series of data analysis passes over the database, and model selection and construction. They developed a framework, named Eiger, that implements their methodology. Another framework to construct performance models was presented by Spafford and Vetter (2012). They used a domain specific language to develop analytical performance models for the three dimensional Fast Fourier Transform (3D FFT).

Boyer *et al.* (2013) proposed a GPU performance modeling framework that predicts both kernel execution time and data transfer time. To address this, they used the framework GROPHECY Meng *et al.* (2011) which was described above. Thus, Authors extended GROPHECY to account for data transfer time. They named this extension as GROPHECY++. They tested with GROPHECY++ on a production data analysis and visualization machine at Argonne National Laboratory. Authors also used different GPU applications from the Rodinia Benchmark suite.

Recently, Konstantinidis and Cotronis (2017) used a classic RooflineWilliams *et al.* (2009) model to propose a GPU kernel performance estimation. They limited this problem by either memory transfer, compute throughput or other latencies. They utilized micro-benchmarking and profiling in a "black box" fashion to get these limits. These values are used in a quadrant-split visual representation, which captured the characteristics of different GPUs in relation to a particular kernel.

Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures has been also proposed. Augonnet *et al.* (2009) considered that a deep knowledge normally required to model the performance of parallel applications. For this reason, authors presented an auto-tuning performance prediction approach based on history tables built during execution of the applications. This technique performed automatic calibration of tasks which are executed on runtime systems.

Most of the work mentioned above, they created models of different approaches to predict the performance of GPU applications. They had to use different analytical models to abstract different phases of a GPU application. Our model is based solely on the BSP model. To obtain a good performance modeling with these main related works is necessary deep knowledge of the GPU application and GPU architecture. In our analytical model, scalability, communication optimization, and conflicts, divergence, and differences between architectures are all adjusted by a single parameter λ. Modeling irregular applications with any analytical model always will fail in certain circumstances. Simple analytical models can predict well applications which scale regularly. This subject is also pointed it out in this work.

Profiling techniques were used to confirm information about caches memories accesses. Profile information is also used to establish parameter values about computation and/or communication. This is done only when these parameters were difficult to find out in the source code of the kernels. This model allows an easy parametrization, well-suited for many CUDA kernels in productions.

Chapter 4.

Machine Learning Techniques to Predict CUDA Kernels

The methods developed in the area of machine learning can be used to predict running times of GPU applications. Therefore, it is always meritorious to research more effective mechanisms to obtain better performance predictions. Information about profiling and traces of heterogeneous parallel applications can be used to improve current Job Managment Systems, which require a better knowledge about the applications (Emeras *et al.*, 2014). Machine learning techniques can learn to capture interactions between CUDA kernels and GPU architectures without manual intervention, but may require large training sets.

In this chapter, we compared three different machine learning approaches: Linear Regression, Support Vector Machines and Random Forests with a BSP-based analytical model, to predict the execution time of GPU applications. Comparison between these two approaches will be presented using the Mean Average Percentage Error (MAPE). Two different methodologies with machine learning techniques were used. First, a fair comparison is done using the same features that our analytical model. Second, a step of features extraction was performed from the profile information. Correlation analysis and hierarchical clustering are performed in this second methodology. Profile information was collected from each application over each GPU to use as data input for the ML algorithms.

The rest of this chapter is organized as follows. Section 4.1 presents background concepts on machine learning techniques and feature extraction. Section 4.2 describes methodology of two different scenarios. Section 4.3 presents the experimental results. Finally, Section 4.4 reviews important related works about performance prediction of GPU applications using machine learning techniques.

4.1 Concepts and Background

Machine learning refers to a set of techniques for understanding data. The theoretical subject of "learning" is related to prediction. ML involves building statistical models for predicting, or estimating an output based on input features. There are different kinds of machine learning techniques based on their learning type. Regression techniques belong to supervised learning. Other classes of learning are unsupervised, semi-supervised and reinforcement learning. Regression models are used when the output is a continuous value. In this work, we used three different machine learning methods: Linear Regression, Support Vector Machines and Random Forest. There exist other machine learning techniques with sophisticated learning processes. However, in this work, we wanted

to use simple models to prove that they achieve reasonable predictions.

4.1.1 Linear Regression (LR)

Linear regression is a straightforward technique for predicting a quantitative scalar response Y on the basis of one or more predictor variables X_p. When $p = 1$, the method is called simple linear regression and when $p \geq 2$ the method is called multiple linear regression. Linear regression assumes that there is approximately a linear relationship between each X_p and Y. It gives to each predictor a separate slope coefficient in a single model. Mathematically, we can write the multiple linear regression model as

$$Y \approx \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p + \epsilon \tag{4.1}$$

where X_p represents the pth predictor, β_p quantifies the association between that variable and the response and ϵ is a mean-zero random error term.

In Equation 4.1, $\beta_0, \beta_1, ..., \beta_p$, are unknown constants that represent the intercept and slope values in the linear model and they are also known as the model coefficients or parameters. Once the training process is done and the values of these constants are estimated, it is possible to predict future samples with the same features used and different values than in the training process. Figure 4.1 shows an example of a linear regression model. In this figure, red points can be the training data of the model, and the blue line is the linear model constructed from the training data.

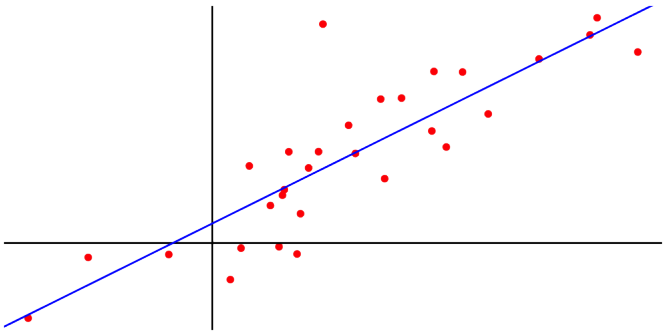

Figure 4.1: *Example of a Linear regression model*

Least square approach is commonly used to find the linear relations between the variables in the process. In this procedure, regression starts with algebraic expressions for the sum of the squared distances between each observed point and a hypothetical line. The least-squares regression function is obtained by finding the orthogonal projection onto the subspace. The expression is then minimized, until finally to find the regression coefficients.

In single regression models, the error can be represented as $\epsilon_i = y_i - \hat{y}_i$. This error is the difference between the ith observed response value and the ith predicted value by the linear single model. So, the residual sum of squares (RSS) is computed as $RSS = e_1^2 + e_2^2 + ... + e_n^2$. RSS is a common measure to validate the accuracy of machine learning models. Another important measures is the RSE (Residual Standard Error). RSE is given by the formula $RSE = \sqrt{RSS/(n-2)}$. RSE is an estimate of the standard deviation of ϵ. The quality of a linear regression model is commonly computed with the RSE and the R^2. R^2 takes the form of a proportion. The values of this error are always between 0 and 1. Where 1 means an excellent representation of the predictions and 0

means the opposite.

Another way to solve the problem multiple linear regression is using a linear algebra approach. In this approach, a linear equation system is build to find the regression coefficients. This equation system has the form $\mathbf{Ax} = b$ for given matrix A, and vector b. This equation system can be solved using linear solvers. QR decomposition method is used in this work to solve this linear equation system.

4.1.2 Support Vector Machines (SVM)

Support Vector Machines is a widely used technique for classification and regression problems. SVMs are generalizations of simple classifier called the maximal margin classifiers, which separate hyperplanes in $p-$dimensional spaces. A hyperplane is a flat subspace of dimension $p-1$. If $p \geq 3$ dimensions, then it can be difficult to visualize a hyperplane, but the notion of $(p-1)$-dimensional subspace still is applied. A natural choice to separate multidimensional hyperplanes is the maximal margin hyperplane.

The maximal margin hyperplane is the separating hyperplane for which the margin is largest and the hyperplane has the farthest minimum distance to the training observations. If a separating hyperplane if possible to separate, then it can be used to construct a natural classifier.

The support vector classifier is an approach for classification for problems of two classes, if the boundary between the two classes is linear. However, in the practice, non-linear class boundaries are regularly faced. SVM is an extension of the support vector classifier. SVMs belong to the general category of kernel methods, which are algorithms that depend on the data only through dot-products.

The dot product can be replaced by a kernel function which computes a dot product in some possibly high dimensional feature space Z. It maps the input vector x into the feature space Z though some nonlinear mapping. The dot product of two r-vector a and b is defined as $\langle a, b\rangle = \sum_{i=1}^{r} a_i b_i$. The dot product of two observations x_i,$x_{i'}$ is generalized by a function kernel of the form $K(x_i, x_{i'}$. These kernels are:

- Equation 4.2 presents the Linear Kernel for SVM.

$$K(x_i, x_{i'}) = \sum_{j=1}^{p} x_{ij} x_{i'j}. \tag{4.2}$$

- Polynomial Kernel for SVM is described by equation 4.3, where $d \geq 1$ and it is related with the degree of the polynomial.

$$K(x_i, x_{i'}) = (1 + \sum_{j=1}^{p} x_{ij} x_{i'j})^d. \tag{4.3}$$

- Radial Kernel SVM is shown in equation 4.4. In equation 4.4 γ is a integer constant.

$$K(x_i, x_{i'}) = exp(-\gamma \sum_{j=1}^{p} (x_{ij} x_{i'j})^2). \tag{4.4}$$

These kernels were tested and used during the experiments.

4.1.3 Random Forests (RF)

Random Forests belong to decision tree methods, capable of performing both regression and classification tasks. In general, a decision tree with M leaves divides the feature space into m regions R_m, $1 \leq m \leq M$. The prediction function of a tree is then defined as $f(x) = \sum_{m=1}^{M} \theta_m I(x, R_m)$, where R_m is a region in the features space, θ_m is a threshold corresponding to region m and I is the indicator function, which is 1 if $x \in R_m$, 0 otherwise. The values of θ_m are determined in the training process.

Figure 4.2a) shows an illustration of a recursive binary partitioning of an input space and Figure 4.2b) the traversal of the binary tree which describes the recursive subdivision of the space by Figure 4.2a). Conditional statements divide the different regions of the space using the values of two inputs x_1, x_2 and the values of θ. Within each region, there is a model to predict the target variable. For example, in our case, regression, we may predict a constant over each region.

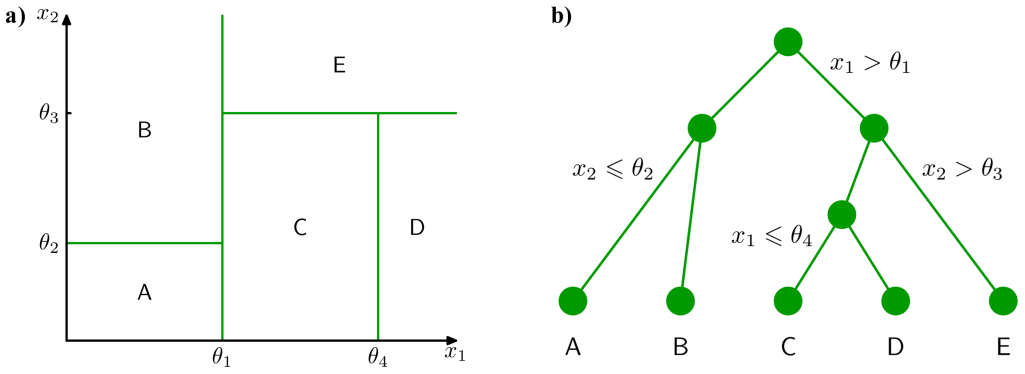

Figure 4.2: . ***a)*** *A two-dimensional space that has been partitioned into five regions and* ***b)*** *its respective binary tree. Figures copied from Bishop (2006)*

Random forest consists of an ensemble of decision trees and uses the mode of the decisions of individual trees. The training algorithm of the random forests implements the general technique of bootstrap aggregating. The random forest presents the following procedure for training:

1. At the current node, randomly select p features from available features D. The number of features p is usually much smaller than the total number of features D.
2. Compute the best split point for tree k using the specified splitting metric and split the current node into nodes and reduce the number of features D from this node on.
3. Repeat steps 1 to 2 until either a maximum tree depth l has been reached.
4. Repeat steps 1 to 3 for each tree k in the forest.
5. Aggregate over the output of each tree in the forest.

4.1.4 Feature Extraction Techniques

Correlation techniques and hierarchical clustering algorithm were used in the phase of feature extraction to reduce the dimensionality of the features. Here, we show a short theoretical background of the techniques used in this work.

There exist different correlation functions, among them, Pearson, Spearman, and Kendal. The Pearson's correlation evaluates the linear relationship between two continuous variables, but omit

variation in different scales. The Spearman's correlation evaluates the monotonic relationship between two continuous or ordinal variables. Besides Spearman correlation consider relations in different scales and spaces of the features due to the rank variables.

Pearson's correlation is commonly represented by the greek letter (ρ) when it is used for populations and by the letter r when it is used for samples. We will denote it with the letter r, the formula for r in a simplified form is

$$r = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y}, \tag{4.5}$$

where, $\text{cov}(X, Y)$ is the covariance between two variables, X and Y, σ_X is the standard deviation of X and σ_Y is the standard deviation of Y.

The Spearman correlation coefficient is defined as the Pearson correlation coefficient between the ranked variables (Myers *et al.*, 2010), for a sample of size n, the n raw scores X_i, Y_i are converted to ranks $\text{rg}X_i$ $\text{rg}Y_i$. Here, this correlation is denoted as r_s, the formula for r_s in a short manner is

$$r_s = \frac{\text{cov}(rgX, rgY)}{\sigma_{rgX} \sigma_{rgY}}, \tag{4.6}$$

where $\text{cov}(rgX, rgY)$ is the covariance of the rank variables, and σ_{rgX} and σ_{rgY} are the standard deviations of the rank variables.

Hierarchical clustering creates groups from a distance matrix. Different metrics or distance functions exist in the literature, among them, Euclidean, Manhattan, Canberra, Binary or Minkowski, however, correlation functions can be also used. A heat map of pair-wise correlations is a simple way to discover relationships between pairs of quantitative variables in a dataset. A dendrogram is a tree diagram frequently used to illustrate the arrangement of the clusters produced by a hierarchical clustering algorithm. In Figure 4.3 is shown a dendrogram with its respective distance matrix using a Spearman correlation function. We can see in this figure a dendrogram with 10 different features about the profile information. In this figure, red cells have a high correlation while blue cells have a low correlation.

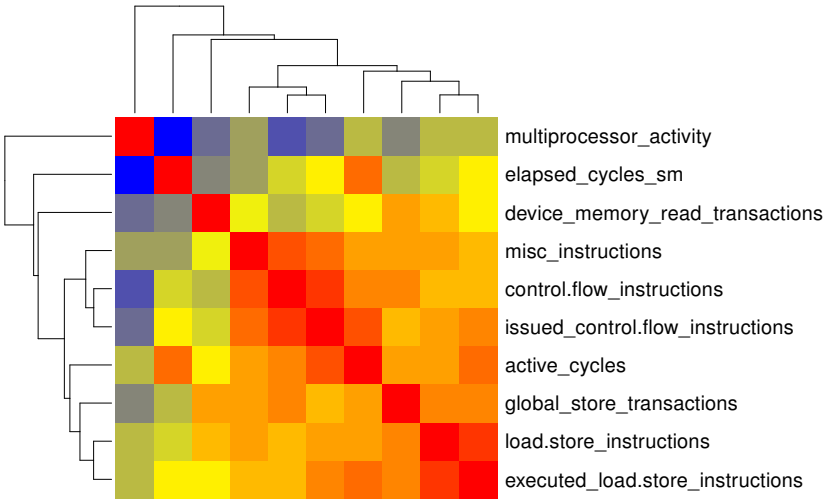

Figure 4.3: *Heatmap and Dendrogram of a correlation matrix with some features in the experiments*

A dendrogram is a tree diagram frequently used to illustrate the arrangement of the clusters produced by hierarchical clustering algorithms. Figure 4.3 shows a dendrogram with its respective correlation matrix. This dendrogram can be cut in a certain height to permit create a specific

number of clusters. More details about these feature extraction techniques are explained in Section 4.2.2.

4.2 Methodology

This section presents the methodological process undertake to conduct this research with machine learning techniques to predict GPU kernels. Two different approaches with machine learning were proposed. In the first approach, we used the same features used on our analytical model, trying to provide a fair comparison; and in the second approach, we performed a phase of feature extraction. We used three different machine learning methods: Linear Regression, Support Vector Machines and Random Forest.

The error of the predictions is presented using the Mean Absolute Percentage Error (MAPE), see Equation 4.7. In this equation, T_k is the predicted time, T_m is the measured time and n is the number of fitted points. With this error, we have analyzed the reliability of our approaches.

$$MAPE = \frac{100}{n} \sum_{t=1}^{n} \left| \frac{T_m - T_k}{T_m} \right| \tag{4.7}$$

We used R to automate the statistical analyses, in conjunction with the `e1071` and `randomForest` packages to use the `svm` and `randomForest` functions respectively. Collected data, experimental results and source codes are openly available[1] under Creative Commons Public License.

To perform our experiments with machine learning techniques, we first collected the profile information (metrics and events) for each kernel and over each GPU. All data was collected using the CUDA profiling tool **nvprof**. This profiling tool enables us to collect data from the command-line reducing the overhead of this process. This process is done without modifications on the application source codes. **nvprof** has four modes to collect information from command-line, these are: summary, GPU-trace/API-trace, event/metric summary and event/metric trace. We have used GPU-trace and event/metric trace modes. The first mode collected data about execution times. The second mode is very expensive computationally. The event/metric trace mode was executed only one time over each GPU, because of the process for one execution spent more than one week and the variance of the metric and event values was negligible.

GPU-trace mode permits collecting data only from CUDA API functions, such as kernel executions, memory transfer throughput between CPU and GPU, thread hierarchy, shared memory configuration, among others. This is made without adding overhead to these functions. Execution times were collected in this mode. A total of 16 features were collected in this mode. Thus, each GPU-trace mode was executed ten times, and for the experiments, we calculated the mean of these samples and verified their confidence interval of 95% with the Student's t-Test.

In event/metric trace mode, all events and metrics are collected for each kernel execution. Although this causes a large overhead in the execution of the kernels, it gives detailed information about the behavior and performance of the executed CUDA kernel functions. All applications were iterated over their selected parameters. The number of events and metrics varied according to the compute capability of the GPUs. All collected information resulted in an approximated size of 12.5GB and the process spent up to 15 days in each GPU. For each sample, the metrics, events, and traces information were collected in different phases.

[1] Hosted at GitHub: `https://github.com/marcosamaris/gpuperfpredict` [Accessed on 6 may 2018]

4.2.1 Machine Learning Without Feature Extraction

These experiments were done with all the kernels presented in Section 2.2.2. We used the two-dimensional applications (matrix multiplication and matrix addition) using CUDA thread blocks equal to 16^2. We varied the input sizes or matrix sizes from 2^8 to 2^{13} increasing the matrix size in a step of 2^8. We used 32 samples per GPU and a total of 256 samples. For the uni-dimensional problems (dot product, vector addition, and maximum sub-array problem), we used input sizes or vector sizes from 2^{17} to 2^{28}. From 2^{17} to 2^{22}, 6 samples were taken, and from 2^{23} to 2^{28}, 63 samples were taken. For these three applications, 69 samples per GPU were selected, resulting in a total of 552 samples.

Many profile information samples of Matrix multiplication resulted in overflow because of the maximum supported integer in some GPUs. The few samples of the kernels of matrix multiplication resulted in a bias of the training phase and consequently bad predictions. For this reason, kernel versions of Matrix multiplication were not used in this second methodology with machine learning techniques.

All Rodinia applications were iterated over their selected parameters shown in Table 2.2. For most kernels of Rodinia applications, we generated many samples, but we could generate only 57 samples on each GPU from the Back-Propagation (BCK) application. To avoid the bias on the learning algorithms, we selected 100 random samples from the other Rodinia applications in both approaches. It was, this sampled data, that was used as the dataset for the ML algorithms in both approaches.

To be fair with the analytical model, we then choose similar communication and computation parameters to use as feature inputs for the machine learning algorithms. We performed the evaluation using cross-validation, that is, for each target GPU, we performed the training using the other GPUs, testing the model in the target GPU. This process was done for each application separately. The features that we used to feed the Linear Regression, Support Vector Machines and Random Forest algorithms are presented in Table 4.1.

Feature	Description
`num_of_cores`	Number of cores per GPU
`max_clock_rate`	GPU Max Clock rate
`Bandwidth`	Theoretical Bandwidth
`Input_Size`	Size of the problem
`totalLoadGM`	Load transaction in Global Memory
`totalStoreGM`	Store transaction in Global Memory
`TotalLoadSM`	Load transaction in Shared Memory
`TotalStoreSM`	Store transaction in Global Memory
`FLOPS SP`	Floating operation in Single Precision
`BlockSize`	Number of threads per blocks
`GridSize`	Number of blocks in the kernel
`No. threads`	Number of threads in the applications
`Achieved Occupancy`	Ratio of the average active warps per active cycle to the maximum number of warps ed on a multiprocessor.

Table 4.1: *Features used as input in the machine learning techniques*

To generate the features `totalLoadGM`, `totalStoreGM` the number of requests was divided by the number of transactions per request.

We first transformed all data to a log_2 scale and, after performing the learning process. We returned to the original scale after the prediction process using a 2^{T_m} transformation (Barnes *et al.* , 2008), where T_m is the measured time. This finished in a reduction of the non-linearity effects. Figure 4.4 shows the difference between the trained model without (left-hand side graph) and with (right-hand side graph) logarithmic scale. All the regression models resulted in poor fitting in the tails, resulting in poor predictions. This problem was solved with the log_2 transformation.

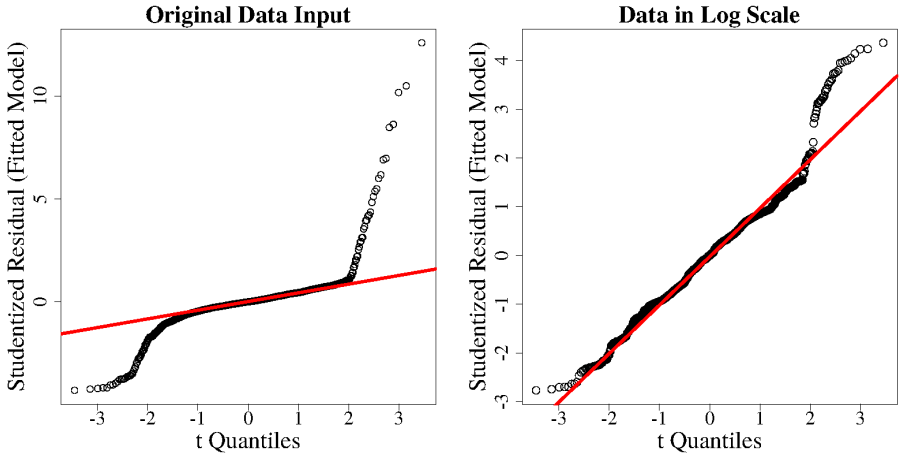

Figure 4.4: *Quantile-Quantile Analysis of the generated models*

4.2.2 Machine Learning With Feature Extraction

In a machine learning process, the accuracy of the predictions depends on the quality of the pre-processing phase (Dasu and Johnson, 2003). In this section, we present a feature extraction process to determine a set of parameters which permits predicting the running time of GPU applications in different contexts or scenarios. As it was mentioned above, we collected separately the set of events, metrics and the execution times of each application of Section 2.2.2 over all GPUs presented in Section 2.2.1.

Kernels of matrix addition, dot product, vector addition, and sub-array maximum problem were used for these experiments. We used the same number of samples as in the last section. All Rodinia applications were used. In total 16 kernels were used for these experiments. We cleaned the collected information, dropping features with no variation among executions and without concordance among GPUs. Each cleaned sample of the dataset resulted in 85 kernel features plus 11 GPU architecture features, for a total of 96 features. We then performed a correlation analysis, followed by a clustering analysis to reduce the number of features (dimensionality reduction) and to find better features to improve the accuracy of the predictions.

We first transformed the data into a log_2 scale and, followed by a normalization, reducing the non-linearity effects (Barnes *et al.*, 2008) (line 2 and 3 of Algorithm 1). To select a small number of application execution features to use with the ML algorithms, we performed correlation and clustering analysis over the data. We performed the correlation analysis using the Spearman Correlation Coefficient (SCC), since it captures relations and variations among features over different scales by using the rank values of the features. We evaluated the SCC for all features against the kernel execution times and applied a threshold of 0.75 of this coefficient (line 4), keeping only the

21 features. These features are shown below:

Algorithm 1: Algorithm of the methodological process done in this second approach

```
   Result: Prediction of the ML techniques
 1 Input: Data sets;
 2 ScaledData = scaleLog(Data sets);
 3 NormalizedData = Normalize(ScaledData);
 4 SCC = corr(NormalizedData, 0.75);
 5 forall No. Param in [5, 10] do
 6 |   dendroG = hclust(corr(SCC));
 7 |   cutDendro = cutTree(dendroG, No. Param);
 8 |   features = variance(cutDendro);
 9 |   forall context in [GPUs, Kernels] do
10 |   |   forall ML in [LM, SVM, RF] do
11 |   |   |   trainingSet != features[context];
12 |   |   |   testSet == features[context];
13 |   |   |   Model = ML( trainingSet);
14 |   |   |   output = predict(testSet);
15 |   |   end
16 |   end
17 end
```

```
 [1] "elapsed_cycles_sm"
 [2] "gld_inst_32bit"
 [3] "gst_inst_32bit"
 [4] "inst_executed"
 [5] "inst_issued1"
 [6] "gld_request"
 [7] "gst_request"
 [8] "active_cycles"
 [9] "global_load_transactions"
[10] "global_store_transactions"
[11] "device_memory_read_transactions"
[12] "l2_read_transactions"
[13] "l2_write_transactions"
[14] "issued_control.flow_instructions"
[15] "executed_control.flow_instructions"
[16] "issued_load.store_instructions"
[17] "executed_load.store_instructions"
[18] "issue_slots"
[19] "control.flow_instructions"
[20] "load.store_instructions"
[21] "misc_instructions"
```

To further reduce the number of features, we used a hierarchical clustering algorithm. We first create a similarity matrix over all features, using the Spearman correlation coefficient. This similarity matrix is the input for the clustering algorithm. This algorithm then performs the hierarchical clustering by first assigning each feature to its own cluster, and then proceeding iteratively, at each stage joining the two most similar clusters, continuing until there is just a single cluster, as shown in Figure 4.5, which present a dendrogram with 5 clusters built from 20 features.

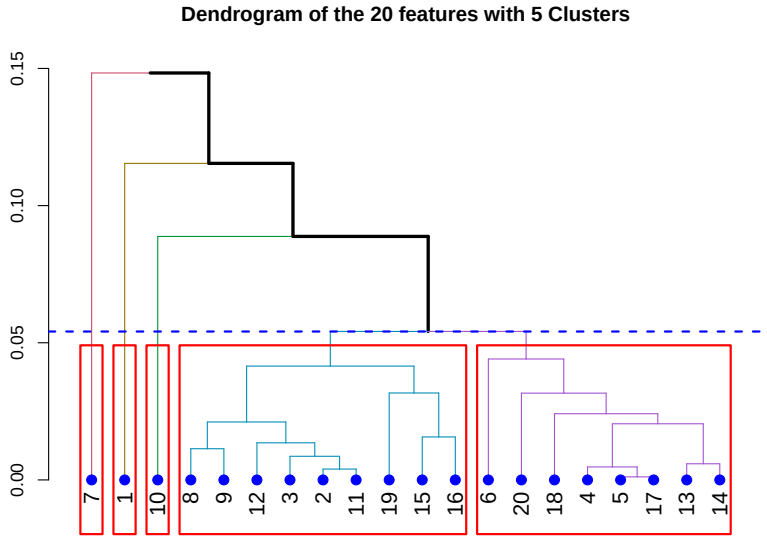

Figure 4.5: *Dendrogram with 5 clusters to select 1 features from each one*

We can then select a number of clusters by cutting the tree at a certain height. For instance, in Figure 4.5, the horizontal blue dashed line cuts the tree in a height that keeps 5 clusters and in Figure 4.5, the horizontal blue dashed line cuts the tree in a height that keeps 10 clusters. Finally, we performed a variance analysis to select a single feature from each cluster.

Regarding the GPU parameters, we performed a correlation analysis and experimented with a different number of parameters. As it was not possible to get a high Spearman Correlation Coefficient with the GPU parameters, the parameters were taken considering the order of the SCC. In this way, we compared the GPU parameters against the kernel execution times using SCC and tested with a different number of GPU parameters. GPU parameters were 11 and during the experiments, we concluded that few GPU parameters captured the relations between hardware and applications. The 11 GPU features are presented below.

```
[1]  compute_capability
[2]  max_clock_rate
[3]  num_of_cores
[4]  num_of_sm
[5]  num_cores_sm
[6]  L2_size
[7]  bus
[8]  memory_clock
[9]  theoretical_flops
[10] bandwidth
[11] global_memory_size
```

In these experiments, we used three different machine learning methods: Linear Regression, Support Vector Machines and Random Forest. These machine learning techniques achieved a high precision in the predictions with the extracted features. Ensemble methods and logistic regression were also tested, but the additional computational complexity did not justify the small improvement in accuracy of the predictions. We evaluated the ML algorithms using sets of 5 and 10 features,

and considered two experimental contexts:

1. **GPUs:** we evaluated if the ML algorithm could predict the execution time over a previously unseen GPU. We used a leave-one-out cross-validation procedure, by selecting all the samples from one GPU as a test set and using all the samples from the other 8 GPUs as the training set.

2. **Kernels:** in this case we predict the execution time of a previously unseen CUDA kernel. We used the same leave-one-out cross-validation procedure, but now selecting the samples from a single kernel as a test set and using the samples from the other 15 kernels as the training set.

4.3 Experimental Results

4.3.1 Results without extraction features

Table 4.2 and 4.3 shows the comparison between both analytical model and machine learning approaches in terms of Mean Absolute Percentage Error (MAPE). This table shows that both analytical model and machine learning techniques obtained reasonable predictions for almost all cases.

The Analytical model presented MAPE around of 9% for almost all the vector/matrix applications. This adjustment with this λ parameters will give us acceptable prediction under certain circumstances, in others, the analytical model always will need calibration. Machine learning techniques can do better predictions whether more and different samples from each class of CUDA kernels, GPUs and better features are available.

Apps	**MAPE ML Techniques**			
	AM	**LR**	**SVM**	**RF**
MMGU	4.41±4.97	16.47±13.40	20.48±14.88	10.65±10.21
MMGC	7.35±3.49	15.53±11.28	13.93±7.49	10.02±7.44
MMSU	7.95±5.31	5.33±1.93	11.79±4.25	4.90±1.99
MMSC	8.31±6.44	14.83±10.22	17.04±8.21	8.41±2.87
MAU	9.71±3.09	69.51±51.77	14.76±9.62	36.76±22.23
MAC	11.00±5.27	12.65±2.77	17.67±8.52	10.83±3.85
dotP	6.89±5.56	3.30±1.42	15.37±8.31	4.67±1.35
vAdd	9.92±7.71	18.98±16.56	14.68±8.15	8.64±4.56
MSA	28.02±13.05	3.02±1.62	10.91±5.02	5.67±4.11

Table 4.2: *MAPE of the predictions of the first context (in %) with the vector/matrix applications*

Table 4.2 reveals that our simple analytical model has reasonably better predictions than machine learning techniques. This is due to the linear growing of the execution time of these vector/-matrix GPU applications. Linear regression and Support vector machine obtained bad predictions because of a large number of features and the few numbers of samples. Random Forests obtained good predictions because it is an ensemble method of decision trees. Random Forests algorithm uses importance variable in the training process and few features are used for this process.

Running times of back-propagation kernels (BCK-K1 and BCK-K2), Heartwall kernel and Hotspot kernel were well predicted with the proposed analytical model. Both back-propagation kernels present vector/matrix operation and their behavior are regular. Samples for training and testing process in both methodologies are selected randomly, while data to test analytical model used the complete execution of each application. Table 4.3 indicates than SVM obtained the best predictions of the running time of CUDA kernels with the same features than the analytical model. Linear regression technique resulted in the worst predictions because of a large number of features and few numbers of samples.

Kernels/Techn.	AM	LR	SVM	RF
BCK-K1	3.92±1.00	19.89±15.57	31.00±49.90	24.49±15.90
BCK-K2	4.86±3.33	86.42±158.74	140.21±283.86	97.49±188.53
GAU-K1	54.09±5.92	65.24±116.70	14.82±3.13	35.31±53.29
GAU-K2	63.72±5.12	26.62±12.38	18.52±11.73	18.60±7.48
HTW	3.71±2.23	100.79±192.76	41.63±38.04	26.29±25.13
HOT	5.53±2.14	167.16±293.56	57.61±83.96	116.92±183.91
LUD-K1	-	27.61±41.32	10.46±8.37	27.45±41.17
LUD-K2	-	57.70±79.18	42.50±54.74	48.16±70.49
LUD-K3	-	41.22±22.08	25.94±24.01	42.13±37.12
NDL-K1	-	16.09±5.27	15.54±10.09	14.03±3.81
NDL-K2	-	15.01±5.11	10.56±4.94	13.20±3.25

Table 4.3: *MAPE of the prediction of the first context (in %) with the Rodinia CUDA kernels*

Figures 4.6 and 4.7 shows a comparison between the accuracy of the Linear Regression (LR), Random Forest (RF) and SVM Regression (SVM) to predict execution times of each application on each target GPU. Figure each box plot represents accuracy per GPU, with each column representing a different technique and each line a different application. These figures show that we could reasonably predict the running time of 16 CUDA kernels over different GPUs belonging to 3 different architectures using machine learning techniques.

Figure 4.6 demonstrates than regardless the MAPEs in Table 4.3 are higher compared to our analytical model, accuracy for most vector/matrix application is between 0.8 and 1.2 of accuracy. Experiments presented in Figure 4.7 were performed using data from the Pascal GPU. This GPU architecture is more powerful computationally compared to the other GPUs in Table 2.2. This resulted in bad predictions of the execution time of the CUDA kernels over the Pascal GPU and this impacted the graphical statistical results.

4.3.2 Results with extraction features

In this section, we will compare only the results of the three machine learning techniques. We first performed the process of feature extraction. After applying the correlation and hierarchical clustering, we selected the 10 features below. The first 5 features are used in the experiments with 5 features, and the complete set is used for the experiments with 10 features.

```
[1] "elapsed_cycles_sm"
[2] "gld_request"
[3] "gst_request"
[4] "executed_control.flow_instructions"
```

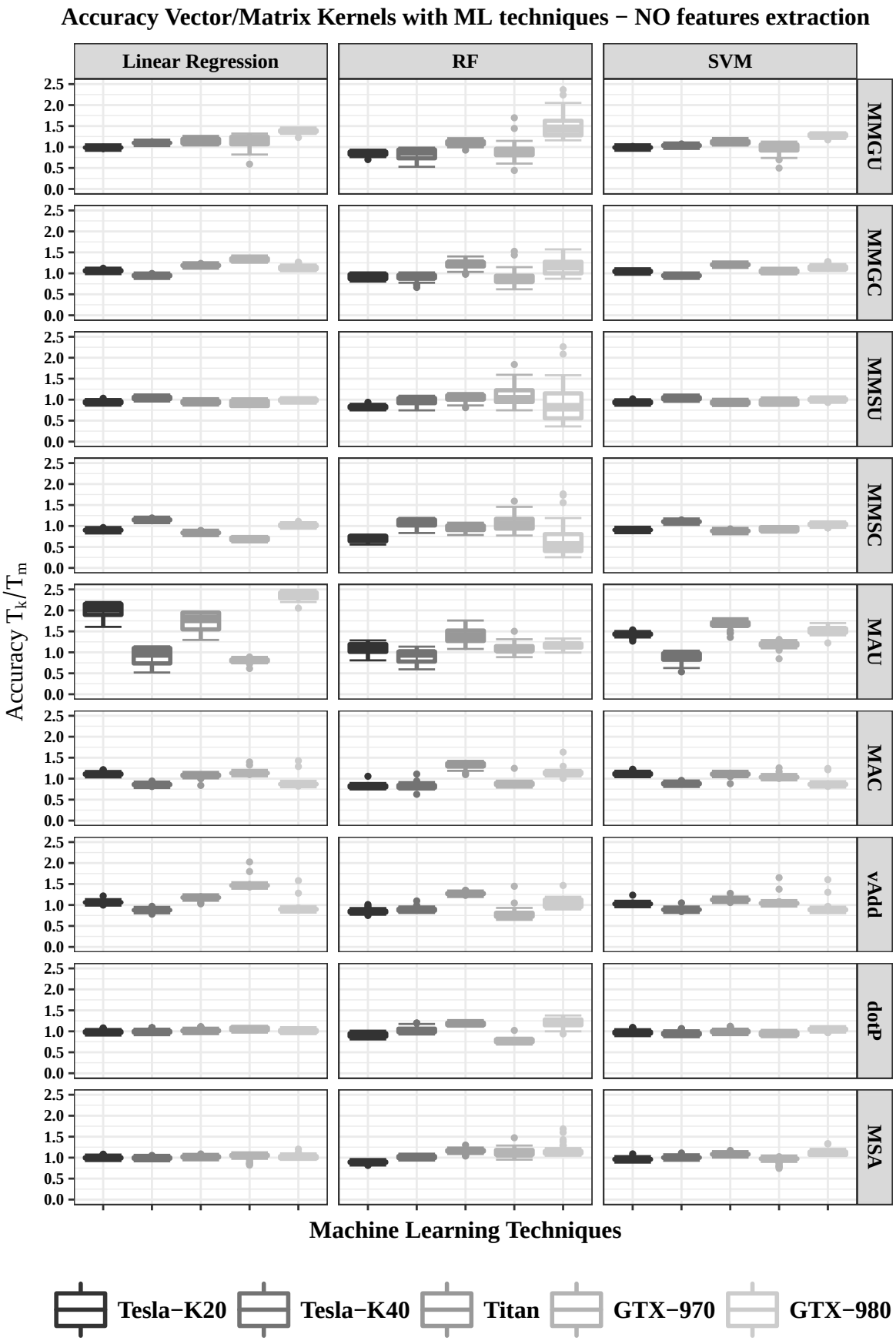

Figure 4.6: *Accuracy Boxplots of the machine learning techniques in the first context of the vector/matrix applications*

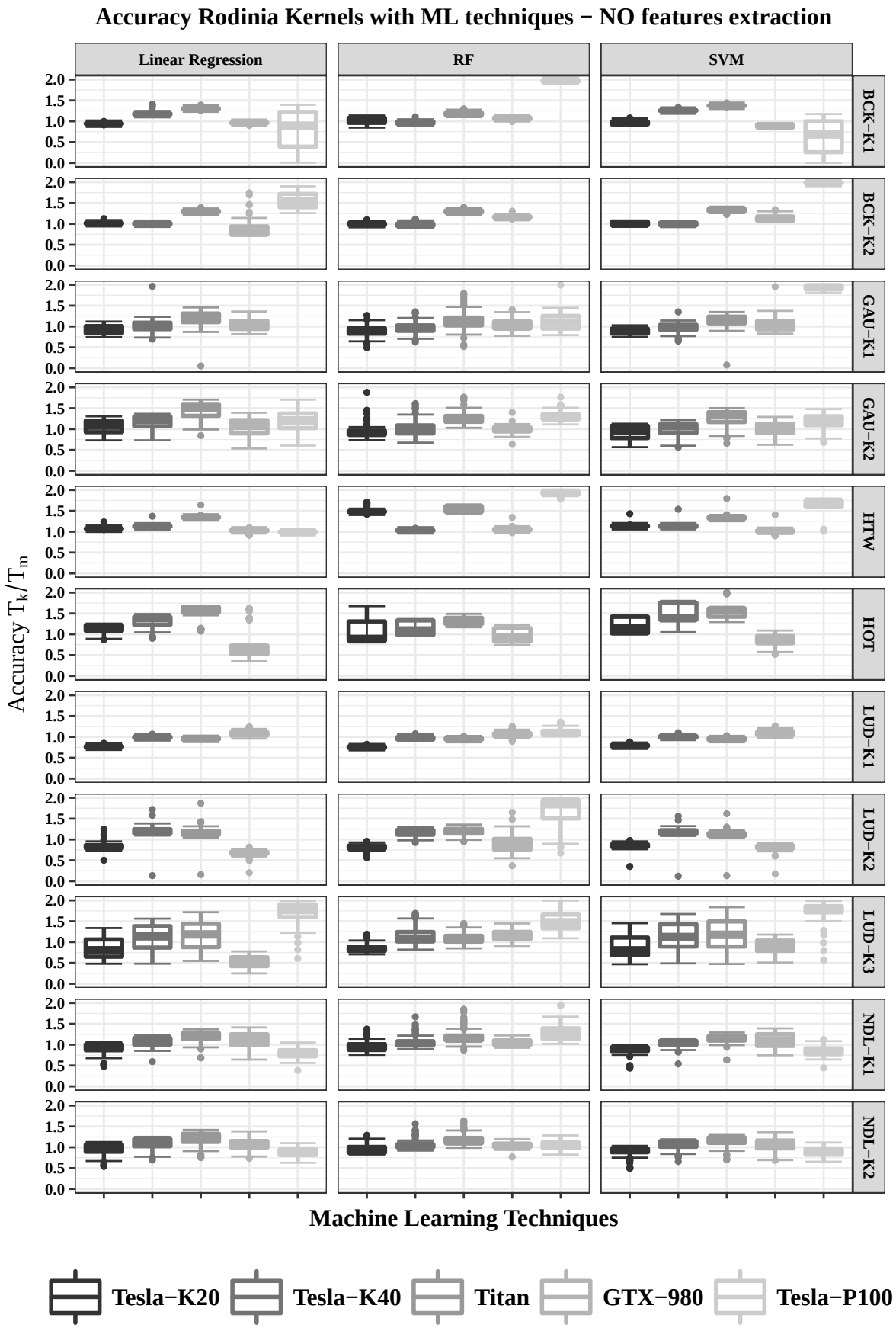

Figure 4.7: *Accuracy Boxplots of the machine learning techniques in the first context of the Rodinia CUDA kernels*

```
[5] "device_memory_read_transactions"
[6] "active_cycles"
[7] "global_store_transactions"
[8] "gst_inst_32bit"
[9] "executed_load.store_instructions"
[10] "misc_instructions"
```

As it was mentioned in the last section, we used two metrics, accuracy and Mean Absolute Percentage Error (MAPE) of the predictions of each machine learning algorithm. The accuracy is the ratio T_k/T_m, between the predicted times T_k and the measured times T_m. MAPE is computed using equation 4.7.

We evaluate the MAPE for each learning algorithm, linear regression (LR), support vector machine (SVM) and random forest (RF), when using 2 GPU parameters (number of cores and amount of L2 cache), when considering either 5 or 10 application features. We justify the selection of these 2 GPU parameters later in this section.

Table 4.4 shows the MAPE for the ML algorithms for the first scenario, where we predict the execution time over a previously unseen GPU. Light gray and dark gray represent the best and worst MAPE for each predicted GPU, respectively. Linear Regression obtained the best MAPE in 4 different GPUs, while Random Forests was better for other 3 GPUs. RF and SVM presented 5 worst cases when they used 10 application features. Prediction of the running time of CUDA kernels over the Tesla-P100 and GTX-680 could improve weather GPUs of the same Compute Capability are added to the training process.

GPUs	**MAPE First Context**					
	LR		**RF**		**SVM**	
	5	**10**	**5**	**10**	**5**	**10**
GTX-680	6.24	6.01	2.92	2.94	6.53	7.40
Tesla-K40	1.34	1.07	0.84	0.88	1.76	1.27
Tesla-K20	1.73	1.31	1.36	1.51	1.97	1.32
Titan	1.31	1.39	2.00	1.94	1.43	1.53
Quadro	2.63	2.25	2.91	3.02	2.81	2.20
TitanX	2.51	2.17	2.87	2.89	2.63	2.38
GTX-970	3.11	3.38	2.46	2.33	3.26	3.47
GTX-980	1.84	1.74	1.81	1.64	2.01	1.82
Tesla-P100	3.67	4.68	8.18	8.64	4.25	4.87

Table 4.4: *MAPE of the prediction of the first context (in %). First scenario varying the GPUs.*

Table 4.5 shows the MAPE of the ML algorithms when predicting the execution time of a previously unseen application. LR and RF with 5 features obtained the best average MAPEs and seem appropriate choices for predicting the execution times. Although LR and RF obtained the best MAPE for 6 and 3 kernels, respectively, the obtained poor MAPEs for some kernels when 10 features are used, resulting in larger average MAPEs. Considering the results from both contexts (Tables 4.4 and Table 4.5), it seems that using LR with 5 parameters is a better overall choice.

Kernels	MAPE Second Context					
	LR		RF		SVM	
	5	10	5	10	5	10
BCK-K1	2.02	2.07	2.29	2.01	2.55	2.47
BCK-K2	2.79	2.38	2.76	2.74	3.60	2.76
GAU-K1	3.98	4.11	3.67	4.17	3.60	3.37
GAU-K2	1.56	1.74	6.87	5.95	2.03	1.98
HTW	3.72	2.50	8.19	5.81	2.98	4.63
HOT	2.72	2.62	2.63	2.45	2.70	2.32
LUD-K1	4.00	3.87	2.53	4.05	3.82	2.93
LUD-K2	1.80	1.81	2.51	2.67	2.17	1.93
LUD-K3	0.97	1.09	1.56	2.16	1.61	1.41
NDL-K1	1.27	1.33	0.50	0.45	1.62	1.01
NDL-K2	1.19	1.19	0.49	0.50	1.55	0.98
MAU	2.13	4.01	12.00	11.83	2.55	4.77
MAC	1.31	1.23	1.92	2.04	1.74	1.88
dotP	2.24	9.24	8.51	5.33	2.32	3.64
vAdd	2.99	4.80	4.09	2.69	3.23	3.67
MSA	18.55	11.07	42.14	39.06	22.20	13.61

Table 4.5: *MAPE of the prediction of the second context (in %). Second scenario varying the CUDA kernels*

For each context, we also tested the number of GPU parameters (features) to use. Figure 4.8 shows the average MAPE of the random forest with different numbers of GPU parameters in both contexts. This figure shows that predictions with only two GPU parameters are good enough to get a high accuracy in the running time predictions.

We can see in both bar plots of Figure 4.8, left-side and right-side, that 2 parameters provided the smaller error for 5 and 10 application features. Experiments in Figure 4.8 were done with CUDA kernels of the Rodinia Benchmark suite in Table 2.4. Consequently, we decided to use 2 GPU parameters in the experiments. The evaluated GPU parameters are shown below, ordered from higher to lower Spearman coefficient against execution times. We showed that using only the number of cores and the amount of L2 cache was enough to obtain reasonable predictions, since our small number of GPUs in the experiments and because those parameters change in the most of GPUs.

```
1 L2_size
2 num_cores_sm
3 memory_clock
4 compute_capability
5 theoretical_flops
6 max_clock_rate
```

Figures 4.9, 4.10 and 4.11 show box plots of the prediction accuracies of the second methodology. Figure 4.9 presents the accuracy of the prediction of the first scenario. For easy interpretation, in this plots is shown only the results of 5 GPUs. Results of the other GPUs are in the same range. Figures 4.10 and 4.11 shown the results for second scenario of vector/matrix CUDA kernels and Rodinia CUDA kernels, respectively. The boxes represent the median and the upper and lower first

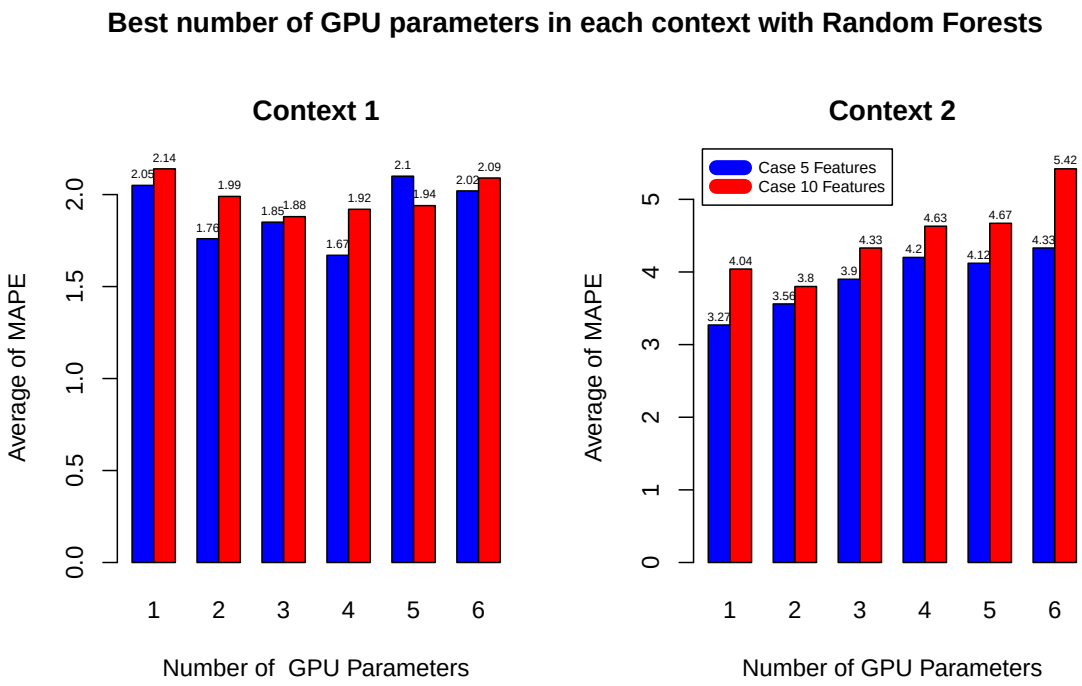

Figure 4.8: *Best number of GPU parameters in each one of the contexts for Random Forests*

quartiles prediction accuracies, with whiskers representing the maximum and minimum possible errors. Outliers are marked as individual points.

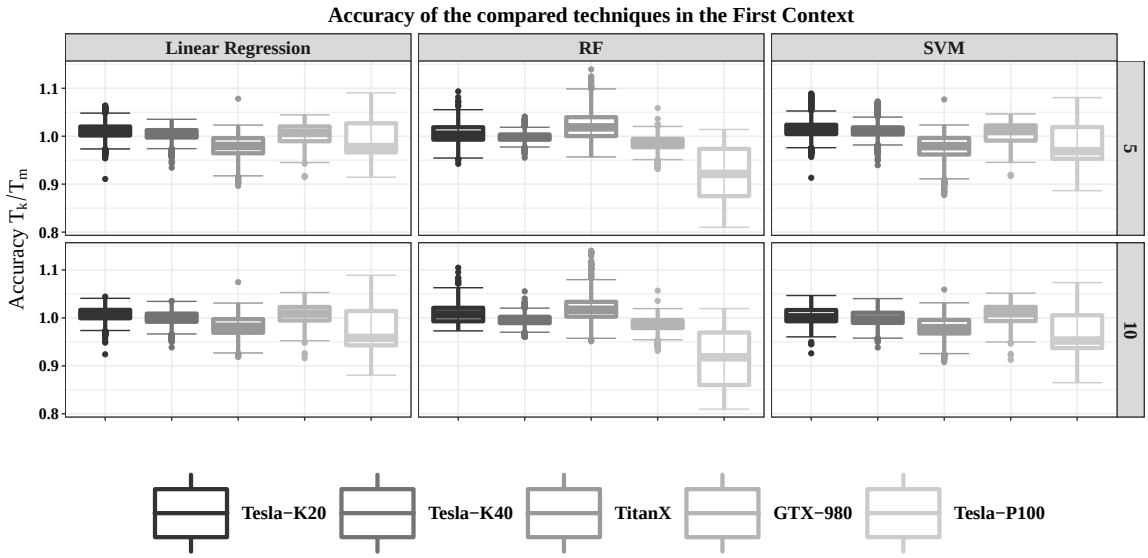

Figure 4.9: *Accuracy (t_k/t_m) of the first scenario with GPUs of 3 different architectures*

Overall, the predictions for the first scenario (Figure 4.9) were good enough, with the most samples with a accuracy between 0.9 and 1.1. For the second scenario (Figure 4.10 and 4.11) the predictions were less accurate for some kernels, but for the SVM and linear regression, they were still mostly between 0.8 and 1.2. SVM and LR obtained similar accuracies because we used a linear kernel for SVMs, since it provided than using nonlinear kernels, such as polynomial, Gaussian (RBF) and sigmoid. For Random Forests, we set the number of trees to 50, and the number of predictor candidates at each split to 3, as these values resulted in better predictions and faster

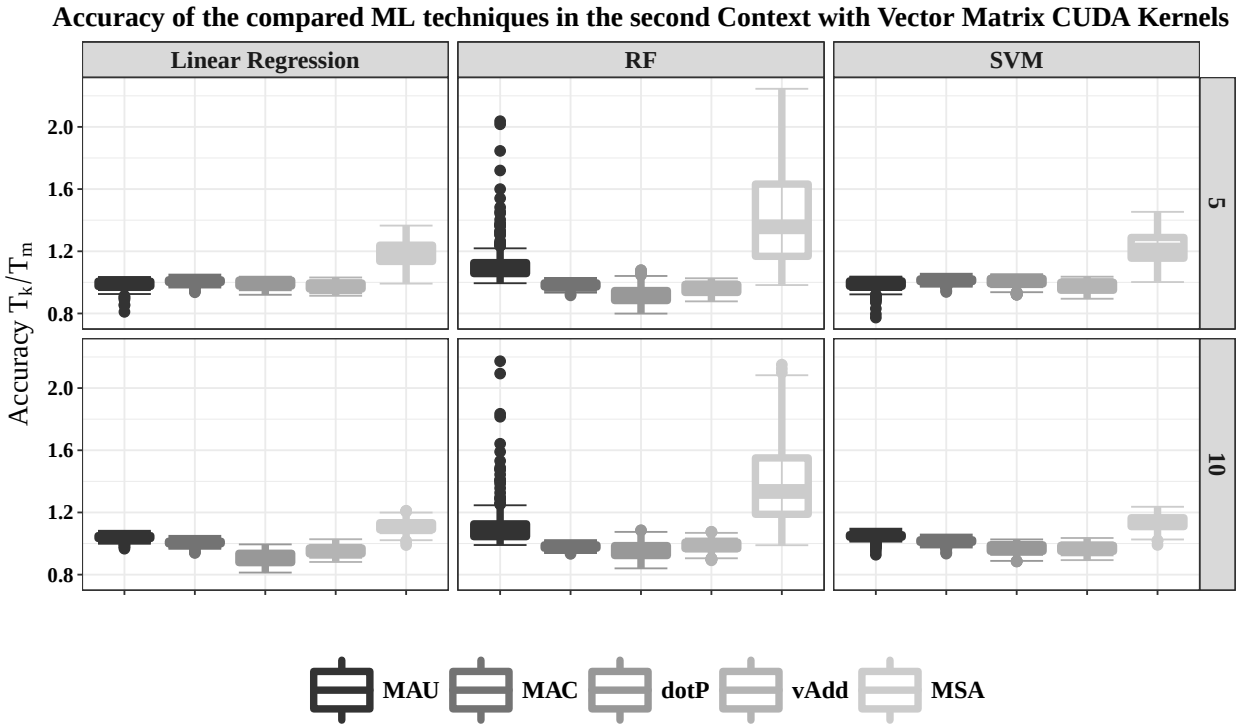

Figure 4.10: *Accuracy of the 5 vector/matrix CUDA kernels used in the second scenarios*

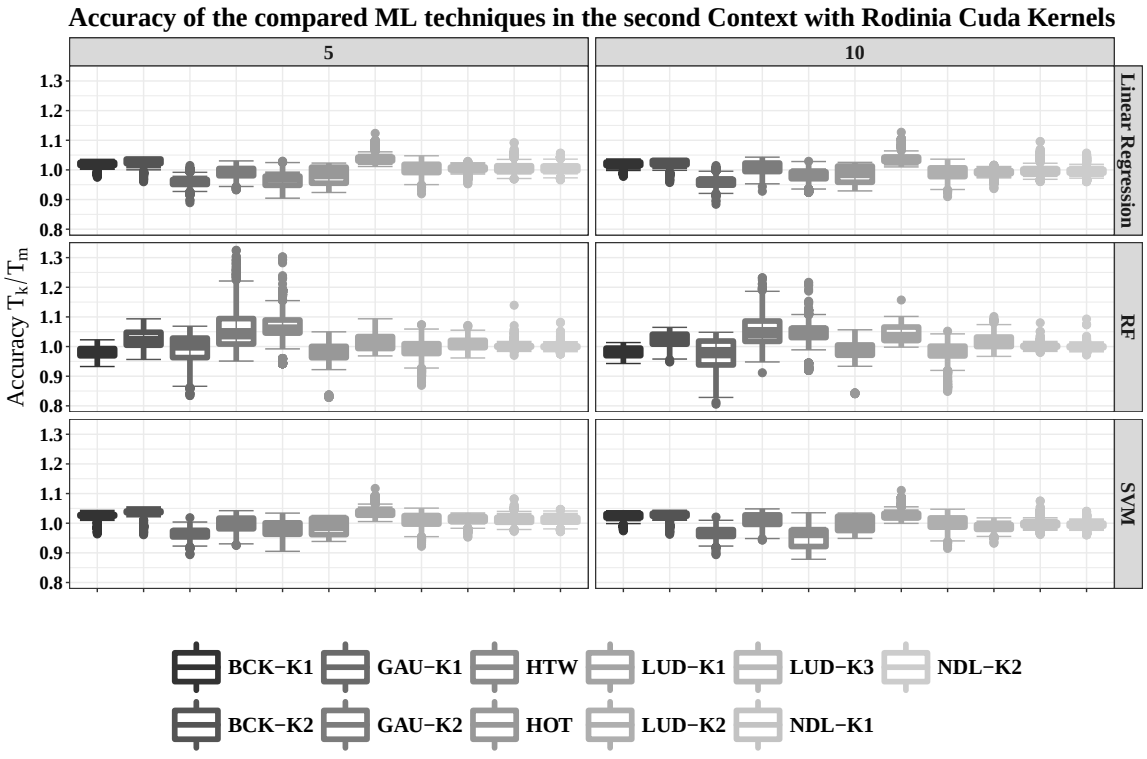

Figure 4.11: *Accuracy of the second scenarios of the Rodinia CUDA kernels*

executions.

4.4 Related Works

Some authors have focused their works on performance prediction of applications using machine learning (Ipek *et al.*, 2005; Li *et al.*, 2009; Matsunaga and Fortes, 2010; Ozisikyilmaz *et al.*, 2008; Singh *et al.*, 2007). The main learning algorithms that they used were statistical methods, Neural Networks, Support Vector Machines, and Random Forest.

Singh *et al.* (2007) were one of the firsts authors to compare performance predictions of parallel computing models Juurlink and Wijshoff (1998), comparing BSP, E-BSP, and BPRAM over different parallel platforms. Some authors have also focused their work on performance prediction of parallel applications using machine learning. All this work is about parallel applications executed over CPUs and not GPU applications.

Kerr *et al.* (2012) developed a methodology for the systematic construction of performance models of heterogeneous processors. They developed a framework, named Eiger, that implements their methodology. This methodology is comprised of experimental data acquisition and database construction, a series of data analysis passes over the database, and model selection and construction. They collected data from a simulator of GPU application named Ocelot, they defined a functional simulator Kerr *et al.* (2009) for specific Nvidia PTX code.

Meswani *et al.* (2012) predicted the performance of HPC applications on hardware accelerators such as FPGA and GPU from applications running on CPU. This was done by identifying common compute patterns or idioms, then developing a framework to model the predicted speedup when the application is run on GPU or FPGA using these idioms. Ipek *et al.* (2005) trained multilayer neural networks to predict different performance aspects of parallel applications using input data from executing applications multiple times on the target platform.

Studies on GPU performance using different statistical and machine learning approaches have appeared. Baldini *et al.* (2014) showed that machine learning can predict GPU speedup from OpenMP applications. They used K-nearest neighbor and SVM as a classifier to know the performance of these applications over different GPUs.

Baldini *et al.* (2014) showed that machine learning can predict GPU speedup of OpenCL applications from OpenMP applications. They used K-nearest neighbor and SVM as a classifier to know the performance of these applications over different GPUs. They showed that a small set of easily-obtainable features can predict the magnitude of GPU speedups on two different high-end GPUs, with accuracies varying between 77% and 90%, depending on the prediction mechanism and scenario.

In recent years, studies on the comparison of GPU performance using analytical modeling, statistical analysis, and machine learning approaches have appeared. Madougou *et al.* (2016) presented a comparison between different GPGPU performance modeling tools, they compare between analytical model, statistical approaches, quantitative methods and compiler-based methods. In this works, authors concluded that the available GPU performance modeling solutions are very sensitive to applications and platform changes, and require significant efforts for tuning and calibration when new analyses are required.

Wu *et al.* (2015) described a GPU performance and power estimation model, using K-means to create sets of scaling behaviors representative of the training kernels and neural networks that map kernels to clusters, with experiments using OpenCL applications over AMD GPUs. Karami *et al.*

(2013) proposed a statistical performance prediction model for OpenCL kernels on NVIDIA GPUs using a regression model for prediction and principal component analysis for extracting features of higher weights, thus reducing model complexity while preserving accuracy.

Hayashi *et al.* (2015) presented a statistical approach on the performance and power consumption of an ATI GPU Zhang *et al.* (2011), using Random Forest due to its useful interpretation tools. Hayashi et al. constructed a prediction model that estimates the execution time of parallel applications based on a binary prediction model with Support Vector Machines for runtime CPU/GPU selection.

In this work, we compare three different machine learning techniques to predict kernel execution times over NVIDIA GPUs. Moreover, we also perform a comparison with a BSP-based analytical model to verify when each approach is advantageous.

Chapter 5.

Conclusions

In this work, we proposed a BSP-based model for predicting the performance of GPU applications. The BSP model offers a solution to tackle parallel problems in massively parallel architectures. We performed a comparison between the BSP-based analytical model and three different machine learning techniques. This comparison was done following two different methodologies. First, machine learning techniques and analytical model used similar parameters and second, a step of features extraction was done. This second methodology was implemented in two different contexts. In the first context, the dataset was grouped by GPUs and in the second context, the dataset was grouped by CUDA kernels.

We have done this comparison using 20 different CUDA kernels belonging to 15 different applications: matrix multiplication with four different kernels; matrix addition with two different kernels; one dot product kernel; a vector addition kernel; a coarse grained solution of subsequence maximum kernel and 11 CUDA kernel functions belonging to 6 applications from Rodinia benchmarking suite (see Table 2.4). All applications were developed in CUDA and were executed on 9 different GPU boards.

5.1 Contributions

By considering two levels of memory, shared and global memories, we could accurately model the performance of applications executed on GPUs. Our simple analytical model predicted reasonably execution times of CUDA kernels that scale regularly (e.g. matrix multiplications, matrix additions, dot product, among others). The usage of one adaptable parameter λ was sufficient to model the effect of data coalescing, divergence and the usage of other memories during the execution of these kernels, which scale regularly. A similar set of parameters also model the effects of cache. The model that we have proposed provides relatively better prediction accuracy than machine learning approaches, but it requires calculations to be performed for each application and at least one execution tuning. Furthermore, the value of λ has to be calculated for each application executing on different GPU architectures.

Machine learning could predict execution times with less accuracy than the analytical model. But, this approach provides more flexibility because prior knowledge about the applications or devices is not demanded as in the analytical model. A machine learning approach is more generalizable for different applications and GPU architectures than an analytical approach. For instance, whether a huge database with information about execution and profiling of GPU applications

over different GPUs, then, ML techniques can predict the execution times of unknown kernels on unknown GPUs with different configurations.

We showed that predictions with high accuracy of running time of GPU applications are possible with an optimum statistical process of feature extraction. We implemented an automated approach for feature extraction based on correlation analysis and hierarchical clustering. We evaluated two scenarios, on which we extract 5 and 10 application features, using three different machine learning techniques: Linear Regression Models, Support Vector Machines, and Random Forests. Each machine learning technique was tested over two different contexts, one on which we predict the execution times over previously unseen GPUs and over unseen CUDA kernels. Over the two contexts evaluated, we have obtained a high precision of the predictions, mostly between 0.9 and 1.1 of the measured execution time.

In our fair comparison, we validated that our simple analytical model had reasonably better predictions than machine learning techniques. This is due to the linear growing of the execution times of vector/matrix CUDA kernels and because of the bias of the prediction process for few and similar data samples. In the comparison between Machine learning techniques, Linear Regression and Random Forests obtained the best MAPEs in both contexts. Considering our results from both contexts, we can also conclude that using machine learning techniques with 5 parameters is a better overall choice to perform a prediction process of running time of CUDA kernels.

5.2 International Contributions Related to This Thesis

The performance of HPC system on heterogeneous many-core system depends on how well a scheduling policy allocates its workload on its processing elements. An incorrect decision can degrade performance and waste energy and power. Job management systems require that users provide an upper bound of the execution times of their jobs (wall time).

During my internship at Grenoble Informatics Laboratory under the supervision of Prof. Denis Trystram, I studied the problem of scheduling parallel applications, represented by a precedence task graph, on heterogeneous multi-core machines. To test the scheduling algorithms, a Benchmark was created with five parallel applications and one fork-join application. This Benchmark was used to distinguish the allocation and the scheduling phases and proposed efficient algorithms with worst-case performance guarantees for both off-line and on-line settings.

During my internship in the University of California Irvine under the supervision of Prof Jean-Luc Gaudiot, I contributed in the development of a dynamic adaptive workload balance (DAWL) scheduling algorithm. It can adaptively adjust workload of the ratio CPU-GPU based on available heterogeneous resources and real-time information. As a case study, a 5 points 2D stencil was used and executed on a runtime. Besides, a profile-based machine-learning estimation model is built to optimize our scheduler algorithm. The estimation model can obtain a customized initial workload on various heterogeneous architectures, as well as how many devices are necessary. With this information, DAWL can reduce its adaptation time.

5.3 Final Considerations and Future Works

The accuracy of a GPU performance model is subject to low-level elements such as instruction pipeline usage and cache hierarchy. GPU performance approaches its peak when the instruction pipeline is saturated but becomes unpredictable when the pipeline is under-utilized. The performance prediction of GPU applications requires a deep knowledge due to different circumstances,

as deep memory hierarchy, threads granularity, programming model and the interaction among these variables. In the context of thousands or millions of threads, the performance is practically unpredictable. The same is valid when throughout in the GPU is underutilized for few threads. Nonetheless, there exist many GPU applications highly used in many scientific areas which scale regularly and simple analytical models, and an adjusting parameter can predict them.

We think it is fair to say that the traditional study of performance prediction of heterogeneous applications from a point of view of the Artificial Intelligence is a very interesting area of research. A contribution of this work was the research of different machine learning techniques which permitted reasonable predictions of running times of GPU Applications.

A future and ongoing research is the comparison of these techniques of performance prediction over scheduling algorithms for heterogeneous resources. We intend to apply these predictions to online scheduling heuristics, to evaluate the gains that these predictions can generate. A recently emerged significant measure of GPU kernels is power consumption. In another future work, we will use machine learning to predict power consumption. Many scheduling strategies for cluster systems are still based on first-come-first-serve (FCFS). Other more sophisticated techniques have been implemented in JMS. For example, backfilling is an optimization that tries to balance the goal of utilization and maintaining FCFS order. It requires that each job also specifies its maximum execution time. Using Machine Learning to predict performance and/or Energy in GPU applications would be very useful, even more, when a scheduling heuristics based on these factors is used.

Bibliography

Agullo *et al.* **(2012)** E. Agullo, G. Bosilca, B. Bramas, C. Castagnede, O. Coulaud, E. Darve, J. Dongarra, M. Faverge, N. Furmento, L. Giraud, X. Lacoste, J. Langou, H. Ltaief, M. Messner, R. Namyst, P. Ramet, T. Takahashi, S. Thibault, S. Tomov and I. Yamazaki. Poster: Matrices over runtime systems at exascale. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1332–1332. doi: 10.1109/SC.Companion.2012.168. Cited in page 89

Amaris *et al.* **(2016)** M. Amaris, R. Y. de Camargo, M. Dyab, A. Goldman and D. Trystram. A comparison of gpu execution time prediction using machine learning and analytical modeling. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pages 326–333. doi: 10.1109/NCA.2016.7778637. Cited in page 5, 75

Amaris *et al.* **(2015)** Marcos Amaris, Daniel Cordeiro, Alfredo Goldman and Raphael Y.de Camargo. A simple bsp-based model to predict execution time in gpu applications. In *High Performance Computing (HiPC), 2015 IEEE 22nd International Conference on*, pages 285–294. Cited in page 4, 37

Amarís *et al.* **(2017)** Marcos Amarís, Clément Mommessin, Giorgio Lucarelli and Denis Trystram. Generic algorithms for scheduling applications on heterogeneous multi-core platforms. *arXiv preprint arXiv:1711.06433*. Cited in page 75, 89, 90

Amaris *et al.* **(2017)** Marcos Amaris, Clément Mommessin, Giorgio Lucarelli and Denis Trystram. Generic algorithms for scheduling applications on hybrid multi-core machines. In *Euro-Par: International Conference on Parallel and Distributed Computing*, pages 220–231. Cited in page 75, 89, 90

Ansel *et al.* **(2014)** Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, Edmonton, AB, Canada. ACM. Cited in page 24

Asanovic *et al.* **(2009)** Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67. ISSN 0001-0782. Cited in page 15

Augonnet *et al.* **(2011)** C. Augonnet, S. Thibault, R. Namyst and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198. ISSN 1532-0626. Cited in page 89

Augonnet *et al.* **(2009)** Cédric Augonnet, Samuel Thibault and Raymond Namyst. Automatic calibration of performance models on heterogeneous multicore architectures. In *European Conference on Parallel Processing*, pages 56–65. Springer. Cited in page 41

Baghsorkhi *et al.* **(2010)** Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp and Wen-mei W. Hwu. An adaptive performance modeling tool for gpu architectures. *SIGPLAN Not.*, 45(5):105–114. ISSN 0362-1340. doi: 10.1145/1837853.1693470. Cited in page 3

Baldini *et al.* **(2014)** I. Baldini, S. J. Fink and E. Altman. Predicting gpu performance from cpu runs using machine learning. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, pages 254–261. doi: 10.1109/SBAC-PAD.2014.30. Cited in page 60

Barnes *et al.* **(2008)** Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski and Martin Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 368–377, New York, NY, USA. ACM. ISBN 978-1-60558-158-3. doi: 10.1145/1375527.1375580. Cited in page 49

Bishop (2006) Christopher M Bishop. Machine learning and pattern recognition. *Information Science and Statistics. Springer, Heidelberg.* Cited in page x, 45

Boyer *et al.* **(2013)** Michael Boyer, Jiayuan Meng and Kalyan Kumaran. Improving gpu performance prediction with data transfer modeling. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1097–1106. IEEE. Cited in page 41

Bruel *et al.* **(2017)** Pedro Bruel, Marcos Amarís and Alfredo Goldman. Autotuning cuda compiler parameters for heterogeneous applications using the opentuner framework. *Concurrency and Computation: Practice and Experience*, 29(22). Cited in page 24, 25

Bukh (1992) Per Nikolaj D Bukh. The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling, 1992. Cited in page 3

Cetin and Yilmaz (2016) Omer Cetin and Guray Yilmaz. Real-time autonomous uav formation flight with collision and obstacle avoidance in unknown environment. *Journal of Intelligent & Robotic Systems*, 84(1):415–433. ISSN 1573-0409. doi: 10.1007/s10846-015-0318-8. URL https://doi.org/10.1007/s10846-015-0318-8. Cited in page 8

Che *et al.* **(2009)** 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)*, IISWC '09, pages 44–54, Washington, DC, USA. IEEE Computer Society. ISBN 978-1-4244-5156-2. Cited in page 4, 14

Che *et al.* **(2010)** Shuai Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, Liang Wang and K. Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11, Atlanta, GA, USA. Cited in page 4

Che *et al.* **(2011)** Shuai Che, Jeremy W. Sheaffer and Kevin Skadron. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 13:1–13:11, New York, NY, USA. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063401. Cited in page 22

Cordeiro *et al.* **(2010)** D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent and F. Wagner. Random graph generation for scheduling simulations. In *ICST (SIMUTools)*. Cited in page 89

Culler *et al.* **(1993)** David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12. ISSN 0362-1340. doi: 10.1145/173284.155333. Cited in page 31

Danalis *et al.* **(2010)** Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju and Jeffrey S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 63–74, New York, NY, USA. ACM. ISBN 978-1-60558-935-0. doi: 10.1145/1735688.1735702. URL http://doi.acm.org/10.1145/1735688.1735702. Cited in page 14

Dasu and Johnson (2003) Tamraparni Dasu and Theodore Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, Inc., New York, NY, USA, 1 ed. ISBN 0471268518. Cited in page 49

de Camargo *et al.* **(2006)** Raphael Y. de Camargo, Andrei Goldchleger, Fabio Kon and Alfredo Goldman. Checkpointing BSP parallel applications on the InteGrade Grid middleware. *Concurrency and Computation: Practice and Experience*, 18(6):567–579. URL http://dx.doi.org/10.1002/cpe.966. Cited in page 30

Dehne *et al.* **(2002)** Frank K. H. A. Dehne, Afonso Ferreira, Edson Cáceres, Siang W. Song and Alessandro Roncato. Efficient Parallel Graph Algorithms for Coarse-Grained Multicomputers and BSP. *Algorithmica*, 33(2):183–200. URL http://dx.doi.org/10.1007/s00453-001-0109-4. Cited in page 4, 31

Du *et al.* **(2012)** Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson and Jack Dongarra. From {CUDA} to opencl: Towards a performance-portable solution for multi-platform {GPU} programming. *Parallel Computing*, 38(8):391 – 407. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/j.parco.2011.10.002. URL http://www.sciencedirect.com/science/article/pii/S0167819111001335. {APPLICATION} {ACCELERATORS} {IN} {HPC}. Cited in page 8

Emeras *et al.* **(2014)** Joseph Emeras, Cristian Ruiz, Jean-Marc Vincent and Olivier Richard. Analysis of the jobs resource utilization on a production system. In Narayan Desai and Walfredo Cirne, editors, *Job Scheduling Strategies for Parallel Processing*, volume 8429 of *Lecture Notes in Computer Science*, pages 1–21. Springer Berlin Heidelberg. ISBN 978-3-662-43778-0. doi: 10.1007/978-3-662-43779-7_1. Cited in page 4, 42

Feitelson *et al.* **(2007)** Dror Feitelson, D TALBY and JP JONES. Standard workload format. *Technical report*. Cited in page 89

Feng and Xiao (2010) Wu-chun Feng and Shucai Xiao. To GPU synchronize or not GPU synchronize? In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 3801–3804. doi: 10.1109/ISCAS.2010.5537722. Cited in page 33

Flynn and Rudd (1996) Michael J Flynn and Kevin W Rudd. Parallel architectures. *ACM Computing Surveys (CSUR)*, 28(1):67–70. Cited in page 27

Fortune and Wyllie (1978) Steven Fortune and James Wyllie. Parallelism in Random Access Machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114–118, New York, NY, USA. ACM. doi: 10.1145/800133.804339. Cited in page 28, 40

Gaj *et al.* **(2002)** Kris Gaj, Tarek A. El-Ghazawi, Nikitas A. Alexandridis, Frederic Vroman, Nguyen Nguyen, Jacek R. Radzikowski, Preeyapong Samipagdi and Suboh A. Suboh. Performance evaluation of selected job management systems. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 260–, Washington, DC, USA. IEEE Computer Society. ISBN 0-7695-1573-8. URL http://dl.acm.org/citation.cfm?id=645610.660898. Cited in page 3

Gaster and Howes (2012) Benedict R. Gaster and Lee Howes. Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck? *Computer*, 45:42–52. ISSN 0018-9162. doi: 10.1109/MC.2012.257. Cited in page 2, 75

Gibbons *et al.* **(1998)** P.B. Gibbons, Y. Matias and V. Ramachandran. The queue-read queue-write asynchronous PRAM model. *Theoretical Computer Science*, 196(1–2):3–29. ISSN 0304-3975. doi: 10.1016/S0304-3975(97)00193-X. URL http://www.sciencedirect.com/science/article/pii/S030439759700193X. Cited in page 29, 40

Goldchleger *et al.* **(2005a)** A. Goldchleger, A. Goldman, U. Hayashida and F. Kon. The implementation of the BSP parallel computing model on the InteGrade Grid middleware. In *Proceedings of the 3rd international workshop on Middleware for grid computing*, pages 1–6. ACM. Cited in page 4

Goldchleger *et al.* **(2005b)** Andrei Goldchleger, Alfredo Goldman, Ulisses Hayashida and Fabio Kon. The implementation of the bsp parallel computing model on the integrade grid middleware. In *Proceedings of the 3rd International Workshop on Middleware for Grid Computing*, MGC '05, pages 1–6, New York, NY, USA. ACM. ISBN 1-59593-269-0. doi: 10.1145/1101499.1101504. URL http://doi.acm.org/10.1145/1101499.1101504. Cited in page 30

Gregg and Hazelwood (2011) Chris Gregg and Kim Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 134–144, Washington, DC, USA. IEEE Computer Society. ISBN 978-1-61284-367-4. doi: 10.1109/ISPASS.2011.5762730. Cited in page 2

Ha *et al.* **(2008)** Phuong Hoai Ha, Philippas Tsigas and Otto J. Anshus. The Synchronization Power of Coalesced Memory Accesses. In Gadi Taubenfeld, editor, *DISC*, volume 5218 of *Lecture Notes in Computer Science*, pages 320–334. Springer. ISBN 978-3-540-87778-3. Cited in page 14

Hayashi *et al.* **(2015)** Akihiro Hayashi, Kazuaki Ishizaki, Gita Koblents and Vivek Sarkar. Machine-learning-based performance heuristics for runtime cpu/gpu selection. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, pages 27–36, New York, NY, USA. ACM. ISBN 978-1-4503-3712-0. doi: 10.1145/2807426.2807429. Cited in page 61

Holl *et al.* **(1998)** Jonathan Holl, Bill McColl, Mark Goudreau *et al.* Standard: BSPlib: the BSP Programming Library. *Parallel Computing*, 24:1947–1980. Cited in page 4, 30

Hong and Kim (2009) Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *ACM SIGARCH Computer Architecture News*, 37(3):152–163. ISSN 0163-5964. doi: 10.1145/1555815.1555775. Cited in page 40, 41

Hou *et al.* **(2008)** Qiming Hou, Kun Zhou and Baining Guo. BSGP: bulk synchronous GPU programming. *ACM Transaction on Graphics*, page 12. Cited in page 2, 4, 30

Huang *et al.* **(2006)** Wei Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron and M. R. Stan. Hotspot: a compact thermal modeling methodology for early-stage vlsi design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):501–513. ISSN 1063-8210. doi: 10.1109/TVLSI.2006.876103. Cited in page 20

IEEE (2008) IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70. doi: 10.1109/IEEESTD.2008.4610935. Cited in page 10, 34

Ipek *et al.* **(2005)** Engin Ipek, Bronis R. de Supinski, Martin Schulz and Sally A. McKee. An approach to performance prediction for parallel applications. In *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, Euro-Par'05, pages 196–205, Berlin, Heidelberg. Springer-Verlag. ISBN 3-540-28700-0, 978-3-540-28700-1. doi: 10.1007/11549468_24. Cited in page 60

Juurlink and Wijshoff (1998) Ben H. H. Juurlink and Harry A. G. Wijshoff. A quantitative comparison of parallel computation models. *ACM Transactions on Computer Systems*, 16(3): 271–318. ISSN 0734-2071. doi: 10.1145/290409.290412. Cited in page 60

Karami *et al.* **(2013)** A. Karami, S. A. Mirsoleimani and F. Khunjush. A statistical performance prediction model for opencl kernels on nvidia gpus. In *The 17th CSI International Symposium on Computer Architecture Digital Systems (CADS 2013)*, pages 15–22. doi: 10.1109/CADS.2013.6714232. Cited in page 60

Kerr *et al.* **(2012)** A. Kerr, E. Anger, G. Hendry and S. Yalamanchili. Eiger: A framework for the automated synthesis of statistical performance models. In *High Performance Computing (HiPC), 2012 19th International Conference on*, pages 1–6. doi: 10.1109/HiPC.2012.6507525. Cited in page 41, 60

Kerr *et al.* **(2009)** Andrew Kerr, Gregory Diamos and Sudhakar Yalamanchili. A characterization and analysis of ptx kernels. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 3–12. IEEE. Cited in page 60

Khronos (2013) Khronos. Opencl - the open standard for parallel programming of heterogeneous systems @ONLINE, Janeiro 2013. URL http://www.khronos.org/opencl/. Cited in page 8

Kirtzic (2012) J. Steven Kirtzic. *A Parallel Algorithm Design Model for the GPU Architecture*. Tese de Doutorado, University of Texas at Dallas, Richardson, TX, USA. AAI3547670. Cited in page 30

Konstantinidis and Cotronis (2017) Elias Konstantinidis and Yiannis Cotronis. A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling. *Journal of Parallel and Distributed Computing*, 107:37–56. Cited in page 41

Kothapalli *et al.* **(2009)** K. Kothapalli, R. Mukherjee, M.S. Rehman, S. Patidar, P. J. Narayanan and K. Srinathan. A performance prediction model for the CUDA GPGPU platform. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 463–472. doi: 10.1109/HIPC.2009.5433179. Cited in page 40

Li *et al.* **(2009)** Jiangtian Li, Xiaosong Ma, K. Singh, M. Schulz, B.R. de Supinski and S.A. McKee. Machine learning based online performance prediction for runtime parallelization and task scheduling. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 89–100. doi: 10.1109/ISPASS.2009.4919641. Cited in page 60

Lindholm *et al.* **(2008)** Erik Lindholm, John Nickolls, Stuart Oberman and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55. ISSN 0272-1732. doi: 10.1109/MM.2008.31. URL http://dx.doi.org/10.1109/MM.2008.31. Cited in page 10

Madougou *et al.* **(2016)** Souley Madougou, Ana Varbanescu, Cees de Laat and Rob van Nieuwpoort. The landscape of {GPGPU} performance modeling tools. *Parallel Computing*, 56:18–33. ISSN 0167-8191. doi: 10.1016/j.parco.2016.04.002. URL http://www.sciencedirect.com/science/article/pii/S0167819116300114. Cited in page 60

Matsunaga and Fortes (2010) A. Matsunaga and J. Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 495–504. doi: 10.1109/CCGRID.2010.98. Cited in page 60

Mei *et al.* **(2014)** Xinxin Mei, Kaiyong Zhao, Chengjian Liu and Xiaowen Chu. Benchmarking the memory hierarchy of modern gpus. In Ching-Hsien Hsu, Xuanhua Shi and Valentina

Salapura, editors, *Network and Parallel Computing*, volume 8707 of *Lecture Notes in Computer Science*, pages 144–156. Springer Berlin Heidelberg. ISBN 978-3-662-44916-5. doi: 10.1007/978-3-662-44917-2_13. Cited in page 32, 35

Meng *et al.* **(2011)** Jiayuan Meng, Vitali A Morozov, Kalyan Kumaran, Venkatram Vishwanath and Thomas D Uram. Grophecy: Gpu performance projection from cpu code skeletons. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–11. IEEE. Cited in page 40, 41

Meswani *et al.* **(2012)** M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden and S. Poole. Modeling and predicting performance of high performance computing applications on hardware accelerators. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1828–1837. doi: 10.1109/IPDPSW.2012.226. Cited in page 60

Mittal and Vetter (2014) Sparsh Mittal and Jeffrey S. Vetter. A Survey of Methods for Analyzing and Improving GPU Energy Efficiency. *ACM Comput. Surv.*, 47(2):19:1–19:23. ISSN 0360-0300. doi: 10.1145/2636342. Cited in page 9

Myers *et al.* **(2010)** Jerome L Myers, Arnold Well and Robert Frederick Lorch. *Research design and statistical analysis*. Routledge. Cited in page 46

Nagasaka *et al.* **(2010)** Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo and Satoshi Matsuoka. Statistical power modeling of gpu kernels using performance counters. In *Green Computing Conference, 2010 International*, pages 115–122. IEEE. Cited in page 4

NVIDIA (2018) NVIDIA. *CUDA C: Programming Guide, Version 9.0*, January 2018. URL http://docs.nvidia.com/cuda/index.html. Cited in page ix, 2, 8, 13, 15, 35

NVIDIA Corporation (2014) NVIDIA Corporation. *CUDA C Best Practices Guide*, August 2014. Cited in page 32, 33, 38

Ozisikyilmaz *et al.* **(2008)** B. Ozisikyilmaz, G. Memik and A. Choudhary. Machine learning models to predict performance of computer system design alternatives. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 495–502. doi: 10.1109/ICPP.2008.36. Cited in page 60

Power *et al.* **(2014)** Jason Power, Joel Hestness, Marc Orr, Mark Hill and David Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *Computer Architecture Letters*, 13(1). ISSN 1556-6056. doi: 10.1109/LCA.2014.2299539. URL http://gem5-gpu.cs.wisc.edu. Cited in page 4

Savadi and Deldari (2014) Abdorreza Savadi and Hossein Deldari. Measurement of the latency parameters of the Multi-BSP model: a multicore benchmarking approach. *The Journal of Supercomputing*, 67(2):565–584. ISSN 0920-8542. doi: 10.1007/s11227-013-1018-4. Cited in page 31

Silva *et al.* **(2014)** Cleber Silva, Siang Song and Raphael Camargo. A parallel maximum subarray algorithm on GPUs. In *5th Workshop on Applications for Multi-Core Architectures (WAMCA 2014). IEEE Int. Symp. on Computer Architecture and High Performance Computing Workshops*, pages 12–17, Paris. Cited in page 15, 18, 33

Singh *et al.* **(2007)** Karan Singh, Engin İpek, Sally A. McKee, Bronis R. de Supinski, Martin Schulz and Rich Caruana. Predicting parallel application performance via machine learning approaches: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(17):2219–2235. ISSN 1532-0626. doi: 10.1002/cpe.v19:17. Cited in page 60

Skillicorn and Talia (1998) David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169. ISSN 0360-0300. doi: 10.1145/280277.280278. Cited in page 3, 28

Spafford and Vetter (2012) Kyle L. Spafford and Jeffrey S. Vetter. Aspen: A domain specific language for performance modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 84:1–84:11, Los Alamitos, CA, USA. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL http://dl.acm.org/citation.cfm?id=2388996.2389110. Cited in page 41

Stratton *et al.* **(2012)** John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng D. Liu and Wen-mei W. Hwu. Parboil: A Revised Benchmark Suite for Scientic and Commercial Throughput Computing. Relatório Técnico IMPACT-12-01, University of Illinois at Urbana-Champaign. Cited in page 14

Suettlerlein *et al.* **(2013)** Joshua Suettlerlein, Stéphane Zuckerman and Guang R. Gao. An implementation of the codelet model. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 633–644, Berlin, Heidelberg. Springer-Verlag. ISBN 978-3-642-40046-9. doi: 10.1007/978-3-642-40047-6_63. URL http://dx.doi.org/10.1007/978-3-642-40047-6_63. Cited in page 75

Szafaryn *et al.* **(2009)** Lukasz G Szafaryn, Kevin Skadron and Jeffrey J Saucerman. Experiences accelerating matlab systems biology applications. In *Proceedings of the Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits*, pages 1–4. Cited in page 20

TOP-500-Supercomputer (2017) TOP-500-Supercomputer. [Web site http://www.top500.org] Visited May 2016, 2017. URL http://www.top500.org. Cited in page 1

Valiant (1990) Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111. ISSN 0001-0782. doi: 10.1145/79173.79181. Cited in page 4, 29

Valiant (2011) Leslie G. Valiant. A bridging model for multi-core computing. *Journal of Computer and System Sciences*, 77(1):154–166. ISSN 0022-0000. doi: 10.1016/j.jcss.2010.06.012. Cited in page 4, 30

Verber (2014) Domen Verber. The future of supercomputers and high-performance computing. *Handbook of Research on Interactive Information Quality in Expanding Social Network Communications*, page 152. Cited in page 1

Williams *et al.* **(2009)** Samuel Williams, Andrew Waterman and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76. Cited in page 41

Wong *et al.* **(2010)** H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246. doi: 10.1109/ISPASS.2010.5452013. Cited in page 32

Wu *et al.* **(2013)** Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang and Xipeng Shen. Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-coalesced Memory Accesses on GPU. *SIGPLAN Not.*, 48(8):57–68. ISSN 0362-1340. doi: 10.1145/2517327.2442523. Cited in page 22, 33, 37

Wu *et al.* **(2015)** G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena and D. Chiou. Gpgpu performance and power estimation using machine learning. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 564–576. doi: 10.1109/HPCA.2015.7056063. Cited in page 60

Zhang *et al.* **(2011)** Y. Zhang, Y. Hu, B. Li and L. Peng. Performance and power analysis of ati gpu: A statistical approach. In *Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on*, pages 149–158. doi: 10.1109/NAS.2011.51. Cited in page 61

Zhang and Owens (2011) Yao Zhang and J.D. Owens. A quantitative performance analysis model for GPU architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382–393. doi: 10.1109/HPCA.2011.5749745. Cited in page 40

Zhong *et al.* **(2012)** Z. Zhong, V. Rychkov and A. Lastovetsky. Data Partitioning on Heterogeneous Multicore and Multi-GPU Systems Using Functional Performance Models of Data-Parallel Applications. In *2012 IEEE International Conference on Cluster Computing (Cluster 2012)*. Cited in page 2, 75

Appendix A.

International Collaborations Related to This Thesis

During my Doctorate, I did two different international internships, in both we obtained great results and research partners with of very high international quality. Follow a short description of both internships. Different international internships, obtaining relevant research results and starting partnership with distinguished international collaborators.

- **Internship at the University of California Irvine**: This internship was of three (3) months. In this internship, I was a visitor in the PArallel Systems & Computer Architecture Lab (PASCAL), in the Electrical Engineering & Computer Science Department at University of California, Irvine, under the direction of Dr. Jean-Luc Gaudiot. PASCAL is a group for the investigation of parallel and distributed computing. The group pursues experimental research in areas such as functional programming, compiler techniques, and computer architectures. During this internship, I participated in different works. One of those work resulted in a paper which was submitted recently in an international conference.

  During this short-term internship at UCI, we worked on a profile-based estimation model with a Dynamic Adaptive Workload balance algorithm (called DAWL for this work) to help effectively utilize CPU and GPU resources in different heterogeneous platforms. Our approach followed two phases: first, in an adaptive way, the algorithm decides what is the best workload balance based on the application's information, obtained in real time. Second, establish a profile-based machine learning estimation model to obtain a suitable initial workload allocation to resources. Then, our dynamic adaptive workload balance algorithm will run with this information (initial workload on GPU and CPUs, number of resources) and adjust workload based on the real-time system information. Part of this work is shown in Section A.1. Since this work is still in revision, we provided the original document that we have submitted. This document is presented in Section A.1.2

- **Internship at Grenoble Informatics Laboratory (LIG - France)**: This internship was of twelve (12) months. LIG research is conducted by 24 teams organized into different high-level areas. I was part of one of DATAMOVE (Data Aware Large Scale Computing) team, which is a joint research project involving INRIA and CNRS, Grenoble INP, University of Grenoble Alpes, in France. This internship was supervised by Prof. Dr. Denis Trystram. During this internship, we implemented part of my doctoral thesis that was a fair comparison

between different machine learning approaches and our simple BSP-based model to predict the execution time of CUDA kernels functions (Amaris *et al.*, 2016).

I also collaborated in the analysis and design of efficient algorithms to schedule applications represented by a precedence task graph on heterogeneous computing resources. One of the contributions was studying the problem with precedence on both off-line and on-line settings and designing algorithms that can be implemented in actual systems (Amarís *et al.*, 2017; Amaris *et al.*, 2017). In this collaboration, we created a benchmark of task-based applications executed over heterogeneous resources, CPUs and GPUs. This benchmark is presented in Section A.2 as well some results of the scheduling algorithms when using this benchmark.

A.1 Performance Predictions on Hybrid CPU-GPU Applications

Most works dealing with accelerators, and in particular Graphics Processing Units (GPUs), tend to follow one of two options (1) Fully offload the most compute-intensive parts of a given application on a GPU, or (2) partition the compute-intensive steps between "GPU-friendly" portions that will run solely on the GPU, and "CPU-friendly" ones. *Co-running* friendly applications, *i.e.*, which can run on both GPU and CPU concurrently, tend to have low memory/communication bandwidth requirements, compared to applications which run the workload on only one part of the system.

The implementation of an algorithm for parallel execution on both CPUs and GPU does not guarantee it will execute with the same efficiency on both architectures. In particular, data-parallel algorithms dealing with large blocks of data, *i.e.*, featuring intense array arithmetic and regular (array-based) data structures, can greatly benefit when implemented exclusively to run on a GPU (Gaster and Howes, 2012; Zhong *et al.*, 2012).

We chose to extend DARTS (Delaware Asynchronous RunTime System, called as EDRT Extended DARTS Runtime for this work) (Suettlerlein *et al.*, 2013) as our test runtime system. The original version of EDRT mainly focuses on homogeneous many-core systems. However, this collaboration targets workload balancing between heterogeneous resources. To validate and verify DAWL, we extended EDRT to support heterogeneous architectures. If an application requires the use of a barrier between the host and the accelerator device in a heterogeneous system, a significant effort must be done for load-balancing to achieve high-performance.

Here we present a collaboration about a profile-based estimation model with a dynamic adaptive workload balance algorithm (DAWL algorithm) to help utilize effectively heterogeneous platforms with CPUs and GPUs. This algorithm is explained in the attached paper, see Section A.1.2 in the Appendix.

We have used a 5-point 2D stencil kernel as a case study. When the stencil problem size is less than the available device memory, $r \gg 1$, all computation is carried on the device. Where r is the ratio of the execution time in both devices, CPUs and GPU, i.e. $r = CPU\_naive/GPU\_naive$. When it is larger than the available device memory, an algorithm (DAWL) chooses how the workload will be distributed in a co-running way. Then, CPU and GPU processing elements can execute the workload concurrently. In addition, in the specific case of our stencil kernel, each time step relies on the results obtained in the previous one. Thus, such a computation requires both the host and the device to synchronize regularly during the lifetime of the application.

To solve this problem, we proposed a profile-based machine-learning estimation model to optimize the algorithm DAWL. A black-box machine learning method, *i.e.*, an automatic model without

any user intervention, is used.

First, the static initial workload on GPU, which should be smaller than the available GPU memory; second, we can obtain the initial workload from our estimation model. It is based on the compute node's hardware configuration, such as the number of CPU cores, last-level of cache (LLC), the GPU type *etc.* The DAWL scheduling algorithm is meant to minimize workload imbalance between heterogeneous processing elements. To overcome the problem of load imbalance for heterogeneous systems, DAWL dynamically adjusts the workload distribution on different computing resources based on real time information.

While DAWL can dynamically adjust the workload according to real-time information, an unsuitable initial workload may drag down the performance when the problem size is relatively small. To solve this problem, we also developed a profile-based machine-learning estimation model to optimize DAWL. A black-box machine learning method, *i.e.*, an automatic model without any user intervention, is used.

In our machine learning black-box estimation model, different algorithms were tested, including linear regression, logistic regression and random forests. The best match and less computationally complex algorithm was chosen to run our statistical model. Table A.1 shows the hardware architecture informationa and the parameters to our black-box estimation model about the hardware. This information is based on the compute node's hardware configuration, such as the number of CPU cores, last-level of cache (LLC), the GPU type *etc.*. Application parameters were size of the problem, GPU wokload, CPU Workload, ratio, among other. To evaluate how well the model fits the data, the statistical measure, $R_{squared}$, is widely used. It is also known as the coefficient of determination. $R_{squared}$ is defined as the percentage of the response variation that is explained by a linear model.

A.1.1 Method and Results

Several heterogeneous nodes with multicores processors were selected for our experiments. They are representative of the current many core-based hardware trends, see Table A.1.

Param. / Machines	CPU Parameters					GPU Parameters					PCIe
	Threads	**Clock**	**# Socket**	**L3 Cache**	**CPU Mem**	**# SM**	**Clock**	**L2 Cache**	**GPU Mem**	**# CE**	
1. Fatnode (K20)	32	2.6 GHz	2	20 MB	64 GB (2x32)	13	0.71 GHz	1.25 MB	4.8 GB	2	6.1 GB/s
2. Super Micro (K20)	40	3 GHz	2	25 MB	256 GB (2x128)	13	0.71 GHz	1.25 MB	4.8 GB	2	6.1 GB/s
3. CCSL (Valinhos) (k40)	8	3.4 GHz	1	8 MB	16 GB	15	0.75 GHz	1.5 MB	12 GB	2	10.3 GB/s
4. Debian (Titan)	12	3.4 GHz	1	12 MB	31 GB	14	0.88 GHz	1.5 MB	6 GB	2	11.5 GB/s
5. Hive (GTX 680)	12	3.2 GHz	1	12 MB	30 GB	8	1.06 GHz	0.5 MB	2 GB	1	5.6 GB/s

Table A.1: *Experiment Hardware Environment*

To build up a profile-based Machine-learning estimation model for heterogeneous architecture, collecting data is the first step. Our target application is co-running friendly (Hybrid CPU-GPU) application corresponding to stencil code which problem sizes are larger than 17000*17000, global memory of all GPUs were limited to up 2GB available memory. We have varied the problem size, number of iterations, initial GPU workload, initial CPU workload and CPU core number; with the next values: starting from 17000 to 35000 with step of 2000; 1, 2, 8; 1000, 2000, 4000, 8000; $GPU\_workload * (0.5, 1.0, 1.5, 2.0)$, 4, 8, 16, ..., largest CPU core number on each server; respectively. Sample sets are different for each machine, due to memory size differences. The total number of collect samples on the 5 servers was 6480.

The second step is to obtain the best-matched model. Multiple regression algorithms (linear regression, logistic regression, random forests) with multiple Meta-functions and Polynomial func-

tions, were tested. The best fit model was the simple linear regressions (Meta-function) whose *R_squared* are between 93% and 94%. Figure A.1 shows a Quantile-Quantile plot of the trained model with the train set, red line show the built model and the point each of of the train samples.

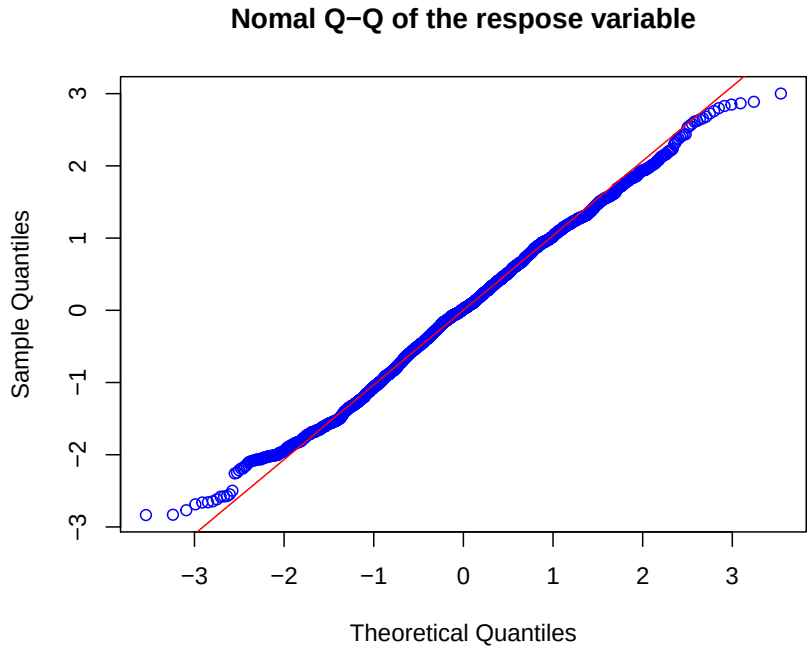

Figure A.1: *Quantiles of the model with all the samples in this experiment*

To measure the progress of the learning algorithm we have used the Mean Absolute Percentage Error (MAPE). With this error, we have analyzed the reliability of our approaches. Table A.2 shows Mean Average Percentage Error (MAPE) of the linear model for each machine.

Table A.2: *Mean Absolute Percentage Error (MAPE)*

Machines	supermicro	fatnode	debian	hive	ccsl
MAPE	7.41%	6.43%	1.68%	3.49%	3.45%

As a second goal in this statistical estimation model, we wanted to know which parameters had more impact on the construction of the model. For this reason, we used the absolute value of the *t-statistic* for each model parameter and calculate the importance of each parameter in the model. We made this experiment with 17 software and hardware parameters, the result was that 6 parameters are enough to predict the performance of the developed Stencil application, these were: NumOfSocket, CPU_Clock, CPU_Cores, Total_Workload, Initial_GPU_Workload, CPU.GPU_Workload_Ratio. Except for 3 workload related parameters, 3 other parameters are all related to the CPUs. In our case, GPUs in the 5 servers are pretty similar, that is the reason why GPUs parameters do not have relevance in our model. The situation will change if more GPUs type are involved in the experiments. GPUs and PCIe information was hidden in parameter Workload parameters. Finally, the best initial workload on GPU and CPU that we have called r, and the best maximum number of cores on each server can be obtained.

Combining our profile-based estimation model and DAWL, we obtained the results shown in Figure A.2. Compared to `EDRT-CPU`, which always uses all the CPUs, this new implementation of

`EDRT-DAWL` uses at most half of the CPUs (depending on the hardware). Our scheduler can obtain up to 6× speedup compared to the sequential version, 1.6× speedup compared to the multiple core version, and 4.8× speedup compared to the pure GPU version.

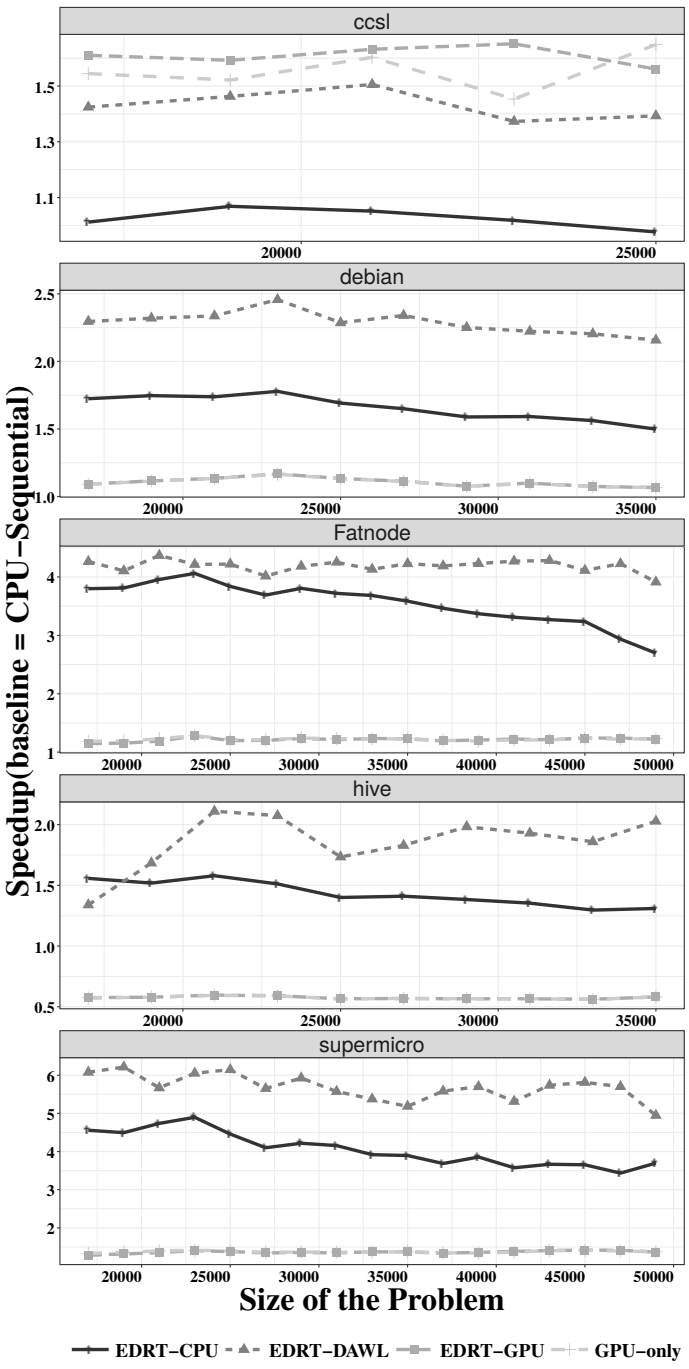

Figure A.2: *Speedup of the different Stencil versions of matrices larger than 17K*

A.1.2 Article: Profile-based Dynamic Adaptive Workload Balance on Heterogeneous Architectures

Profile-based Dynamic Adaptive Workload Balance on Heterogeneous Architectures*

Author names omitted for blind review

ABSTRACT

While High Performance Computing (HPC) systems are increasingly based on heterogeneous cores, the effectiveness of these heterogeneous systems depends on how well the scheduler can allocate workload onto appropriate computing devices and how communications or computations can be overlapped. With different types of resources integrated into one system, GPUs and CPUs, the complexity of the scheduler correspondingly increases. Moreover, for applications with varying problem sizes on different heterogeneous resources, the optimal scheduling approach may vary accordingly.

This paper presents a profile-based dynamic adaptive workload balance scheduling approach to efficiently utilize heterogeneous systems. Our approach follows two phases: first, we build up a dynamic adaptive workload balance algorithm. It can adaptively adjust a workload based on available heterogeneous resources and real time situation. Second, we establish a profile-based device-specific estimation model to optimize the scheduling algorithm. Our scheduling approach is tested on a standard 2D 5-point stencil computation in our Extended EDRT platform[1]. Experimental results show that when the problem size allows the working set to remain within the available GPU memory, our scheduler can obtain speedups up to 8.75× compared to the sequential version, 7× compared to the pure multi-CPU version, and stays on-par with pure GPU version. When the problem size causes the working to no longer fit in the available GPU memory, our scheduler can obtain up to 6× compared to the sequential version, 1.6× compared to the pure multi-CPU version, and 4.8× compared to the pure GPU version.

CCS CONCEPTS

• **General and reference → Performance**; • **Theory of computation → Streaming models**; • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering → Synchronization**; **Concurrent programming structures**;

KEYWORDS

Heterogeneous Computing, CUDA, Adaptive modeling, Load balancing

*Produces the permission block, and copyright information

[1]We replaced the real name of the runtime for double-blind review.

ICS 2018, June 2018, Beijing, China

ACM ISBN 978-1-4503-nnnn-n.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Author names omitted for blind review. 2018. Profile-based Dynamic Adaptive Workload Balance on Heterogeneous Architectures. In *Proceedings of ACM International Conference on Supercomputing (ICS 2018).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Nowadays, most High-Performance Computing (HPC) platforms feature heterogeneous hardware resources (CPUs, GPUs, FPGAs, storage, etc.). For instance, the number of platforms of the Top500 equipped with accelerators has significantly increased during the last years [12]. In the future it is expected that the nodes of such platforms' heterogeneity will increase even more: they will be composed of fast computing nodes, hybrid computing nodes mixing general purpose units with accelerators, I/O nodes, nodes specialized in data analytics, etc. The interconnect of a huge number of such nodes will also lead to more heterogeneity. Resorting to heterogeneous platforms can lead to better performance and power consumption through the use of more appropriate resources according to the computations to perform. However, it has a cost in terms of code development and more complex resource management.

GPU boards are integrated with multi-core chips on a single compute node to boost the overall computational processing power. When scientific applications rely on large amounts of data, this poses some restrictions on how to offload some of the computation to the accelerator device, *e.g.*, a GPU, as their memory capacity is limited, and data transfers incur very limited latencies and bandwidth [11]. CUDA is a high-level platform for developing GPU applications, comprising a compiler, a library, and a runtime system. CUDA applications are organized in kernels, which are functions executed on GPUs.

The main task of a load-balancing mechanism in a heterogeneous system is to devise the best data division among the general-purpose CPUs and the accelerator—in our case, the GPU. Simple heuristic load division, such as evenly dispatch work to all the CPU cores and GPU, may result in worse performance and consume more power.

This work stems from the following observations: with a few exceptions (see Section 5), most work dealing with so-called accelerators, and in particular Graphics Processing Units (GPUs), tend to follow one of two options (1) Fully offload the most compute-intensive parts of a given application on a GPU, or (2) partition the compute-intensive steps between "GPU-friendly" portions that will run solely on the GPU, and "CPU-friendly" ones. In this paper, we attempt to answer the following question: Do we *really* need to use all the computation resources simultaneously? When parts of an application is *co-running*, *i.e.*, when it runs on both CPU and GPU cores, is the resulting performance always higher than running, say, fully on either the GPU or CPUs? As will be shown in the remainder of the paper, there is no clear-cut answer, and it all depends on a wealth of parameters, both hardware and software. Indeed, the question will yield a totally different answer when hardware

architectures, schedulers, applications or even just the problem size of a specific application change.

We consider a compute node is made of *heterogeneous hardware* when it is comprised of general-purpose CPUs, as well as some kind of *accelerator* hardware, such as GPUs or reconfigurable fabric (say, one or more FPGAs). For the purpose of this work, we will focus on GPU-based heterogeneous platforms. We consider two main types of heterogeneous architectures: integrated CPU/GPU architectures and multi-CPUs with discrete GPUs.

Co-running friendly applications, *i.e.*, which can run on both GPU and CPU concurrently, tend to have low memory/communication bandwidth requirements, compared to applications which run the workload on only one part of the system. Hence, the computation-memory ratio and computation patterns can help identify the suitability of the workload to a resource. For example, if the application is characterized such that the communication time (data transfers) between CPUs and GPUs is far higher than the computation time of a given workload, and if there is no overlap between computation and communication, this application will belong to the "co-running unfriendly class." However, the categorization may change when the application is developed in different hardware architectures or even in same hardware architecture with a changing dataset.

Furthermore, we must evaluate if the performance of a friendly co-running application using all available computing resources, *i.e.*, CPUs and GPUs, results in better performance than, say, running on a lower number of compute cores (CPUs and/or GPUs). Many researchers such as Van Craeynest *et al.* [22], J. Power *et al.* [16], V. García *et al.* [7], F. Zhang *et al.* [24], Q. Chen *et al.* [5], or C. Yang *et al.* [23] proved that heterogeneous system architectures are significantly impacted by various parameters, such as number and type of cores, the organization or topology of cores, the memory hierarchy, bandwidth and memory access pattern, the communication congestion and synchronization mechanism, as well as other hardware or software factors.

We propose a profile-based estimation model augmented with a dynamic adaptive workload balance algorithm to help effectively utilize CPU and GPU resources in different heterogeneous platforms. Our approach follows two phases: first, adaptively decide what is the best workload balance based on the application's information, obtained in real time. Second, establish a profile-based machine learning estimation model to obtain a suitable initial workload allocation to resources. Then, our dynamic adaptive workload balance algorithm will run with these information (initial workload on GPU and CPUs, number of resources) and adjust workload based on the real-time system information.

The structure of this document is as follows. Section 2 reviews the main concepts of this work; Section 3 describes our methodology; Section 4 describes our main experimental results; in Section 5 we review the literature about the area and present various related papers. Finally, Section 6 concludes this work and presents the planned future work.

2 BACKGROUND

2.1 Runtime System

We chose to extend EDRT [1, 2, 19] as our test runtime system. It is an implementation of the Codelet Model and the Codelet Abstract Machine (CAM) on which it relies. The CAM models a many-core architecture with two types of cores: synchronization units (SUs), which perform resource management and scheduling, and computation units (CUs), which carry out the computation. A CAM is an extensible, scalable and hierarchical parallel machine model. One SU, one or more CUs, and some local memory make up a *cluster*. Clusters, alongside some memory, can be grouped together to form a chip. One or more chips, coupled with some memory, consist of a node. Finally, a full CAM is composed of at least one node. The communication between and within components of each level of the hierarchy is done by the interconnection network. EDRT is based on the Codelet Model, proposed by paper [26]. It is a fine-grain event-driven program execution model. The Codelet Model is a hybrid between the dataflow [6] and von Neumann models of computation. A codelet *fires* (*i.e.*, is scheduled for execution) when all its dependencies (data and resources) are met.

2.2 Heterogeneous Computing

Heterogeneous computing is about efficiently using all processors in the system, including CPUs and GPUs. In our heterogeneous systems, GPUs have been connected to a host machine by a high bandwidth interconnect such as PCI Express (PCIe). Host and device have different memory address spaces, and data must be copied back and forth between the two memory pools explicitly by the data transfer functions. There are two main aspects to consider when attempting to run an application on both CPUs and GPUs concurrently, namely the communication costs incurred by CPU-GPU communications, and how to efficiently overlap computation and data transfers to reduce said cost.

2.2.1 Heterogeneous Hardware Communication Costs. On integrated CPU/GPU architectures, F. Zhang *et al.* [24] suggested that the architecture differences between CPUs and GPUs, as well as limited shared memory bandwidth, are two main factors affecting co-running performance. On SMP systems connected to GPU architectures, beside architectural differences, communications between CPUs and GPUs are another important factor. GPUs and CPUs are bridged by a PCIe bus. Data are copied from the CPU's host memory to PCIe memory first, and are then transferred to the GPU's global memory. The PCIe bandwidth is always a crucial performance bottleneck to be improved. However, the bandwidth between the CPU host memory and the GPU memory is far less than PCIe bandwidth. Nvidia [14] provides ways to pin memory, also called paged-locked memory. A pinned page can be remapped to a PCIe buffer to eliminate data transfer between the host memory and the PCIe buffer. However, consuming too much page-locked memory will reduces overall system performance.

Servers with multi-cores and multi-accelerator devices are primarily connected by PCI-Express (PCIe) bus. Congestion control mechanisms have a significant impact on communications. Moreover, the PCIe congestion behavior varies significantly depending on the conflicts created by communication. Martinasso *et al.* [13] have explored the impact of PCIe topology, which is one major parameter influencing the available bandwidth. They developed a congestion-aware performance model for PCIe communication. They found that bandwidth distribution behavior is sensitive to the

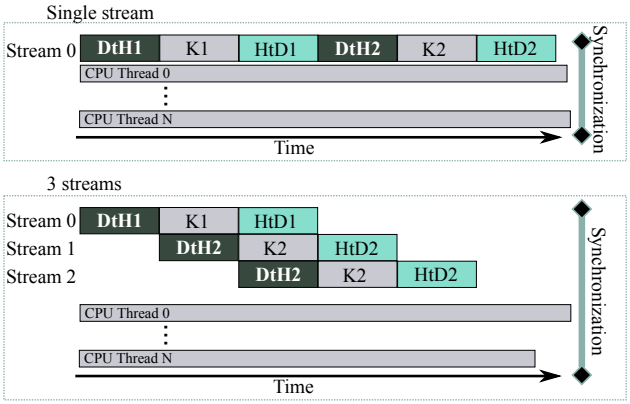

Figure 1: Concurrent Streams overlapping data transfers

transfer message size. PCIe congestion can be hidden if the overlapping communications transfer very small message sizes. However, when the message size reaches some limit, congestion will significantly reduce the theoretical transfer bandwidth efficiency.

2.2.2 Concurrent Streams on GPU. CUDA's programming model provides constructs based on streaming which are capable to schedule multiple kernels concurrently. One CUDA stream can encapsulate multiple kernels, and they have to be scheduled so they strictly follow a particular order. However, kernels from multiple streams can be scheduled to run concurrently. Operations in different streams can be interleaved and overlapped, which can be used to hide data transfers between host and device.

A main optimization of the developed application was to overlap data transfers across the PCIe bus [13]. This is only possible using CUDA streams and pinned memory in the host. Using pinned host memory enables asynchronous memory copies, lowers latency, and increases bandwidth. This way, streams can run concurrently. However, this goal is constrained by the number of kernel engines and copy engines exposed by GPUs, and synchronization must be explicit in the stream kernels.

There are GPUs with only a single copy engine and a single kernel engine. In this case, data transfer overlapping is not possible. Different streams may execute their commands concurrently or out of order with respect to each other. When an asynchronous CUDA stream is executed without specifying a stream, the CUDA runtime uses the default stream 0; but when a set of independent streams are executed with different ID numbers, these streams avoid serialization, achieving concurrency between kernels and data copies.

Figure 1 explains the streaming model which we used to improve the performance of our target GPU application. In this figure we compare the sequential computation of two different kernels with their respective data transfers: one single stream vs. 3 different kernels with their respective data transfers using 3 streams. The second method is only possible in GPUs with two copy engines, one for host-to-device transfers and another for device-to-host transfers.

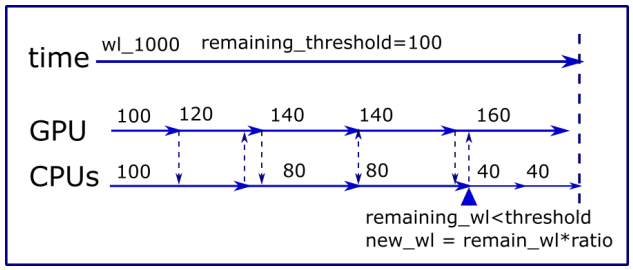

Figure 2: DAWL example

3 METHODOLOGY

In this section, we will describe our approach consisting of two parts: we propose a dynamic adaptive workload algorithm (DAWL) as a first step, and a profile-based machine-learning estimation model as an optimization over it.

3.1 Algorithm: Dynamic Adaptive Workload between Heterogeneous resources

While GPGPU (General-propose computing on GPUs) have evidenced outstanding results in many scientific areas, tools to further capitalize this parallel devices and more refined implementations are still to emerge [14]. Researchers in this area can create applications with a concept of parallelism that are able to run in both CPU and GPU architectures. However, the implementation of an algorithm for parallel execution on both CPUs and GPU does not guarantee it will execute with the same efficiency on both architectures. In particular, data-parallel algorithms dealing with large blocks of data, *i.e.*, featuring intense array arithmetic and regular (array-based) data structures, can greatly benefit when implemented exclusively to run on a GPU [9, 25].

If an application requires the use of a barrier between the host and the accelerator device in a heterogeneous system, a significant effort must be put into load-balancing to achieve high-performance. As a result, the workload behavior for both the host and the device(s) must be carefully analyzed. Among the various parameters that must be gathered, we can cite the number of general purpose and device processing elements, the size and topology of the host's cache hierarchy, the devices's main memory capacity, the PCIe bandwidth, the maximum number of concurrent transfers that can occur between the host and the devices, as well as the problem size to be distributed over both the host and its device(s).

$$GPU_{naive} = \frac{Transfer_{(H \rightarrow D)} + Compute_{(D)} + Transfer_{(D \rightarrow H)}}{NumThreads_{(D)}} \quad (1)$$

$$CPU_{naive} = \frac{Compute_{(H)}}{NumThreads_{(H)}} \quad (2)$$

$$r = \frac{CPU\_naive}{GPU\_naive} \quad (3)$$

Based on the hardware configuration information of the PCIe, GPU, *etc.*, we can roughly obtain an initial idea by using Equations 1, 2, and 3.

Algorithm 1: Dynamic Adaptive workload balance between Heterogeneous Resources

```
1  Function main(Hardware_Info,WL(problem_size),GPU_WL,CPU_WL,Limit_WL,
2                GPU_Change_ratio,CPU_Change_ratio):
3      step1: decision = hardware_choose(Hardware_Info,WL)
4      step2: if decision = Co_running then
5          Co_running_WL_balance(Hardware_Info,total_WL, GPU_WL,CPU_-
             WL,Limit_WL,GPU_Change_ratio,CPU_Change_ratio)
6      else if decision = CPU then
7          run_CPUs(Hardware_Info,WL)
8      else
9          run_GPU(Hardware_Info,WL)
10     end
11 Function hardware_choose(Hardware_Info,WL):
12     if WL < GPU_memory_available_size then
13         if r»1 then
14             decision = GPU
15         else
16             decision = CPU
17         end
18     else
19         decision = Co_running
20     end
21 Function
     Run_Func(Hardware_Info,type,WL,Remaining_WL,Limit_WL,Change_ratio):
22     if type = CPU then
23         CPU_Func(Hardware_Info,WL)
24     else
25         GPU_Func(Hardware_Info,WL)
26     end
27     if Remaining_WL<Limit_WL then
28         WL = Remaining_WL;
29     else
30         (faster,Ratio) = check(CPU_status,GPU_status)
31         if faster = CPU then
32             TWL = WL * (1-change_ratio)
33         else
34             TWL = WL * (1+change_ratio)
35         end
36         if Remaining_WL<TWL then
37             WL = Remaining_WL * Ratio
38         else
39             WL = TWL
40         end
41     end
42     Remaining_WL -= WL
43     sync_GPU_CPU(Remaining_WL,MEM)
44     renew(WL_min,WL_max)
45     Run_Func(Hardware_Info,type,WL,Remaining_WL,Limit_WL)
46 Function SYNC_Rebalance_Func(Hardware_Info,
     CPU_WL_Info,GPU_WL_Info):
47     CPU_WL = Rebalance(CPU_WL_Info)
48     GPU_WL = Rebalance(GPU_WL_Info)
49     IsChange = check_Hardware(Hardware_Info)
50     if IsChange=true then
51         CPU_WL = CPU_Update(CPU_WL,Hardware_Info)
52         GPU_WL = GPU_Update(GPU_WL,Hardware_Info)
53     end
```

For the remainder of this paper, we will use a 5-point 2D stencil kernel as a case study. We will leverage it to explain our Dynamic Adaptive Work-Load (DAWL) balance scheduling algorithm below, and outlined in algorithms 1 and 2.

When the stencil problem size is less than the available device memory, $r \gg 1$. Hence, all computation will be carried on the device. When it is larger than the available device memory, Algorithm 1's *hardware_choose* function will choose how the workload will be

Algorithm 2: Asynchronous Parallel function and load balance

```
1  Function Co_running_WL_balance(Hardware_Info,total_WL,
     GPU_WL,CPU_WL,Limit_WL,GPU_Change_ratio,CPU_Change_ratio):
2      GPU_initialize(Hardware_Info,GPU_WL)
3      CPU_initialize(Hardware_Info,CPU_WL)
4      Remaining_WL = total_WL - GPU_WL-CPU_WL
5      do
6          PARALLEL EXECUTION: GPU and CPU
7          GPU: Run_Func(Hardware_Info,GPU,GPU_WL,Remaining_WL,
8                        Limit_WL,GPU_Change_ratio)
9          CPU: Run_Func(Hardware_Info,CPU,CPU_WL,Remaining_WL,
10                       Limit_WL,CPU_Change_ratio)
11         SYNC_Rebalance_Func(Hardware_Info,CPU_WL_Info,
             GPU_WL_Info)
12     while Iteration !=0
```

distributed in a co-running manner. Then, CPU and GPU processing elements can execute the workload concurrently. In addition, in the specific case of our stencil kernel, each time step relies on the results obtained in the previous one. Thus, such a computation requires both the host and the device to synchronize regularly during the application's lifetime. There are two cases which will determine how the workload will be partitioned between the host and the device. First, the static initial workload on GPU, which should be smaller than the available GPU memory; second, we can obtain the initial workload from our estimation model, outlined in Section 3.2. It is based on the compute node's hardware configuration, such as the number of CPU cores, last-level of cache (LLC), the GPU type *etc.* The DAWL scheduling algorithm is meant to minimize workload imbalance between heterogeneous processing elements. To overcome the problem of load imbalance for heterogeneous systems, DAWL dynamically adjusts the workload distribution on different computing resources based on real time information. Algorithm 2 is composed of six steps (steps 3 and 4 are described in Figure 2):

(1) Initialize CPU and GPU configurations, *e.g.*, determine the number of processing elements, their initial workload, *etc.*.
(2) Run the tasks on CPUs and GPU simultaneously.
(3) Monitor the computation. For example, when CPUs finish computing, CPUs will check the GPU status. If computing on CPUs is faster than on GPU, it means the CPUs must be given a larger workload. Hence, the workload will be adjusted for CPUs in the next tasks group. The GPU will be targeted similarly, with a few limitations, *e.g.*, the GPU workload must fit in the device's memory size.
(4) Adapt to borderline cases. When the remaining workload reaches a given threshold, *e.g.*, 10% of the total workload, CPUs or GPU will only take half of remaining workload as next task no matter which one finishes first. The first half will be allocated to whichever set of processing elements finishes first. Thus, the first and second halves may run on different or the same type of cores.
(5) Synchronize and re-balance workload when all the compute tasks in one time step finish. The load-balancing function will check all the workload information, and compute the mean workload for each processing element type. The hardware status will also be re-checked: if the hardware changed, *e.g.*, if several CPUs are not available at this time, the system must adjust the workload on both CPUs and GPU.

(6) Reset CPUs and GPU and free allocated memory.

3.2 Optimization: profile-based machine-learning estimation model

While DAWL can dynamically adjust the workload according to real-time information, an unsuitable initial workload may drag down the performance when the problem size is relatively small. As shown in Figure 3, when problem sizes are close to a specific drop point, an unsuitable initial workload (such as an initial workload on GPU close to the problem size) lowers the performance. It is not a guaranteed behavior; Figure 3 shows the performance of a small initial GPU workload (EDRT-DAWL-2GB-1) can be better than a big initial workload (EDRT-DAWL-2GB-1). However, assuming a small workload will yield better results by default cannot be our golden rule, especially for stencil types of application. Indeed, such applications feature heavy data dependencies, need to be synchronized when partitioning the workload into different tasks. A smaller workload means more communication, as the number of tasks and workloads are in inverse proportion, and the number of tasks and shared data are proportional. A suitable initial workload can help us fully utilize the computing resources with reasonable amounts data transfers.

To solve this problem, we propose a profile-based machine-learning estimation model to optimize DAWL. A black-box machine learning method, *i.e.*, an automatic model without any user intervention, is used. Our estimate model follows three phrases:

(1) Collect hardware architecture information (see Table 1) as parameters to our black-box estimation model.
(2) Collect application runtime profile information as training data (samples).
(3) Utilize the information gathered from the first two steps to build a profile-based estimation model for a given heterogeneous platform, and furthermore obtain customized initial workload on different heterogeneous architectures and how many devices are actually necessary.

In our machine learning black-box estimation model (step 3), different algorithms were tested, including linear regression, logistic regression and random forests. The best match and less computationally complex algorithm was chosen to run our statistical model. To evaluate how well the model fits the data, the statistical measure, $R_{squared}$, is widely used. It is also known as the coefficient of determination. $R_{squared}$ is defined as the percentage of the response variation that is explained by a linear model. $R_{squared} = Explained\ variation/Total\ variation$. *R_squared* is always between 0 and 100%. 0% indicates that the model explains none of the variability of the response data around its mean and 100% indicates that the model explains all the variability of the response data around its mean.

Once the best matched statistical model is obtained, an estimation formula can be built up to predict the application performance on this specific heterogeneous platform. For a given problem size, a minimization multi-variable function can be used to obtain an estimation of the initial workload.

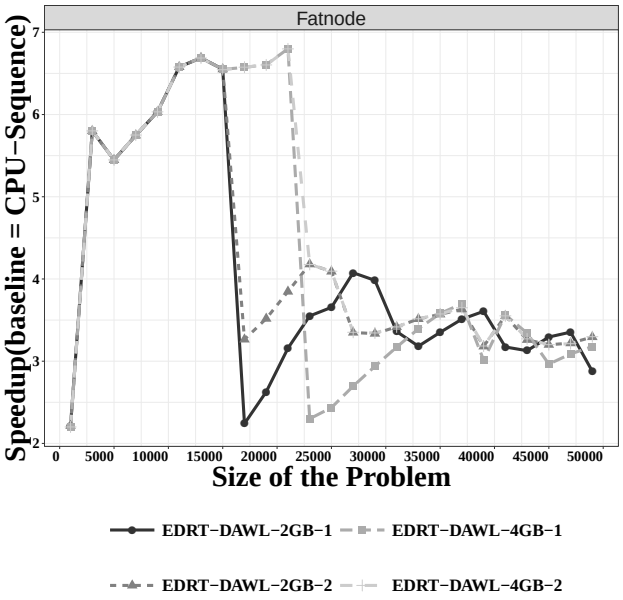

Figure 3: Stencil application: available GPU memory (av_GPU): 2GB vs 4GB, initial workload (GPU=CPU): $0.5 \times av\_GPU$ **(1) vs** 2000×2000 **(2)**

4 EXPERIMENTAL RESULTS

4.1 Experimental Testbed

4.1.1 Experiment Hardware Overview. Several heterogeneous SMP nodes with multicores were selected for our experiments. They are representative of the current many core-based hardware trends. The main characteristics of those systems are shown in Table 1. Each system is equipped with Intel processors and Nvidia GPUs. `fatnode`'s general purpose CPUs are made of Intel Xeon® E5-2670, with hyper-threading activated, and a Tesla K20® (Kepler architecture with Compute Capability 3.5) board. `Super Micro` sports two Intel Xeon®E7 v2, and embeds 4 Tesla K20 boards®. `CCSL Valinhos` contains one processor Intel i7-4770® and one Tesla K40®. `Debian` has one processor i7-4930K®, with one Nvidia Titan®board and one GeForce GT 630®. `Hive` embeds one Intel i7-3930k® and two GeForce GTX 680®.

4.1.2 Runtime System: Extending EDRT. The original version of EDRT mainly focuses on homogeneous many-core systems. However, this work targets workload balancing between heterogeneous resources. To validate and verify DAWL, we extended EDRT to support heterogeneous architectures. We will only describe the logical model without deep explanations about implementation details in the implementation because of space limitation.[2] To handle GPU tasks, we developed a new type of Codelet, named *GPU-Codelet*, consisting of two parts: CUDA host code and CUDA kernel code. The host and kernel codes can run concurrently or sequentially. To guarantee each code runs on its intended device (GPU or CPU), a

[2]We intend to publish the source code of all our experiments and modifications to EDRT upon acceptance of this paper.

Table 1: Experiment HardWare Environment

Param. / Machines	CPU Parameters					GPU Parameters					PCIe
	CPUThreads	Clock	# Socket	L3 Cache	CPU Mem	# SM	Clock	L2 Cache	GPU Mem	# CE	
1. Fatnode (K20)	32	2.6 GHz	2	20 MB	64 GB (2x32)	13	0.71 GHz	1.25 MB	4.8 GB	2	6.1 GB/s
2. Super Micro (K20)	40	3 GHz	2	25 MB	256 GB (2x128)	13	0.71 GHz	1.25 MB	4.8 GB	2	6.1 GB/s
3. CCSL (Valinhos) (k40)	8	3.4 GHz	1	8 MB	16 GB	15	0.75 GHz	1.5 MB	12 GB	2	10.3 GB/s
4. Debian (Titan)	12	3.4 GHz	1	12 MB	31 GB	14	0.88 GHz	1.5 MB	6 GB	2	11.5 GB/s
5. Hive (GTX 680)	12	3.2 GHz	1	12 MB	30 GB	8	1.06 GHz	0.5 MB	2 GB	1	5.6 GB/s

GPU-CPU codelet scheduling method was designed. A mechanism was added to monitor CPU-codelets and GPU-codelets, so that the two types of tasks can synchronize with each other.

4.2 Experimental Results: DAWL

DAWL was evaluated on the five systems we described in Section 4.1 (see Table 1 for details). In the following experiments, all systems were configured so that only 2 GB were seen as available by the runtime system. This was intended to evaluated DAWL more fairly, by reducing the "parameters surface" to explore. All our experiments were run using a 5-point 2D stencil computation kernel, with a fixed number of time steps (the convergence test was left out to make the experiments more deterministic). All matrices hold double precision values.

In this section, we adopt a static initial workload methodology to validate and verify the DAWL algorithm. The initial GPU and CPU workload both were 2000×2000 (*rows* $\times$ *columns*).

4.2.1 Performance Analysis: Full Resource Usage. To comprehensively characterize DAWL, we performed a series of workload performance analyses. We compared the `EDRT-DAWL` performance with `GPU-Only`, `CPU-Seq`, `EDRT-CPU`, `EDRT-GPU` executions (see Table 2 for details). Here, `EDRT-DAWL` is the implementation of DAWL on EDRT. Depending on the application, problem size, and hardware information, `EDRT-DAWL` may run on multiple CPUs or GPUs, or be co-running on both CPUs and GPUs.

Figure 4 shows the speedup of different stencil variants, using the `CPU-Seq` version as a baseline. Everything was run on all systems. Figure 5 zooms in Figure 4 for matrix sizes 17000×7000 and onward. `EDRT-GPU` uses the concurrent streaming mechanism described in Section 3.2, for all problem sizes. From Figure 4, we can see the performance of `EDRT-GPU` is stable. We also wrote another variant of `EDRT-GPU`, which uses solely the traditional one-stream method (not shown here due to space limitations). Its performance is very bad when the problem size is larger than the GPU's memory capacity. The `GPU-Only` variant was slightly optimized: When the problem size was smaller than the GPU memory capacity, we used the one-stream method to avoid superfluous synchronizations between the host and device. We used concurrent streaming when problem sizes were larger than the GPU's memory capacity, in order to overlap communication and computation.

Figure 4 shows that, even though there are a lot of differences, overall, `EDRT-DAWL`'s performance drops dramatically at 17000×17000. Based on the *hardware_choose* function (detailed in Algorithm 1), when the problem size is smaller than 17000×17000, the application belongs to the GPU-friendly category and all the workload will be on GPU. Figure 4 validates our solution. On all systems, `EDRT-DAWL`'s performance is very close to `GPU-Only`. If the GPU's

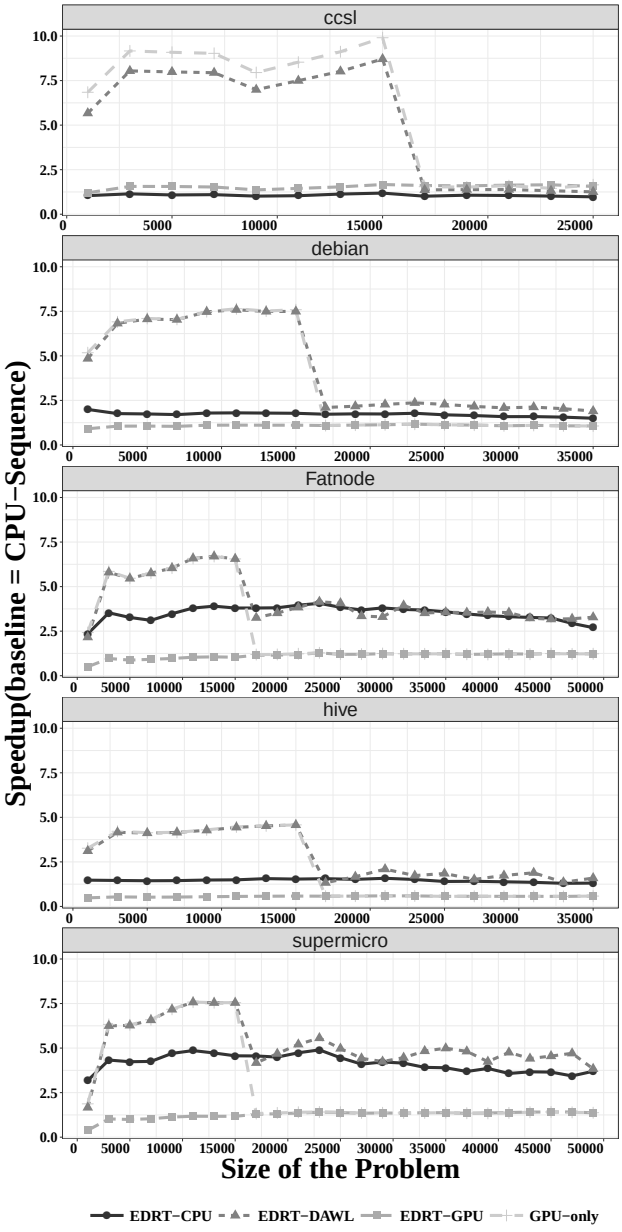

Figure 4: Speedup of the different Stencil versions

Table 2: Stencil kernel implementation

Implementation	Illustration
CPU-Seq	sequential c++ code
GPU-Only	cuda code
EDRT-CPU	multi-threads c++ code
EDRT-GPU	CUDA code on EDRT (concurrent streams)
EDRT-DAWL	DAWL hybrid code on EDRT

memory capacity changes, the inflection point also changes. As shown in Figure 3, the inflection point shifts from 17000 × 17000 to 23000× when the GPU's memory capacity changes from 2GB to 4GB. When problem sizes are larger than 17000 × 17000, the application changes to the co-running friendly category, and computation will run on both CPUs and GPU. Figure 5 shows that the speedup ratios are quite different on different systems with different problem sizes. On the fatnode and Super Micro, EDRT-DAWL and EDRT-CPU are alternately faster; on the CCSL Valinhos, EDRT-DAWL and GPU-Only are similar; on Hive and debian, EDRT-DAWL is faster than EDRT-CPU.

From Figure 3, we can observe the initial workload significantly affects performance when problem sizes are relatively small; their impact is reduced when the problem size keeps increasing. This is because the bigger the problem size, the more time EDRT-DAWL has to balance the workload. Except for the initial workload, our profile-based analytical model can also help us obtain the competitiveness change ratio. We will describe it in Section 4.3.

4.2.2 Performance Analysis: Varying the CPU Number. Figure 6 shows the experimental results related to our second question: do we really need all the computation resource simultaneously when we run a co-running friend application? Due to page limitations, we only show one system running with different threads. This servers stands for a classical type of hardware configuration: a "regular" GPU and a two-way SMP chip multiprocessing system. We used two different strategies to pin threads to physical processing elements: *spread* and *compact*. According to the hardware topology gathered by the runtime system, the *spread* strategy attempts to allocate software threads to processing elements (*i.e.*, a physical core or thread) as far from each other as possible. On the contrary, the *compact* strategy attempts to allocate software threads as closely as possible on the available processing elements. *Spread* can potentially help performance when fewer threads are involved in the computation, as each thread can benefit from more shared LLC or private L1-L2 cache space. However, this may incur a higher overhead if threads are sharing a fair amount of data. On the contrary, the *compact* pinning strategy ensures that locality is maximal, but may end up under-utilizing the underlying hardware if too few threads are involved in the overall computation. Figure 7 shows the topology of fatnode.

Even though the *compact* and *spread* methods affect plenty the performance, the rough trend is the same. When the threads number reaches its limit, increasing the number of threads does not improve performance—which is expected, because of memory conflicts. Section 4.3 will detail how our analytical model can help

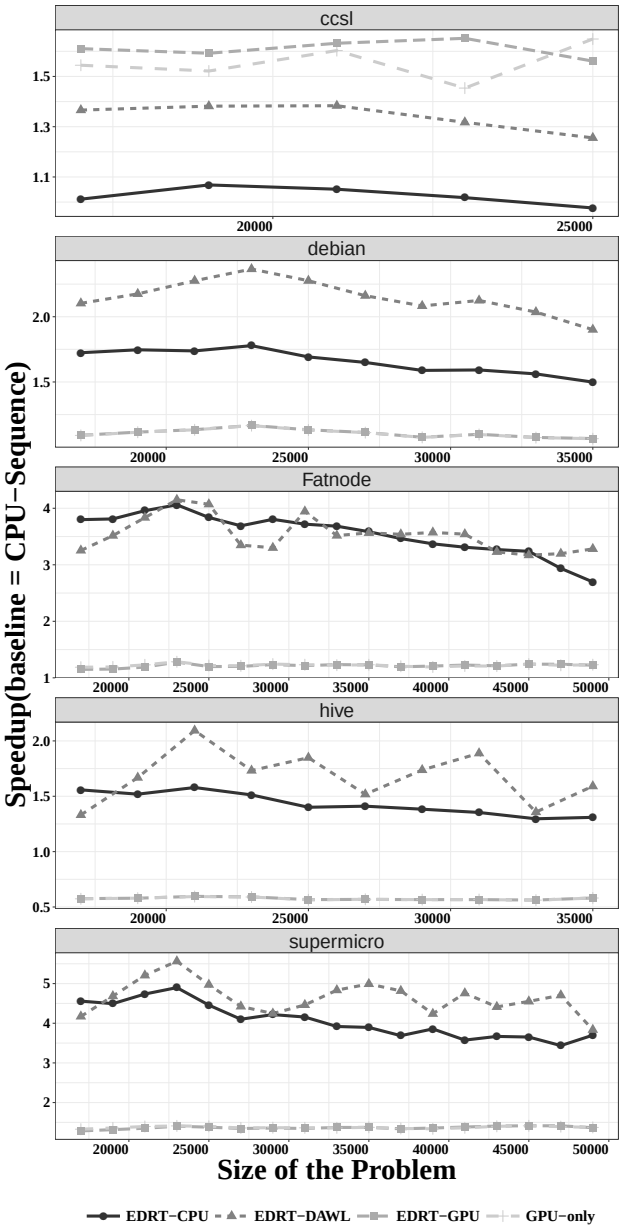

Figure 5: Speedup of the different Stencil versions of matrices higher than 17K

obtain a suitable threads number based on the application and hardware configuration.

4.3 Experimental Results: Profile-based Machine-learning Estimation Model

As we describe in Section 3.2, to build up a profile-based Machine-learning estimation model for heterogeneous architecture, collecting data is the first step. Sampling method is good way to

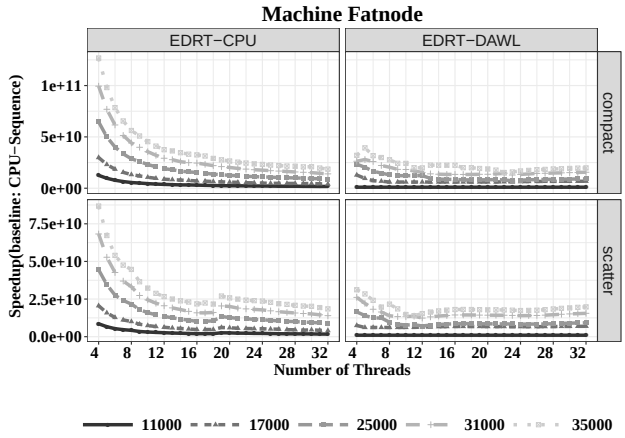

Figure 6: Performance with a varying number of general-purpose processing elements on `fatnode`

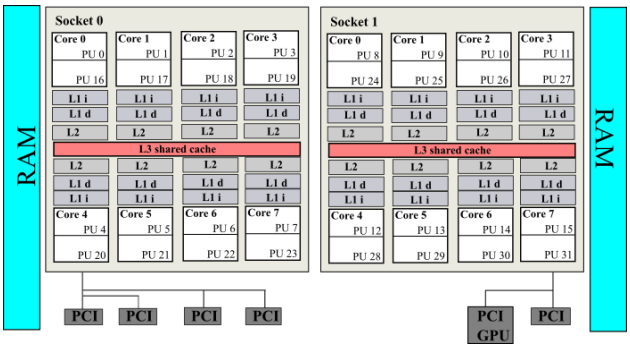

Figure 7: Topology of `fatnode`

obtain both training data and test data. Our target application is co-running friendly application corresponding to stencil code which problem size are larger than 17000*17000 with 2GB available GPU memory. Our approach is to utilize small problem size data to predict big problem size execution time and data with small time step(iteration) to predict large time step. Total sample set including test and training data set consists of several parts: initial GPU workload (1000,2000,4000,8000), initial CPU workload $GPU\_workload * (0.5, 1.0, 1.5, 2.0)$, iteration (1,2,8) problem size (starting from 17000 to 35000 with step of 2000), and CPU core number (4,8,16,..largest CPU core number on the server). The sample sets are a little bit of different on different servers, such as the largest ccsl server problem size sample will be 25000 since main memory is smaller than the others. Total sample on 5 servers is 6480.

Second step is to obtain the best matched model. multiple regression algorithms (linear regression, logistic regression) with multiple Meta-functions and Polynomial functions, and ensemble learning method (random forests) *etc.* are test. Finally, the best fit mode is the simple linear regressions (Meta-function) whose $R\_squared$ are

Table 3: Mean Absolute Percentage Error (MAPE)

Machines	supermicro	fatnode	debian	hive	ccsl
MAPE	7.41%	6.43%	1.68%	3.49%	3.45%

between 93 and 94%, Figure 8 explains the variance of the data in this percentage.

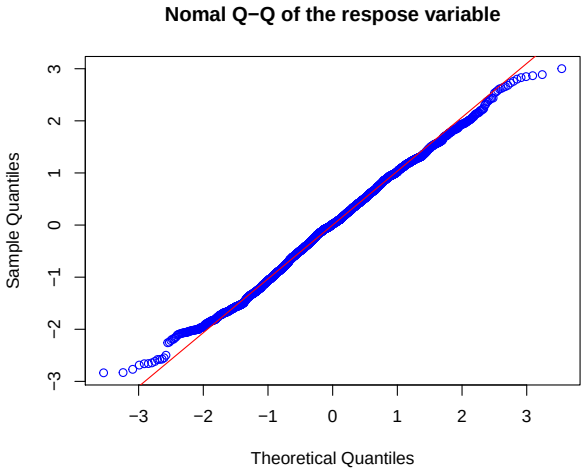

Figure 8: Quantiles of the model with all the samples

To measure the progress of the learning algorithm we have used the Mean Absolute Percentage Error (MAPE). With this error we have analyzed the reliability of our approaches. Table 3 shows Mean Average Percentage Error (MAPE) of the linear model for each machine.

As a second goal in this statistical estimation model, we wanted know which parameters had more impact in the construction of the model. For this reason, we used the absolute value of the $t-statistic$ for each model parameter and calculate the importance of each parameters in the model, we made this experiment with 17 software and hardware parameters, the result was that 6 parameters are enough to predict the performance of the developed Stencil application, these were: NumOfSocket, CPU_Clock, CPU_Cores, Total_Workload, Initial_GPU_Workload, CPU.GPU_Workload_Ratio. Except 3 workload related parameters, 3 other parameters are all related to the CPUs. In our case, GPUs in 5 servers are pretty similar, that is the reason why GPUs parameters do not have relevance in our model. The situation will change if more GPUs type are involved in the experiments. GPUs and PCIe information were hidden in parameter Workload parameters.

Finally, by using estimation model(execution estimation formula) and mathematical optimization method (minimize multi-variable function), the best initial workload on GPU and CPU, most efficient device number can be obtained.

Combining our profile-based estimation model and DAWL, we obtained the the results shown in Figure 9. Compared to `EDRT-CPU`, which always uses all the CPUs, this new implementation of `EDRT-DAWL`

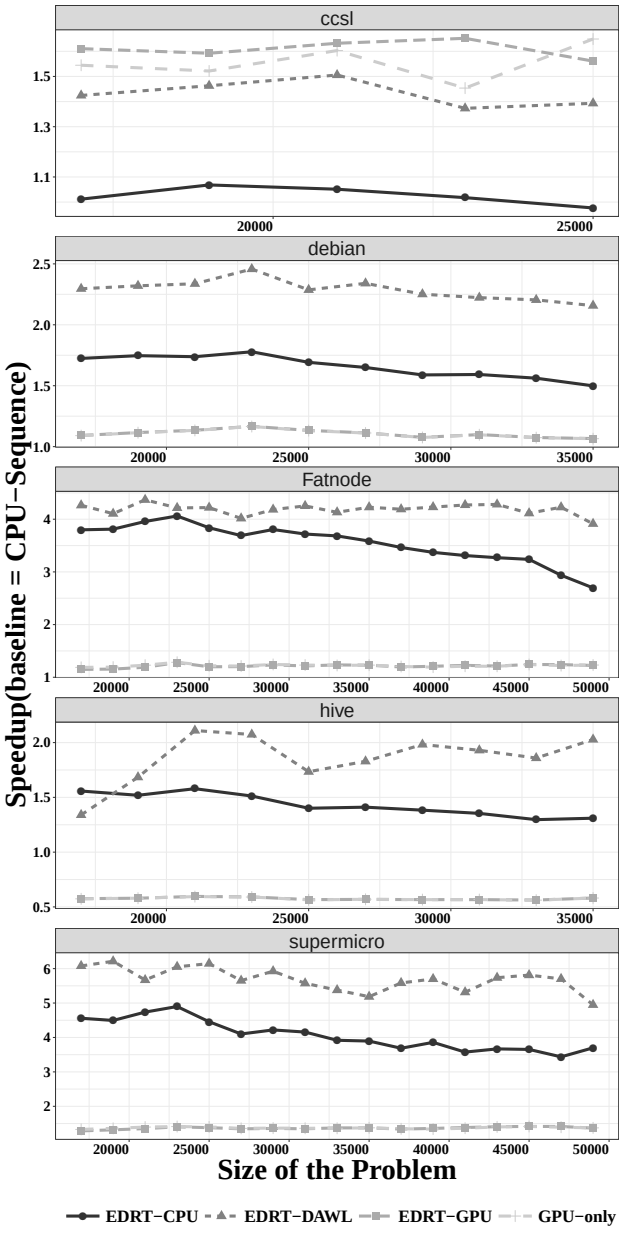

Figure 9: Speedup of the different Stencil versions of matrices higher than 17K

uses at most half of the CPUs (depending on the hardware). Our scheduler can obtain up to 6× speedup compared to the sequential version, 1.6× speedup compared to the multiple core version, and 4.8× speedup compared to the pure GPU version.

5 RELATED WORK

The efficient utilization of the computing capacity offered by CPU-GPU heterogeneous resources is a challenging problem. Moreover, with the development of chip multiprocessing technology, new types of heterogeneous platforms, equipped with a deep memory hierarchy system, Non-Uniform Memory Access (NUMA) and multiple different types computation devices has become very common in the academic and industry areas, and even in daily life. As a result, a number of compilers, runtime systems, scheduling methods and frameworks [3, 8, 10, 17, 21] have been introduced.

Teodoro *et al.* [21] have proposed and implemented a performance variation aware scheduling technique along with an estimation optimization model to collaboratively use CPUs and GPUs on a parallel system. A number of scheduling methods or library [15, 18, 20] were combined with StarPU [3], a task programming library for hybrid architectures based on task-dependency graphs, to perform scheduling and handle task placement in heterogeneous systems. In StarPU, the user provides different kernels and tasks for each target device and specifies inputs and outputs of each task. The runtime ensures that data dependencies are transfered to the devices for each task. Furthermore, the user can specify explicit dependencies between tasks.

Panneerselvam and Swift proposed Rinnegan [15], a Linux kernel extension and runtime library, and implemented and validated that decisions of where to execute a task must consider not only execution time of the task, but also current heterogeneous system conditions. Sukkari *et al.* [20] proposed an asynchronous out-of-order task-based formulation of the Polar Decomposition method to improve hardware occupancy using fine-grained computations and look-ahead techniques.

The main challenge of the load-balancing mechanism is to precisely divide workload on processing units. A simple heuristics devision approach may result in worse performance. Belviranli *et al.* proposed a dynamic load-balancing algorithm for heterogeneous GPU clusters named the Heterogeneous Dynamic Self-Scheduler (HDSS) [4]. Sant'Ana *et al.* described a novel profile-based load-balance algorithm [18] named PLB-Hec for data-parallel applications in heterogeneous CPU-GPU clusters. PLB-HeC algorithm performs an online customized estimation of performance curve models for each devices (CPU or GPU). Like a typical data-parallel application, data in PLB-HeC is divided in blocks, which can be concurrently processed by multiple threads. The granularity of the block size for each processing unit is crucial for performance: incorrect block sizes will produce idleness in some processing units and reduce performance. To get good block sizes, PLB-HeC solves the problem in three phrases: first, it dynamically computes performance profiles for different processing units at runtime; second, using a non-linear system model, they determine the best distribution of block size among different processing units; third, they rebalance the block size during execution. PLB-Hec obtained higher performance gains with more heterogeneous clusters and larger problems sizes.

All the works presented above rely on StarPU to implement their various strategies. Of all of them, Belviranli *et al.* and Sant'Ana *et al.*'s work are probably closest to our own (they rely on online profiling, or resort to some machine-learning techniques to perform load-balancing decisions). However, this work and most of the previous ones tend to focus on loosely synchronized parallel workloads, where specific tasks are often run only a specific type of processing element (*e.g.*, CPU or GPU). On the contrary, our work

focuses on workloads that are iterative in nature, feature heavy data dependences, and require regular and possibly frequent synchronization operations between the device and the host. The work itself is "homogeneous," but it can be run on either the host or the device, depending on their state of idleness, the remaining work size to perform, *etc.*

6 CONCLUSIONS AND FUTURE WORK

The performance of HPC system on heterogeneous many-core system dependents on how well a scheduling policy allocates its workload its processing elements. An incorrect decision can degrade performance and waste energy and power. This paper makes the following contributions: first, a dynamic adaptive workload balance (DAWL) scheduling algorithm is proposed. It can adaptively adjust workload based on available heterogeneous resources and real time information; second, a profile-based machine-learning estimation model is built to optimize our scheduler algorithm. The estimation model can obtain a customized initial workload on various heterogeneous architectures, as well as how many devices are necessary. With these information, DAWL can reduce its adaptation time. As a case study, we used a 5 points 2D stencil and ran it on an extended implementation of the EDRT runtime system. We reported that, when the workload fit in the GPU memory, we could get speedups as high as 8.75× compared to the sequential baseline, 7× when comparing to a pure multicore version, and stay on-par with a pure GPU version. When the workload size goes beyond the GPU memory capacity, our scheduling system can reach speedups up to 1.6× the pure multicore version, and 4.8× compared to the pure GPU version.

We plan to augment our model with power-consumption parameters to enrich a profile-based DAWL and determine good trade-offs between performance and power on heterogeneous architectures. This will result in a more complex analytical model.

REFERENCES

[1] Authors omitted for blind review purposes 2017. Title omitted for blind review purposes. *ACM Trans. Archit. Code Optim.* 14, 4, Article 47 (Dec. 2017), P pages.

[2] Authors omitted for blind review purposes 2017. Title omitted for blind review purposes. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. P–P.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.* 23, 2 (Feb. 2011), 187–198. https://doi.org/10.1002/cpe.1631

[4] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. 2013. A Dynamic Self-scheduling Scheme for Heterogeneous Multiprocessor Architectures. *ACM Trans. Archit. Code Optim.* 9, 4, Article 57 (Jan. 2013), 20 pages. https://doi.org/10.1145/2400682.2400716

[5] Q. Chen and M. Guo. 2017. Contention and Locality-Aware Work-Stealing for Iterative Applications in Multi-socket Computers. *IEEE Trans. Comput.* PP, 99 (2017), 1–1. https://doi.org/10.1109/TC.2017.2783932

[6] J. B. Dennis. 1974. First Version of a Data Flow Procedure Language. In *Programming Symposium, Proceedings Colloque Sur La Programmation*. Springer-Verlag, London, UK, UK, 362–376. http://dl.acm.org/citation.cfm?id=647323.721501

[7] V. García, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Pena. 2016. Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. https://doi.org/10.1109/IISWC.2016.7581277

[8] Francisco Gaspar, Luis Taniça, Pedro Tomás, Aleksandar Ilic, and Leonel Sousa. 2015. A Framework for Application-Guided Task Management on Heterogeneous Embedded Systems. *ACM Trans. Archit. Code Optim.* 12, 4, Article 42 (Dec. 2015), 25 pages. https://doi.org/10.1145/2835177

[9] Benedict R. Gaster and Lee Howes. 2012. Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck? *Computer* 45 (2012), 42–52. https://doi.org/10.1109/MC.2012.257

[10] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. 2013. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 1299–1308. https://doi.org/10.1109/IPDPS.2013.66

[11] Chris Gregg and Kim Hazelwood. 2011. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '11)*. IEEE Computer Society, Washington, DC, USA, 134–144. https://doi.org/10.1109/ISPASS.2011.5762730

[12] TOP500 Supercomputer List. [n. d.]. http://www.top500.org (visited on Nov. 2017). ([n. d.]).

[13] Maxime Martinasso, Grzegorz Kwasniewski, Sadaf R. Alam, Thomas C. Schulthess, and Torsten Hoefler. 2016. A PCIe Congestion-aware Performance Model for Densely Populated Accelerator Servers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, Article 63, 11 pages. http://dl.acm.org/citation.cfm?id=3014904.3014989

[14] NVIDIA. 2015. *CUDA C: Programming Guide, Version 7.*

[15] Sankaralingam Panneerselvam and Michael Swift. 2016. Rinnegan: Efficient Resource Use in Heterogeneous Architectures. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 373–386. https://doi.org/10.1145/2967938.2967964

[16] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 457–467.

[17] Vignesh T. Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. 2010. Compiler and Runtime Support for Enabling Generalized Reduction Computations on Heterogeneous Parallel Configurations. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10)*. ACM, New York, NY, USA, 137–146. https://doi.org/10.1145/1810085.1810106

[18] L. Sant'Ana, D. Cordeiro, and R. Camargo. 2015. PLB-HeC: A Profile-Based Load-Balancing Algorithm for Heterogeneous CPU-GPU Clusters. In *2015 IEEE International Conference on Cluster Computing*. 96–105. https://doi.org/10.1109/CLUSTER.2015.24

[19] Authors omitted for blind review purposes 2013. Title omitted for blind review purposes. In *Proceedings of the 19th International Conference on Parallel Processing (Euro-Par'13)*. Springer-Verlag, Berlin, Heidelberg, P–P.

[20] D. Sukkari, H. Ltaief, M. Faverge, and D. Keyes. 2017. Asynchronous Task-Based Polar Decomposition on Single Node Manycore Architectures. *IEEE Transactions on Parallel and Distributed Systems* PP, 99 (2017), 1–1. https://doi.org/10.1109/TPDS.2017.2755655

[21] G. Teodoro, T. M. Kurc, T. Pan, L. A. D. Cooper, J. Kong, P. Widener, and J. H. Saltz. 2012. Accelerating Large Scale Image Analyses on Parallel, CPU-GPU Equipped Systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 1093–1104. https://doi.org/10.1109/IPDPS.2012.101

[22] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. 2012. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 213–224. http://dl.acm.org/citation.cfm?id=2337159.2337184

[23] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu. 2010. Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing. In *2010 IEEE International Conference on Cluster Computing*. 19–28. https://doi.org/10.1109/CLUSTER.2010.12

[24] F. Zhang, J. Zhai, B. He, S. Zhang, and W. Chen. 2017. Understanding Co-Running Behaviors on Integrated CPU/GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems* 28, 3 (March 2017), 905–918. https://doi.org/10.1109/TPDS.2016.2586074

[25] Z. Zhong, V. Rychkov, and A. Lastovetsky. 2012. Data Partitioning on Heterogeneous Multicore and Multi-GPU Systems Using Functional Performance Models of Data-Parallel Applications. In *2012 IEEE International Conference on Cluster Computing (Cluster 2012)*.

[26] Authors omitted for blind review purposes 2011. Title omitted for blind review purposes. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT '11)*. ACM, New York, NY, USA.

A.2 Proposed Benchmark of Task-based Heterogeneous Applications for Scheduling algorithms

In this collaboration, we presented efficient algorithms for scheduling an application represented by a precedence task graph on hybrid and more general heterogeneous computing resources. For this, we studied the above problem on both off-line and on-line settings. The goal was to design algorithms through a solid theoretical analysis that can be practically implemented in actual systems. For complete this work, I have developed a benchmark to test these scheduling algorithms, stored in Standard Workload Files (SWF) format (Feitelson *et al.*, 2007). This benchmark was my main subject in this work and I will describe it here. For more information about the scheduling algorithms, see papers (Amarís *et al.*, 2017; Amaris *et al.*, 2017).

Those SWF files are the result of applications executed on different heterogeneous machines with CPUs and GPUs. This benchmark is freely available for research purpose under Creative Commons Public License[1]. The SWF was defined in order to ease the use of workload logs and models. With it, programs that analyze workloads or simulate system scheduling need only to be able to parse a single format, and can be applied to multiple workloads. The execution time of each task and their respective dependencies were collected and formatted.

The benchmark is composed of six parallel applications. Five of them have been generated from Chameleon, a dense linear algebra software which is part of the MORSE project (Agullo *et al.* , 2012), while the sixth has been generated with GGen, a library for generating directed acyclic graphs (Cordeiro *et al.*, 2010), and it corresponds to a more irregular application.

This benchmark has been generated for the purpose of two scientific papers:

1. **Marcos Amaris**, Clement Mommessin, Giorgio Lucarelli and Denis Trystram. Generic algorithms for scheduling applications on hybrid multi-core machines. In *2017 Euro-Par: International Conference on Parallel and Distributed Computing*, pages 220–231.

2. **Marcos Amaris**, Clement Mommessin, Giorgio Lucarelli and Denis Trystram. Generic algorithms for scheduling applications on heterogeneous multi-core platforms *arXiv:1711.06433 preprint* in revision[2]. Submmited for *Concurrency and Computation: Practice and Experience*.

A.2.1 Chameleon Software

The five applications from Chameleon software are `getrf_nopiv`, `posv`, `potrs`, `potri` and `potrf`, these applications are composed of multiple sequential basic tasks of linear algebra. The kernels used are: `SYRK` (symetric rank update), `GEMM` (general matrix-matrix multiply) and `TRSM` (triangular matrix equation solver), among others (see Table A.3). Figure A.3 shows the DAG of the application Cholesky Factorization with $nb\_blocks = 5$, which is the application `potrf`. The Chameleon applications (Agullo *et al.*, 2012) were executed with the runtime StarPU (Augonnet *et al.*, 2011) and the execution traces were collected. At first, all the applications were executed on CPUs and then were forced to execute on GPUs to have the processing times of each task of the application for different resources (CPUs and GPUs).

To generate the applications, different tilings of the matrices have been used, varying the number of sub-matrices denoted by *nb_blocks* and the size of the sub-matrices denoted by *block_size*.

[1] Hosted at: https://github.com/marcosamaris/heterogeneous-SWF [Accessed on 13 March 2018]
[2] https://arxiv.org/abs/1711.06433

Apps / Kernels	getrf_nopiv	posv	potrf	potri	potrs
syrk		x	x	x	
gemm	x	x	x	x	x
trsm	x	x	x	x	x

Table A.3: *Basic kernel of linear algebra of each application*

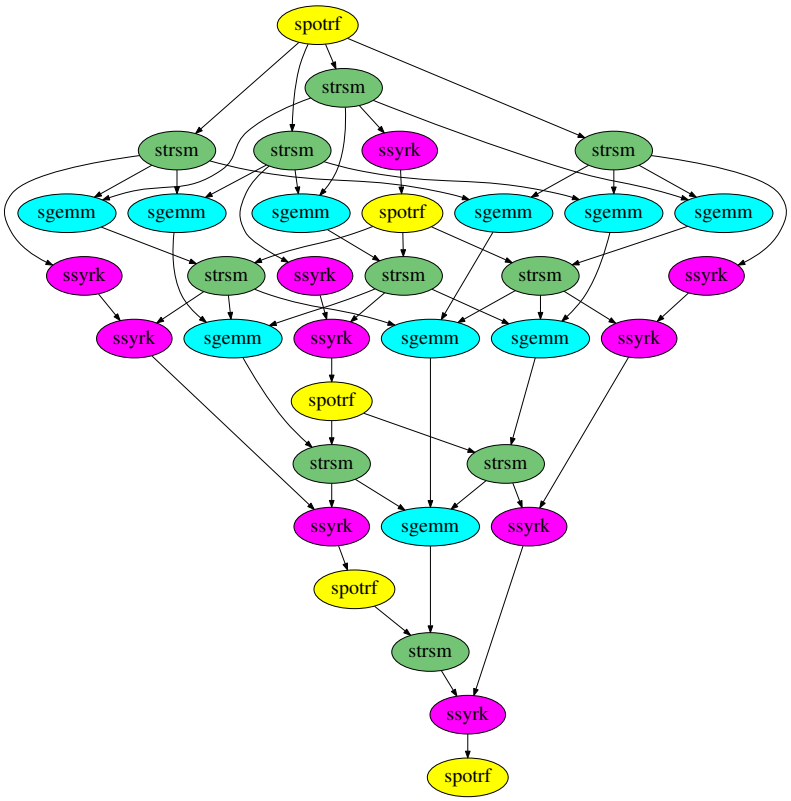

Figure A.3: *Direct Acyclic Graph of the application spotrf with nb_blocks = 5 (Cholesky Factorization)*

Here, tiling is related to the number of sub-matrices that problem is divided. The different values of *nb_blocks* were 5, 10 and 20 and the different values of *block_size* were 64, 128, 320, 512, 768 and 960, for a total of 18 configurations per application. Table A.4 shows the total number of tasks for each application and each value of *nb_blocks*. Notice that the value of *block_size* does not impact the number of tasks.

These applications were executed and data was collected over two different machines, one machine with one type of CPU and one type of GPU (2 resources) and another with one type of CPU and 2 different GPUs (3 resources). Amaris *et al.* (2017) used the dataset from machine with 2 type of resources and dataset from the machine with 3 type of resources was used by (Amarís *et al.*, 2017).

For the setting with 2 resource types, the chameleon applications were executed on a machine with two Xeon E7 v2 with a total of 10 physical cores with hyper-threading of 3 GHz and 256 GB of RAM. The machine had 4 GPUs NVIDIA Tesla K20 (Kepler architecture) with each 5 GB of memory and 200 GB/s of bandwidth. For 3 resource types, the applications were executed on an Intel i7-5930k machine with a total of 6 physical cores with hyper-threading of 3.5 GHz and 12 GB

Apps / Nb_blocks	getrf_nopiv	posv	potrf	potri	potrs
5	55	65	35	105	30
10	385	330	220	660	110
20	2870	1960	1540	4620	420

Table A.4: *Total number of tasks in function of the number of blocks*

of RAM. This machine had 2 NVIDIA GPUs: a GeForce GTX-970 (Maxwell architecture) with 4 GB of memory and 224 GB/s of bandwidth; and a Quadro K5200 (Kepler architecture) with 8 GB of memory and 192 GB/s of bandwidth. The running time of each task composing the applications was stored for each resource type.

A.2.2 Fork-join Application

This application was generated with GGen. *fork-join* represents applications that starts by executing sequentially and then forks to be executed in parallel with a specific diameter (number of parallel tasks). When the parallel execution has completed, results are aggregated by performing a join operation. This procedure can be repeated several times depending on the number of phases. Figure A.4 shows a fork-join application with 3 phases and a diameter of 5 tasks in parallel. For our experiments, we used 2, 5 and 10 phases with a diameter of 100, 200, 300, 400 and 500 for a total of 15 different configurations. Table A.5 shows the total number of tasks for each configuration of this application.

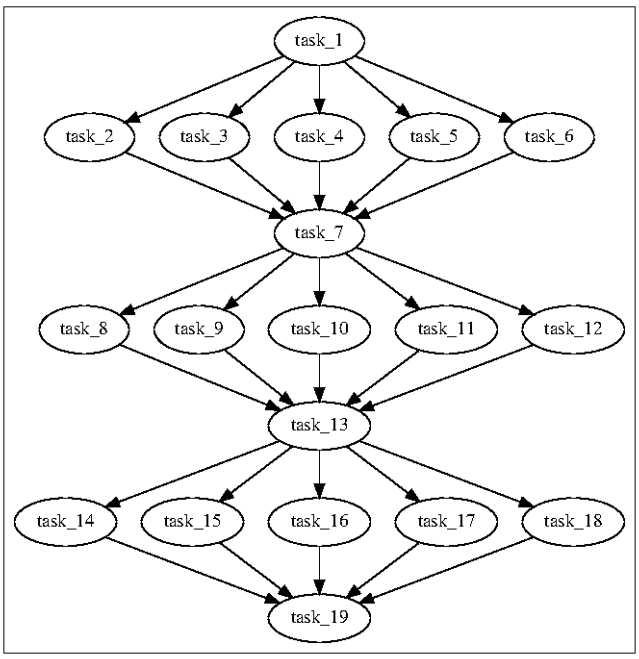

Figure A.4: *Fork-join application with 3 phases and 5 parallel and concurrent tasks*

The running time of each task was computed using a Gaussian distribution with center p and standard deviation $\frac{p}{4}$, where p is the number of phases. We have decided to establish various acceleration factors in each diameter of the fork. In this way, for all the configurations there are five parallel tasks with an acceleration factor between 0.1 and 0.5 while the others have an acceleration

Nb_phases \ Diameter	100	200	300	400	500
2	203	403	603	803	1003
5	506	1006	1506	2006	2506
10	1011	2011	3011	4011	5011

Table A.5: *Total number of tasks in function of the number of phases and the width of the phase*

factor between 0.5 and 50.

A.2.3 Method and Results

This benchmark was used to test algorithms for both the off-line and on-line settings of the addressed problem. In this document, we will show briefly some results of the on-line setting. For more information about this work, please read papers mentioned in page 89.

In this experiments was used the scheduling algorithm ER-LS (Enhanced Rules - List Scheduling), this is a set of combination rules with a greedy List Scheduling policy that schedules each task as early as possible on the CPU or GPU side already decided by the rules. We compared the performance, in terms of makespan, of the algorithm ER-LS (Enhanced Rules - List Scheduling), with 3 baseline algorithms: *EFT*, which schedules a task on the processor which give the earliest finish time for that task; *Greedy*, which allocates a task on the processor type which has the smallest processing time for that task; and *Random*, which randomly assigns a task to the CPU or GPU side. For the algorithms Greedy and Random, we used a List Scheduling algorithm to schedule the tasks once the allocations have been made.

The algorithms were implemented in Python (v. 2.8.6). For the machine configurations, we determined different sets of pairs (Nb_CPUs, Nb_GPUs). Specifically, we used 16, 32, 64 and 128 CPUs with 2, 4, 8 and 16 GPUs for a total of 16 machine configurations for the case with 2 resource types. For the case with 3 resource types, we determined different sets of triplets (Nb_CPUs, Nb_GPU1s, Nb_GPU2s) with the name numbers of CPUs and for either types of GPUs, for a total number of 64 machine configurations.

We executed the algorithms only once with each combination of application and machine configuration since all algorithms are deterministic, except Random. The running times of the algorithms were similar and took at most 5 seconds each for the biggest instances of applications.

Figure A.5 (left) compares the ratios between the makespan of each of the on-line algorithms and LP^* (the optimal solution of the linear program). Due to large differences between the performances of Random and the 3 other algorithms, we kept only the algorithms ER-LS, EFT and Greedy. Results show that Greedy is on average outperformed by ER-LS and EFT, and that EFT creates less outliers than the 2 other algorithms.

In this collaboration, we also studied the performance of the 3 algorithms with respect to the theoretical upper bound. Figure A.5 (right) shows the mean competitive ratio of ER-LS, EFT and Greedy along with the standard error as a function of $\sqrt{\frac{m}{k}}$ associated to each instance. To simplify the lecture, we only present the applications *potri* and *fork-join*, since other *Chameleon* applications showed similar results. We observed that the competitive ratio is smaller than $\sqrt{\frac{m}{k}}$ and far from the theoretical upper bound of $4\sqrt{\frac{m}{k}}$ for ER-LS. Where m is the number of identical CPUs and k of GPUs in the scheduling experiments.

Figure A.6 compares Greedy and ER-LS (left), and EFT and ER-LS (right), by showing the ratio between the makespans of the two algorithms. We can see that ER-LS outperforms Greedy on average, with a maximum for the potri application where ER-LS performs 11 times better than Greedy. More specifically, there is an improvement of between 8% and 36% on average for ER-LS depending on the application considered, except for *potrs* whose makespans are on average 10% greater than for Greedy.

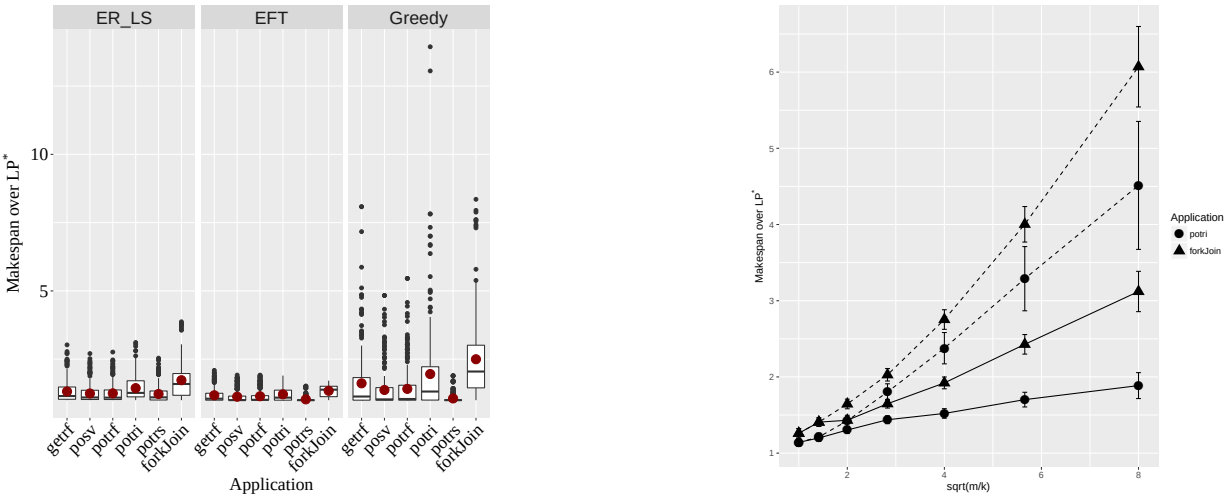

Figure A.5: *Ratio of makespan over LP^* for each instance, grouped by application for the on-line algorithms with 2 resource types (left). Mean competitive ratio of ER-LS (plain), EFT (dashed) and Greedy (dotted) as a function of $\sqrt{\frac{m}{k}}$ (right).*

Comparing EFT and ER-LS, we can see that ER-LS is outperformed by EFT with a decrease of 11% on average and up to 60% for certain instances of fork-join. However, the worst-case competitive ratio for EFT can be directly obtained from the proof of the worst-case approximation ratio for HEFT, presented in the paper attached. More specifically, if the adversary presents to EFT the list of tasks ordered as in the counter-example for HEFT, i.e., by decreasing order of the rank, then we obtain the same lower bound.

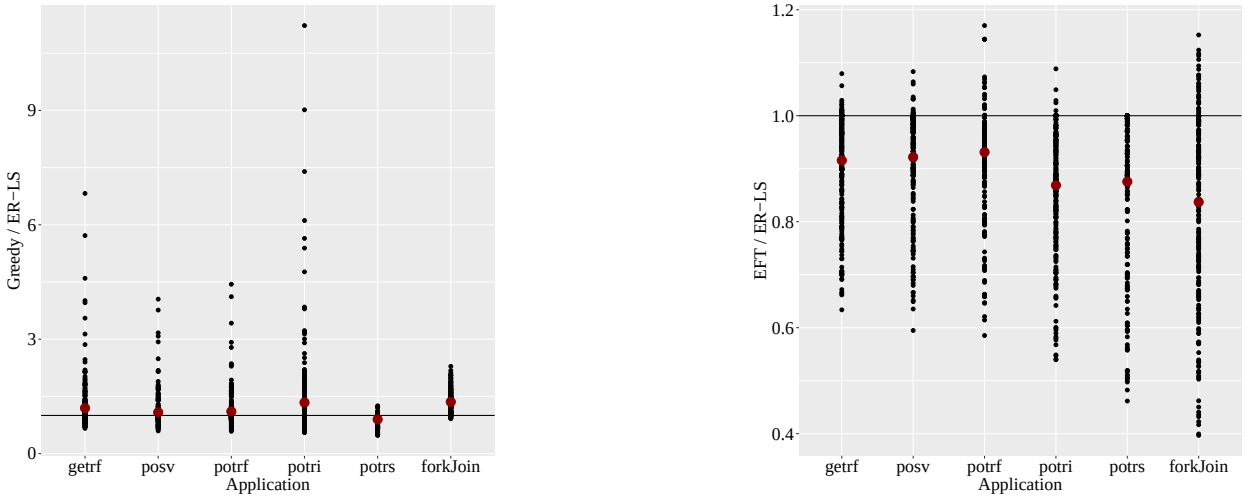

Figure A.6: *Ratio between the makespans of Greedy and ER-LS (left), and EFT and ER-LS (right) for each instance, grouped by application.*

Bibliography

Agullo *et al.* **(2012)** E. Agullo, G. Bosilca, B. Bramas, C. Castagnede, O. Coulaud, E. Darve, J. Dongarra, M. Faverge, N. Furmento, L. Giraud, X. Lacoste, J. Langou, H. Ltaief, M. Messner, R. Namyst, P. Ramet, T. Takahashi, S. Thibault, S. Tomov and I. Yamazaki. Poster: Matrices over runtime systems at exascale. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1332–1332. doi: 10.1109/SC.Companion.2012.168. Cited in page 89

Amaris *et al.* **(2016)** M. Amaris, R. Y. de Camargo, M. Dyab, A. Goldman and D. Trystram. A comparison of gpu execution time prediction using machine learning and analytical modeling. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pages 326–333. doi: 10.1109/NCA.2016.7778637. Cited in page 5, 75

Amarís *et al.* **(2017)** Marcos Amarís, Clément Mommessin, Giorgio Lucarelli and Denis Trystram. Generic algorithms for scheduling applications on heterogeneous multi-core platforms. *arXiv preprint arXiv:1711.06433*. Cited in page 75, 89, 90

Amaris *et al.* **(2017)** Marcos Amaris, Clément Mommessin, Giorgio Lucarelli and Denis Trystram. Generic algorithms for scheduling applications on hybrid multi-core machines. In *Euro-Par: International Conference on Parallel and Distributed Computing*, pages 220–231. Cited in page 75, 89, 90

Augonnet *et al.* **(2011)** C. Augonnet, S. Thibault, R. Namyst and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198. ISSN 1532-0626. 89

Cordeiro *et al.* **(2010)** D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent and F. Wagner. Random graph generation for scheduling simulations. In *ICST (SIMUTools)*. Cited in page 89

Feitelson *et al.* **(2007)** Dror Feitelson, D TALBY and JP JONES. Standard workload format. *Technical report*. Cited in page 89

Gaster and Howes (2012) Benedict R. Gaster and Lee Howes. Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck? *Computer*, 45:42–52. ISSN 0018-9162. doi: 10.1109/MC.2012.257. Cited in page 2, 75

Suettlerlein *et al.* **(2013)** Joshua Suettlerlein, Stéphane Zuckerman and Guang R. Gao. An implementation of the codelet model. In *Proceedings of the 19th International Confer-*

ence on Parallel Processing, Euro-Par'13, pages 633–644, Berlin, Heidelberg. Springer-Verlag. ISBN 978-3-642-40046-9. doi: 10.1007/978-3-642-40047-6_63. URL http://dx.doi.org/10.1007/978-3-642-40047-6_63. Cited in page 75

Zhong *et al.* **(2012)** Z. Zhong, V. Rychkov and A. Lastovetsky. Data Partitioning on Heterogeneous Multicore and Multi-GPU Systems Using Functional Performance Models of Data-Parallel Applications. In *2012 IEEE International Conference on Cluster Computing (Cluster 2012)*. Cited in page 2, 75