UNIVERSIDADE FEDERAL FLUMINENSE

ROMMEL ANATOLI QUINTANILLA CRUZ

Analyzing and Estimating the Performance of Concurrent Kernels Execution in GPUs

NITERÓI

2017

UNIVERSIDADE FEDERAL FLUMINENSE

ROMMEL ANATOLI QUINTANILLA CRUZ

Analyzing and Estimating the Performance of Concurrent Kernels Execution in GPUs

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic Area: Visual Computing.

Advisor:
Prof. D.Sc. ESTEBAN WALTER GONZALEZ CLUA

Co-advisor:
Prof. D.Sc. LÚCIA MARIA DE ASSUMPÇÃO DRUMMOND

NITERÓI

2017

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e do Instituto de Computação da UFF

Q7 Quintanilla Cruz, Rommel Anatoli

Analyzing and estimating the performance of concurrent kernels execution in GPUs. / Rommel Anatoli Quintanilla Cruz. – Niterói, RJ : [s.n.], 2017.

55 f.

Dissertação (Mestrado em Computação) - Universidade Federal Fluminense, 2017.

Orientadores: Esteban Walter Gonzalez Clua, Lúcia Maria de Assumpção Drummond.

1. Programação (Computação). 2. Kernel (Computação). 3. Unidade de processamento gráfico. 4. Escalonamento de tarefa. I. Título.

CDD 005.1

ROMMEL ANATOLI QUINTANILLA CRUZ

Analyzing and Estimating the Performance of
Concurrent Kernels Execution in GPUs

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic Area: Visual Computing.

Approved on September 2017.

APPROVED BY

Prof. D.Sc. Esteban Walter Gonzalez Clua - Advisor, UFF

Prof. D.Sc. Lúcia Maria de Assumpção Drummond - Co-advisor, UFF

Prof. D.Sc. Anselmo Antunes Montenegro, UFF

Prof. D.Sc. Cristiana Barbosa Bentes, UERJ

Niterói
2017

To my family, for all their love and support.

Acknowledgements

I want to thank professor Esteban Clua who have shared their time guiding me in my research.

I am grateful to Lucia Drummond and Cristiana Bentes for their invaluable support and availability.

I would like to thank my parents, for everything they have done for me.

I would especially like to thank Deyni for the love, support, and constant encouragement.

Also many thanks to the financial support of CAPES for providing a scholarship.

Finally, special thanks to Gabriel Gazolla who gave their time and effort to improve this work.

Resumo

As GPUs estabeleceram uma nova linha de base em relação à eficiência de energia e capacidade de computação, oferecendo maiores largura de banda e mais unidades de computação em cada nova geração. As GPUs modernas suportam a execução simultânea de kernels para maximizar a utilização dos recursos, permitindo que kernels possam explorar melhor os recursos ociosos. Entretanto, a decisão da execução simultânea de kernels diferentes é realizada pelo hardware e muitas vezes as GPUs não permitem a execução de blocos remanescentes de outros kernels, mesmo com a disponiblidade de recursos. Neste trabalho, realizamos um estudo aprofundado sobre a execução simultânea de kernels na GPU. Apresentamos as condições necessárias para executar kernels simultaneamente, listamos os fatores que influenciam a concorrência e propomos um modelo que descreve a redução de desempenho. Finalmente, validamos o modelo utilizando kernels de aplicações reais com diferentes intensidades de computação e uso de memória.

Palavras-chave: Kernels concorrentes, Multiprogramação, computação em GPU.

Abstract

GPUs have established a new baseline for power efficiency and computing power, delivering larger bandwidth and more computing units in each new generation. Modern GPUs support the concurrent execution of kernels to maximize resource utilization, allowing kernels to better exploit idle resources. However, the decision on the simultaneous execution of different kernels is made by the hardware, and sometimes GPUs do not allow the execution of remaining blocks of other kernels, even with the availability of resources. In this work, we present an in-depth study on the simultaneous execution of kernels in the GPU. We present the necessary conditions for executing kernels simultaneously, we list the factors that influence competition and propose a model that describes performance degradation. Finally, we validate the model using kernels of real-world applications with different use intensities of computation and memory.

Keywords: Concurrent Kernels, Multiprogramming, GPU computing.

List of Figures

List of Tables

List of Abbreviations

CUDA	:	Compute Unified Device Architecture;
CPU	:	Central Processing Unit;
GPU	:	Graphics Processing Unit;
GPGPU	:	General Purpose Graphics Processing Unit;
HPC	:	High Performance Computing;
SIMT	:	Single Instruction Multiple Threads;
SM	:	Streaming Multiprocessor;
SP	:	Streaming Processor;

Contents

Chapter 1

Introduction

This chapter presents the context of the dissertation, problem statement, research general objective, specific objetives, applicability, key contributions and closes with the dissertation outline.

1.1 Context

Graphic Processing Units (GPUs) [1] play a major role in diverse knowledge areas such as physics [14], medical imaging [10], artificial intelligence [50], bioinformatics [15], among others. Compared to traditional multi-core processing units, GPUs provide a high capacity of computation, performing the same calculation several orders of magnitude faster, which allows solving problems considered unfeasible some years ago [29].

In addition to their computing power, GPUs also feature a high-energy efficiency [18], allowing to embed them in many kind of devices: laptops, robots, game consoles, VR headsets and cars. Hence GPUs is no longer only a tool for scientific research but became a commodity component [30].

Due to the increasing computational capacity, often one single application cannot take full advantage of the available resources in GPUs. To address this issue, current GPU architectures allow the execution of multiple applications concurrently.

Despite all the public information available for GPUs, there are no accurate details about how the scheduling of concurrent applications is made [2, 24]. According to Wong

[1] Traditionally GPUs are devices dedicated to graphics rendering. The term General Purpose Graphics Processing Unit (GPGPU) includes the use of GPUs for general-purpose computation. Consequently, both terms may found in the literature as synonyms. Hereafter, we will refer to GPGPUs as simply GPUs.

et al. [54], a kind of reverse engineering to find out some hardware behaviors can be possible. Therefore several authors agree that the internal scheduling of thread blocks follows a round-robin way, though few workloads share the GPU efficiently [25, 42].

With the objective of overcoming the limitations of the current scheduling policy, alternative multitasking techniques have been studied. However, most of them are hardware extensions validated on GPU simulators [4]. Also, researchers propose to optimize the execution of concurrent kernels, but few works take into account the possible negative interactions [20]. We present a performance estimation model that measures the expected run-time slowdown of a kernel taking into account the current distribution of resources for concurrent applications on real hardware.

1.2 Problem Statement and Relevance

Although GPU multitasking is extensively researched, the current support of concurrent execution is primitive and unfair, because resources are assigned between concurrent kernels in an unbalanced way. Suppose a situation where several kernels are scheduled for execution at the same time. There might be cases when the first scheduled kernel drains most of the resources even without completely using them, with the following kernels remaining idle. This behavior prevents the execution of the subsequent kernels, resulting in an ineffective resource utilization.

Several GPU optimization techniques use estimations for taking accurate scheduling decisions. For instance, a successful technique is the use of estimated information to perform auto-tuning of parameters. In the same way, to know the behavior of the inter-application impact before the execution of such programs could be useful [7, 29, 30].

Sharing resources in the cloud age with reliable, easy to use and cheap taxes, brought additional challenges to single-user GPUs devices. The increasing porting of typical CPUs applications to many-core systems demonstrated that it is mandatory that GPUs shortly implement an efficient multitasking support to be adapted to this growing trend [41, 46].

The sharing of GPUs is part of a new technological trend where multiple users can simultaneously submit workloads on-demand. Clouds and supercomputers with GPUs as accelerators are some examples that require concurrent execution support [9]. Frameworks such as StarPU [19] and rCUDA [44] have the capacity to manage this kind of workloads. In the light of the recent efforts to handle efficiently concurrent tasks, our work could be implemented in these frameworks in order to improve the scheduling of applications.

1.3 Research Objectives and Contributions

The objective of this work is to develop a slowdown estimation model for concurrent kernels in GPUs based on resource usage. The key idea is to estimate the deceleration of the execution time that applications may suffer before they execute. This performance degradation can be due to the resource sharing with previous applications that could have been executing. This projection is based on the configuration parameters related to the required resources at compile-time.

This general objective can be broken down into four more specific objectives that would together achieve the overall goal of this dissertation as follows:

- **Investigate the limitations for concurrent execution on GPUs** - The current implementation of concurrent kernel execution on GPUs has limitations. For instance, the scheduling decisions are performed by the hardware in runtime; such decisions depend on the order of the kernel submission that can affect the efficient resource usage. Also, the available GPU resources are not shared equally, some kernels could be executed on a small subset of resources, drastically affecting their overall execution time.

- **Identify scenarios where concurrency of applications is allowed** - It is unclear how the GPU scheduler distributes the available resources. A deeper understanding of how GPUs face simultaneous execution may lead to overcoming the limitations. Also, if we take into account those conditions, the concurrent execution of future workloads will be guaranteed.

- **Identify GPU applications that can benefit from concurrent execution** - Each application demands different resources with mixed intensities, but most of them are unable to entirely occupying all the GPU resources. The possibility of characterizing applications which are concurrent-friendly would allow programmers to avoid some combinations which result in considerable performance loss.

- **Propose a model to estimate the deceleration of a GPU application in concurrent execution** - The analytical model is based on the kernels configurations and the maximum resource limits of a target GPU.

- **Compare the accuracy of the proposed model by using synthetic and real-world applications** - To validate the model, we measure the difference between the estimated and the real performance degradation.

Following are the major contributions of this work:

1. **Determination of the conditions under which is possible to get concurrency of several kernels** - There are some cases where a kernel is prevented from executing their blocks even when there are sufficient resources to support it.

2. **Determination of the point from which two kernels are concurrently executed** - Multiple factors, including the current scheduling policy or resource utilization of each kernel, could cause a high, low or no overlap in concurrent execution.

3. **Specification of the slowdown in individual situations** - We can estimate the performance decrease by making an analytic examination focused on the available resources.

4. **Validation of the proposed model in a broadly known benchmark suite** - These results confirm that the proposed model can be used in different scenarios.

1.4 Dissertation Outline

This work is structured as follows: Chapter 2 describes the background of GPU computing and the details of the current GPU scheduling. Chapter 3 introduce the related works for slowdown estimation in concurrent kernels. Chapter 4 shows the analysis of the concurrency achieved in modern hardware. Chapter 5 introduces our proposed model to estimate the slowdown. Chapter 6 presents the experimental results and discussion. Chapter 7 presents the conclusions and some suggested future works.

Chapter 2

Background

In this chapter, we provide an overview of the GPU programming model and basic scheduling concepts, which are useful to understand our proposed model.

2.1 GPU Programming Model

GPUs implement an execution model known as Single Instruction Multiple Threads (SIMT). The model introduces parallel work organization through concurrent threads and the memory hierarchy of the GPU with different access costs. Opposite to CPUs, GPUs can manage thousands of threads simultaneously. To process some calculations on GPU, the user must define a task, which is defined through functions called kernels. Data is transferred before and after kernels execution to get/set the results. The most popular frameworks for programming GPUs are CUDA and OpenCL, both are based on the C programming language [21].

2.1.1 CUDA

CUDA is a framework that allows the development of applications for NVIDIA GPUs [34]. According to the CUDA programming model, a *kernel* is a function compiled that runs on a CUDA-capable device. A typical parallel CUDA application can execute thousands of threads simultaneously. The set of threads that execute the kernel compose a *grid* of thread blocks. CUDA programmers must define a configuration before executing the kernel; it determines the number of threads per block and the number of blocks per grid.

The launch configuration must be set according to the maximum resources available in the target GPU. Any configuration out of the limits might make impossible to run a

kernel.

When the host invokes a CUDA kernel, the grid is allocated on the GPU resources. GPUs are composed of Streaming Multiprocessors (SMs), each SM has a set of Streaming Processors (SPs) or CUDA cores. Also, each SM has available a set of registers and an amount of shared memory. Figure 2.1 shows the diagram of a standard GPU architecture.

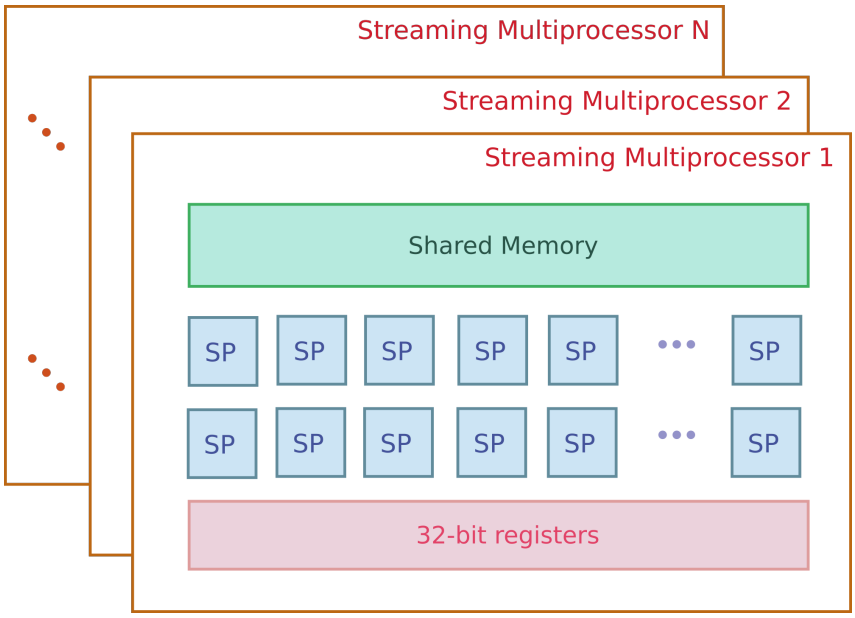

Figure 2.1: A typical GPU is composed of SMs, where each SM contains a set of SPs. Also, each SM has a programmable cache called shared memory and a pool of registers.

At the hardware level, the threads in a block are grouped into *warps* (i.e. groups of 32 threads). Blocks are mapped to SMs in runtime. Each block is scheduled and executed from the beginning to end in one SM.

In CUDA it is possible to represent the dependency of tasks by means of *streams*. A CUDA stream is like a queue, where memory transfers and kernel invocations are enqueued and executed in the same order. Furthermore, the execution of these groups of tasks could be overlapped. This corresponds to another level of concurrency allowed by the use of different streams. The tasks enqueued in more than one stream may be performed in parallel depending on the resources and capabilities of the target device.

2.2 GPU Multitasking

Traditional GPU programming is based on the utilization of the entire GPU by a single kernel that solves a problem which exposed a considerable amount of data parallelism. Therefore, GPU kernels should be able to launch thousands of threads that will work

on different data independently, using as much as possible of the resources. However, several works point out that most of the kernels have unbalanced workloads [1, 41], i.e., kernels often present irregular memory access patterns, and complex control flow behaviors resulting in lower occupancy.

To address this problem, modern GPUs also support task parallelism by allowing the execution of multiple kernels simultaneously. Although the scheduling of kernels is stated in the hardware, and their details are closed and entirely specified by their vendor, various authors proposed some extensions based on one or a combination of the techniques described below:

Cooperative Multitasking - Similar to a sequential execution in the sense that a new kernel is executed on GPU only when the previous kernel voluntarily leaves it. The problem so far is that a kernel not necessarily uses all the resources. Furthermore, small kernels could be severally affected because of long waiting times. Figure 2.2 shows the timeline following the cooperative multitasking technique of two kernels, where k_2 is invoked after k_1. The first kernel, even with a low rate of GPU utilization, does not allow the second kernel to be executed simultaneously.

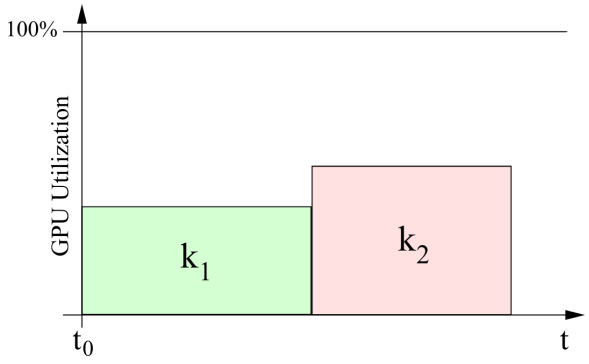

Figure 2.2: Cooperative multitasking. The kernel k_2 even when is ready to be executed must wait to the kernel k_1 to finish before start its execution.

Spatial Multitasking - It is characterized by the potential sharing of the available GPU resources. Figure 2.3 shows an example of two kernels running at the same time in the GPU. Note that under this approach the under-utilization problem is just partially solved. To demonstrate this point, suppose that a kernel k_2 has greater resource requirements that do not fit within the remaining resources left by k_1. Therefore, the resulting behavior will be the same that the cooperative multitasking.

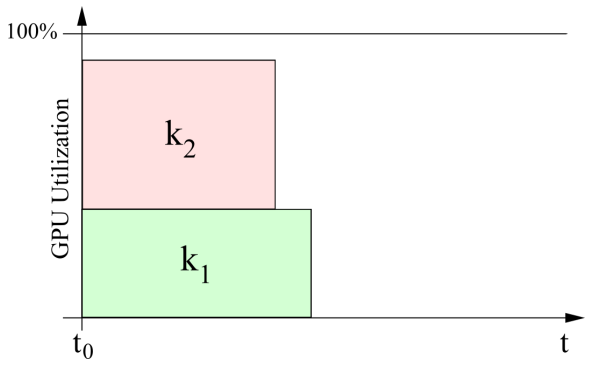

Figure 2.3: Spatial multitasking. Two kernels are executed concurrently whenever there are available resources.

Preemptive Multitasking - The GPU can interleave the execution of concurrent kernels. Moreover, any kernel cannot avoid the execution of subsequent kernels, but after a time slot, it has to give up the GPU resources. Figure 2.4 illustrates the continuous context swapping that allows the execution of kernels simultaneously.

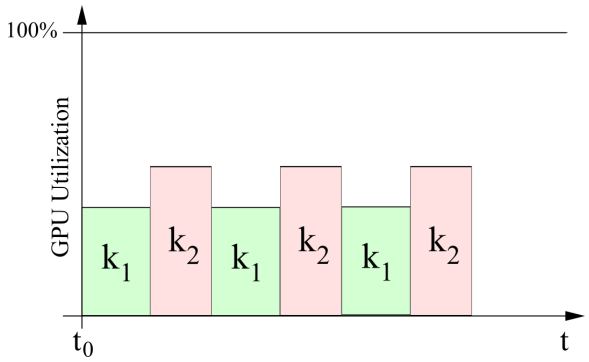

Figure 2.4: Preemptive multitasking. Both kernels k_1 and k_2 are executed on the GPU switching the use of resources until the end of their executions.

Among all of the architectures released by NVIDIA, Pascal GPUs are the only ones that implement preemption at various levels, however due to the lack of public information a better understanding will require many efforts which are out of the scope of this dissertation.

2.3 Occupancy calculation

Occupancy is a metric that represents the amount of parallel work per multiprocessor. Higher occupancy results in higher performance most of the time. Although it depends on several factors, this metric may represent the overall utilization of the resources. Occupancy is defined as the number of active warps on a multiprocessor over the maximum

number of active warps supported by the multiprocessor in the target device, see Equation 2.1.

$$occupancy = \frac{ActiveWarps}{LimitActiveWarps} \tag{2.1}$$

For instance, suppose a GPU that supports 16 active warps per SM, 2 active blocks with 256 threads per block (8 warps per block with 32 threads per warp) results in 16 active warps, and 100% theoretical occupancy. The number of active warps is calculated in Equation 2.2.

$$ActiveWarps = WarpsPerBlock \times ActiveBlocks \tag{2.2}$$

$$WarpsPerBlock = \left\lceil \frac{threadsPerBlock}{limThPerWarp} \right\rceil \tag{2.3}$$

As stated in Equation 2.4, the number of active blocks is defined as the minimum value between the allocatable number of blocks regarding each of the three main resources in GPU, namely, number of threads, number of registers and amount of shared memory.

$$ActiveBlocks = \min(blocksLimByNT, blocksLimByNR, blocksLimBySH) \tag{2.4}$$

Equations 2.5, 2.6 and 2.7 define the number of blocks that may be active considering the total capacity of each type of resource separately.

$$blocksLimByNT = \min\left(limBlocksPerSM, \left\lfloor \frac{limWarpsPerSM}{warpsPerBlock} \right\rfloor\right) \tag{2.5}$$

$$blocksLimByNR = \left\lfloor \frac{limWarpsDueRegs}{warpsPerBlock} \right\rfloor \times \left\lfloor \frac{limRegsPerSM}{limRegsPerBl} \right\rfloor \tag{2.6}$$

$$blocksLimBySH = \left\lfloor \frac{limShMemPerSM}{shPerBlock} \right\rfloor \tag{2.7}$$

2.4 Kernel Concurrency

The need of concurrency in GPUs emerges from the under-utilization of resources when a single kernel cannot leverage all the available resources. Although CPUs and GPUs have significant differences, it is natural that many techniques applied to CPUs will also

be implemented in GPUs. Next, we describe some related technologies that support concurrency in GPUs.

2.4.1 Hyper-Q

GPUs based on NVIDIA Fermi [36] architecture were the first to support concurrent kernel execution in a limited way [32]. To illustrate this limitation, multiple instances of two kernels named as $kernel\_A$ and $kernel\_B$ are invoked in different streams. Both kernels have a grid with only one block, each block containing one thread, so it can be assumed that any combination of up to 16 kernels (a limit on Fermi's) should run concurrently. Figure 2.5 shows the resulting timeline that evidences the problem denominated false-serialization, where all streams queues share a unique hardware queue which causes false dependencies between kernels of the same stream. Thus, independent executions of kernels wait for the finalization of previous executions of the same kernel unnecessarily, even though both calls are in different streams. These timelines were extracted using the NVIDIA Visual Profiler [40].

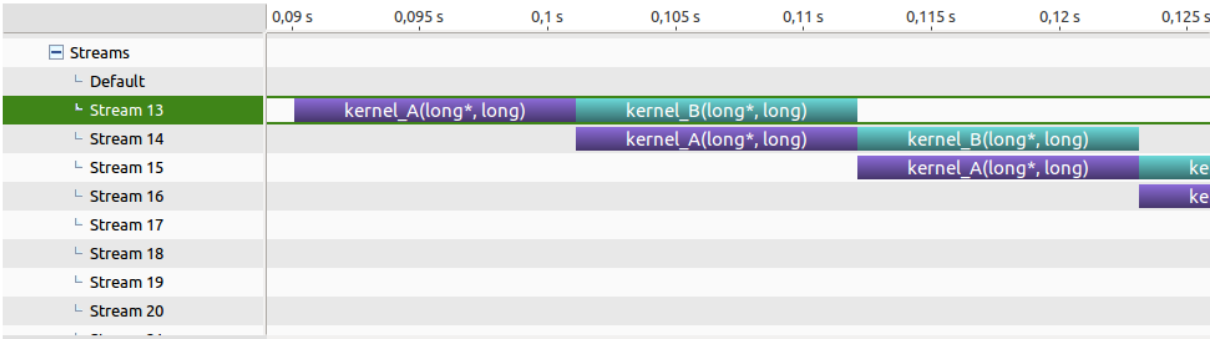

Figure 2.5: Timeline execution in a device without Hyper-Q. The timeline reveals that the concurrent execution is prevented because of the false dependencies between kernels that belong to the same stream.

Starting on the Kepler architecture, a modern hardware solution implements the Hyper-Q technology [25], which is a feature that enables multiple CPU cores to simultaneously utilize up to 32 hardware queues on a single Kepler GPU as seen in Figure 2.6 [1].

2.4.2 Multi-Process Service

CUDA requires a different context for each application. Also, the GPU does not allow the concurrent execution of them. In this scenario, Multi-Process Service (MPS) [38] is

[1]Environment variable CUDA_DEVICE_MAX_CONNECTIONS must be set to 32 (default is 8)

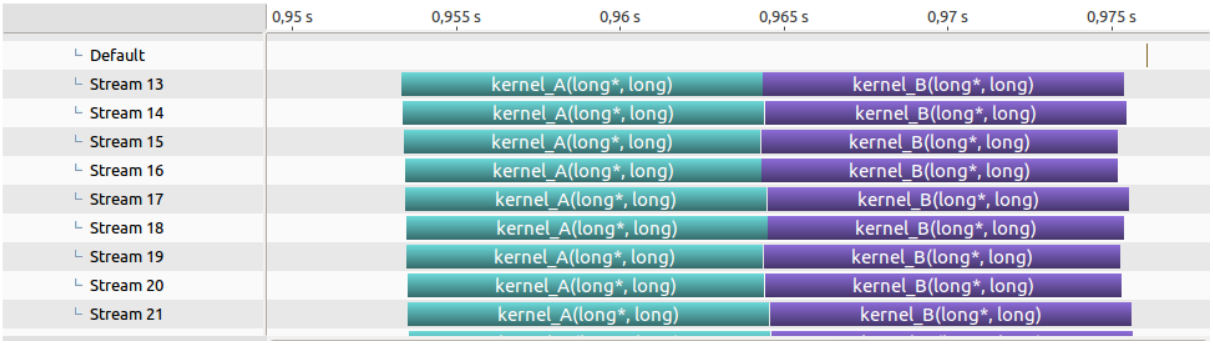

Figure 2.6: Timeline execution in a device with Hyper-Q. Each stream is assigned to a different hardware queue allowing real concurrency.

a software mechanism that allows the concurrent kernel execution by joining different context from various processes leveraging the Hyper-Q technology. MPS shares one only set of scheduling resources between several processes allowing higher overall utilization of the GPU.

Figure 2.7 depicts the timeline execution for all CUDA applications from the Rodinia benchmark managed with MPS. Each horizontal row represents the timeline execution of one application. Blue and green activities represent kernel executions, and the yellow bars represent CPU-GPU memory transfers. Bars overlapped in vertical direction represents the activities that occur at the same time in the GPU. Hence, vertical columns with more than one blue or green activity represent concurrent kernel execution. It can be observed that there is a limited competition for the GPU resources because the kernel invocations are performed in scattered moments. In our work, we implemented a framework with similar behavior that is able to isolate the execution of kernels from other host routines like memory transfers.

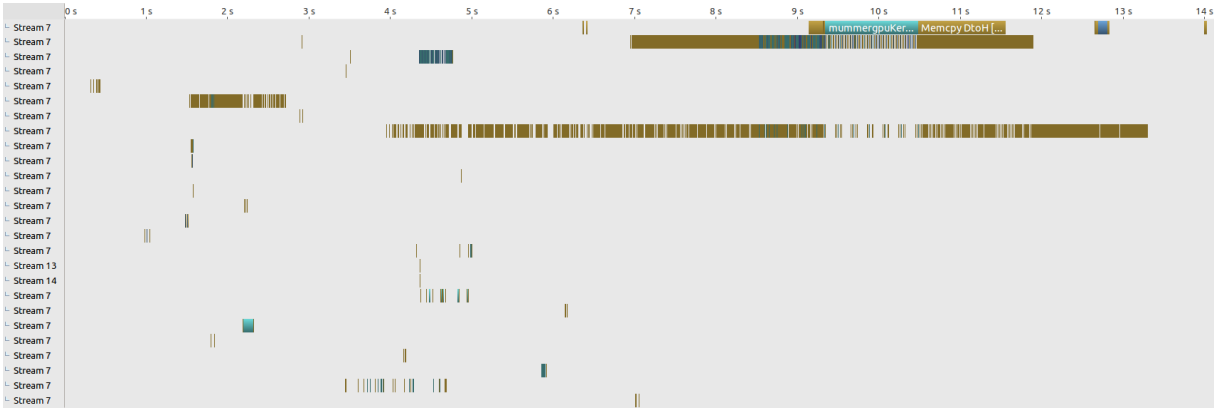

Figure 2.7: Execution timeline of the Rodinia benchmark-suite using the NVIDIA MPS.

2.4.3 Left-over scheduling policy

The current NVIDIA scheduling policy is not publicly available. Some previous work performed microbenchmark experiments to disclose it [17]. The speculation is that the hardware uses a *leftover* policy that assigns as many resources as possible for one kernel and then assigns the remaining resources to another kernel if there are sufficient leftover resources.

Figure 2.8 shows a timeline execution that evidences this leftover policy. Besides that two kernels potentially could share GPU resources from the beginning, the first kernels take the GPU control completely and execute their blocks preventing sharing the remaining resources.

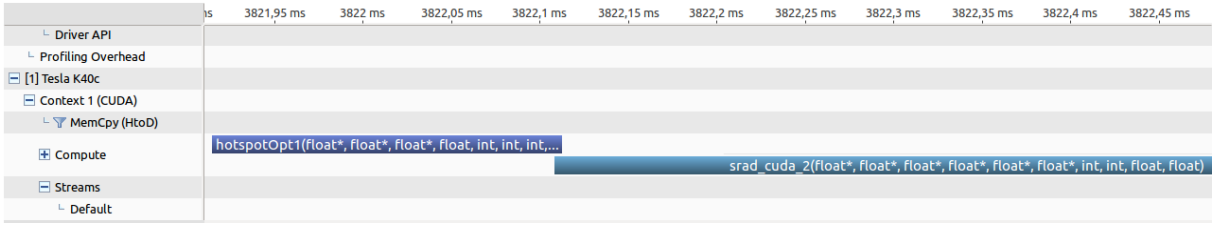

Figure 2.8: Timeline evidencing the leftover policy. There is just a few time slice where both kernels are overlapping.

Chapter 3

Related Work

In this chapter, we present previous work that composes the state-of-art in the literature about how performance estimation of concurrent kernels has evolved. We also present and discuss several works related to our proposed method.

3.1 Concurrent Kernel Execution

Prior to the hardware support of the concurrent execution of kernels, several works address this limitation through developing software techniques that *merge* multiple kernels into one at the source code level [13, 45].

Since the emergence of programmable GPUs, NVIDIA Fermi [36] architecture was the first generation that supports the execution of concurrent kernels. However, due to the existence of one single hardware work queue, some stream dependencies arise and limit the real concurrency [52].

In 2012, NVIDIA released Kepler [37] that is the next generation of GPUs that features Hyper-Q technology. Opposite to its predecessor, Kepler aims to decrease the number of idle resources enabling up to 32 hardware managed queues. Even with Kepler, various authors reported significant rates of resource underutilization [1]. As a consequence, there were proposed techniques such as reordering, preemption, and spatial partitioning.

Some works conclude that it is possible to obtain a permutation that has a better performance than the original order. Accordingly, kernel reordering arises from the overall impact caused by the scheduling order or the limitations imposed by the hardware.

To face the hardware restrictions, Wende et al. [53] explore the performance gains by improving the overall throughput of small-sized GPU kernels within multi-threaded

CUDA applications. The authors introduce a kernel reordering mechanism to avoid false-serialization of kernels caused by the single task queue on Fermi's GPUs. The model is validated with synthetic kernels and a real-world program where it was able to considerably increase the level of concurrency.

Li et al. [23] propose a reordering scheme for GPU architectures with Hyper-Q support. Their scheduling algorithm places priority on executing kernels with complementary resource requirements and a low predicted power consumption.

Breder et al. [6] introduce a reordering approach that can simulate the kernel distribution to the hardware queues aiming to maximize the resource utilization. With that goal, the kernel assignments to the resources are seen as a series of knapsack problems solved with a dynamic programming algorithm. The authors report significant increases of both, system throughput and the average turnaround time with low overhead.

Similarly to reordering, preemption was proposed as another approach to support multiprogramming efficiently. The key idea is to interleave the execution of kernel workloads instead of allocating the needed resources to a kernel until the end of their execution [43, 48, 55].

Alternatively, spatial partitioning divides all multiprocessors between the concurrent applications. Diverse works point out that this method shows improvements over the cooperative execution [2, 24, 46, 49]. Adriaens et al. [1] present an in-depth study of this scheduling technique evaluating four partitions schemes to divide the GPU multiprocessors among applications.

By contrast, we study the current behavior of GPUs regarding concurrent kernel execution where the left-over policy behaves as a spatial scheduling technique unlike most of the previous techniques which propose extensions to the hardware and validate its findings in simulators.

3.2 Performance Estimation Models

A performance estimation model is capable of giving accurate estimations of the execution time or power consumption of a kernel. They are useful in several scenarios, for example, auto-tuning, bottlenecks identification, workload balancing [28]. Hence, different methods were implemented such as analytical models and statistical approaches and those based on simulation [26].

The method proposed by Hong et al. [16] abstracts the CUDA execution model as a set of equations fed mainly with static kernel information. They claim that the overall execution time is closely related to the memory warp parallelism and then their model estimates the cost of memory operations by analyzing the kernel source code.

Another approach is based on the use of machine learning techniques. In this case, the methods find patterns by analyzing some features like performance counters and the hardware characteristics. The model proposed by Wu et al. [56] projects how an application may scale in different GPU configurations running a previously trained model.

Otherwise, a simulator implement in software the execution model of a target device. They allow execute programs compiled for GPUs in a controlled way, however as a drawback, they usually have long execution times.

The three estimation models mentioned in this section consider just one kernel at a time in the GPU. Our work presents a slowdown estimation model considering the execution of more than one kernel concurrently.

3.3 Inter-application Interference in CPUs

On the side of CPUs, diverse works aims to estimate the slowdown of inter-application interference. Subramanian et al. [47] proposed an interference model based on the observation that the concurrent access of applications to memory resources is correlated to their overall slowdown corresponded to the sequential execution.

Alves & Drummond [3] present a quantitative study model addressing the multi-application interference in a cloud multi-core environment. They take into account a number of concurrent accesses to the shared resources and the similarity between those applications accesses. Their model is the result of a multiple regression analysis of multiple executions of synthetic applications. In contrast, our estimation model is based on the current behavior of concurrent applications in GPUs.

3.4 Inter-application Interference in GPUs

Jog et al. [20] observe that applications may have their performance limited because of the contention in the memory subsystem. They implement two memory scheduling policies to improve instruction throughput and the weighted speedup considering the bandwidth

use, and L2-cache misses. However, opposite to our work, they do not propose a slowdown estimation model.

The focus of our work is on the execution of kernels, we assume that the data needed by any kernel is already in GPU memory. Some works explore concurrency between data transfers and kernel computation in-depth [5, 12, 22, 27, 51].

Closer to our work, Hu et al. [17] proposed a slowdown estimation model focused on memory contention of concurrent kernels; their experiments show an error of 8% overall possible two pair combinations of 15 GPU applications. They compare their model with two slowdown estimation models for CPUs applications. Different to our work, their proposed model is an extension of the hardware and was validated in a controlled simulation environment considering a spatial division of the GPU.

Table 3.1 presents a summary of the related works closer to our research. Most of the previous work implement their solutions as hardware extensions in a simulator. In contrast, we propose and validate a model for slowdown estimation with both synthetic and real-world kernels. Additionally, we present a deep study of the necessary conditions for real simultaneous execution in current GPUs considering the actual leftover scheduling policy.

Table 3.1: Summary of the main related works.

Work	**Objective**	**Scheduling Policy**	**Solution**
Adriaens et al. [1]	Multitasking and Kernel characterization	Spatial-Even	Hardware extension
Aguilera et al. [2]	Multitasking	Spatial-Fair	Hardware extension
Lian et al. [24]	Multitasking	Spatial	Software
Jog et al. [20]	Multitasking and Memory analysis	Spatial-Even	Hardware extension
Hu et al. [17]	Slowdown estimation	Spatial-Even	Hardware extension
Our work	Slowdown estimation	Leftover	Software

Chapter 4

Analyzing Concurrency in GPUs

This chapter describes the behavior of hardware when several kernels are scheduled for execution. The different cases of execution overlapping between kernels are described.

4.1 Defining the Factors that Influence Concurrency

The use of streams in CUDA applications should not be considered as sufficient condition to run kernels concurrently; such decision is affected by the thread block scheduler based on the resource usage of each kernel.

Given two kernels k_1 and k_2 submitted to concurrent execution in this respective order in a particular GPU, their execution may fall into one of these three cases: (A) the two kernels are executed concurrently from the beginning, (B) the second kernel start its execution when the first kernel begins to release resources or (C) the two kernels are executed sequentially.

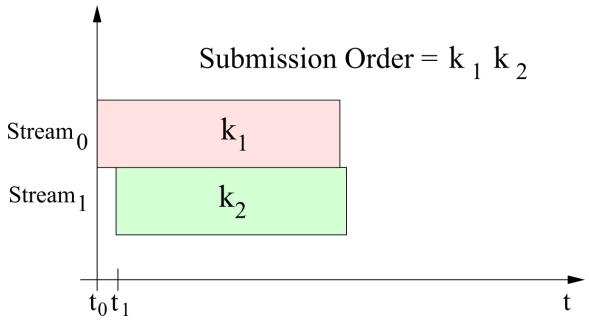

Figure 4.1: Kernels are executed concurrently from beginning (Case A).

Figure 4.1 illustrates the case when two kernels in different streams are invoked, and the device has enough resources for both kernels. In fact, there is an overhead between

both executions, and for this case, the overhead represented by the difference between $t1$ and $t0$ is negligible as the time of the execution of kernels is considerably larger.

The partial concurrent execution of two kernels is illustrated by Figure 4.2. Kernel k_2 starts its execution before the end of kernel k_1 execution. Also, this event is characterized by a larger $t_1 - t_0$ value concerning the launch overhead.

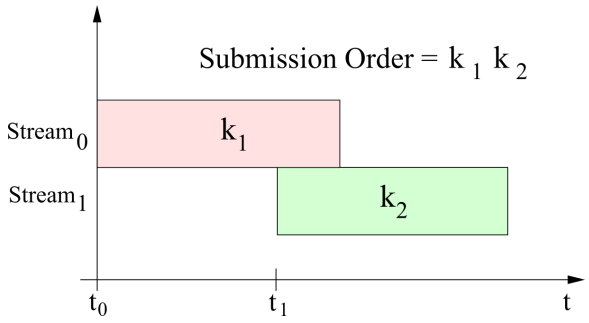

Figure 4.2: Kernels are executed concurrently but not from beginning (Case B).

The third case, as illustrated by Figure 4.3, represents the timeline execution of two kernels when no concurrency was possible.

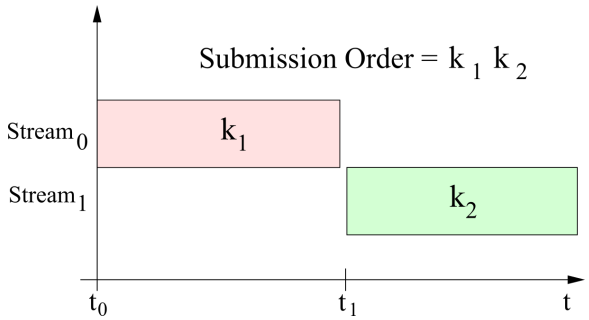

Figure 4.3: Kernels are executed sequentially (Case C).

4.1.1 Unveiling the Concurrent Behavior

We conduct a test to unveil the point from which two kernels can run simultaneously. The experiment begins running two synthetic kernels with few blocks to ensure concurrent execution and then gradually increase that blocks number until getting less concurrency.

The program listing 4.1 represents the invocation of two kernels followed by a call function to synchronize the GPU in order to wait for the finalization of both kernels before the next iteration. The parameters J and K varies between 1 and 256, and the parameter *blocksize* varies between [256,512,1024].

After running the complete suite of tests, we can infer that whenever $Blocks_{k1}$ is

lesser than ($ActiveBlocks_{k1} \times nSM$) both kernels will be executed from the beginning. Otherwise, the second kernel starts when all the blocks from the first kernel were scheduled. Figure 4.4 illustrates the execution of the previous listing when $Blocks_{k1}$ becomes greater than ($ActiveBlocks_{k1} \times nSM$).

Listing 4.1: Host code that executes two kernels with different grid sizes.

```
void testSize(int K, int J,int blocksize)
{
  for(int I=0;I<K;I++)
  {
    synthKernel1<<<I,blockSize,0,stream[0]>>>();
    synthKernel2<<<J,blockSize,stream[1]>>>();
    cudaDeviceSynchronize();
  }
}
```

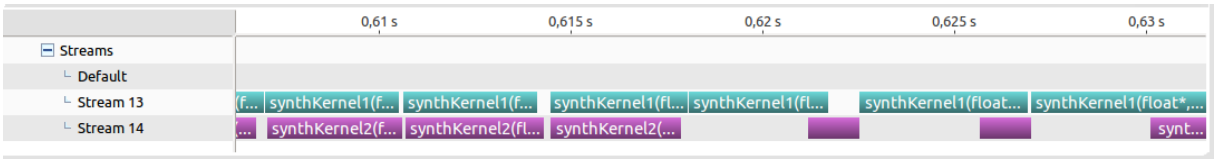

Figure 4.4: Timeline execution when J=1, blockSize=1024, K=64 in GPU Titan X with nSM=24.

4.1.2 Concurrent Scheduling

Algorithm 1 presents the conditions we identified experimentally in section 4.1.1 under which two kernels are executed simultaneously in the GPU. Given the information from compilation and execution of two kernels and the hardware limit configurations of the device, the algorithm identifies whether the hardware will actually execute simultaneously two kernels and at what point that happens. Suppose that the number of SMs in the GPU is nSM and that the overhead of launching a kernel is $Launch\_overhead$. Also suppose that $Blocks_{k1}$ is the total number of blocks of k_1, $ExecTime_{k1}$ is the execution time of k_1.

Initially, the algorithm tests if k_1 execution time is greater than the overhead of launching a kernel on the target device G. If k_1 executes for less than the launching overhead time, the time the GPU takes to launch k_2, k_1 has finished, and no concurrency is achieved.

Algorithm 1 Concurrent scheduling

```
1: function AreExecutedConcurrently(k1, k2, G)
2:     if ExecTime_k1 > Launch_overhead then
3:         cond1 ← Blocks_k1 < (ActiveBlocks_k1 × nSM)
4:         cond2 ← GetAllocatableBlocksPerSM(k1, k2, G) > 0
5:         if (cond1 = true) and (cond2 = true) then
6:             return Case A                ▷ Kernels are executed concurrently from beginning
7:         else if (Blocks_k1 mod (ActiveBlocks_k1 × nSM)) > 0 then
8:             return Case B                ▷ Kernels are executed concurrently but not from beginning
9:         end if
10:    end if
11:    return Case C                        ▷ Kernels are executed sequentially
12: end function
```

The launch overhead could be relevant when the algorithm faces some small kernels with execution time in the order of microseconds. Their value depends on the overall system, as well as the kernel.

For k_1 and k_2 to run concurrently from the beginning (Case A), two conditions must be satisfied. First, k_1 blocks must not occupy all the SMs entirely, so the number of blocks of k_1 must be smaller than the number of blocks of k_1 that can be active in all SMs ($ActiveBlocks_{k1} \times nSM$, which means that k_1 is leaving space for another kernel execution. The second condition tests if at least one block of k_2 can be allocated in the SMs. The function $GetAllocatableBlocksPerSM$ returns the number of blocks from k_2 that can be allocated in a SM after the blocks from k_1 have been already allocated.

The function $GetAllocatableBlocksPerSM$ is shown in Algorithm 2. It examines if there is space for k_2 blocks, according to the amount of leftover resources from k_1 in terms of registers, number of threads and shared memory. First, the algorithm computes the unused resources, $freeRegs$, $freeThreads$, $freeShMem$, by subtracting k_1 allocation from the hardware limits for all these resources. After that, the algorithm computes k_2 allocation on these resources dividing the amount of free resource by the k_2 request on each resource. The number of blocks allocated for k_2 is the minimum of all the possible allocations.

There is also a possibility of concurrent execution, when the number of blocks of k_1 exceeds the amount of blocks that can be active in all SMs (($Blocks_{k1}$ mod ($ActiveBlocks_{k1} \times nSM$)) > 0). In this case, k_1 must be allocated in *waves*. A *wave* represents an interval of time where the maximum number of active thread blocks per multiprocessor from k_1 is executed. For example, suppose that k_1 has 128 blocks, and the GPU allows 8 active blocks to execute on one SM. For a GPU with 2 SMs, k_1 will execute in 8 waves. In regular kernels, we could assume that each wave will have the same value approximately.

Algorithm 2 Calculating allocatable blocks based on free resources

```
1: function GetAllocatableBlocksPerSM(k1, k2, G)
2:     freeRegs ← limitRegsPerSM − (regsPerBlock_k1 × activeBlocks_k1)
3:     freeThreads ← limitThreadsPerSM − (threadsPerBlock_k1 × activeBlocks_k1)
4:     freeShMem ← limitShMemPerSM − (shMemPerBlock_k1 × activeBlocks_k1)

5:     allocBlByRegs ← freeRegs/regsPerBlock_k2
6:     allocBlByThreads ← freeThreads/threadsPerBlock_k2

7:     if shMemPerBlock_k2 = 0 then
8:         allocBlByShMem ← limitBlocksPerSM
9:     else
10:        allocBlByShMem ← freeShMem/shMemPerBlock_k2
11:    end if

12:    return min(allocBlByRegs, allocBlByShMem, allocBlByThreads)
13: end function
```

The number of *waves* is calculated as shown in Equation 4.1.

$$Waves = \left\lceil \frac{Blocks}{ActiveBlocks \times nSM} \right\rceil \tag{4.1}$$

We distinguish the *last wave* as the wave at which there maybe resources left for concurrent execution. In the last wave, if $(Blocks_{k1} \bmod (ActiveBlocks_{k1} \times nSM)) > 0$, it means that k_2 can run concurrently with the remaining blocks of k_1 (Case B).

When k_1 execution time is smaller than the overhead of launching a kernel, or k_1 blocks occupy all the SMs, or the last wave of k_1 does not leave space for k_2 execution, the kernels are executed sequentially (Case C).

$ActiveBlocks_{k1}$ can be calculated by means of Equation 2.4 and represent the number of blocks from k_1 that can be active in one SM, considering these restrictions.

Figure 4.5 represents the occupation of blocks from two kernels when the axis X represents the SMs available in a GPU and the axis Y represents the time. Each kernel execution is composed of several waves.

4.2 Resource Allocation

The current GPU scheduler will first allocate the available resources for k_1 and, if there are leftover resources, will allocate resources for k_2, the number of blocks which can execute concurrently with k_1 on an SM is limited by:

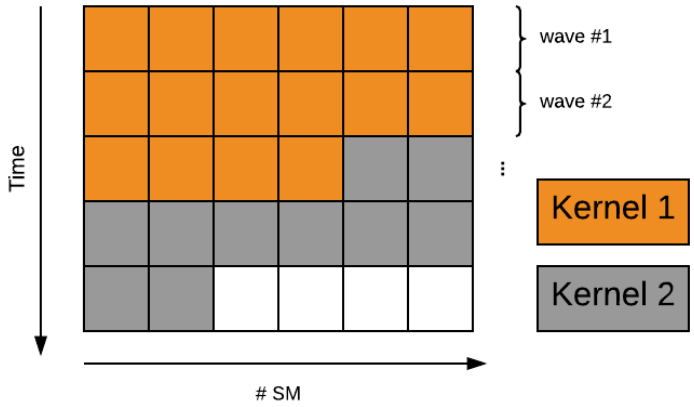

Figure 4.5: The execution of a kernel is composed of waves.

I. The number of thread blocks available on each SM,

II. The maximum active thread blocks imposed by the hardware,

III. The number of thread blocks that the shared memory can accommodate given the consumption of each thread block,

IV. The number of thread blocks that the registers can accommodate given the consumption of each thread block.

The first two resources and only the shared memory which is statically declared is obtained at compile time. The remaining of the shared memory, called as dynamic shared memory is known just after their allocation, at runtime.

For each of the resources mentioned above, we set the maximum limits according to the microarchitecture on which it is based. The list of hardware features associated with each device is best known as their *Compute Capability* [31].

Chapter 5

Proposed Model

This chapter presents the analysis of the concurrent execution and the model we proposed to quantify such effects.

5.1 Model Overview

We observed that it is possible to execute another kernel over the limited resources left by a previous already scheduled kernel.

Given two kernels scheduled for concurrent execution named as k_1 and k_2, we realize that exists some time intervals at which the second kernel k_2 can also be executed with limited resources because there is room to schedule blocks from k_2 in the same multiprocessors that k_1 or because blocks from k_1 were scheduled in just a subset of the multiprocessors. Therefore, the slowdown will be proportional to the available resources.

Also, following the leftover policy, we assume that k_1 is not affected by the scheduling of blocks from k_2, so the interference in this work is related to the use of resources than the contention in deeper levels of warp execution inside multiprocessors like memory accesses.

5.2 Slowdown Estimation

We define the estimated slowdown of kernel 2 on Equation 5.1:

$$slowdown = \frac{WavesWithLimRes}{Waves} \tag{5.1}$$

The value of $WavesWithLimRes$ represent the number of waves in k_2 execution,

considering that there are limited resources available. The idea is to account for the allocatable blocks from k_2 according to the resources leftover by k_1. This is computed according to Equation (5.2).

$$WavesWithLimRes = \left\lceil \frac{Blocks_{k2}}{(nFree \times ActiveBlocks_{k2}) + (nOccupied \times AllocBl_{k2})} \right\rceil \quad (5.2)$$

The computation of the number of waves with limited resources has to consider two cases of the allocation of k_1 blocks on the SMs. In the first case, k_1 blocks fulfill a number of SMs, and leave $nFree$ SMs completely free. In the second case, k_1 blocks use all the SMs but do not fulfill them, leaving space in for k_2 blocks. The number of k_2 blocks that can be allocated per SM in this case is $AllocBl_{k2}$, that is computed by the function $GetAllocatableBlocksPerSM$ presented in Algorithm 2. So, the computation of $WavesLimRes$ is performed by dividing the number of blocks of k_2 by $nFree$ multiplied by the number of blocks from k_2 that can be active in one SM, $ActiveBlocks_{k2}$, or divided by the number of SMs occupied by k_1, $nOccupied$, multiplied by the number of k_2 blocks that can be allocated per SM, $AllocBl_{k2}$. In Table 5.1 we list all needed parameters and their respective source.

Table 5.1: Summary of input parameters used in the proposed model.

Model Parameter	**Definition**	**Source**
Blocks	Total number of blocks in the grid	Kernel configuration
ActiveBlocks	Number of concurrently running blocks on one SM	Theoretical calculation Equation 2.4
regsPerBlock	Number of registers requested by the kernel	Kernel configuration
threadsPerBlock	Number of threads requested by the kernel	Kernel configuration
shMemPerBlock	Amount of shared memory requested by the kernel	Kernel configuration
nSM	Number of multiprocessors of the target device	Device specifications
limitRegsPerSM	Maximum number of register per SM	Device specifications
limitThreadsPerSM	Maximum number of threads per SM	Device specifications
limitShMemPerSM	Maximum amount of shared memory per SM	Device specifications
limitBlocksPerSM	Maximum number of resident blocks per SM	Device specifications
maxWarpsPerSM	Maximum number of warps per multiprocessor	Device specifications

To illustrate an application of our model, consider a GPU G with the specifications of Table 5.2 and two kernels k_1 and k_2, scheduled for execution on the device G with the configurations of Table 5.3.

Thus, we want to estimate the slowdown for k_2 since kernel k_1 is scheduled first. The number of active blocks will be calculated according to the Equation 2.4. In our example

Table 5.2: The specifications of the target GPU.

Parameter	**Value**
nSM	16
limitRegsPerSM	32768
limitThreadsPerSM	1024
limitShMemPerSM	4096
limitBlocksPerSM	8
maxWarpsPerSM	32
threadsPerWarp	32

Table 5.3: Configuration of kernels used in the illustrative example.

Configuration	k_1	k_2
Blocks	16	256
regsPerBlock	512	256
threadsPerBlock	512	256
shMemPerBlock	1024	1024

$ActiveBlocks$ for k_1 and k_2 is 2 and 4, respectively.

The number of waves is established by the Equation 4.1, then k_1 has 1 wave and k_2 has 4 waves.

Since we have that $Blocks_{k1}$ is 16 and $ActiveBlocks_{k_1}$ is 2, the number of free and occupied SMs is 8 for both variables.

Subsequently, the number of the allocatable blocks of k_2 in the SMs occupied by blocks from k_1 is 0, as result of applying the Algorithm 2.

Finally, $WavesWithLimRes$ is equal to 8 and given that $Waves_{k2} = 4$; therefore, the estimated slowdown will be 2, that means that the kernel k_2 will run two times slower than its execution time alone.

Chapter 6

Experimental Results

This chapter presents the hardware and software configurations we used in this study as well as the description of the experiments and the discussion of the results.

6.1 Experimental Setup

Hardware Environment. The experiments were conducted on a dedicated and exclusive machine running Ubuntu 14.04.5 LTS with 8 GB of RAM and powered by an Intel i7-950 @ 3.06 GHz with four cores. The target GPU was an NVIDIA GPU Tesla K40C. Table 6.1 shows the specifications of the GPU used to perform all experiments.

Software settings. The estimation framework was implemented in C++, together with the host codes of the benchmarks, were compiled with g++ version 4.8.4 [11]. The CUDA source codes were compiled with NVCC [33] and CUDA version 7.5 [35].

6.2 Framework for Concurrent Execution

As shown in Section 2.4.2, even when it is possible to run several CUDA applications at once through the MPS framework, few of them share the GPU resources effectively. Most of the time others sections of code dominate the kernel execution, such as data initialization, correctness routines, and GPU-CPU memory transfers.

In order to isolate the execution of kernels, a framework for concurrent execution of kernels was implemented. Given a set of kernels scheduled for execution, the kernels are placed in a waiting queue until all of their associated initialization routines are performed. After that, the framework calls each kernel specifying different CUDA streams to allow

Table 6.1: Target GPU configuration.

	K40
Number of cores	2,880
RAM	12GB
Memory Bandwidth	288 GB/s
Capability	3.5
Number of SMs	15
Shared Memory per SM	48KB
Number of Registers per SM	64K
Max number of threads per SM	2048
Max thread blocks per SM	16
Max registers per thread	255
Maximum thread block size	1024
Architecture	Kepler

concurrency, followed by an explicit synchronization barrier so that after it complementary routines are executed.

6.3 Validation Experiments

In this section, we present the experiments that validate our model. Also, we show the results and discuss them.

6.3.1 Synthetic Applications

In an attempt to evaluate the behavior of our model in scenarios with little or none memory contention, we apply the model to synthetic kernels with variate resource requirements.

Moreover, we generate 100 synthetic kernels consisting of few dummy arithmetic operations working on register values and a loop to consume clock cycles. Each kernel has its corresponding number of blocks, number of threads and shared memory randomly chosen and contained within the limits of the hardware capacity. The configurations of all synthetic kernels are presented in Table 6.2. In this experiment, the first 50 kernels were created to allow concurrency from the beginning (Case A) in such a way that their number of blocks satisfies $Blocks_{k1} < (ActiveBlocks_{k1} \times nSM)$ (Algorithm 1) where the odd-numbered kernels were derived from the first set and the even-numbered were derived from the second set.

Table 6.2: Synthetic applications characteristics

Kernel	# **Blocks**	# **Threads**	**Sh Memory (B)**
S1	110	256	1024
S2	450	256	0
S3	100	256	4096
S4	60	256	0
S5	42	512	256
S6	120	128	0
S7	90	256	896
S8	467	512	256
S9	35	256	2048
S10	130	512	1024
S11	35	256	0
S12	230	256	0
S13	29	512	768
S14	237	512	384
S15	47	512	1536
S16	305	256	512
S17	109	256	1664
S18	292	256	512
S19	65	256	1792
S20	409	128	512
S21	26	256	6400
S22	207	256	9984
S23	165	128	4096
S24	235	128	0
S25	150	256	1024
S26	393	512	8704
S27	26	512	3072

Table 6.2 Continued from previous page

Kernel	# **Blocks**	# **Threads**	**Sh Memory (B)**
S28	245	512	0
S29	21	768	11776
S30	485	128	2560
S31	40	256	1024
S32	400	768	4096
S33	20	1024	8704
S34	72	512	1792
S35	71	256	256
S36	216	128	4096
S37	24	512	7168
S38	478	128	4096
S39	45	256	9728
S40	191	1024	5120
S41	75	256	4096
S42	156	256	9728
S43	37	256	10752
S44	232	128	6912
S45	55	512	3840
S46	441	128	768
S47	51	256	0
S48	305	256	3328
S49	69	128	8448
S50	165	256	6400
S51	18	128	4864
S52	398	256	512
S53	36	512	1024
S54	362	1024	2048

Table 6.2 Continued from previous page

Kernel	# **Blocks**	# **Threads**	**Sh Memory (B)**
S55	70	128	8704
S56	510	1024	4864
S57	19	256	7168
S58	222	512	256
S59	66	128	5120
S60	412	256	11008
S61	72	128	4608
S62	194	1024	6144
S63	94	256	256
S64	65	512	1024
S65	101	128	4352
S66	248	512	512
S67	38	256	4608
S68	219	512	8960
S69	61	256	1024
S70	383	128	768
S71	41	512	1792
S72	201	512	256
S73	45	256	11008
S74	368	128	3328
S75	63	128	1280
S76	155	1024	10752
S77	84	256	4864
S78	488	256	2560
S79	102	256	2048
S80	278	128	512
S81	28	128	256

Table 6.2 Continued from previous page

Kernel	# **Blocks**	# **Threads**	**Sh Memory (B)**
S82	128	128	4864
S83	83	512	8704
S84	387	256	0
S85	24	512	6656
S86	189	512	2048
S87	27	512	1024
S88	175	512	5120
S89	15	1024	2048
S90	340	1024	7936
S91	88	256	0
S92	161	512	2048
S93	64	256	256
S94	101	512	4864
S95	32	512	4096
S96	127	1024	0
S97	28	1024	7168
S98	186	128	1792
S99	33	512	256
S100	109	256	5888

6.3.1.1 Results

The estimated slowdown is compared to the actual slowdown. The real slowdown is given by the ratio between the execution time when the kernel is executed concurrently and the execution time when the kernel is executed alone. The profiling information was obtained using the NVIDIA command-line profiler [39]. Each experiment was repeated 30 times, and the average slowdown was computed.

We test the accuracy of our model by evaluating pairs of kernels from the applications

listed. Table 6.3 introduce the results for the synthetic kernels. The percentage error is computed as in equation 6.1.

$$error = \frac{estimated - actual}{actual} \times 100\% \tag{6.1}$$

Table 6.3: Estimated vs Actual slowdown (synthetic kernels)

Kernel Pair $k1 - k2$	**Estimated**	**Actual**	**Perc. Error**
S1-S2	11.25	11.31	0.55%
S3-S4	3.00	3.02	0.61%
S5-S6	2.00	2.00	0.12%
S7-S8	4.00	3.94	1.59%
S9-S10	1.30	1.33	2.49%
S11-S12	1.50	1.49	0.39%
S13-S14	2.00	1.97	1.36%
S15-S16	4.00	4.06	1.38%
S17-S18	9.00	7.01	28.41%
S19-S20	2.00	2.00	0.01%
S21-S22	1.25	1.25	0.00%
S23-S24	4.00	3.91	2.26%
S25-S26	1.29	1.38	6.53%
S27-S28	1.60	1.60	0.01%
S29-S30	1.67	1.67	0.01%
S31-S32	1.36	1.46	6.92%
S33-S34	2.00	2.00	0.20%
S35-S36	1.50	1.50	0.03%
S37-S38	1.33	1.33	0.03%
S39-S40	2.00	1.89	6.04%
S41-S42	2.00	2.00	0.02%
S43-S44	2.00	2.00	0.00%

Table 6.3 Continued from previous page

Kernel Pair $k1 - k2$	**Estimated**	**Actual**	**Perc. Error**
S45-S46	11.50	10.28	11.89%
S47-S48	1.67	1.67	0.07%
S49-S50	4.00	4.06	1.47%
S51-S52	1.00	1.02	1.55%
S53-S54	2.00	2.31	13.52%
S55-S56	2.00	1.99	0.42%
S57-S58	1.25	1.26	0.71%
S59-S60	2.00	2.01	0.74%
S61-S62	1.86	1.89	1.61%
S63-S64	3.00	3.07	2.17%
S65-S66	1.60	1.80	11.13%
S67-S68	1.50	1.51	0.70%
S69-S70	2.00	2.00	0.18%
S71-S72	2.75	2.76	0.33%
S73-S74	3.44	3.44	0.10%
S75-S76	1.33	1.83	27.28%
S77-S78	2.80	2.81	0.26%
S79-S80	4.00	4.06	1.52%
S81-S82	1.00	1.00	0.10%
S83-S84	1.50	1.52	1.48%
S85-S86	1.33	1.50	11.14%
S87-S88	2.00	1.96	2.22%
S89-S90	1.92	1.96	2.25%
S91-S92	3.00	3.67	18.19%
S93-S94	2.00	2.01	0.72%
S95-S96	2.00	2.03	1.54%
S97-S98	12.00	12.05	0.45%

Table 6.3 Continued from previous page

Kernel Pair $k1 - k2$	**Estimated**	**Actual**	**Perc. Error**
S99-S100	3.00	2.94	2.03%

6.3.2 Real-World Applications

To test our proposed model we use different GPU applications adapted from the Rodinia benchmark suite [8]: k-Nearest Neighbors (kNN), Path Finder (PF), Hotspot 3D (HS3), Breadth-First Search (BFS), Hotspot 2D (HS2), Speckle Reducing Anisotropic Diffusion version 2 (SRAD), LU Decomposition (LUD), and Particle Filter (PFL). Table 6.4 summarizes the kernels used.

Table 6.4: Rodinia applications characteristics

App	**Kernel**	**#Registers**	**# Blocks**	**# Threads**	**Sh Memory (B)**
kNN	euclid	8	3840	256	0
PF	dynproc_kernel	13	463	256	2048
HS3	hotspotOpt1	36	1024	256	0
BFS	Kernel	19	1954	512	0
HS2	calculate_temp	38	1849	256	3072
SRAD	srad_cuda_2	21	16384	256	5120
LUD	lud_diagonal	32	1	16	1024
PFL	KernelFindIndex	13	47	128	0

6.3.2.1 Results

We test the accuracy of our model by evaluating pairs of kernels from the benchmark list. Among all of the possible pair combinations, we present the pairs in which k_1 is the kernel PFL because allows real concurrency from the beginning (Case A), since $Blocks_{PFL} < (ActiveBlocks_{PFL} \times nSM)$. For each pair k_1, k_2, kernel k_1 is issued for execution first, and k_2 is issued next.

Table 6.5 list the results of the pairs of PFL with all the other applications. The comparison of the actual and the estimated slowdown led to an average error of 10.6

Table 6.5: Estimated vs Actual slowdown with PFL first.

Kernel Pair $k1 - k2$	Estimated	Actual	Perc. Error
PFL-kNN	1.250	1.185	5.46%
PFL-PF	1.250	1.252	0.17%
PFL-HS3	1.290	2.409	46.46%
PFL-BFS	1.240	1.357	8.60%
PFL-HS2	1.240	1.260	1.55%
PFL-SRAD	1.250	1.117	11.94%
PFL-LUD	1.000	1.004	0.36%

It can be observed that the highest error was produced by the combination PFL-HS3. For that experiment, our model estimated a smaller slowdown than the real achieved value.

Because of the highest percentage of usage of the memory bandwidth of HS3 application, around 60%, the higher error suggests that the interference due to memory contention is a significant factor. For the other applications, we can observe that the main source of performance reduction in concurrent execution is the amount of resources leftover by PFL kernel.

Chapter 7

Conclusions and Future Works

This chapter concludes this dissertation. It briefly reiterates the achievements, and the results are wrapped up on section 7.1. It also checks how the outcome matched the general research objective set out in Section 1.3, and its contribution to academic community. Finally, Section 7.2 present some directions for extension of this research.

7.1 Concluding Remarks

This research developed an study on the concurrent kernel execution in the GPU. In order to boost the actual resource utilization of GPU, current architectures allow the execution of multiple kernels simultaneously. Nonetheless, GPUs delivers a leftover policy which assigns resources as can be conceded for one kernel and then distributes the remaining resources to another kernel.

Taking into consideration this policy, a resource-hungry kernel will anticipate the execution of other small kernels. Due to this, we have listed the main conditions for parallel execution and also presented an algorithm which details the time when concurrency takes part.

This work concludes that a model to estimate the slowdown of performance is feasible. We corroborated our slowdown model in synthetic and real-life applications. Thus, the model presented was qualified to estimate the slowdown in diverse resource requirements situations.

For the synthetic applications, which do not have memory interference, the model estimated the slowdown with an average error of 3.49%. For real-life applications, the error was higher, reaching up to 10.6% on average. The application with the most important

memory bandwidth demand was the one that had the biggest error.

Our results confirm that the GPU resources are distributed between the kernels, but the hardware does not consider a fair scheduling policy. Therefore, it is essential to establish the kernel characteristics to co-schedule applications with complementary resource requirements.

7.2 Future Works

The proposed and implemented model opens numerous research opportunities for future works which can be improved and extended. Our future work includes assessing the impact of memory interference in the model. Further, we plan to investigate memory contention in real-life applications to adapt it to our model. Besides, we aim to study the impact of applying the model presented in different GPU architectures.

References

[1] Adriaens, J. T.; Compton, K.; Kim, N. S.; Schulte, M. J. The case for GPGPU spatial multitasking. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on* (2012), IEEE, pp. 1–12.

[2] Aguilera, P.; Morrow, K.; Kim, N. S. Fair share: Allocation of GPU resources for both performance and fairness. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on* (2014), IEEE, pp. 440–447.

[3] Alves, M. M.; de Assumpção Drummond, L. M. A multivariate and quantitative model for predicting cross-application interference in virtual environments. *Journal of Systems and Software 128* (2017), 150–163.

[4] Bakhoda, A.; Yuan, G. L.; Fung, W. W.; Wong, H.; Aamodt, T. M. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (2009), IEEE, pp. 163–174.

[5] Belviranli, M. E.; Khorasani, F.; Bhuyan, L. N.; Gupta, R. Cumas: Data transfer aware multi-application scheduling for shared GPUs. In *Proceedings of the 2016 International Conference on Supercomputing* (2016), ACM, p. 31.

[6] Breder, B.; Charles, E.; Cruz, R.; Clua, E.; Bentes, C.; Drummond, L. Maximizando o uso dos recursos de GPU através da reordenação da submissão de kernels concorrentes. In *Anais do WSCAD 2016 Simpósio de Sistemas Computacionais de Alto Desempenho* (2016), Editora da Sociedade Brasileira de Computação (SBC), pp. 98–109.

[7] Brodtkorb, A. R.; Hagen, T. R.; Sætra, M. L. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing 73*, 1 (2013), 4–13.

[8] Che, S.; Boyer, M.; Meng, J.; Tarjan, D.; Sheaffer, J. W.; Lee, S.-H.; Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (2009), Ieee, pp. 44–54.

[9] Diab, K. M.; Rafique, M. M.; Hefeeda, M. Dynamic sharing of gpus in cloud systems. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International* (2013), IEEE, pp. 947–954.

[10] Eklund, A.; Dufort, P.; Forsberg, D.; LaConte, S. M. Medical image processing on the GPU–past, present and future. *Medical image analysis 17*, 8 (2013), 1073–1094.

[11] GNU. Gcc, the gnu compiler collection. `https://gcc.gnu.org`.

[12] Gómez-Luna, J.; González-Linares, J. M.; Benavides, J. I.; Guil, N. Performance models for asynchronous data transfers on consumer graphics processing units. *Journal of Parallel and Distributed Computing 72*, 9 (2012), 1117–1126.

[13] Gregg, C.; Dorn, J.; Hazelwood, K. M.; Skadron, K. Fine-grained resource sharing for concurrent gpgpu kernels. In *HotPar'12 Proceedings of the 4th USENIX conference on Hot Topics in Parallelism* (2012), USENIX Association Berkeley.

[14] Harju, A.; Siro, T.; Canova, F. F.; Hakala, S.; Rantalaiho, T. Computational physics on graphics processing units. In *International Workshop on Applied Parallel Computing* (2012), Springer, pp. 3–26.

[15] Harvey, M.; De Fabritiis, G. A survey of computational molecular science using graphics processing units. *Wiley Interdisciplinary Reviews: Computational Molecular Science 2*, 5 (2012), 734–742.

[16] Hong, S.; Kim, H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News* (2009), vol. 37, ACM, pp. 152–163.

[17] Hu, Q.; Shu, J.; Fan, J.; Lu, Y. Run-time performance estimation and fairness-oriented scheduling policy for concurrent GPGPU applications. In *Parallel Processing (ICPP), 2016 45th International Conference on* (2016), IEEE, pp. 57–66.

[18] Huang, S.; Xiao, S.; Feng, W.-c. On the energy efficiency of graphics processing units for scientific computing. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (2009), IEEE, pp. 1–8.

[19] Janzén, J.; Black-Schaffer, D.; Hugo, A. Partitioning GPUs for improved scalability. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2016 28th International Symposium on* (2016), IEEE, pp. 42–49.

[20] Jog, A.; Kayiran, O.; Kesten, T.; Pattnaik, A.; Bolotin, E.; Chatterjee, N.; Keckler, S. W.; Kandemir, M. T.; Das, C. R. Anatomy of GPU memory system for multi-application execution. In *Proceedings of the 2015 International Symposium on Memory Systems* (2015), ACM, pp. 223–234.

[21] Kim, H.; Vuduc, R.; Baghsorkhi, S.; Choi, J.; Hwu, W.-m. Performance analysis and tuning for general purpose graphics processing units (GPGPU). *Synthesis Lectures on Computer Architecture 7*, 2 (2012), 1–96.

[22] Lázaro-Muñoz, A. J.; González-Linares, J. M.; Gómez-Luna, J.; Guil, N. Efficient opencl-based concurrent tasks offloading on accelerators. *Procedia Computer Science 108* (2017), 2353–2357.

[23] Li, T.; Narayana, V. K.; El-Ghazawi, T. A power-aware symbiotic scheduling algorithm for concurrent GPU kernels. In *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on* (2015), IEEE, pp. 562–569.

[24] Liang, Y.; Huynh, H. P.; Rupnow, K.; Goh, R. S. M.; Chen, D. Efficient GPU spatial-temporal multitasking. *IEEE Transactions on Parallel and Distributed Systems 26*, 3 (2015), 748–760.

[25] Liang, Y.; Li, X. Efficient kernel management on GPUs. *ACM Transactions on Embedded Computing Systems (TECS) 16*, 4 (2017), 115.

[26] Lopez-Novoa, U.; Mendiburu, A.; Miguel-Alonso, J. A survey of performance modeling and simulation techniques for accelerator-based computing. *IEEE Transactions on Parallel and Distributed Systems 26*, 1 (2015), 272–281.

[27] Luley, R. S.; Qiu, Q. Effective utilization of CUDA hyper-q for improved power and performance efficiency. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International* (2016), IEEE, pp. 1160–1169.

[28] Meng, J.; Morozov, V. A.; Kumaran, K.; Vishwanath, V.; Uram, T. D. Grophecy: GPU performance projection from cpu code skeletons. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), ACM, p. 14.

[29] Mittal, S.; Vetter, J. S. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys (CSUR) 47*, 4 (2015), 69.

[30] Navarro, C. A.; Hitschfeld-Kahler, N.; Mateu, L. A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Communications in Computational Physics 15*, 2 (2014), 285–329.

[31] NVIDIA. CUDA c programming guide v8.0. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities`.

[32] NVIDIA. Kepler tuning guide. `http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html#concurrent-kernels`.

[33] NVIDIA. Nvidia CUDA compiler. `http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc`.

[34] NVIDIA. Nvidia CUDA parallel computing platform. `http://www.nvidia.com/object/cuda_home_new.html`.

[35] NVIDIA. Nvidia CUDA toolkit. `https://developer.nvidia.com/cuda-toolkit`.

[36] NVIDIA. NVIDIA fermi architecture whitepaper. `https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[37] NVIDIA. NVIDIA kepler GK110 architecture whitepaper. `https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[38] NVIDIA. Nvidia multi-process service. `https://docs.nvidia.com/deploy/mps/index.html`.

[39] NVIDIA. Nvidia profiler tool. `http://docs.nvidia.com/cuda/profiler-users-guide/index.html`.

[40] NVIDIA. Nvidia visual profiler. `https://developer.nvidia.com/nvidia-visual-profiler`.

[41] O'Neil, M. A.; Burtscher, M. Microarchitectural performance characterization of irregular GPU kernels. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on* (2014), IEEE, pp. 130–139.

[42] Pai, S.; Thazhuthaveetil, M. J.; Govindarajan, R. Improving GPGPU concurrency with elastic kernels. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 407–418.

[43] Park, J. J. K.; Park, Y.; Mahlke, S. Chimera: Collaborative preemption for multitasking on a shared GPU. *ACM SIGARCH Computer Architecture News 43*, 1 (2015), 593–606.

[44] Peña, A. J.; Reaño, C.; Silla, F.; Mayo, R.; Quintana-Ortí, E. S.; Duato, J. A complete and efficient cuda-sharing solution for hpc clusters. *Parallel Computing 40*, 10 (2014), 574–588.

[45] Peters, H.; Koper, M.; Luttenberger, N. Efficiently using a CUDA-enabled GPU as shared resource. In *IEEE 10th International Conference on Computer and Information Technology (CIT)* (2010), IEEE, pp. 1122–1127.

[46] Ravi, V. T.; Becchi, M.; Agrawal, G.; Chakradhar, S. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the 20th international symposium on High performance distributed computing* (2011), ACM, pp. 217–228.

[47] Subramanian, L.; Seshadri, V.; Ghosh, A.; Khan, S.; Mutlu, O. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th International Symposium on Microarchitecture* (2015), ACM, pp. 62–75.

[48] Tanasic, I.; Gelado, I.; Cabezas, J.; Ramirez, A.; Navarro, N.; Valero, M. Enabling preemptive multiprogramming on GPUs. In *ACM SIGARCH Computer Architecture News* (2014), vol. 42, IEEE Press, pp. 193–204.

[49] Ukidave, Y.; Kalra, C.; Kaeli, D.; Mistry, P.; Schaa, D. Runtime support for adaptive spatial partitioning and inter-kernel communication on gpus. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on* (2014), IEEE, pp. 168–175.

[50] Upadhyaya, S. R. Parallel approaches to machine learning–a comprehensive survey. *Journal of Parallel and Distributed Computing 73*, 3 (2013), 284–292.

[51] Van Werkhoven, B.; Maassen, J.; Seinstra, F. J.; Bal, H. E. Performance models for CPU-GPU data transfers. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on* (2014), IEEE, pp. 11–20.

[52] Wang, L.; Huang, M.; El-Ghazawi, T. Exploiting concurrent kernel execution on graphic processing units. In *High performance computing and simulation (HPCS), 2011 international conference on* (2011), IEEE, pp. 24–32.

[53] Wende, F.; Cordes, F.; Steinke, T. On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering. In *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on* (2012), IEEE, pp. 74–83.

[54] Wong, H.; Papadopoulou, M.-M.; Sadooghi-Alvandi, M.; Moshovos, A. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on* (2010), IEEE, pp. 235–246.

[55] Wu, B.; Liu, X.; Zhou, X.; Jiang, C. Flep: Enabling flexible and efficient preemption on GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017), ACM, pp. 483–496.

[56] Wu, G.; Greathouse, J. L.; Lyashevsky, A.; Jayasena, N.; Chiou, D. GPGPU performance and power estimation using machine learning. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on* (2015), IEEE, pp. 564–576.