International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 5, Number 1, pages 86–104, January 2015

Linear Performance-Breakdown Model: A Framework for GPU kernel programs performance analysis

Chapa Martell Mario Alberto
Department of Electrical Engineering,
The University of Tokyo. Japan

Sato Hiroyuki
Information Technology Center
The University of Tokyo. Japan

Received: August 1, 2014
Revised: October 31, 2014
Accepted: December 3, 2014
Communicated by Susumu Matsumae

Abstract

In this paper we describe our performance-breakdown model for GPU programs. GPUs are a popular choice as accelerator hardware due to their high performance, high availability and relatively low price. However, writing programs that are highly efficient represents a difficult and time consuming task for programmers because of the complexities of GPU architecture and the inherent difficulty of parallel programming. That is the reason why we propose the Linear Performance-Breakdown Model Framework as a tool to assist in the optimization process. We show that the model closely matches the behavior of the GPU by comparing the execution time obtained from experiments in two different types of GPU, an Accelerated Processing Unit (APU) and a GTX660, a discrete board. We also show performance-breakdown results obtained from applying the modeling strategy and how they indicate the time spent during the computation in each of the three Mayor Performance Factors that we define as processing time, global memory transfer time and shared memory transfer time.

Keywords: GPGPU, OpenCL, APU, Performance Modeling

1 Introduction

The Graphic Processing Unit (GPU) is a computer device originally designed as specialized hardware that freed the CPU from computational-heavy graphic-related task. However since the beginning of the 21st century computer programmers realized that the rate at which the GPU could perform floating-point operations was higher than that of the CPU by a wide margin. There was a drawback: the only way to program the GPU was by using graphics-related programming languages and APIs, making it difficult and non-intuitive. Nonetheless, the results of using the GPU for general-purpose computations were good and the new field of General Purpose Computations on GPU (GPGPU) was born. Because of its initial success, many research communities and GPU vendors joined the efforts to make GPGPU available for more all programmers and not only those with computer graphics

knowledge. As a result, the development of GPU-specific programming models like like CUDA [6] and OpenCL which targets any accelerator hardware and multi-core and many-core processors [10] have made GPU programming more accessible and enabling features that allow its application in more areas besides graphics processing.

GPUs are made up of a large number of Processing Elements (*PE*) (the exact number varies in each GPU model, but it is typically several hundreds) that allow a high level of parallelism, which is one of the sources of their high computational throughput, thus achieving high performance is the main purpose of recurring to GPGPU. It is desirable that programs use the GPU are as efficient as possible, inefficient use of the hardware results in detriments in performance which is contradictory to the original objective. It is, however, due to GPU's complex architecture that it is difficult and time consuming to develop GPU programs that make full use of the potential of the hardware the run on. Despite the existence of general programming models and tools for GPU programming, writing correct, efficient programs for GPUs is a difficult, time consuming task. The reason of the difficulties is that GPUs are highly sophisticate hardware designed to create and manage thousands of parallel threads, sophistication that is achieved through architectural complexity. There is a user-managed memory hierarchy that is different of that of the CPU, race conditions, multi-port memory, add complexity that is responsibility of the programmer.

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. We present our Linear Performance-Breakdown Model, a performance modeling tool that shows the bottlenecks to the programmer to aid in the optimization process.

The rest of this document is organized as follows: In section 2 we present an overview of the OpenCL programming model and its relationship with GPU hardware. We describe our proposed Linear Performance-Breakdown Model and the framework. Section 4 presents the results obtained from applying our framework to three case studies, Single-precision General Matrix Multiplication, Fast Fourier Transform and Reduction. A review on the literature related to this study is presented in Section 5. Finally, in section 6 we present the conclusions drawn from this work and a brief description of the intended future work regarding the LPBM framework.

2 GPU Programming Model

In order to program the GPU for general purpose computations, a special API is necessary. For our experiments we opt to use OpenCL. By using OpenCL, it is possible to run the same program in GPUs from different vendors. This enables us to perform experiments in GPU with different architectures. To help clarifying our model methodology, in this section we describe the OpenCL programming model and its relationship with the architecture of the GPU. This is necessary to understand how GPU are constituted and how the different concepts in the programming model are related to the performance achieved by GPU kernels.

The execution of a GPU kernel typically starts from the CPU in a program that is commonly referred as *host program*. The host program prepares necessary resources in the host computer, like system memory to store the data that will be sent to and from the GPU as input and output data; as well as compile and send to the GPU the *kernel program file*, that is the file that contains the source code of the program that will be executed in the GPU. Once the kernel code is compiled, the host program can send kernel code and data to the GPU for its execution.

Figure 1 shows a block diagram of the GPU architecture at its basic level. Modern GPUs architecture follows a hierarchical block mode in which the top level is made of vector processors, called Compute Units (*CU*) that operate in a Single Instruction Multiple Data (*SIMD*) fashion. Down to the next level, each CU contains an array of Processing Cores (*PE*). All the PEs inside a CU are able to communicate through an on-chip, programmer-managed memory known as shared memory. [16, 21].

When a kernel is submitted to the GPU for execution a index space containing all the Work-Items, specified in the kernel program, is created. Each Work-Item will have a coordinate which

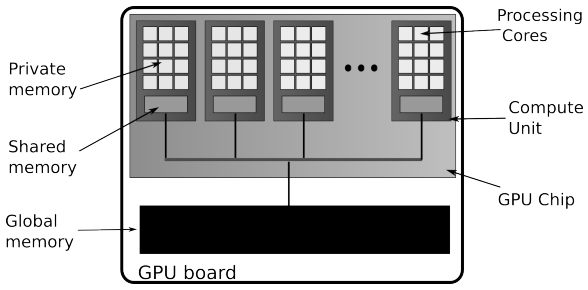


Figure 1: General block diagram of a GPU device. A GPU can be regarded as a collection of Compute Units, which in turn are constituted by an array of Processing Cores.

purpose is to provide a means for Work-Items to distinguish from each other and for the programmer to control which data elements each Work-Item must operate on. It is then responsibility of the programmer to divide the work to be done and ensure each Work-Item is correctly indexed and accessing the correct data elements.

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 memory and the register states for the next task must be loaded back, an operation that can cost a high number of cycles. 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. The impact of this mechanism is that the GPU uses the fast context-switching ability coupled with the capability to manage a large number Work-Item to hide data latency. However, it is important to consider that even though the number of register banks is large, it still has a limit that directly impacts the number of Work-Items that can reside in a Computing Unit at any given moment. 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.

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. A kernel program can run on a GPU device with any arbitrary number of CUs, the GPU's scheduler will distribute groups to all available units. This translates to the general rule that more CU reduce computation time as more work is done in parallel. Unfortunately more factors intervene to determine the actual computation time. 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. The number of Work-Items that can be scheduled for execution is known as *Wave-Front* (WF). The WF size is different for each GPU vendor, in current generation hardware, for AMD devices, the WF size is 64 Work-Items whereas for Nvidia hardware is 32. This means that if some number different from the WF size needs to be executed, the schedulers will still create 64 Work-Items and since not all of them have useful work to perform, some will be idle. The result is Processing Elements being idle during some cycles, reducing the efficiency of the GPU.

Although slower than the register file, shared memory is larger 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. GPU's global memory have the characteristic of featuring a very high bandwidth which is achieved by partitioning the memory cell into several banks, known as *partitions*. Figure 2 shows a description of how the different concepts in the OpenCL standard map to the correspondent GPU hardware component. For example, a Work-Item executes in a PE and each Work-Group is assigned to a CU.

For the experiments presented in this paper we use two GPUs which feature different architectures, an AMD Accelerated Processing Unit (APU) and a NVIDIA GeForce series GTX 660. The

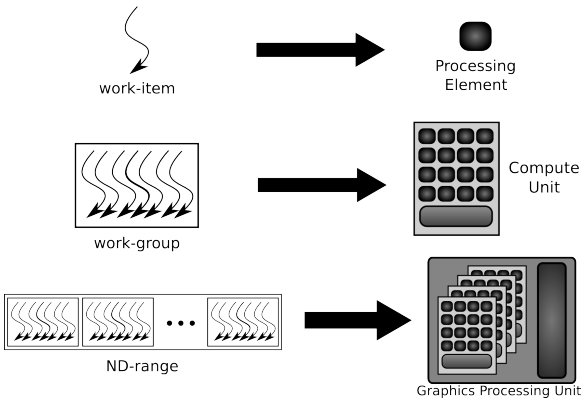


Figure 2: Correspondence between OpenCL concepts and the GPU hardware.

distinguishing characteristic between these devices is that the APU is an integrated GPU and the GTX is a discrete GPU board.

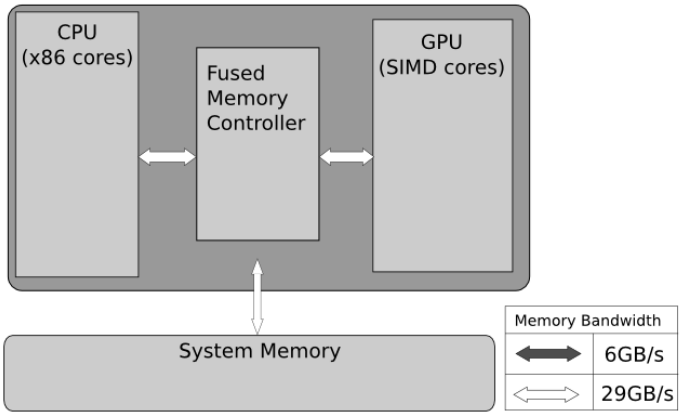


Figure 3: Block diagram of an APU. The memory connections and placement of computing cores as well as the speed of the memory buses are shown

The architecture of an integrated GPU is shown in Figure 3. An APU is a special type of computer processor in which 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 memory spaces. 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 is transferred, data is moved from one RAM location to the other. However, 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 between different memory locations. On the other hand discrete GPUs, like the GTX 660, are GPUs that are contained into its own electronic board. The block diagram of a discrete GPU can be seen in Figure 4. The GPU board is attached into the host computer system using the PCI express port. The advantage of a discrete board is that there is neither, space nor transistor count limits to build the GPU chip, which is the reason discrete GPU are generally more powerful than integrated GPUs. 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[6]. Since a discrete board is far from the main system memory, and separated by 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 transfers of small quantity of data, the long data transfer overhead time overcomes the advantage of the high throughput offered by the GPU.

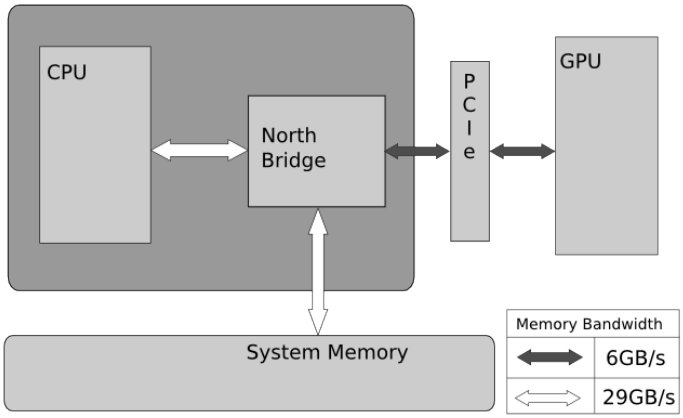


Figure 4: Block diagram showing how a discrete GPU board is attached to a host system. The memory buses speed is indicated by the colored arrows.

At the lowest memory hierarchy level, GPUs feature 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 the next section we describe the framework on which our modeling methodology is based. The framework features several modules that perform specific tasks related to building the model from the source code and obtain the performance-breakdown.

3 LPBM Framework

In this section we describe our proposed modeling strategy, the Linear Performance-Breakdown Model *LPBM*. We also describe the framework we developed to use the LPBM as a GPU kernel optimization tool. Our framework for the LPBM consist of a group of modules that interact with each other to process a GPU program and produce a Performance-Breakdown graph. The framework takes the host and kernel code as input, follows several steps to process the relevant information in the files that compose the GPU program and produces profiling information that is used to obtain the Performance-Breakdown formula. The modules that make up the framework are the Auto-Profiling Module(APM), the Performance Formula Generation Module (PFGM), the Linear-Regression Module(LRM) and the Performance-Breakdown Extraction Module (PEM). The modules are responsible of obtaining all the necessary information to calculate the breakdown of performance from the kernel code in the local machine. Figure 5 shows a block diagram of the framework modules, showing inputs and outputs of each module.

In the rest of this section, we describe each module, their necessary inputs and outputs, and the functions they perform.

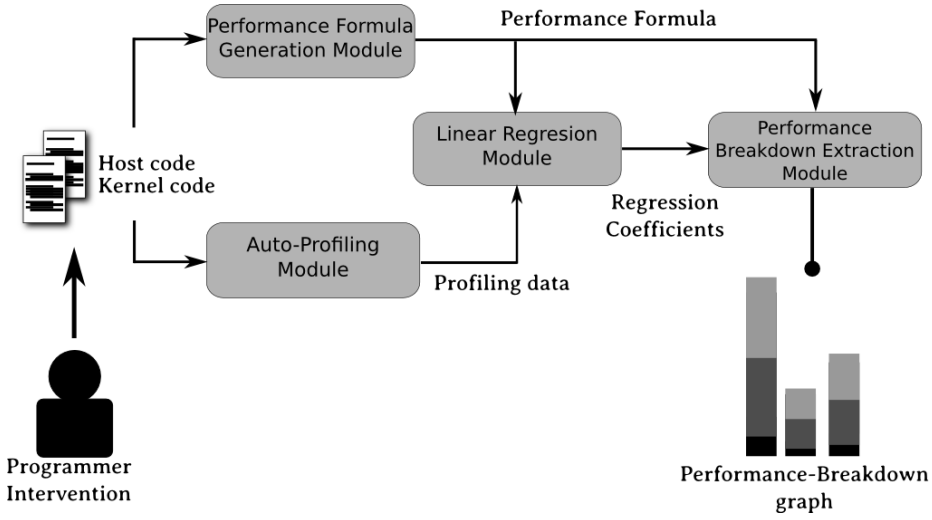


Figure 5: LBPM Framework block diagram. The modules that make up the framework are depicted. The framework is a semi-automated system, it requires the user intervention.

3.1 Auto-Profiling Module (APM)

For our LPBM framework like in other modeling methods, in order to extract the breakdown of performance, program parameter information and machine parameter information are necessary [17]. In our system, the machine information is obtained using a list of known GPU device specifications that are available in the documentation of the hardware provided by the GPU manufacturers, in addition to information available using the OpenCL function *clGetDeviceInfo* as well as parameters defined in the program code. Program parameters include register usage, wavefronts in the kernel launch, etc. Program parameter information is obtained by the Performance Formula Generation Module. The LPBM Framework requires kernel program profiling data to calculate the regression coefficients that are used for breakdown of performance calculation. This profiling data must be a set of measured execution times with different Work-Group settings. The purpose of the APM is the gathering of the profiling information. Execution time of a program contains both, system and program information and, as expected, depends on the GPU device and the program code, where most of the characteristics are defined by the kernel program.

To correctly identify the relevant information from the kernel code, the APM requires the programmer to prepare the source code. First, The Command Queue that will make the call to the *clEnqueueNDRangeKernel* function must be created with the *enable profile information* option set. This instruct the Command Queue to gather profiling information that is used to obtain the execution time information. Second, the correspondent code must be added before and after the call of kernel execution functions to compute and print out the gathered information and thus allow the APM to collect the execution data. A last adjustment involves the variable or constant used to determine the local and global-work size in the host code. Such variables or constant must be changed to a predefined variable names, *LWS* and *GWS*, that stand for Local Work Size and Global Work Size respectively. This is necessary to conform with the module in our framework and enable it to define sets of different values for both, local and global work size and gather the execution time information.

The implementation of this module is through shell scripts. The shell script specifies the global and local sizes at compile time using the gcc flag *-D*. The module then generates a set of program runs with the different parameters and collects the profiled data (execution time against Work-Group size). As mentioned previously, it makes use of both, kernel and host code files as input, and the output is a data file that contains the profiling information organized in a line-wise order. The included information will be the settings of global and global work size plus the timing information

in nanoseconds obtained from the OpenCL API facilities.

3.2 Performance Formula Generation Module (PFGM)

The execution time of a computer program is difficult to predict in modern processors. Out-of-order execution, branch prediction, sophisticated caches systems and other improvements are difficult to model due to their non-deterministic nature. However, we consider that even though a large number of factors and the complex interactions that they present determines the real performance of a computation in GPUs, it is possible to work with the assumption that among all factors, there are three of them that have the largest impact on the execution time of any kernel program. In a computer, the total execution time of a program be calculated as the sum of the time used for computations and the time used for data transfers. In a GPU system, because there is a clear difference between the two levels of memory, data transfers are in turn divided into global memory to shared memory transfers, and shared memory to private memory transfers. Thus, we define the Major Performance Factors (*MPF*) as 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*). The reason for choosing the MPF is that because of the increasing gap between CPU and memory performance, the execution time of programs in modern computers is largely dominated by the data transfers time[18]. In the case of GPUs the only difference is that the memory hierarchy is an heterogeneous, multi-level structure that combines different memory technologies. Typically the memory levels that can be found are the global memory, off-chip memory implemented using GDDR technology. Global memory is large, with capacity in the order of GB, but access latency is high. The next level is the Shared memory, on-chip memory implemented with SDRAM technology. It is faster than global memory by an order of magnitude but also much more limited capacity, typically in the order of kB. The last memory hierarchy level is the private memory, the closest memory to the processing units and the fastest memory type. To determine the performance formula for each performance factor (*PUT, G2ST, S2PT*) it is necessary to analyze 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.

3.2.1 Performance Factors and Performance Formula designation

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 1.

$$RunningTime = ComputationTime + DataTransferTime \tag{1}$$

A more refined model suited for GPU computations considers that the data transfers in a GPU are divided into two categories. The first category refers to the data transfers from global memory to shared memory, the second, from the shared memory to the private memory, i.e. the registers of the processing cores. In our model we do not consider transfers between system memory and global memory because, in the case of discrete GPUs, they cannot be avoided as the GPU and main system memory are separated from each other; and in the case of integrated GPUs, zero-copy data

buffers are be used, which eliminates any transfer. For that reason host-device memory transfers are not considered, being this the main advantage of an APU system [7]. 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 2 that its a linear combination of three terms.

$$\begin{aligned} RunningTime = & \alpha_1 \cdot ComputationTime + \\ & \alpha_2 \cdot LocalMem.TransferTime + \\ & \alpha_3 \cdot GlobalMem.TransferTime \end{aligned} \tag{2}$$

The α parameters in the equation define the weight of each individual Mayor Performance Factor in 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 efforts can be applied in the correct direction. As a starting point for constructing the formulas that make up the performance factor formulas, let us consider the case where the system runs under a sequential execution style. In such sequential system, the total execution time 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 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 a certain degree of concurrency. The degree of concurrency that can be achieved depends not only on the capabilities of the hardware, but also the program characteristics. Each GPU has an upper limit of wavefronts that can be executing concurrently in the device. That number and the actual number of concurrent wavefronts being executed when a kernel is running are different under ordinary situations. The reason is that Work-Items will require a given number of registers when they are executed and if there is not enough register to service several wavefronts at the same time, the effective number of wavefronts executed concurrently will be reduced. The exact number of registers per Work-Item depends on the program code. It follows to calculate the concurrency level, the total number of Wave-Fronts required to execute all the Work-Items in a kernel launch is calculated. Knowing that value, 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.

3.2.2 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 to hide the latency of memory operations. The maximum number of blocks that are executed concurrently per CU is a combination of device properties and kernel characteristics. It is crucial to evaluate the impact on the in-flight blocks that can be executed in a CU of tuning parameters in kernel programs as the performance impact of this idling is a reduced computational throughput.

In a computer system, the total execution time of a program can be estimated as the number of floating-point operations necessary to finish the task. This would yield the total amount of

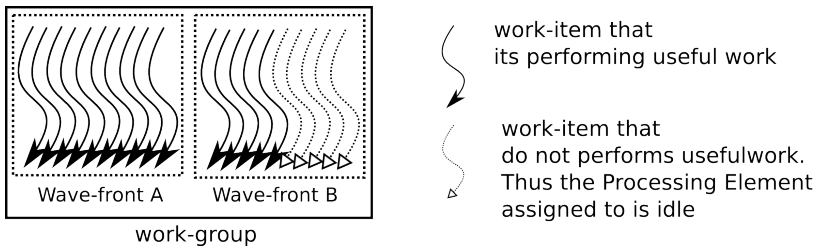


Figure 6: Wavefronts are the smallest scheduling unit. In this figure we can observe a single Work-Group composed by two Wave-Fronts. Because the Work-Group size is not a integer multiple, some Work-Items do not perform useful work.

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[20]. 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. 3).

$$ProcessingUnitsTime = \frac{BigOcomplexity}{ConcurrencyLevel} \tag{3}$$

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.

$$ConcurrencyLevel = \frac{Wavefronts_{NDRange}}{Wavefronts_{MaxConcurrent}} \tag{4}$$

3.2.3 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 Performance 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 [5, 13]. 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. Furthermore, irregularities like memory camping, where consecutive elements access memory location which memory bank interface is the same. 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.5).

$$G2ST = \frac{N \cdot WF_{wg}}{WG_{size}} \tag{5}$$

3.2.4 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 [20]. 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.6.

$$S2PT = \frac{WG_{total}}{Wavefronts_{MaxConcurrent}} \tag{6}$$

3.3 Building the Performance Factor Formulas

Memory transfers in GPU kernels can be identified by observing assignment statements in the source code. As mentioned previously, the module locates all assignments in the kernel program code, locating all the memory transfers that are to take place when the program is executed. Each assignment instance is classified according to the left, and right side of the assignment and the type of the variables that appear in each side. We classify the memory transfers in relation to the slowest memory type involved in the assignment. Because of their limited size, The used amount of shared and private memory, not only the access pattern but also the amount used by the kernel program greatly influence the execution time. Once the three predictors have been defined, we integrate them into a linear combination formula. Table 1 shows a summary of the performance predictors used to calculate the MPFs.

Table 1: Summary of performance predictors

Symbol	**Meaning**
WF_{size}	Wavefront size
WG_{size}	Work-Group size
WG_{total}	Total number of Work-Groups
WF_{wg}	Wave-Fronts in a Work-Group
WF_{ndr}	Wave-Fronts in the ND range
GP_{wg}	GP registers per Work-Group
WF_{max}	Max. number of concurrent Wave-Fronts

3.4 Linear Regression Module (LRM)

If we consider that the performance formulas as a math expressions were coefficients that will have some predictors affect them in a directly proportional way and other predictors affect them in a inversely proportional way, we can define a *meta-formula* for the performance-breakdown. The exact rate at which the predictor affect the performance factor is given by a complex and non-deterministic interaction of factors that depend on the original program code. Using Regression as a tool to find the impact of this factors and allow the calculation of the performance-breakdown is the objective of this module. 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* [9] and its module nnlsm. The factors are 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. 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[17]. 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 $y = y_1, \ldots, y_n$. For a point i in this space, let y_i denote its response variable and $x_i = x_{i,1}, \ldots, x_{i,p}$ 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 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.

$$f(y_i) = \sum_{j=1}^{p} \alpha_j g_j(x_{i,k}) \tag{7}$$

3.5 Performance Breakdown Extraction Module (PBEM)

This is the last module in the framework. It calculates the performance-breakdown using the regression coefficients obtained by the LRM and the performance formula determined by the CLPM. The output is a bar graph that shows the decomposition of total execution time into the time attributed to each of the MPFs. This module produces the final output of the framework, a bar graph where the user can observe the performance-breakdown graph.

4 Case Studies

This section presents the results obtained from applying our model to three different kernel routines, Single-precision General Matrix Multiplication (SGMM) which kernel is obtained from the sample code found in [21], The Fast Fourier Transform (FFT) using a kernel obtained from the sample code found in [22] and Reduction which kernel programs is taken from obtained from the SHOC Benchmark suite, a collection of benchmark programs testing the performance and stability of systems using accelerator computing devices for general purpose computing[1].

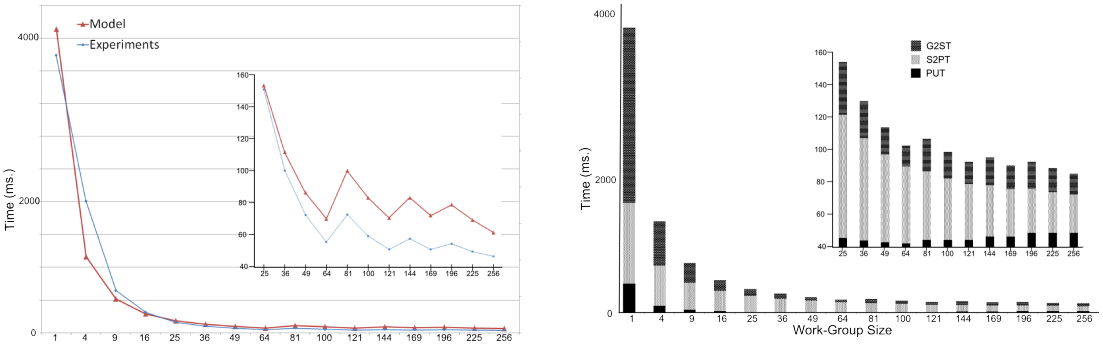
In all the experiments we employ two GPU devices whose characteristics are summarized in table 2. The results for both GPU are calculated using the same model and the correspondent profiled execution times as input. We run the programs 1000 times and the execution time used to train the model is an average of all the measured timings for each tile size.

Table 2: GPU Characteristics

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

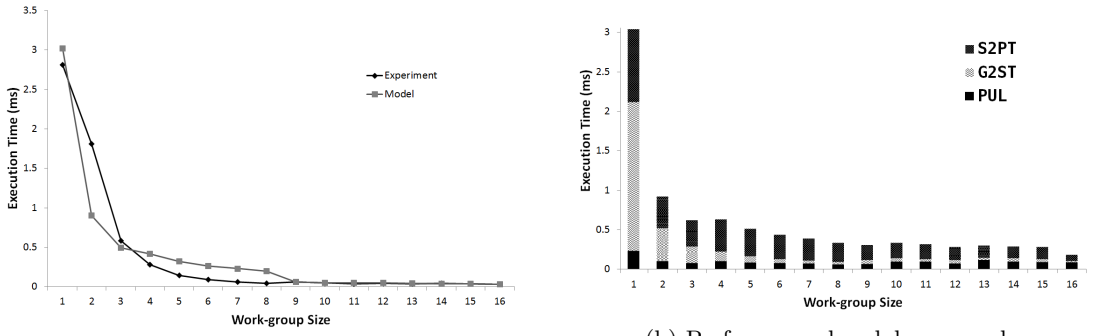
4.1 Single-precision General Matrix Multiplication

For our first case study, we use a SGMM program. This kernel program takes two matrices and compute the *matrix product*. Listing 4.1 shows the kernel code used in this experiment. The program was run with all the possible ranges of values for local work size for a matrix size of 1000-by-1000. The local work size values range is from 1 to 16. It is not possible to create a tile of a width larger than 16 because of the limit of Work-Groups size that is 256 elements.



(a) Execution time. (b) Performance breakdown graph.

Figure 7: Results of the SGMM on the APU.



(a) Execution time. (b) Performance breakdown graph.

Figure 8: Results of the SGMM on the GTX660

Listing 1: SGMM kernel with tiling

```
float output_value = 0;
        for(int m = 0; m < Width/TILE_WIDTH; m++) {
        // Global − Local memory transfers
        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);
                // Local − Private memory transfers
                for(int k = 0; k < TILE_WIDTH; k++)
                        // Processing Units time
                  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. Figures 7 and 8 show the results for the SGMM

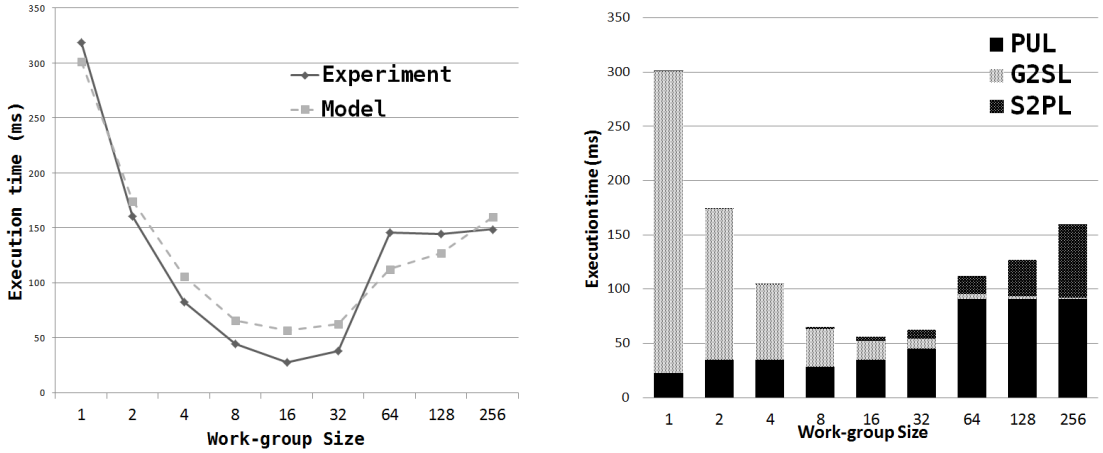
kernel experiments. In subfigures 7a and 8a it is possible to observe the comparison between the profiled execution time and the execution time obtained from the model. It is possible to observe that the results of the model closely matches experimental for both devices. The performance-breakdown results can be observed in subfigures 7b for the APU and 8b for the Nvidia card. The performance breakdown graph for this kernel shows the positive effect the increase of tile size has on performance [8], correctly reflecting that effect in the graphs. Additionally, when the Work-Group size is large and close to the maximum possible, the PUT show a noticeable increment, a detail that is observed in both GPUs. This is due to the increase in the computational load compared to the memory transfer load ameliorated by the increase of Work-Group size, a situation that is observed in either architecture.

4.2 Fast Fourier Transform

The Fast Fourier Transform *FFT* is our second case study for our model. In this case the kernel is show in Listing 4.2. This kernel computes the Fourier Transform. For the FFT kernel we can observe that there several memory transfers involved in the computation, but most of them involve values that are classified as private memory variables. This has two consequences, this values do not need to be transfered from slow memory spaces, but at the same time, they increase the register count for the kernel. As discussed previously, when a kernel requires a large number of registers, the amount of in-flight wavefronts that the device is capable of execute, decreases. Figures 9a and 10a show the comparison between the real values and the values obtained using the model for the FFT kernel in the APU and the GTX660 respectively. The x axis represents Work-Group size, in powers of 2 as required by the algorithm and y represents the execution time in ms. The graph shows how the execution time changes for different values of Work-Group size and that the model closely matches the profiled data. Figures 9b and 10b show the results Regarding the breakdown of performance into the MPFs. We can observe again the positive effects of increasing the Work-Group size until a certain value is reached, because unlike the SGMM case, the highest value does not achieve the best performance. Observing the performance breakdown graph, it is possible to note that the amount estimated for PUT increase considerably. Because the Work-Group size increase, the registers needed to service all the generated Work-Items increases as well. This results in a detriment in the concurrency level which in turn causes the increase in the PUT that we can observe in the graphs.

Listing 2: Fast Fourier Transform Kernel code

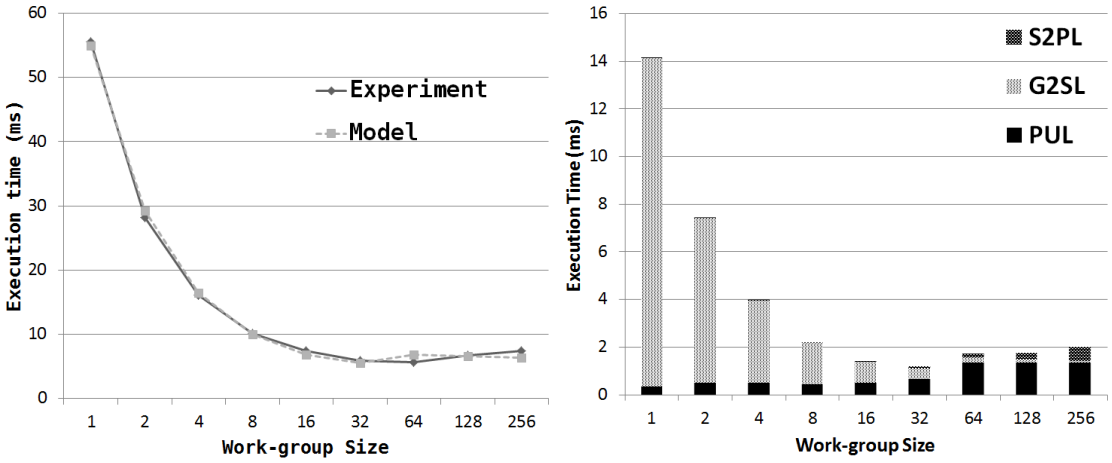
```
unsigned int gid = get_global_id(0);
unsigned int nid = get_global_id(1);
int butterflySize = 1 << (iter−1);
int butterflyGrpDist = 1 << iter;
int butterflyGrpNum = n >> iter;
int butterflyGrpBase = (gid >> (iter−1))∗(butterflyGrpDist);
int butterflyGrpOffset = gid & (butterflySize−1);
int a = nid ∗ n + butterflyGrpBase + butterflyGrpOffset;
int b = a + butterflySize;
int l = butterflyGrpNum ∗ butterflyGrpOffset;
float2 xa, xb, xbxx, xbyy, wab, wayx, wbyx, resa, resb;
xa = x[a];
xb = x[b];
xbxx = xb.xx;
xbyy = xb.yy;
wab = as_float2(as_uint2(w[l]) ^ (uint2)(0x0, flag));
wayx = as_float2(as_uint2(wab.yx) ^ (uint2)(0x80000000, 0x0));
wbyx = as_float2(as_uint2(wab.yx) ^ (uint2)(0x0, 0x80000000));
resa = xa + xbxx∗wab + xbyy∗wayx;
resb = xa − xbxx∗wab + xbyy∗wbyx;
x[a] = resa;
x[b] = resb;
```



(a) Execution time.

(b) Breakdown of performance.

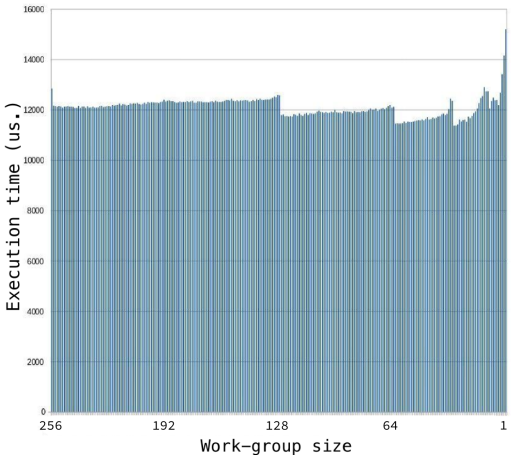
Figure 9: Experiment results of the FFT kernel on the APU.



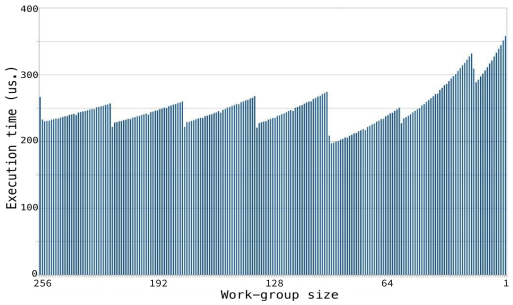
(a) Execution time.

(b) Breakdown of performance.

Figure 10: Experiment results of the FFT kernel on the GTX660



(a) Execution time on the APU



(b) Execution time on the GTX660

Figure 11: Reduction kernel results

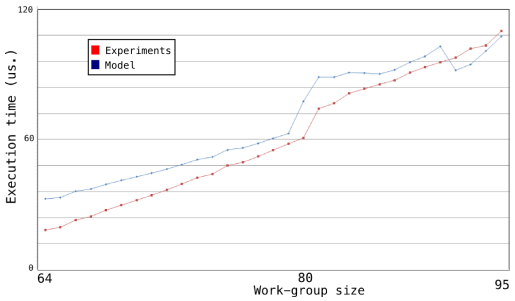
4.3 Reduction

Reduction is a vector operation, the input is a vector of length N, and the objective is to add the values of each independent vector element successively until a single output result is obtained. Reduction if not computational heavy, but it involves a high number of data transfers. The GPU implementation of this process presents several design challenges in regard of the type of memory used to store intermediate results, the Work-Item indexing strategy, etc. Each strategy will produce different performance results. Listing 4.3 shows the reduction kernel code.

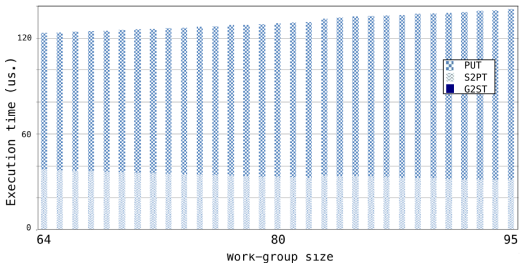
Listing 3: Reduction

```
sdata[tid] = 0;
// Reduce multiple elements per thread, strided by grid size
while (i < n){
        sdata[tid] += g_idata[i] + g_idata[i+blockSize];
        i += gridSize;
}
barrier(CLK_LOCAL_MEM_FENCE);
// do reduction in shared mem
for (unsigned int s = blockSize / 2; s > 0; s >>= 1){
        if (tid < s){
        sdata[tid] += sdata[tid + s];
}
barrier(CLK_LOCAL_MEM_FENCE);
// Write result back to global memory
if (tid == 0){
        g_odata[get_group_id(0)] = sdata[0];
}
```

The results obtained in the reduction kernel experiments with a vector size of 1 million elements can be observed in Figure 11. Figure 11a shows the results for the APU and in Figure 11b the results for the GTX660 are shown. It is possible to see that even in kernels that perform a simple operation like reduction both devices show different behavior when modifying the Work-Group size and that both of them exhibit a different optimal Work-Group size, the APU shows the minimum execution time at a work-size of 64 and the GTX660 at 128 elements per Work-Group. Because for this case study the experiment incorporated a large number of trials, Figures 12 and 13 show a sample from the whole experiment value. This is to make possible to observe the breakdown of

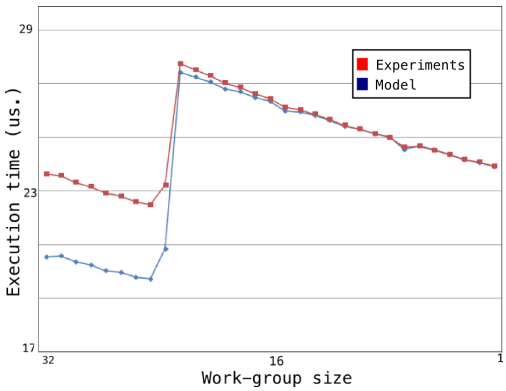


(a) Execution time.

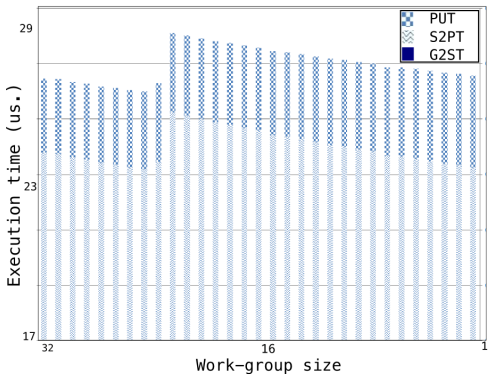


(b) Performance-breakdown.

Figure 12: Reduction kernel results on the APU



(a) Execution time.



(b) Performance-breakdown

Figure 13: Reduction kernel results on the GTX660

performance. From the definition of the operation and the kernel code it is possible to understand that this kernel does not involve a large number of computations, but the intensive work is about transferring data. This is reflected in the results we can observe in Figure 12b for the APU and Figure 13b for the Nvidia card, the performance-breakdown revealed by the LPBM shows that the value of PUT is very low, almost zero. Another important point to note is that the amount that corresponds to S2PT is larger than G2ST. Because the kernel is using local memory to perform the reduction operations, the time spent transferring data from and to global memory is reduced, and this is reflected in the performance breakdown formula.

5 Related Work

When an application is developed, the main reason GPU is preferred over CPU for computation is GPU's high performance paired with a higher energy consumption efficiency. Thus, it is natural that a larger share of the research effort have been invested towards application development, to demonstrate the usefulness of GPUs and the expansion of the range of applications where GPU is a good option for accelerate computations. Companies like Nvidia and AMD focus all their efforts into the improvement of the hardware and the GPU architectural details, whereas users of the technology are mainly interested in developing applications. On the other hand, 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 programing model and hence applicable only to Nvidia GPUs [19, 4, 2, 11]. 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.

In [24], 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 [15] 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, [12] 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. [11] 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 [14], the author proposes a model for execution-less performance modeling for linear algebra algorithms in CPU machines. The authors develops 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 a 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 level of the GPU memory hierarchy.

In [23] the author discuss the effects of factors such as sequential code, barriers cache coherence and scheduling in general shared memory multiprocessors. The author parts from the Amdahl's law and analyze shared memory systems (GPUs belong to this classification) to derive several models, one for each separate factor. Our approach is instead combine the most important factors into a single equation using a special case of Shared memory system and apply then the LSM method to evaluate the impact of each factor.

6 Concluding Remarks

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 since Our original motivation was to investigate a performance model for that particular device that has the characteristic of being a Integrated GPU which overcomes the most important drawback of the GPGPU technology, i.e. the need to transfer data between the GPU memory and the main memory of the host computer. Nonetheless after applying the same methodology to experimental data obtained from a Nvidia GPU, we obtained promising results and demonstrated that the LPBM

could be used for both, integrated and discrete GPUs. These shows that the model is capable of capture the hardware characteristics of different types of GPU and offer an accurate breakdown model. We built a frame that facilitates its use as a optimization assistance tool. Starting from the source host and kernel code, the different modules that make up the framework analyze the GPU program and deliver the Performance-Breakdown graph. In future improvements we plan to extend the scope of the model to performance-affecting factors that might have an important role such as partition camping. At the present stage, the model can be used to analyze single kernel functions which uses a data-parallel decomposition of the problem, this is due to the focus on the Work-Group analysis. For the next step we plan to perform kernel developments test supported by the LBPM framework where. Starting from a simple kernel that lacks optimizations, we first extract the breakdown of performance to posteriorly observe where the execution of the kernel is spending the most time. Once the bottleneck is identified, a optimization aimed to relieve that particular bottleneck will be applied. This constitutes a LPBM *pass*. A second pass will reveal the effects of the optimization not only for the execution time, but also for each MPF, which will provide users with additional insight about the kernel program that will be aimed to reduce the time spent in optimization cycles. As future work the model will be tested with more benchmark programs involving routines that involve more complex execution flow, including but not limited to N-body simulation algorithms, sorting algorithms, etc. One more aspect of the future work on this research is using more GPU devices with different characteristics; for example a Radeon HD discrete card. The GPU cores on the APU chip are of the Radeon HD family, it follows that a discrete card of the same family of hardware will provide valuable information about the effects of different memory technologies and different program behavior. This will allow to collect more data on the effects of hardware characteristics on the performance of kernel code, as well as provide an opportunity to increase the understanding of the effects of architectural designs and optimization effects on them.

References

[1] The scalable heterogeneous computing (shoc) benchmark suite. *Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processors 2010*, 2010.

[2] Sara Baghsorkhi and Matthieu Delahaye. An adaptive performance modeling tool for gpu architectures. *PPoPP'10*, 2010. Bangalore, India.

[3] Eli Bendersky. Pycparser homepage. `https://travis-ci.org/eliben/pycparser`, 2014.

[4] Michael Boyer and Jiayuan Meng. Improving gpu performance prediction with data transfer modeling. *Argone National Laboratory*, 2011.

[5] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Newnes, first edition, 2012.

[6] Nvidia Corporation. Nvidia cuda homepage. `http://www.nvidia.com/object/cuda_home_new.html`, 2014.

[7] Advanced Micro Devices. Amd accelerated parallel processing programing guide revision 2.6. `http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.6.pdf`, November 2013.

[8] Rob Farber. *CUDA Application Design and Development*, chapter 1-5. Elsevier, first edition, 2011.

[9] The R Foundation for Statistical Computing Platform. R language. `https://Rlanguage.org`, 2014.

[10] Kronos Group. Opencl homepage. `http://www.khronos.org/opencl/`, 2014.

[11] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *ISCA'09*, 2009. Austin, Texas.

[12] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH*, 37(3):152–163, June 2009.

[13] Lee Howes and David R. Kaeli. *Heterogeneous Computing with OpenCL*. Newnes, first edition, 2013.

[14] Roman Iakymchuk and Paolo Bientinesi. Modeling performance through memory-stalls. *SIGMETRICS*, pages 86–91, October 2012.

[15] Kothapalli K., Mukherjee R., and Rehman M.S. A performance prediction model for the cuda gpgpu platform. *International Conference on High Performance Computing (HiPC'09)*, pages 463–472, December 2009.

[16] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*, chapter 1-5. Newnes, 2nd edition, 2012.

[17] Benjamin C. Lee, David M. Brooks, and Bronis R. de Supinski. 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.

[18] Nihar R. Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck: Problems and solutions. *Crossroads*, 5(3es), April 1999.

[19] Jiayuan Meng and Vitali A. Morozov et. al. Grophecy: Gpu performance projection from cpu code skeletons. *ACM SC11*, November 2011.

[20] Jason Sanders and Edward Kandrot. *Cuda by Example: An Introduction to General-purpose GPU Programming*, chapter 4-6. Newnes, 2nd edition, 2011.

[21] Matthew Scarpino. *OpenCL in Action: How to Accelerate Graphics and Computation*. Manning Publications Company, first edition, 2011.

[22] Ryoji Tsuchiyama and Takashi Nakamura. The opencl programming book. `http://www.fixstars.com/en/opencl/book/`, 2014.

[23] 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.

[24] Yao Zhang and John Owens. A quantitative performance analysis model for gpu architectures. *HPCA'11*, 2011.