

# 博士論文

A Framework for Performance Analysis and  
Optimization for GPU Kernel Programs using  
Linear Performance-Breakdown Model

(線形性能分解モデルを用いたGPUカーネルプログラムの性能解析と最適化のためのフレームワーク)

Chapa Martell Mario Alberto

チャパ マルテル マリオ アルベルト



**A Framework for Performance Analysis and  
Optimization for GPU Kernel Programs using  
Linear Performance-Breakdown Model**

線形性能分解モデルを用いたGPUカーネルプログラムの  
性能解析と最適化のためのフレームワーク

by

Chapa Martell Mario Alberto

Submitted to the Department of Electrical Engineering and  
Information Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

UNIVERSITY OF TOKYO

September 2014

© Chapa Martell Mario Alberto, MMXIV. All rights reserved.

The author hereby grants to The University of Tokyo permission to  
reproduce and to distribute publicly paper and electronic copies of  
this thesis document in whole or in part in any medium now known or  
hereafter created.

Thesis Supervisor: SATO Hiroyuki Associate Professor



# A Framework for Performance Analysis and Optimization for GPU Kernel Programs using Linear Performance-Breakdown Model

線形性能分解モデルを用いた GPU カーネルプログラムの性能解析と  
最適化のためのフレームワーク

by

Chapa Martell Mario Alberto

Submitted to the Department of Electrical Engineering and Information Science  
on June 02, 2014, in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy in Computer Science and Engineering

## Abstract

Graphics Processing Units (GPU) have evolved into computational devices with Tera-flop performance capabilities. However, a programmer writing code for GPUs faces a more challenging task than when programming for a CPU. First, it is necessary to correctly identify parallel computation. Second, Optimize the placement and movement of data in the multi-level memory hierarchy of the GPU. In addition, a program that has been optimized for one architecture might fail to reach the expected performance on a different GPU architecture. The described scenario leads to a situation where large amounts of time and effort are needed to produce a correct and efficient GPU program. GPU application programming can be aided by performance models and frameworks, and in this document we describe our performance modeling framework for GPU programs. We propose the Linear Performance-Breakdown Model (LBPM), a tool designed to help the programmer to locate performance bottlenecks, facilitating the optimization process and a framework to apply the LBPM model. The objective of the framework is to provide a performance tuning tool to facilitate the optimization of GPU kernel programs. By extracting the breakdown of the execution time of kernel programs into three main components (global memory transfers, local memory transfers and floating-point operations time) the model can serve as a tool to guide optimization efforts. We demonstrate the effectiveness of our model to calculate the breakdown of performance by applying it to several case-tests: SGMM, FFT, Reduction. We confirmed the modeling methodology works with two different GPU devices: A8-3870 AMD Accelerated Processing Unit (GPU) and a GTX 660 Nvidia GPU.

**Keywords:** GPGPU, Performance Modeling,



## Acknowledgments

I would like to express my deep gratitude to Professor Sato Hiroyuki, my research supervisor, for his patient guidance, useful critiques and invaluable feedback for this research work.

I would also like to extend my thanks to the CONACyT and the SEUT Fellowship for founding of my studies in Japan.

I wish to thank my parents and my wife for their support and encouragement throughout all the stages of my life.

Finally, Special thanks should be given to Luis Carlos Castro Madrid, who tended his hand to me when I was in great need, and if not for his help, things would have been very different.

THIS PAGE INTENTIONALLY LEFT BLANK

# Contents

<b>1</b>	<b>Introduction</b>	<b>15</b>
1.1	Example Matrix-Matrix Multiplication . . . . .	18
<b>2</b>	<b>Literature Review</b>	<b>21</b>
<b>3</b>	<b>Background</b>	<b>25</b>
3.1	GPU architecture and OpenCL programming model . . . . .	26
3.2	Discrete and Integrated GPU . . . . .	28
3.3	Performance prediction . . . . .	29
3.3.1	Profile-based prediction . . . . .	30
3.3.2	Simulation-based prediction . . . . .	31
3.3.3	Analytical Modeling . . . . .	32
3.3.4	Modeling with Scalar Parameters . . . . .	33
3.3.5	Modeling with Functions . . . . .	33
3.3.6	Statistical Models . . . . .	34
<b>4</b>	<b>Linear Performance Breakdown Model</b>	<b>35</b>
4.1	Model Design . . . . .	35
4.1.1	Calculating Kernel Execution Time . . . . .	35
4.2	Modeling Methodology . . . . .	36
4.2.1	Performance Factors . . . . .	37
4.2.2	Integrating performance factors . . . . .	41

4.3	Performance Analysis of Tiled Matrix-Matrix Multiplication based on LPBM . . . . .	42
4.4	Framework modules . . . . .	43
4.4.1	Auto-Profiling Module (APM) . . . . .	43
4.4.2	CLParser Module (CLPM) . . . . .	44
4.4.3	Linear Regression Module (LRM) . . . . .	44
4.4.4	Performance Breakdown Module (PBM) . . . . .	44
<b>5</b>	<b>Experiments and Evaluation</b>	<b>45</b>
5.1	Model Evaluation . . . . .	45
5.1.1	Experiments Setup . . . . .	45
5.2	Experiments and Results . . . . .	46
5.2.1	Results Analysis . . . . .	48
<b>6</b>	<b>Conclusions and Future work</b>	<b>53</b>
6.1	Conclusion and Future Plan . . . . .	53
6.1.1	Future Directions . . . . .	54

# List of Figures

1-1	Our proposed framework. The programmer supplies the kernel and host codes and the output of the system is the breakdown of performance.	19
3-1	General block diagram of a GPU device. It is at its most basic level a collection of Compute Units, which in turn are constituted by an array of Processing Cores. The different levels of the memory hierarchy are also shown. . . . .	26
3-2	Correspondence between OpenCL concepts and the GPU hardware. .	27
3-3	Block diagram that highlights the memory connections and placement of an integrated GPU . . . . .	28
3-4	Block diagram showing a discrete board GPU and the key memory connections . . . . .	30
4-1	Wavefronts are the smallest scheduling unit. In the figure we can observe a single work-group composed by two wave-fronts. Because the work-group-size is not a integer multiple. . . . .	39
5-1	Comparison of the experiments profiling versus the model fitting for the SGMM as executed in the APU device. . . . .	47
5-2	Performance Breakdown graph of the SGMM kernel code. The graph shows the execution time with performance breakdown versus tile size.	48

5-3	Results from a Matrix-Matrix Multiplication kernel program executed in the APU device. The left size shows the accuracy of the model approximation. The right side show the performance-breakdown bar graph. . . . .	49
5-4	Results obtained from a FFT kernel program executed in the APU device. The left size shows the accuracy of the model approximation. The right side show the performance-breakdown bar graph. . . . .	50
5-5	Results obtained from a FFT kernel program executed in the GTX device. The left size shows the accuracy of the model approximation. The right side show the performance-breakdown bar graph. . . . .	50

# List of Tables

4.1	Summary of performance predictors . . . . .	41
5.1	GPU Characteristics . . . . .	46

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 1

## Introduction

General Purpose computation with Graphics Processing Units (GPGPU), also known as GPU computing have become an important technology in the High Performance Computing community. The reason is that GPUs, through its evolution from a fixed-pipeline, graphics-processing specialized device to a more flexible, General-Purpose processor, it presents several advantages:

- High arithmetic throughput
- Low price-to-performance ratio
- High availability

Since 2002 with the development of programming models and the evolution of graphics hardware, programming environments like CUDA and OpenCL were developed[16, 7]. Because of this simplified and more general programming environments compared to the domain-specific graphics-related tools and languages, more scientists joined the field and an active research and development community was formed. These programming environments and communities have constantly evolved to make GPU programming more accessible and enabling features that allow its application in more areas. GPUs are highly sophisticate hardware designed to create and manage thousands of parallel processes which is the source of their high computational throughput, nonetheless this sophistication is carried out at the expense of great architectural complexity.

Due to the complexity of the GPU architecture and different operation when compared to CPU, it is difficult and time consuming to develop GPU programs that make full use of the potential of the hardware they run on (i.e. high efficiency). We recognize the need of attacking the problem of the difficult and time-consuming task of producing optimal GPU kernel programs and attack it by developing a tool that can be used as a guide to understand the behavior of an application when executing in a GPU device.

Because of the increasing gap between processor-memory performance, the execution time of programs in modern computers is largely dominated by the data transfers times. This fact is true also in the case of the GPU. Not different when using GPU, the only difference is that the memory hierarchy in the GPU is a multi-level structure that combines different memory technologies: The global memory, available to all the computing units in the GPU, off-chip memory implemented using GDDR technology. Global memory is large, with capacity in the order of GB, but is the slowest memory. The next level is the local memory, on-chip memory that resides inside each computing unit. Implemented with SDRAM technology, it is faster than global memory by an order of magnitude but also much more limited capacity, around 32 kB. The last memory hierarchy level is the private memory, the closest memory to the processing units, the fastest and smaller memory type. Because of their limited size, for shared and private memory, not only the access pattern but also the amount used by the kernel program greatly influence the execution time. There exist several tools for profiling GPU kernel programs. However, all of them are specific to certain platforms and devices, which makes their use difficult.

- **AMD APP profiler (deprecated since November 2013) and AMD CodeXL:** Available for AMD devices on Microsoft's Windows Operating System and some Linux operative systems (only rpm and deb packages are available from the official web page.) [12]
- **Nvidia Visual Profiler:** A tool for GPU kernel programs written in CUDA C/C++. Available for Linux, Mac OS X and Windows. [8]

- **Parallel Nsigth:** Another tool for Nvidia GPUs. There are two versions, the free version does not include the kernel program analyzer which is distributed in the pay version. To use all the features of this tool, it is necessary to use either, two machines connected over the network or a single machine with at least two GPUs. [8]

To solve the presented problem, we propose the division of the execution time of a kernel program into three mayor components: The transfer time between global and local memory (Global-to-Shared Transfer *G2ST*), the transfer time between the local and private memory (Shared-to-Private Transfer *S2PT*), and the time for floating-point operations (Processing Units Time *PUT*). In order to extract the breakdown into the three performance factors of execution time of a GPU kernel program and provide guidelines to find out the bottlenecks of performance. The core of the model is the incorporation of three elements, the Global-to-Shared Memory Latency, Shared-to-Private Latency and Processing Units Latency. These three factors are integrated into a performance model formula by applying the Normalized Least Squares Method (*NLMS*) and the resulting parameters are used reveal the effects of each element in the total execution time of a kernel routine. The block diagram of the proposed system can be observed in figure

The proposed system features the following sub-systems:

**Auto-Profiling Module (APM)** This module generates a set of runs to collect profiling data (execution time against work-group size). It makes use of both, kernel and host code files.

**CLParser Module (CLPM)** Takes the host and kernel program codes and generates an Abstract Syntax Tree for each. The trees are transversed in order to locate the key control loops, variables and function calls; information used to discover movement of data between the different levels of the memory hierarchy.

**Linear Regression Module (LRM)** This module uses the profiling information and the performance formula to calculate the Regression Coefficients by applying the Least Squares Method with non-negativity constraint.

**Performance Breakdown Module (PBM)** The last module calculates the performance-breakdown using the Regression Coefficients and the performance formula.

In general, the input to the system is the files containing the host and kernel code. The output after being processed by the different modules will be a graph showing the performance breakdown. The main contributions of this thesis include:

- The Linear Performance-Breakdown Model
- Semi-automatic performance model generation
- Integration into a script-based framework

The model shows good accuracy. The breakdown of the performance corresponds with the theoretical knowledge. In each case the LBPM shows a good correspondence with the theoretical knowledge.

## 1.1 Example Matrix-Matrix Multiplication

The remaining chapters are organized as follows: In Chapter 2 we discuss the related works and compare them to our proposal. Chapter 3 includes a review of the OpenCL programming model, a brief introduction to General Purpose computations on GPU (GPGPU) and performance modeling. Chapter 4 describes our modeling methodology and the LBPM framework. The results are discussed in Chapter 5 and finally, Chapter 6 includes the conclusions and the discussion of future work.

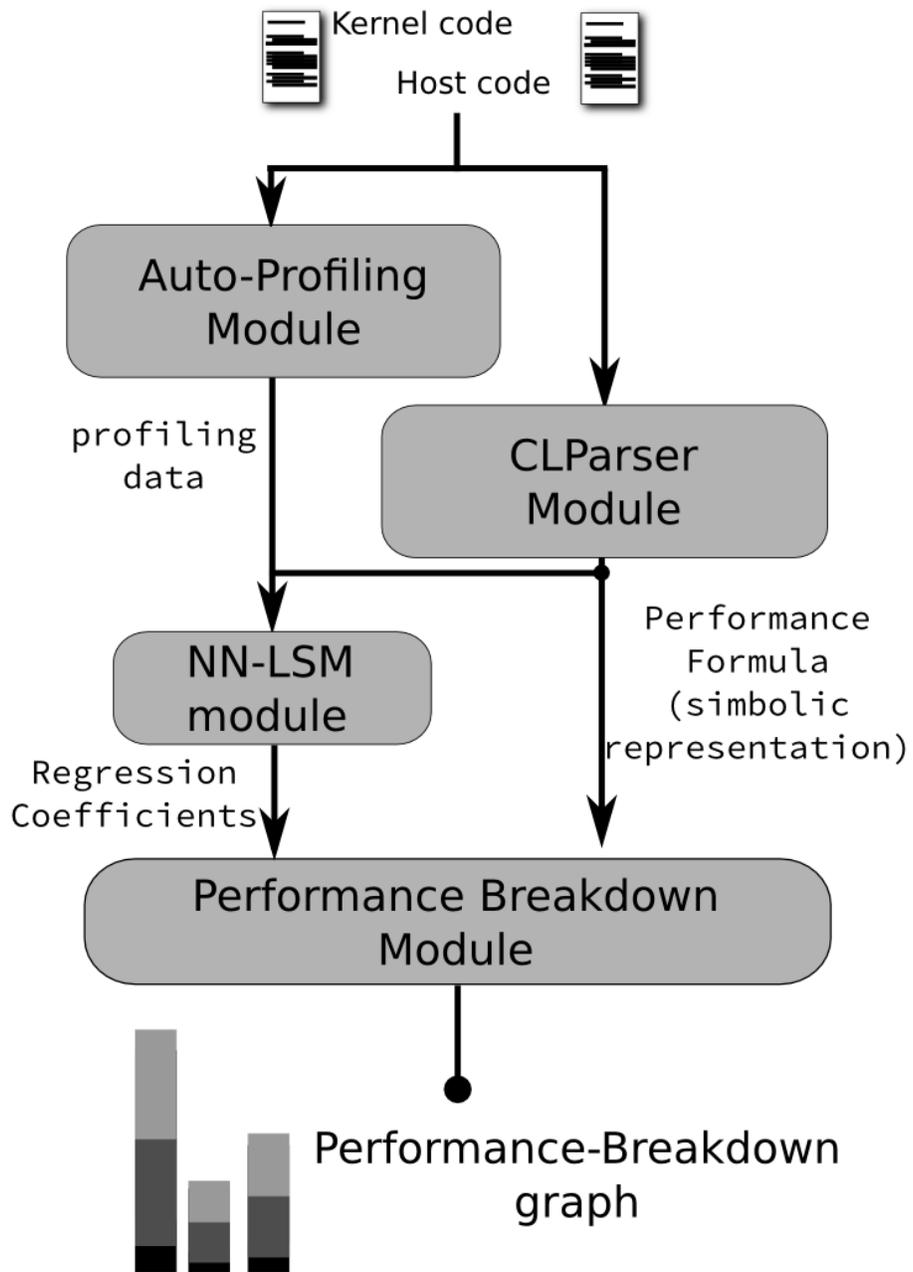


Figure 1-1: Our proposed framework. The programmer supplies the kernel and host codes and the output of the system is the breakdown of performance.

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 2

## Literature Review

In this chapter we present the existing work related to this research.

In the landscape of GPU programming, the main purpose why GPU is preferred over CPU for computation is the high performance it can offer paired with the lowest energy consumption. It seems natural that most of the effort in the development of the field has been devoted to application development. Companies like Nvidia and AMD focus all their efforts into the improvement of the hardware and the GPU architectural details. Users of the technology are mainly interested in developing

Although there have been a large amount of effort devoted to application development and algorithms optimization in the GPU community, the effort devoted to performance modeling and analysis tools is not as prominent. Whilst there are several works on the literature regarding performance models for GPU, it is worth mentioning that a large share of them focus on the CUDA programming model and hence applicable only to Nvidia GPUs [18, 21, 23, 26]. Furthermore, many of them make use of hardware performance counters and specific characteristics of the GPU architecture. The drawback of such approach is that the models and tools can only be applied to Nvidia GPUs and there is no warranty that they might be valid for future generations hardware. However, this studies provide insight about the inner workings of Nvidia GPUs that serve as a base for understanding the general architecture of the GPU device, allowing us to extract this insights and applying it to our own model. Most of the available works are designed to target a specific type of GPU and a specific task.

In [30], the authors present a micro-benchmark derived model that works to identify the bottlenecks of kernel programs in the Nvidia's GeForce 200 series. The authors present a model based on the GPU native instruction set that achieves a low error rate. However, the disadvantage of this approach is that cannot be applied to devices other than the GeForce 200 series.

In [19] the authors relate to the lack of a GPU performance model and recognize the difficulties it derives like the possibility of evaluate the suitability of the GPU to solve a particular problem. Likewise, [27] address the importance of having access to a modeling framework recognizing the fact that, for GPU programs, developers should devote large amounts of time to write a program that produce the correct results, and utilizing the hardware to its best performance its a more time-consuming task. The result of the aforementioned works is a framework to generate predictive models that allows the comparison between GPU and CPU performance in a per-application base. We chose instead to focus the analysis in the GPU architecture, since we believe that at this point it is clear that if the task exhibits a good amount of parallelism, the GPU will present better performance than CPU.

Additionally, the authors base their modeling strategies in performance counters and other metrics that are available in the CUDA programming model. The disadvantage of this fact is only GPU produced by Nvidia support CUDA. Other authors build their models based exclusively on the CUDA programing In 2009, Hong et al. [26] presented a study on GPU power consumption and performance modeling. In their study, the authors demonstrate the development of an analytical model based on an abstraction of Nvidia Fermi architecture and then execute the related experiments to validate the model. In contrast, we first design and execute the set of experiments that provide us with the execution times that will serve as input for our model. Then we make use of our model to decompose the costs of the three mayor performance predictors (floating-point computations, Shared-to-Private and Global-to-Shared memory transfer costs), while maintaining a device-independent approach by identifying the most important aspects of a kernel program execution as it is processed in general by any GPU device.

In [22], the author proposes a model for execution-less performance modeling for linear algebra algorithms in CPU machines. The authors develop their model focusing on *L1* cache misses, and analyze the correspondence between their model and the experimental results obtained in a Barcelona AMD CPU and an Intel Penryn. Contrary to the CPU where the cache is transparent to the programmer, the memory hierarchy is explicitly managed by the programmer in the GPU. For that reason, our model output, a Performance Breakdown Graph shows how much of the execution time is being spent between the various levels of the GPU memory hierarchy.

In [29] the author discusses the effects of factors such as sequential code, barriers, cache coherence and scheduling in general shared memory multiprocessors. The author starts from Amdahl's law and analyzes shared memory systems (GPUs belong to this classification) to derive several models, one for each separate factor. Our approach is instead to combine the most important factors into a single equation using a special case of a shared memory system and apply then the LSM method to evaluate the impact of each factor.

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 3

## Background

Currently, there exist a limited number of programming environments to develop programs for GPU. Each environment consist of a ecosystem of necessary tools to write program code, compile and produce executable files. The available environments are Microsoft's DirectX, NVIDIA's CUDA and Kronos group's OpenCL. DirectX is mainly targeted to develop three dimensional video game applications, and it is limited to Windows Operating System. CUDA is designed specially for GPGPU applications and it is limited to NVIDIA GPUs. Finally. OpenCL is designed to be compatible with a wide range of accelerator, multi-core devises including GPU and to be cross-platform. For this research we chose to use the OpenCL programming model for its universality; a kernel program written using the OpenCL standard can be executed in any compliant GPU, independently of the vendor or the device's family. Additionally, to empathize this property, our model do not require use of internal performance counters or other device-specific metrics, which improves its usability. The remaining of this chapter provides background on the GPGPU technology, and system performance modeling that will be discussed in later chapters.

## 3.1 GPU architecture and OpenCL programming model

In this section we briefly describe the OpenCL programming model and its components are mapped to the different functional units in a GPU. This is necessary in order to provide understanding on how GPU are constituted and the particular choice we made of the three main performance factors.

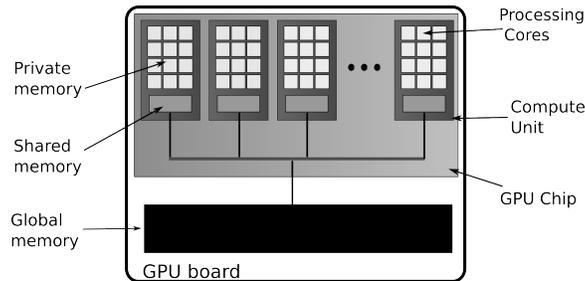


Figure 3-1: General block diagram of a GPU device. It is at its most basic level a collection of Compute Units, which in turn are constituted by an array of Processing Cores. The different levels of the memory hierarchy are also shown.

Figure 3-4 shows the GPU architecture at its basic level. GPUs are made up of hundreds of processing cores that allow a high level of concurrency. In modern GPUs architectures, the processing elements have a two-level hierarchical architecture [10, 25]. The top level is made of vector processors, called Compute Units (*CU*) that operate in a Single Instruction Multiple Data (*SIMD*) fashion. In the next level, each vector unit contains an array of processing elements (*PE*). All the PEs inside a CU are able to communicate through an on-chip, user-managed memory known as shared memory. When a work-group is created within a program, all work-items in that work-group are assigned to the same CU. The purpose is to ensure scalability, allowing a program to run across different generations of GPUs with different number of CUs. Although the vector processors can process an arbitrary number of work-items within some constraints, the scheduler cannot schedule an arbitrary number of work-items for execution. It is always in groups of 64 work-items is known as *Wave-front* (*WF*), the smallest scheduling unit. This means that if some number smaller than 64 work-items

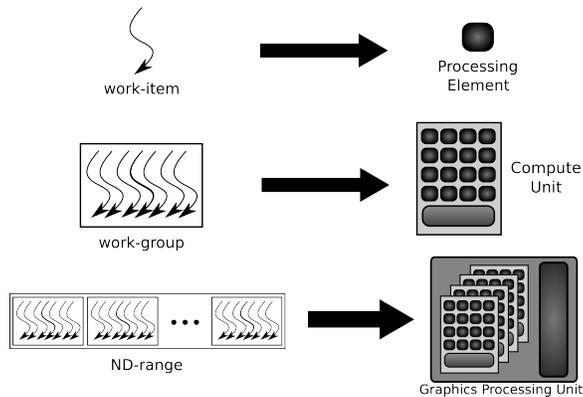


Figure 3-2: Correspondence between OpenCL concepts and the GPU hardware.

need to be executed, the schedulers in the CU will still create 64 work-items and since not all the work-items have useful work to do, some of them will be idling. The shared memory is a portion of memory that can be accessed by all PE in a CU, hence the name. This memory is slower than the register file, but it is also bigger, and most importantly, it is accessible to all work-items in the same work-group. All CUs in the GPU have access to the global memory that is the largest and slowest of all the memory types in a GPU. Figure 3-2 shows a description of how the different concepts in the OpenCL standard have a relationship with a specific GPU concept.

When a kernel is submitted for execution a index space containing a defined number of software threads (called work-items in the OpenCL terminology) is defined. Because the GPU operates with the SPMD model, the index space purpose is to provide a mean for work-items to distinguish from each other and know which data elements they must operate on. It is responsibility of the programmer to ensure each work-item is correctly indexed and to divide the task. GPU and CPU support threads in very different ways. The CPU has a small number of registers per core that are used to execute any task given to the CPU. Context switching on CPUs is expensive in terms of time because the state of all registers for the current task must be saved to RAM and the register states for the next task must be loaded from RAM. On the other hand, GPUs have multiple banks of registers, and each bank is assigned to a different task. A context switching involves a bank selector instead of expensive

RAM loads and stores, making context switching less time-consuming in a GPU. The impact of this mechanism is that the GPU uses that ability of fast context-switching and the availability of a large number of work-items to hide data latencies. However, it is important to know that the number of register banks is limited, and it imposes a limit to how many Work-items can reside in a Computing Unit. Work-items are grouped unto a larger logical instance called Work-group. The most important aspect of work-groups, is that communication is enabled across work-items only if they belong to the same work-group.

### 3.2 Discrete and Integrated GPU

In this work we make use of two different GPU devices that have different architectures: The first device is an AMD Accelerated Processing Unit (APU) and the other one is a NVIDIA GeForce series GTX 660 device. The main distinguishing characteristic of these devices is that the APU is a Integrated GPU whereas the GTX is a discrete board.

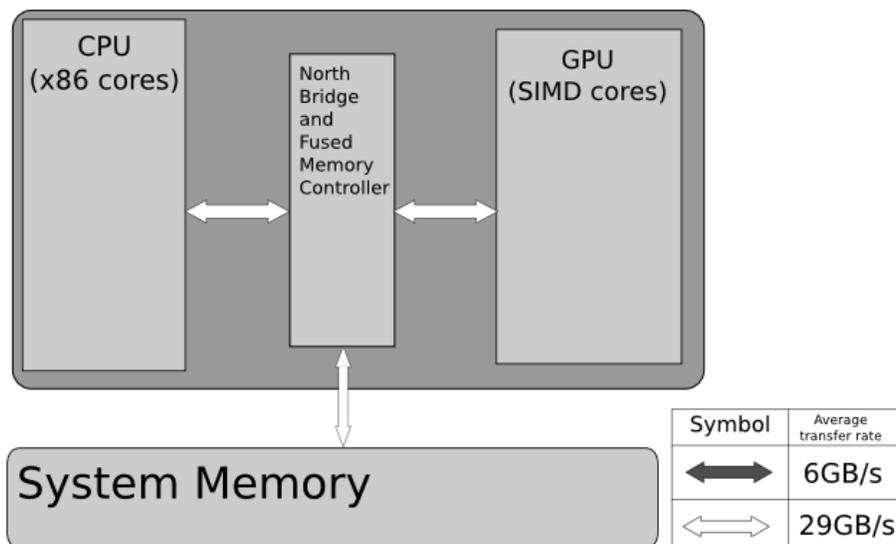


Figure 3-3: Block diagram that highlights the memory connections and placement of an integrated GPU

The APU is an example of integrated CPU-GPU architecture. The GPU cores are

built into the same silicon chip as the CPU cores. Integrated designs like the APU do not need to make use of the PCIe port to transfer data between CPU and GPU memory spaces. Instead, the main system memory is used for both, CPU and GPU data. The only operation that needs to be performed is reserving a memory block for exclusive GPU use when a GPU program is launched. When data between GPU and CPU must be shared, data is transferred from one RAM location to the other. If pinned memory is used, it is possible to make a memory mapping, where the GPU and CPU can read and write to the same memory area, the transfer step is skipped and the cost of the transfer is now the overhead of the mapping which is usually much lower than the time of an actual transfer.

The GTX 660 is a GPU chip that is contained into its own electronic board. This board is attached into the host computer system using the PCI express port. The advantage of a discrete board is that there is no space nor transistor count limits for the GPU chip. Discrete boards have their own on-board memory storage, typically implemented using GDDR technology and ranging from 512 MB up to 2GB on commercial-level boards[7]. Because the space constraint is not present, discrete boards have a higher peak performance compared to discrete boards. Nonetheless, since they are located from a far distance from the CPU and the main system memory, in addition to having to cross the PCIe port, the time needed to transfer data between the on-board GDRAM and the main system memory is significant. Such impediment is one of the major reasons GPU technology cannot be widely adopted to accelerate any arbitrary computation. If a program requires a memory transfer with a small quantity of data, the long transfer memory time overcomes the advantage of the high performance offered by GPU.

### **3.3 Performance prediction**

Regarding the field of computer science, performance prediction refers to the process of estimating the execution time of a program executed on a determined computing system. Performance prediction is used to evaluate new computer designs, analyze

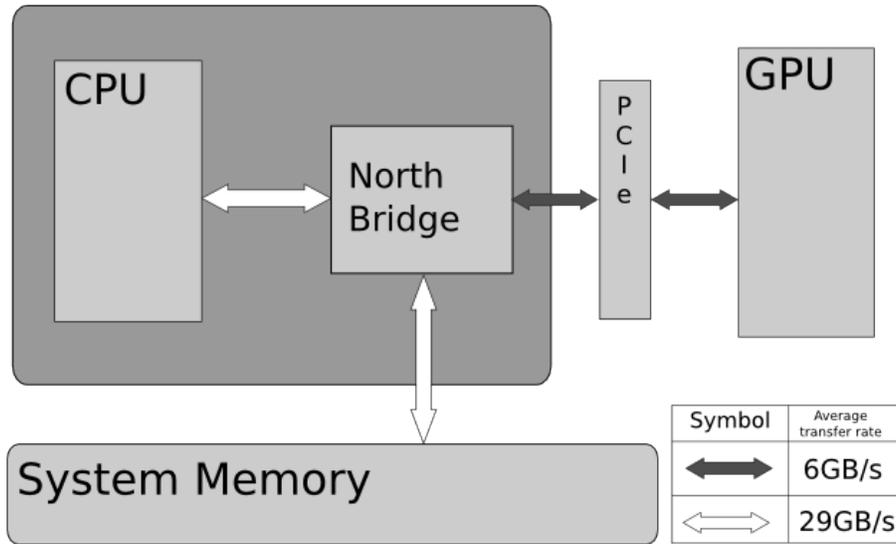


Figure 3-4: Block diagram showing a discrete board GPU and the key memory connections

compiler optimizations, and code tuning. Performance prediction can make use of several performance metrics, like cache misses/hits, instruction counts, branch prediction misses/hits, etc. to calculate and estimate execution time.

There are several approaches to predict performance on computers, here we summarize three of them [2]

- Profile-based prediction
- Simulation-based prediction
- Analytical modeling-based prediction

### 3.3.1 Profile-based prediction

The simplest of all approaches to performance prediction. In this approach, the program is seen as a set of basic blocks that follow an execution path. The execution time of a program is calculated as the sum of the execution time of each block multiplied by number of times the block is executed. The information about basic blocks is generated using a tool called profiler, that generates data pertaining the execution

time of blocks from instruction set information and also the execution frequency of each block.

Profile-based prediction is useful for single instruction issue, in-order execution processors, but for modern processors the approach is not accurate. Modern processors feature advance optimization techniques such as multiple instruction issue, branch prediction and out-of-order execution, These techniques dramatically cause alterations in the execution time of programs that cannot be accounted for using the concept of basic blocks, rendering profile-based prediction ineffective.

### **3.3.2 Simulation-based prediction**

Involves the creation of a computer program that can generate performance data by representing the key characteristics of the computer system being studied like functional units, instruction sets, etc. Each instruction of the target program is actually dynamically executed given a particular input data set. The model represents the computer system and the simulation represents the operation of the system over time. Simulators have the capacity of predicting program's performance very accurately. However when the size of the program being evaluated increases, the simulation time can exceed tractable simulation time.

Trace-based simulation is another approach pertaining simulation-based prediction. It requires the creation of a trace-file which store important program events. Because only key elements of the program run are stored in the trace file, the simulation does not include every instruction, reducing the simulation time. The problems with trace-file simulation is that it loses flexibility and accuracy but in some cases the faster completion time is necessary. One more disadvantage is that the generation of the trace file often consumes considerable amounts of storage space and can severely impact the runtime of applications if large amount of data are recorded during execution.

### 3.3.3 Analytical Modeling

Analytical modeling techniques abstract the features of a parallel system as a set of parameters or parametrized functions. It is worth mentioning that analytical modeling have been widely used for the modeling of parallel systems. A mathematical model of the system that is being studied will provide an abstract representation of both, the hardware and the software. The user is responsible for determining the model parameters, which requires detailed knowledge of the system and the modeling technique, this will greatly determine the accuracy and robustness of the model. The disadvantage of this approach is that the models are not as accurate as real program executions due to simplifications generated when creating the abstract model, details are abstracted away in the modeling process. In general, analytical modeling have a low computational cost. The only exemption is when a statistical process require long simulation times.

The development of analytical modeling involves two decisions that will affect the accuracy of the model. The first one is the system detail level, known as level of abstraction. It is determined by the parameters accounted in the model and their representation. A model with more parameters have a lower level of abstraction, is a more detailed model. therefore increasing the modeling cost. Regarding the representation of system parameters, scalar parameters are simpler than functions and statistical tools, but less flexible. Functions require that the programmer determines not only their coefficients, but also their shape. Statistical tools require a specialized knowledge that average parallel programmers do not have. Another important factor is the difficulty of determining the parameter values. If the parameters are too specific, they can be hard to capture due to a lack of suitable tools or knowledge about the application. If the parameters are too high-level, it is necessary to determine representative behaviors of the system.

When developing analytical models, there are three types of approaches that differ by how the models are expressed. In the next paragraphs a description of each analytical modeling strategy is described.

### 3.3.4 Modeling with Scalar Parameters

This approach uses a set of scalar parameters to model the behavior of a parallel system. These parameters express the average behavior of the parallel system under specified conditions.

In [9] the authors process communication in modeling parallel systems with the *LogP* model. In that model, there are four main parameters that are used to characterize the target system:  $L$  Latency of operations,  $o$  communication overhead,  $g$  the gap between operations, and  $P$  as the memory and processor units. LogP considers communication costs and the number of processors involved in the computation, improving the accuracy of the predictions. This framework is useful for evaluating parallel systems over various paradigms, because its parameters are machine independent. However, it imposes heavy restrictions and cannot be used when complicated events like saturation of networks links have significant impact on the system's performance.

The use of scalar parameters to characterize parallel systems simplifies significantly the modeling of systems, but at the same time, parameters can be difficult to correctly determine. If the parameters are not correctly set, or if important parameters are left out, the model will be inaccurate. Even if the scalar parameters are not difficult to obtain, they represent simplified assumptions about the behavior of the system, which will be a source of inaccuracy. Models produced with this technique are the result of the modeler's understanding about the system being characterized, so these modeling techniques require that the modeler possess an adequate understanding of the underlying architecture to successfully define model parameters.

### 3.3.5 Modeling with Functions

Using functions improves the flexibility and expressiveness of the models, but can also increase the complexity of the models, due to the need to determine the shape and coefficients of the functions. Scalar parameter models are a simplified case of modeling with functions, where the functions are constants.

An experimental approach for performance optimization is presented in [5] the authors propose a method where the idea is to have a library containing various implementations of common procedures and their models. When including several libraries there will be a certain probability of finding one that performs best in the tested architecture. The model consist of environment variables, sets of values for the environment variables, and the possible terms that will compose the formula model. The models are regression fittings of the terms given by the user to some profiling data. The results indicate a good prediction of the best implementation for a given architecture, but the numerical predictions are not accurate.

### 3.3.6 Statistical Models

One more approach to modeling makes use of statistical modeling tools as variable distributions and Markov models. It differs from the scalar parameter approach in the sense that the parameters are statistical tools.

A measurement-based model of the execution of computationally-bound parallel applications is presented in [13]. Monte Carlo simulation is used to solve the model and predict the completion time distribution of the given application with the measured workload. The constructed model is used to evaluate scheduling policies, performance effects of multiprogramming, and scalability of real workloads using Markov models to express the effects of execution parameters.

We should note that these statistical parameters try to represent the asymptotic behavior of the modeled systems. They are usually used to analyze parallel systems whose workload characteristics are well known, being particularly useful for evaluating architectures across a wide range of applications. Nonetheless, Markov models and Petri nets are not as simple to use as mathematical functions or variable distributions. As with other analytical techniques, statistical models do not give any explanation of system behavior, and also require the user to express his understanding about the parallel system behavior as parameters to the statistical tool.

# Chapter 4

## Linear Performance Breakdown Model

In this chapter we describe the methodology upon which our model is built, as well as a description of the modules that make up the LBPM framework used to apply our model

### 4.1 Model Design

In this section we first briefly discuss how the total execution time of a kernel program as executed in a GPU device; then we describe how the model was developed and the characteristics of the GPU hardware that have a major impact on the behavior of the performance curve.

#### 4.1.1 Calculating Kernel Execution Time

A simple formula for modeling the execution time of any given algorithm in a computer system can be obtained by dividing the number of clock cycles required by the algorithm and the duration of one clock cycle. For simplicity, we measure elapsed time instead of clock cycles. Taking one step ahead in the analysis of computation time, The time required to complete an algorithm can be divided into the time spend

into performing numerical computations and time spend copying data from one level of the memory hierarchy to another as shown in equation 4.1.

$$RunningTime = ComputationTime + DataTransferTime \quad (4.1)$$

A more refined model suited for GPU computations considers that the data transfer inside a GPU is divided into two categories. The First category refers to the data transfers from global memory to shared memory inside the GPU, then from the shared memory to the registers of the processing cores. In our model we do not consider transfers between system memory and global memory because we assume zero-copy data buffers, the main advantage of an APU system [11]. An important observation is that it is not possible to measure with perfect accuracy the time elapsed for either the floating point operations or the memory transfers. Taking this into consideration, we obtain equation 4.2 that its a linear combination of three terms.

$$RunningTime = \alpha_1 \cdot ComputationTime + \alpha_2 \cdot LocalMem.TransferTime + \alpha_3 \cdot GlobalMem.TransferTime \quad (4.2)$$

The  $\alpha$  parameters in the equation define the weight of each individual term of the equation. They also help to interpret the breakdown of the performance, indicating which component represents a greater contribution to the total processing time so that optimization effort can be applied in the correct direction.

## 4.2 Modeling Methodology

To produce our model, we apply regression modeling to obtain estimates of the most important performance factors. Making use of the execution time values as input, our model then computes a collection of three parameters; each of these parameters corresponds to one of the three steps in the computation of a general purpose pro-

gram in the GPU, we propose the division of the execution time of a kernel program into three major components: The transfer time between global and local memory (Global-to-Shared Transfer *G2ST*), the transfer time between the local and private memory (Shared-to-Private Transfer *S2PT*), and the time for floating-point operations (Processing Units Time *PUT*). This process is essential to capture the performance characteristics of programs and allow us to define a suitable performance model. Each of the three performance predictor formulas are defined independently from each other and they are integrated into a linear combination formula using the Least Squares Method described above to approximate the model curve to the experimental data.

### 4.2.1 Performance Factors

The total execution time of a kernel program is dominated by three main factors derived from the inner workings of the GPU architecture. Each of the  $\alpha_n$  terms is estimated separately as described in the following discussion. As a starting point we consider a sequential processing system. In such sequential system,  $P_1$  can be estimated as the number of floating-point operations necessary to finish the task. This would yield the total amount of *Time Slots* to complete the execution of an algorithm, and by multiplying the number of time slots by the execution time of a single time slot, we can have an estimate of the computation time. However, in a multi-core system such as the GPU, operations are executed with some degree of concurrency. The key is to estimate the degree of concurrency that can be achieved by taking into account the capabilities of the hardware. To calculate it, an important parameter is the total number of wave-fronts required to execute all the work-items in a kernel launch, and in turn, the total number of wave fronts that can be executed concurrently in the GPU. There is a number of factors that affect the actual level of parallelism achievable. The resource usage per work-item (registers and shared memory) and the number of CU in a device are the most notorious. Taking this into consideration will yield an estimate of the time slots required to complete a kernel execution. The maximum number of wave-fronts that can be executed concurrently

in the GPU. To determine total amount of wave-fronts that can be executed at the same time across the GPU, we must know how many wave-fronts can be executed in a single CU. The maximum number of concurrent wave-fronts value depends on per work-group memory needs. With that information it is possible to calculate how many work-groups can reside in a CU, then the total amount of work-group that can execute concurrently across the GPU is the obtained value multiplied by the amount of CUs in the GPU. We will refer to this value as *Concurrency Level*, it is be one of the most important values that determine the execution time of a kernel program. GPUs have multiple banks of registers, and each bank is assigned to a different task. A context switching involves a bank selector instead of expensive RAM loads and stores, making context switching less time-consuming in a GPU. However, the number of banks is limited, and it imposes a limit on how many Work-groups can be executed concurrently, making this mechanism an important performance factor. In general terms, our model takes a set of execution time measurements and applies a formula that is composed of several parameters that encompass the characteristics of GPU hardware and the execution of tasks in the functional units. That realization brought important feedback about the most important parameters that define how well a kernel program will execute in the GPU

### **Processing Units Latency**

During execution not all Work-groups run necessarily concurrently. Each Work-group is assigned to a Compute Unit (CU) where it is executed until the completion of all its instructions. Each CU maintains the context for multiple blocks. Different Work-groups assigned to each CU will be swapped in and out of execution with the objective of hiding the latency of memory operations. The maximum number of blocks per CU is a combination of device properties and kernel characteristics. The most important consideration to evaluate the impact of tuning parameters in kernel programs is the performance impact of this idling is a reduced computational throughput.

In a sequential system,  $P_1$  can be estimated as the number of floating-point op-

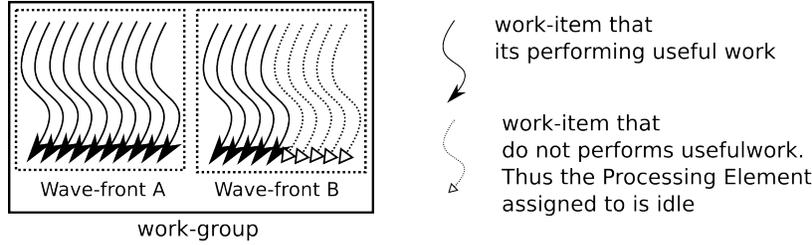


Figure 4-1: Wavefronts are the smallest scheduling unit. In the figure we can observe a single work-group composed by two wave-fronts. Because the work-group-size is not a integer multiple.

erations necessary to finish the task. This would yield the total amount of *Time Slots* to complete the execution of an algorithm, and by multiplying the number of times slots by the execution time of a single time slot, we can have an estimate of the computation time. However, in a multi-core system such as the GPU, operations are executed with some degree of concurrency. The key is to estimate the degree of concurrency that can be achieved by taking into account the capabilities of the hardware. An important parameter is the total number of wave-fronts required to execute all the work-items in a kernel launch, and in turn, the total number of wave fronts that can be executed concurrently in the GPU. There is a number of factors that affect the actual level of parallelism achievable, however the influence of such factors is not as critical as the hardware resources mentioned. Taking this into consideration will yield an estimate of the time slots required to complete a kernel execution. The maximum number of wave-fronts that can be executed concurrently in the GPU is determined by the resources needed by each work-item and those available in each CU, These resources are the number of registers and the size of the Local Data Store (LDS), a especial portion of memory inside each CU[28, 14]. We define this factor in terms of the number of floating-point operations necessary to perform the task (BigO complexity) and the level of concurrency (Eq. 4.3).

$$ProcessingUnitsTime = \frac{BigOcomplexity}{ConcurrencyLevel} \quad (4.3)$$

In turn, the level of concurrency is determined by the number of wave-fronts in the

ND-Range  $Wavefronts_{NDRange}$  and the Maximum number of these wave-fronts that can be scheduled for execution at the same time across the GPU  $Wavefronts_{MaxConcurrent}$  as shown in Eq.4.4.

$$ConcurrencyLevel = \frac{Wavefronts_{NDRange}}{Wavefronts_{MaxConcurrent}} \quad (4.4)$$

### Global-to-Shared Memory Latency

The global memory is the outermost level in the hierarchy and hence the slower memory. The number of memory transactions and how they occur from global memory to shared memory is very important. work-group size and memory access pattern have a strong impact on this factor. Global memory instructions are issued in order and a work-item will stall if the data to operate with is not ready, the GPU needs enough work-items to have a large pool of wave-fronts to swap in and out of execution to hide latency [6, 20]. However, the interaction between work-group size and performance is complex because not only there is the need for enough work-items, but also it is important to have enough memory transactions in flight to saturate the memory bus and fully utilize its bandwidth. We define this predictor in terms of the work set size  $N$ , the number of wave-fronts in a work-group  $WF_{wg}$ , and the work-group size  $WG_{size}$  (Eq.4.5).

$$G2ST = \frac{N \cdot WF_{wg}}{WG_{size}} \quad (4.5)$$

### Shared-to-Private Memory Latency

Once data has been moved to the global memory, it has to be transferred to the Private memory so it can be readily available for execution by the PE inside the CUs. Shared memory is faster to access than global memory, but access patterns must be aligned to be efficient [17]. We define this factor as a function of the total number of work-groups in the NDRange  $WG_{total}$  and the maximum concurrent wave-fronts  $Wavefronts_{MaxConcurrent}$  as show in Eq.4.6.

$$S2PT = \frac{WG_{total}}{Wavefronts_{MaxConcurrent}} \quad (4.6)$$

Once the three predictors have been defined, we integrate them into a linear combination as described in the following section 4.2.2. Table 4.1 shows a summary of the performance predictors used to calculate the LBPM model.

Table 4.1: Summary of performance predictors

$WF_{sizeinset}$	Wavefront size
$WG_{size}$	Work group size
$WF_{wg}$	Wave fronts in a work group
$WF_{ndr}$	Wave fronts in the ND range

## 4.2.2 Integrating performance factors

To integrate the performance factors we follow a general approach where the response of the system is modeled as a weighted sum of predictor variables plus random noise[4]. We have a subset of  $n$  observations for which values of the system response (execution time) are known. Let us denote the vector of responses as  $y = y_1, \dots, y_n$ . For a point  $i$  in this space, let  $y_i$  denote its response variable and  $x_i = x_{i,1}, \dots, x_{i,p}$  denote its  $p$  predictor. In our model  $p = 1, 2, 3$  each representing one of the computation steps described previously. Let  $P = 1, 2, 3$  denote the set of regression coefficients used in describing the response as a linear function of predictors as shown in Equation 4.7.  $g(x_{i,k})$  are the functions defined for each of the predictors. Each predictor function is defined from GPU architecture parameters and kernel program parameters. They are discussed in section 4.2.1.

$$f(y_i) = \sum_{j=1}^p \alpha_j g_j(x_{i,k}) \quad (4.7)$$

After this step we fit the regression model to the observations. The method of Least Squares is commonly used to identify the best fitting model by minimizing  $S(\alpha)$ , the sum of squared deviations of predicted responses given by the model from actual observed responses.

$$S(\alpha_1, \dots, \alpha_p) = \sum_{i=1}^n w_i \left( y_i - \sum_{j=1}^p \alpha_j x_{i,j} \right)^2 \quad (4.8)$$

Solving the system of equations of partial derivatives of  $S(\alpha)$  with respect to  $\alpha_j$  reveal the significance of response-predictor correlations. The  $\alpha_i$  values are used to calculate the breakdown of the total execution time into the three major components as illustrated by Eq.4.9.

$$ExecutionTime = \alpha_1 \cdot PUT + \alpha_2 \cdot G2ST + \alpha_3 \cdot S2PT \quad (4.9)$$

### 4.3 Performance Analysis of Tiled Matrix-Matrix Multiplication based on LPBM

In order to obtain the experimental data necessary to apply the LSM and validate the model, a program that executes the kernel function and profiles the execution time was used. The kernel code is shown in listing 1. The program was run with all the possible ranges of values for  $t$  for a matrix size of 1000. The  $t$  values range is from 1 to 16. It is not possible to create a tile of a width larger than 16 because currently OpenCL restricts the maximum size of work-groups to 256 elements.

ソースコード 4.1: SGMM kernel with tiling

```
float output_value = 0;
for(int m = 0; m < Width/TILE_WIDTH; m++) {
    local_tile_a[ty][tx] =
input_a[Row * Width + (m*TILE_WIDTH + tx)];
    local_tile_b[ty][tx] =
input_b[(m*TILE_WIDTH + ty) * Width + Col];
barrier(CLK_LOCAL_MEM_FENCE);
for(int k = 0; k < TILE_WIDTH; k++)
    output_value += local_tile_a[ty][k] * local_tile_b[k][tx];
}
```

```
barrier(CLK_GLOBAL_MEM_FENCE);  
output[Row * Width + Col] = output_value;
```

In our algorithm we work with single precision floating-point numbers grouped in tiles ranging from one elements to blocks of 16 by 16 elements, to store the values for our MMM kernel, we need at most  $16^2 \cdot 4$  bytes of memory for one tile. Since a work-group will be composed of three tiles, each tile need 3 kB, hence the maximum allowed number of Work-groups is equal to the maximum possible amount. With this information, it is possible to calculate how many work-groups can reside in each CU and the total amount of work-group that can execute concurrently across the GPU by multiplying by the number of CUs in the GPU.

## 4.4 Framework modules

As introduced previously in chapter 1, the proposed system features four main sub-systems or modules that carry out the extraction of the performance-breakdown from the program code files. Here we describe each module and its functions.

### 4.4.1 Auto-Profiling Module (APM)

In order to extract the breakdown of performance as with any modeling strategy, it is necessary to have both, program parameter information and machine parameter information. For the LBPM the machine information includes a list of known parameters specified in the available documentation of the hardware, as well as parameters defined in the program code. Besides that information, the actual execution time of a program is used as another machine parameter information source. That is the purpose of this module. It requires the user to substitute some predefined constants that will be specified at compile time using the gcc flag *-D*. The module then generates a set of runs to collect profiling data (execution time against work-group size). It makes use of both, kernel and host code files.

#### 4.4.2 CLParser Module (CLPM)

To determine the performance formula for each performance factor (*PUT*, *G2ST*, *S2PT*) it is necessary to analyse the kernel program. To achieve an automatic code analysis the source code is transformed into an Abstract Syntax Tree (AST), which is the first task performed by this module. This is done using the python extension *PyCParser*[3]. After generating the AST of the program code, this module locates the relevant variables i.e. those qualified as global memory-residing and local memory-residing variables, and the control loops (such as *for* loops) to determine the performance formulas. It needs the host and kernel program codes as input and generates an Abstract Syntax Tree for each. The trees are transversed in order to locate the key control loops, variables and function calls; information used to discover movement of data between the different levels of the memory hierarchy and determine the formulas for each performance factor as output.

#### 4.4.3 Linear Regression Module (LRM)

Once the profiling information is available, the regression coefficients are calculated by this module. This module uses the profiling information and the performance formula to calculate the Regression Coefficients by applying the Least Squares Method with non-negativity constraint. The process is carried out using the *R language*[15] and its module *nls*.

#### 4.4.4 Performance Breakdown Module (PBM)

The last module calculates the performance-breakdown using the Regression Coefficients obtained by the LRM and the performance formula determined by the CLPM. The final output of the system is the performance-breakdown graph.

# Chapter 5

## Experiments and Evaluation

This chapter shows the results obtained in the experiments, we discuss the interpretation of these results.

### 5.1 Model Evaluation

In this section we describe the setup for the conducted experiments to collect performance information. Then we present the results obtained from applying our modeling methodology to the obtained data.

#### 5.1.1 Experiments Setup

For our experiments we make use of two kernel programs: a Matrix Matrix Multiplication (MMM) routine and a Fast Fourier Transform (FFT) routine. We also employ two GPU devices whose characteristics are summarized in table 5.1. The results for both GPU are calculated using the same model and the correspondent measured execution times as input. To obtain the execution time information, we make use of the timers provided by the OpenCL specification to inquire the total kernel execution time. The timers have a resolution of  $1ns$ . Each was program was run 1000 times and the execution time used to train the model is an average of all the measured timings for each tile size.

## 5.2 Experiments and Results

After running the experiments and collecting the data, the LSM is applied to estimate the  $\alpha_n$  parameters in our model equation. The calculated performance from the experimental results as well as the values calculated from the model is shown in figure 2. As shown in figure 5.2, we can observe that the obtained results for the model closely match those for the experiments. The reason for the discrepancies are manifold, they can be attributed to non-uniformity in the execution of the kernel, like bank conflicts and not enough latency hiding in some cases; as well as another execution details not considered in the model like the influence of caches. Caches are relatively a new addition in the graphics hardware and were not considered in this paper for simplicity purposes. However, the accuracy achieved with the proposed model reflects that the considered parameters are those who have a major impact on the performance like the maximum number of wave-fronts that can run concurrently across the GPU and the impact of the tile size in the number of memory request that must be generated to transfer all the data elements. These factors explain the observed saw tooth pattern when the tile size is greater than 8, because in the ideal case the performance should continue scaling like the observed curve while the tile size is less than eight. The values of the parameters in the model is also useful to observe the different impact on the performance each separate component have, how much they contribute to the total time amounted for the execution of a kernel. If we breakdown the total execution times, so we can observe each term contribution, we obtain the column chart depicted in figure 3. In this figure we can observe that, as expected in a GPU device, the term  $P_1$  that corresponds to the floating-point calculation time adds a small portion of the execution time and the major portion of

Table 5.1: GPU Characteristics

<b>GPU Model</b>	AMD A8-3820	Nvidia GTX 660
<b>Clock freq.</b>	3.0 GHz	1.0 GHz
<b>Compute Units</b>	5	5
<b>Device Memory</b>	256 MB	2 GB
<b>Local Memory</b>	32 kB	512 kB

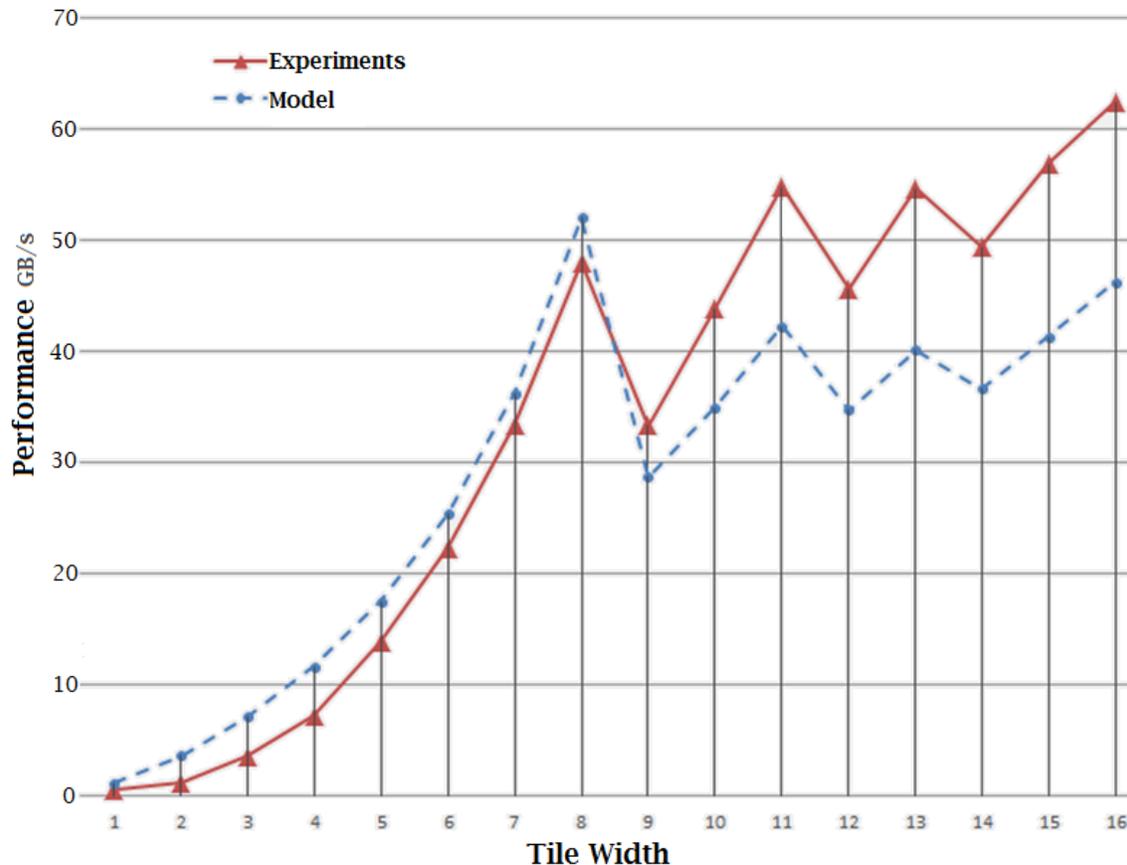


Figure 5-1: Comparison of the experiments profiling versus the model fitting for the SGMM as executed in the APU device.

the execution time is contributed by the memory transfers. Specially global memory that in the case of a tile size of 1 (i.e, no tiling applied) it amends to more than 60% of the total execution time. With the increase of the tile width, the global memory transfer time is greatly reduced for the reasons explained in section 2. It is also worth observing that at some point, the reduction in execution time is not significant anymore with larger tile widths. This hints that enabling the hardware for a larger number of threads per work group will not be synonymous of an important improving in tiling algorithms. It is also worth noticing that since the global memory transfer times is reduced for large values of tile width, an important portion of the total time is attributed to the local memory transfers. This means that a improvement on the nature of this kind of memory could bear a good impact on the performance of the algorithm. As mentioned in the previous sections, another advantage of the

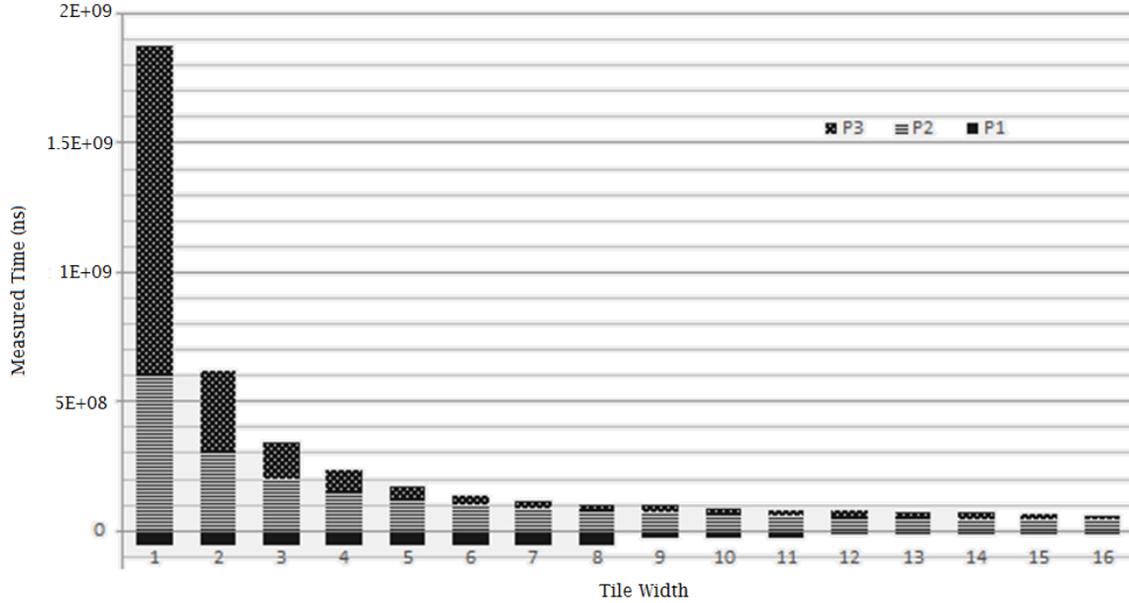


Figure 5-2: Performance Breakdown graph of the SGMM kernel code. The graph shows the execution time with performance breakdown versus tile size.

LSM method is that it provide us with the estimation for the model parameters. This parameters can be used to produce a performance breakdown graph like the one show in figure 5.2. Each corresponding  $\alpha_n$  tell to what extent each of the computation steps adds to the total computation time. This is useful because programmers can have a better understanding of where the optimization efforts can provide the best gains or to know in which steps the application is not performing as it is expected.

### 5.2.1 Results Analysis

Figure 5-3 shows the results obtained from applying our model to a MMM kernel program. On the right side of the figure it is possible to observe the performance breakdown of the execution time. In this case, as the tile size increase, the G2SL is greatly reduced. This can be explained by the global memory reads mechanism. With a small work-group size, there is a larger number of memory requests and these requests have a small number of memory words on them. The effect in the memory system is a large number of request with high latency that will fail to occupy

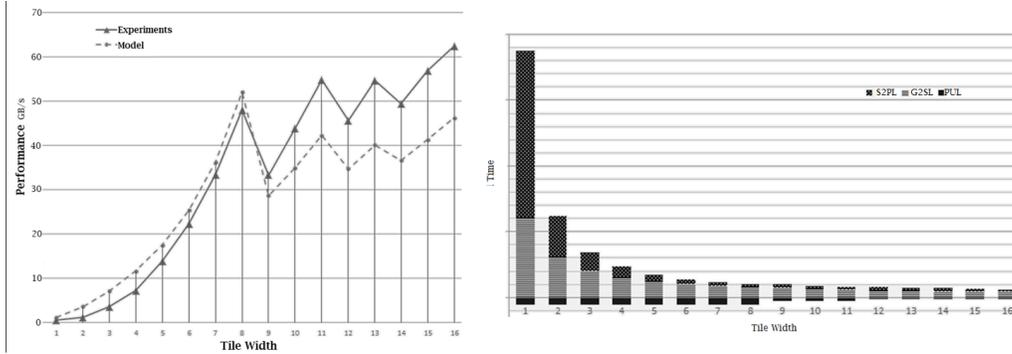


Figure 5-3: Results from a Matrix-Matrix Multiplication kernel program executed in the APU device. The left size shows the accuracy of the model approximation. The right side show the performance-breakdown bar graph.

the bandwidth of the memory channels. The left-hand sub figure figure shows the total execution times for different values of work-group sizes configuration for a fixed problem size, as in the case of matrix multiplication. Each horizontal bar is divided in three different regions, one for each performance predictor in our model. Observing and analyzing these results we find out that for a small work-group size, most of the kernel execution time is spent in global-to-shared memory transactions. The result is that most of the execution time is spent copying data to and from global memory, as our performance-breakdown graph shows. With larger tile sizes the global memory usage improves and the performance increases. The optimal performance is achieved with the largest possible tile size.

In the case of the FFT kernel program, the effects of the tile size are more complicated. While in MMM the optimal tile size was the maximum possible due to the memory transfer efficiency, in this case that does not holds true. The memory transfers improve but only until a certain point is reached, after that point, increasing the tile size degrades the performance. The model reveals a increase in the PUL computation time.

In figures 5-4 and 5-5 This corresponds with the theoretical knowledge of bandwidth being wasted with single-item memory requests to the memory controllers in that particular case of a work-group size of size 1. As the work group size increases, the relative time spent in global-to-shared memory is reduced. After a critical point,

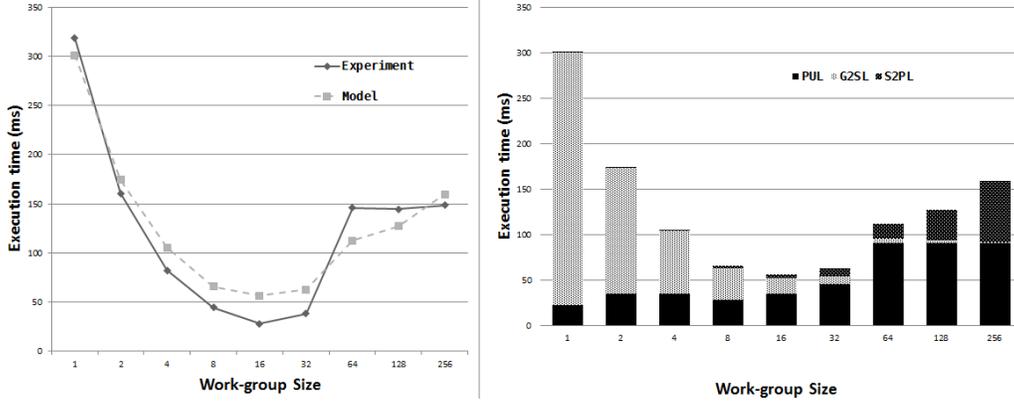


Figure 5-4: Results obtained from a FFT kernel program executed in the APU device. The left size shows the accuracy of the model approximation. The right side show the performance-breakdown bar graph.

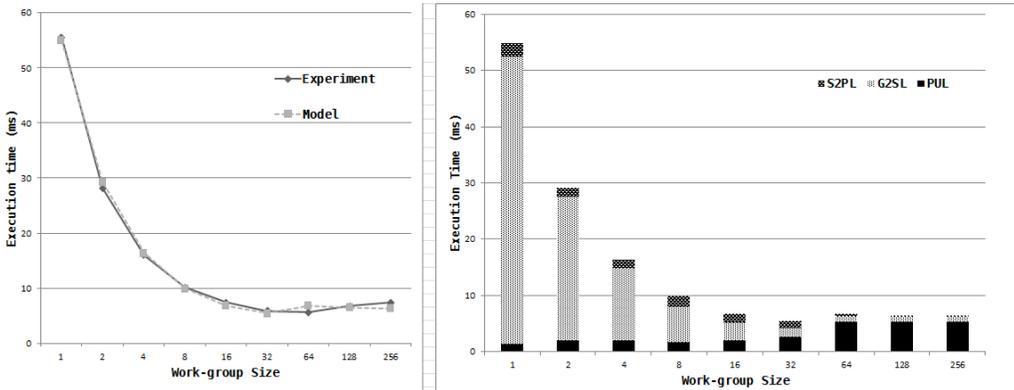


Figure 5-5: Results obtained from a FFT kernel program executed in the GTX device. The left size shows the accuracy of the model approximation. The right side show the performance-breakdown bar graph.

the execution time do not improve further but it degrades instead. This can be explained by the saturation of resources in the GPU. By increasing the number of work-items in a work-group, the occupancy of the GPU improves. this is, the PE will have more Work-items to occupy them and increase the throughput of the computations. However, each work-item require a number of registers and on-chip memory to keep operands, its stack, and other types of data required for its execution. So increasing the work-group size will improve the performance up to some point where register spilling will occur and the performance will degrade. When a work-group is large,

it will need more registers than the Compute Units in the GPU can provide, hence causing register spilling and reducing performance. In the performance-breakdown graphs for the FFT, we can observe a slight increment in the PUL after the point where the work-group size causes a performance degradation. That increase in PUL hints to a high arithmetic latency, where PE do not have enough work to process with the data they receive in order to hide the arithmetic pipeline latency. This hints to the need of a increased Instruction Level Parallelism via loop unrolling, measure that might provide a improvement of the performance at large work-group sizes. From this reasons it is easy to assume that all kernels will have a similar behavior, that increasing the work-group size will increment performance until some point, and using that heuristic it would be possible to find the optimal work-group size. However, fir example, in the case of matrix-matrix multiplication, the optimal work-group size is the largest possible size, the saturation of resources do not present a larger performance sink as reducing the global memory traffic benefit it gives. It is clear then, that finding the optimal parameters presents a challenge for the programmer, because the performance trade off of different parameters is not possible to guess or find a fixed rule that will apply to all cases. Our model can provide a programmer with some insight about what is happening inside the GPU and could be the best parameters for it, and the decide about further optimizations like loop unrolling etc.

[5]

THIS PAGE INTENTIONALLY LEFT BLANK

# Chapter 6

## Conclusions and Future work

In this chapter we summarize our work and discuss the future directions to extend the work presented in this dissertation.

### 6.1 Conclusion and Future Plan

We have developed and tested a Linear Breakdown Performance Model for GPU devices. We originally developed the model to be used with the AMD's Accelerated Processing Unit. Our original motivation was investigate a performance model for that particular device that has the characteristic of being a Integrated GPU. However, after applying the same methodology to experimental data obtained from a Nvidia GPU, we demonstrated the capabilities of the model using two different GPU devices: an AMD's APU (integrated GPU device) and a Nvidia Geforce GTX 660 (discrete board) using two different kernel programs, GSMM and FFT . Both of the used devices have a different architecture, one being a Nvidia card and the other being a AMD card, additionally to the fact that the GTX 660 is a discrete card and the APU is an integrated device. This shows that the model is capable of capture the hardware characteristics of different types of GPU and offer an accurate breakdown model.

### 6.1.1 Future Directions

There are three possible future direction to follow from our previous results:

- **Extensive testing of our model methodology** using a wider range of kernel routines that present one or both of compute-bound routines and memory-bound routines. Then analyze and interpret the results obtained with our model and prove its effectiveness. Available collection of kernel routines written using the OpenCL standard include the Scalable Heterogeneous Computing (SHOC) Benchmark[1], a collection of benchmarking programs aimed to test the performance and stability of heterogeneous systems.
- **Further refinement of the model** by two different directions: incorporating more detailed hardware and execution model features and utilize analytical modeling techniques that improve the model accuracy. One of the possible sources of inaccuracy in the model is attributed to interaction between the three predictors, and non-linearity in these interactions. To deal with this problem we propose the use of Spline functions [4]. Splines can be used to divide the predictor domain into a piece-wise functions divided by *knots*. We consider taking the approach of setting knots in fixed points in the predictor domain corresponding to the points where the number of wave-fronts required to cover a work-group changes. This is because, as was observed in previous results, this is an important point where a change in the tendency of the performance curve is often observed. Incorporating detailed execution information like instruction counters, cache hits and misses counters is not considered because of their unavailability in some platforms. Although there exist several profiling tools for GPU computing, they are not easy to use nor are they available for all Operating Systems and GPU models. We are focusing on OpenCL as our base programming model because of its universality, we consider that universality is an important characteristic of this project.
- **Automatic model generation** The previous models were generated by hand. Starting from the analysis of the kernel routines, its computational complexity

and memory access patterns we developed the set of three separate formulas that make up the model. However, to improve the usefulness of the model, an automatic model generation framework is considered. Although this direction is the most interesting one, it is also difficult to pursue. Nonetheless, studies like the one presented in [24] introduce an abstract interpretation of GPU kernels, Work Flow Graph, used in their work to estimate the execution time of the kernel. A tool based on the flow chart idea can be used to help set the modeling of our performance model.

On the other hand, the other two aforementioned points, increasing the model accuracy and test and analyze a wider selection of routines to improve the model, is a direction that we consider feasible and will produce a tool that will be useful for GPU programmers to reduce the time cost to produce programs with high performance.

THIS PAGE INTENTIONALLY LEFT BLANK

# Bibliography

- [1] . The scalable heterogeneous computing (shoc) benchmark suite. *Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processors 2010*, 2010.
- [2] S. Balsamo, A. di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *Software Engineering, IEEE Transactions on*, 30(5):295–310, May 2004.
- [3] Eli Bendersky. Pycparser homepage. <https://travis-ci.org/eliben/pycparser>, 2014.
- [4] Bronis R. de Supinski Benjamin C. Lee, David M. Brooks. Methods of inference and learning for performance modeling of parallel applications. *In Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '07)*, pages 249–258, 2007. New York, NY, USA.
- [5] Eric A. Brewer. High-level optimization via automated statistical modeling. *SIGPLAN Not.*, 30(8):80–91, August 1995.
- [6] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Newnes, first edition, 2012.
- [7] Nvidia Corporation. Nvidia cuda homepage. [http://www.nvidia.com/object/cuda\\_home\\_new.html](http://www.nvidia.com/object/cuda_home_new.html), 2014.
- [8] NVIDIA Corporation. Nvidia parallel insigth. <http://developer.nvidia.com>, 2014.
- [9] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauer, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.
- [10] Wen-mei W. Hwu David B. Kirk. *Programming Massively Parallel Processors: A Hands-on Approach*, chapter 1-5. Newnes, 2nd edition, 2012.
- [11] Advanced Micro Devices. Amd accelerated parallel processing programming guide revision 2.6. [http://developer.amd.com/wordpress/media/2013/07/AMD\\_](http://developer.amd.com/wordpress/media/2013/07/AMD_)

Accelerated\_Parallel\_Processing\_OpenCL\_Programming\_Guide-rev-2.6.pdf,  
November 2013.

- [12] Advanced Micro Devices. Ati stream profiler. <http://developer.amd.com>, 2014.
- [13] R.T. Dimpsey and R.K. Iyer. A measurement-based model to predict the performance impact of system modifications: a case study. *Parallel and Distributed Systems, IEEE Transactions on*, 6(1):28–40, Jan 1995.
- [14] Rob Farber. *CUDA Application Design and Development*, chapter 1-5. Elsevier, first edition, 2011.
- [15] The R Foundation for Statistical Computing Platform. R language. <https://Rlanguage.org>, 2014.
- [16] Kronos Group. Opencl homepage. <http://www.khronos.org/opencl/>, 2014.
- [17] Edward Kandrot Jason Sanders. *Cuda by Example: An Introduction to General-purpose GPU Programming*, chapter 4-6. Newnes, 2nd edition, 2011.
- [18] et. al. Jiayuan Meng, Vitali A. Morozov. Grophecy: Gpu performance projection from cpu code skeletons. *ACM SC11*, November 2011.
- [19] Rehman M.S Kothapalli K., Mukherjee R. A performance prediction model for the cuda gpgpu platform. *International Conference on High Performance Computing (HiPC'09)*, pages 463–472, December 2009.
- [20] David R. Kaeli Lee Howes. *Heterogeneous Computing with OpenCL*. Newnes, first edition, 2013.
- [21] Jiayuan Meng Michael Boyer. Improving gpu performance prediction with data transfer modeling. *Argonne National Laboratory*, 2011.
- [22] Paolo Bientinesi Roman Iakymchuk. Modeling performance through memory-stalls. *SIGMETRICS*, pages 86–91, October 2012.
- [23] Matthieu Delahaye Sara Baghsorkhi. An adaptive performance modeling tool for gpu architectures. *PPoPP'10*, 2010. Bangalore, India.
- [24] Matthieu Delahaye Sara S. Baghsorkhi. An adaptive performance modeling tool for gpu architectures. *SIGPLAN'10*, 5(45):105–114, January 2010.
- [25] Matthew Scarpino. *OpenCL in Action: How to Accelerate Graphics and Computation*. Manning Publications Company, first edition, 2011.
- [26] Hyesoon Kim Sunpyo Hong. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *ISCA '09*, 2009. Austin, Texas.

- [27] Hyesoon Kim Sunpyo Hong. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH*, 37(3):152–163, June 2009.
- [28] Nicholas Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley, first edition, 2013.
- [29] Zhang X. Performance measurement and modeling to evaluate various effects on a shared memory multiprocessors. *IEEE Transactions on Software Engineering*, 17(1):87–93, January 1999.
- [30] John Owens Yao Zhang. A quantitative performance analysis model for gpu architectures. *HPCA '11*, 2011.