Analyzing and Estimating the Performance of Concurrent Kernels Execution on GPUs

Rommel Cruz[1], **Lucia Drummond**[1], **Esteban Clua**[1], **Cristiana Bentes**[2]

[1]Institute of Computing, Federal Fluminense University, RJ - Brazil

[2]Department of System Engineering, State University of Rio de Janeiro, RJ - Brazil

{rquintanillac,esteban,lucia}@ic.uff.br, cris@eng.uerj.br

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 other 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 blocks from other kernels, even with the availability of resources. In this work, we present an in-depth study on the simultaneous execution of kernels on the GPU. We present the necessary conditions for executing kernels simultaneously, we define the factors that influence competition, and describe a model that can determine performance degradation. Finally, we validate the model using synthetic and real-world kernels with different computation and memory requirements.*

1. Introduction

Graphics Processing Units (GPUs) have emerged as a cost-effective platform for high performance computing and has become ubiquitous as accelerator devices. Modern GPUs contain thousands of computing cores, very large register files, hardware thread management, and access to fast on-chip and high-bandwidth external memories. Programming models like CUDA exploit this processing power using massive thread-level parallelism, where applications are offloaded to the GPU as *kernels*.

As the GPU resources continue to increase, sharing these resources between different applications becomes imperative. However, GPUs need to be able to efficiently handle a variety of applications and provide compatible throughput to be used as shared devices. Still, GPUs are not multiprogrammed devices with an operating system as are the current CPUs. NVIDIA GPUs have hardware support for simultaneous execution of kernels. However, the hardware policy used in scheduling the kernels is proprietary, and no explicit information has been made available. Previous experiments indicate that the resource partitioning is not even among kernels, but the scheduling policy follows a *leftover* strategy [Pai et al. 2013, Aguilera et al. 2014]. The first kernel allocates all the resources needed and, then, the leftover resources are distributed to the next kernel. So, even if the kernels are independent and submitted to run concurrently, the first kernel may consume too many resources and contrain the concurrent execution with another kernel.

Although some spatial multitasking mechanism have been proposed to improve the GPU throughput [Adriaens et al. 2012, Janzén et al. 2016], they are not currently implemented in the hardware. In the current scheduling policy, the order in which kernels are submitted for execution and their resource usage have great impact on the system throughput, occupancy rate, and GPU utilization.

This work presents an in-depth study on the simultaneous execution of kernels in the GPU. We studied the necessary conditions for real simultaneous execution, proposing an algorithm that identifies whether the hardware will actually execute two kernels simultaneously. We also analyze the effects of the concurrent execution on the performance of the two kernels and propose a model for slowdown estimation. Our concurrency algorithm and slowdown estimation model are validated with both synthetic and real-world kernels that have different computation and memory requirements.

The remainder of this paper is organized as follows. Section 2 presents previous work on concurrent execution on the GPU. Section 3 explains the concurrent execution environment of a NVIDIA GPU. Section 4 shows the algorithm proposed to determine whether two kernels will execute concurrently on the GPU. Section 5 presents the slowdown estimation model. Section 6 validates our algorithm and model with synthetic and real-world applications execution. Finally, section 7 presents our conclusions and directions for future work.

2. Related Work

In contrast to CPU multiprogramming, GPU multiprogramming is a relatively new trend, and still largely unexplored. There are only a few works that address the interference on concurrent kernel execution. Most of these works focus only on memory interference. Hu *et al.* [Hu et al. 2016] introduced a slowdown estimation model for GPUs, whose focus is on memory contention of concurrent kernels. They applied two CPU interference models known as MISE and ASM [Subramanian et al. 2015] to some GPU applications. These multicore models are based on the observation that the concurrent access of applications to memory resources is correlated to their overall slowdown compared to their sequential execution. However, this approach resulted in predictions with low accuracy. In a similar direction, and also seeking a balanced execution, Jog *et al.* [Jog et al. 2015] proposed a low-level memory scheduling mechanism that extends the hardware memory scheduler to a more fair policy relying on the bandwidth and L2 behavior of kernels. In contrast to our work, these interference studies consider an ideal case where the GPU resources are statically assigned to each kernel, while we show the behavior of the actual thread block scheduler of modern GPUs. Consequently, we perform our experiments in real hardware instead of validating our proposal using a GPU simulator.

There are also some works that focus on the CPU-GPU memory interference. Ausavarungnirun [Ausavarungnirun 2017] analyzed three types of memory interference in a CPU-GPU system, and propose an application-aware CPU-GPU memory request scheduler. Jeong *et al.* [Jeong et al. 2012] proposed a memory scheduler that guarantees the performance of GPU applications by prioritizing graphics applications over CPU applications. Our work, on the other hand, focuses on the interference of concurrent kernels execution.

On a different direction, improving GPU utilization with concurrent execution was studied in several previous works. Software techniques, such as reordering [Wende et al. 2012, Li et al. 2015, Breder et al. 2016] focus on the order in which GPU kernels are invoked on the host side. Hardware techniques, such as preemption [Tanasic et al. 2014, Park et al. 2015] control the resource usage by applying preemptive multitasking on the GPU. GPU virtualization techniques [Li et al. 2011, Suzuki et al. 2014] allow multiple VMs to share the GPU resources among the cores in a heterogeneous system.

Several studies on GPU benchmark characterization [Che et al. 2010, Goswami et al. 2010, Lal et al. 2014, Ukidave et al. 2015] contribute to demonstrate that applications with irregular memory access patterns and complex control flow behaviors usually produce kernels that do not take advantage of all the GPU resources. According to Pai *et al.* [Pai et al. 2013], the Parboil2 benchmark suite uses only from 20% to 70% of the Fermi GPU resources. Adriens *et al.* [Adriaens et al. 2012] perform similar studies for 12 real-world applications, and show that most of them exhibit unbalanced GPU resource utilization.

3. Concurrent Kernel Execution

In CUDA, the parallel task submitted to run on the GPU is called a *kernel*. Each kernel can have tens of thousands of threads organized into *blocks* that are assigned to run on Streaming Multiprocessors (SMs).

The first generations of NVIDIA GPUs were not able to execute more than one kernel at a time, and the hardware resources were exploited using only thread-level parallelism. Concurrent kernel execution was introduced in the Fermi architecture, with the concept of *streams*. CUDA streams allows the programmer to express execution independence. Kernels that belong to different streams do not depend on each other and can execute concurrently. On the Fermi architecture, the hardware was responsible for multiplexing the kernels into a single work queue, where two successive kernels can execute concurrently if they were assigned to different streams. This earliest form of kernel concurrency in the GPU, however, can cause false-serialization. If two successive kernels on the queue belong to the same stream, they have to complete before additional kernels in different streams can be executed. The Kepler architecture provides the Hyper-Q technology, where 32 hardware work queues were introduced. False serialization can still occur for more than 32 streams. Recently, NVIDIA introduced the Multi-Process Service (MPS) that enables kernels from different applications to share the GPU resources. The MPS server filters the work from different processes and submit to concurrent execution.

The GPU scheduler assigns blocks to run on a specific SM, without the possibility of runtime migration. Within each SM, the threads are scheduled in the GPU scheduling unit called *warp*. The warp scheduler multiplexes the warps to execute in the CUDA cores of the SM. All the threads that belong to the same warp are executed simultaneously. A warp is considered *active* from the time its threads begin executing to the time when all threads in the warp have exited from the kernel. There is a maximum number of warps which can be active on a SM described by the compute capability of the device.

The NVIDIA scheduling policy is, however, proprietary. No published material describes the policy used for block and warp scheduling. Some previous work performed microbenchmark experiments to disclose it [Hu et al. 2016]. 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. Using this policy, a resource-hungry kernel can prevent the concurrent execution of other small kernels. According to Pai *et al.* [Pai et al. 2013], around 50% of the kernels from the Parboil2 and Rodinia 2 benchmark suites consume too many resources and prevent concurrent execution of other kernels.

Another important aspect of the concurrent kernel execution is the interference that one kernel cause in the other due to simultaneous execution. Inter-kernel interference can have a negative impact in the application execution time. In the following sections, we propose a thorough analysis of concurrent execution in the GPU. We first identify whether the concurrency can occur between two independent kernels. Then, we model the slowdown caused by inter-kernel interference. The correct characterization of the concurrency capability may be an important step for maximizing GPU usage and kernels throughput.

4. Analyzing Concurrency in GPUs

Although concurrent kernel execution can be easily expressed using streams or the MPS tool, the actual simultaneous execution depends on the hardware leftover policy. We propose here an algorithm to identify whether the hardware will actually execute simultaneously two kernels submitted for concurrent execution.

Suppose two kernels k_1 and k_2 were submitted to concurrent execution in this order in a particular GPU. Basically, each attempt to run k_1 and k_2 concurrently may fall into 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, (c) the two kernels are executed sequentially. Figure 1 illustrates these cases.

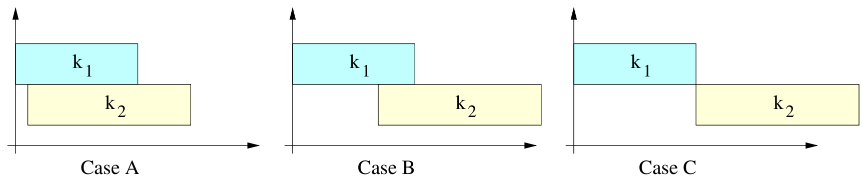

Figure 1. Concurrent execution possibilities.

We present in Algorithm 1 what we identified experimentally as the guidelines to actual concurrent execution. 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.

Since the 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: (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. We define $ActiveBlocks_{k1}$ as the number of blocks from k_1 that can be active in one SM, considering these restrictions.

Initially, the algorithm tests if k_1 execution time is greater than the overhead of launching a kernel on the target device. 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($k_1, k_2, G$)
 2:     if $ExecTime_{k1} > Launch\_overhead$ then
 3:         $cond1 \leftarrow Blocks_{k1} < (ActiveBlocks_{k1} \times nSM)$
 4:         $cond2 \leftarrow GetAllocatableBlocksPerSM(k1, k2, G) > 0$
 5:         if $(cond1 = true)$ and $(cond2 = true)$ then
 6:             return $Case\ A$                ▷ Kernels are executed concurrently from beginning
 7:         else if $(Blocks_{k1} \bmod (ActiveBlocks_{k1} \times nSM)) > 0$ 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
```

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.

Algorithm 2 Calculating allocatable blocks according free resources

```
 1: function GetAllocatableBlocksPerSM($k_1, k_2, G$)
 2:     $freeRegs \leftarrow limitRegsPerSM - (regsPerBlock_{k1} \times activeBlocks_{k1})$
 3:     $freeThreads \leftarrow limitThreadsPerSM - (threadsPerBlock_{k1} \times activeBlocks_{k1})$
 4:     $freeShMem \leftarrow limitShMemPerSM - (shMemPerBlock_{k1} \times activeBlocks_{k1})$

 5:     $allocBlByRegs \leftarrow freeRegs/regsPerBlock_{k2}$
 6:     $allocBlByThreads \leftarrow freeThreads/threadsPerBlock_{k2}$

 7:     if $shMemPerBlock_{k2} = 0$ then
 8:         $allocBlByShMem \leftarrow limitBlocksPerSM$
 9:     else
10:         $allocBlByShMem \leftarrow freeShMem/shMemPerBlock_{k2}$
11:     end if

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

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 ($(Blocks_{k1} \bmod (ActiveBlocks_{k1} \times nSM)) > 0$). In this case, k_1 must be allocated in *rounds*. A round represents an execution of part of the blocks of k_1. 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 rounds. The number of rounds is calculated as shown in equation (1).

$$Rounds = \left\lceil \frac{Blocks}{ActiveBlocks \times nSM} \right\rceil \tag{1}$$

We distinguish the *last round* as the round at which there maybe resources left for concurrent execution. In the last round, 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 round of k_1 does not leave space for k_2 execution, the kernels are executed sequentially (Case C).

5. Slowdown Estimation

The algorithm explained in Section 4 exposes the conditions under which two kernels actually execute concurrently in the GPU. In this section, we analyze the effects of the concurrent execution on the performance of the two kernels. We propose a slowdown estimation model that quantifies the slowdown in k_2 when it is executed concurrently with k_1 since the beginning (Case A).

According to the leftover policy, we assume that k_1 execution is not affected by the scheduling of k_2 blocks. Our slowdown estimation model considers only the performance reduction due to the lack of SM resources to the second kernel. It does not consider the interference caused by contention in memory access, which is quite difficult to quantify. Usually it requires a heavily instrumented GPU simulator and also requires a fair partition of the SMs across the concurrent applications [Jeong et al. 2012, Ausavarungnirun 2017]. GPU architectures evolve rapidly, and there is no available GPU simulator for the current architecture.

We define the estimated slowdown of k_2 as follows:

$$slowdown = \frac{RoundsLimRes}{Rounds} \tag{2}$$

$RoundsLimRes$ accounts for the number of rounds 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 (3).

The computation of the number of rounds 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 are allocated to $nOccupied$ SMs, and leave $nFree$ SMs completely free. In the second case, k_1 blocks use all the SMs but do not fill them, leaving space 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 $RoundsLimRes$ 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, $AllocBl_{k2}$.

$$RoundsLimRes = \left\lceil \frac{Blocks_{k2}}{(nFree \times ActiveBlocks_{k2}) + (nOccupied \times AllocBl_{k2})} \right\rceil \quad (3)$$

6. Experimental Results

In this section, we validate our slowdown model through the execution of pairs of concurrent kernels on the GPU. We first show the results using synthetic applications whose size and resource usage can be varied experimentally. After that, we execute pairs of real-world kernels from the Rodinia benchmark suite.

6.1. Hardware Environment

The results were obtained by direct measurements on a GPU K40. Table 1 shows the device specifications. The profiling information was obtained using the NVIDIA command-line profiler. Each experiment was repeated 30 times and we measured the average slowdown.

Table 1. GPUs configurations

	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

6.2. Framework for Concurrent Execution

In most of the GPU applications, before launching a kernel, the application has to transfer data from the CPU to the GPU and after the kernel finishes, the data has to be copied back from the GPU to the CPU. So, if two applications are submitted to execute at the same time, there maybe no actual concurrency among their kernels depending on the time taken for memory transfers. Since we are interested in the actual kernel concurrency, we implemented a framework that isolates the execution of the kernels.

Our framework guarantees that the kernels are submitted to execute at the same time, generating potential concurrency. A set of kernels are placed in a waiting queue until all of their associated initialization and memory transfers were performed. After that, the framework launches the kernels on different CUDA streams. The framework inserts a synchronization barriers before and after launching the kernels.

The framework was implemented in C++ and built with g++ version 4.8.4 together with the host codes of the benchmark applications. The GPU kernels were compiled with NVCC CUDA version 7.5.

6.3. Synthetic Applications

In order to evaluate scenarios where there is no memory contention between the kernels, we created a set of synthetic kernels. These kernels evaluate the slowdown when different resource requirements are established.

Each synthetic kernel k_i performs a set of arithmetic operations on register values. The number of blocks, number of threads and shared memory requirements are created randomly.

In this experiment, two sets of 50 kernels were created. In the first set, we select k_1 kernels and in the second set, we select k_2 kernels. The kernels in the first set were created to allow concurrency from the beginning (Case A), so their number of blocks satisfies $Blocks_{k1} < (ActiveBlocks_{k1} \times nSM)$ (Algorithm 1). A huge number of experiments were set, but we show here only a random sample of these experiments. Table 2 shows 12 kernels, numbered from 1 to 12, where the odd-numbered kernels were derived from the first set and the even-numbered were derived from the second set.

Table 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

6.3.1. Results

Table 3 shows the results of the estimated and the actual slowdown for a set of pairs from the 12 first kernels. The estimated slowdown is computed by equation (2). The actual slowdown is the ratio between the execution time when the kernel is executed concurrently and the execution time when the kernel is executed alone. The percentage relative error is computed by equation (4).

$$error = \frac{estimated - actual}{actual} \times 100\% \tag{4}$$

Table 3. Estimated vs Actual slowdown (synthetic kernels)

Kernel Pair $k1 - k2$	Estimated	Actual	Perc. Error
S1-S2	11.25	11.312	0.55%
S3-S4	3.00	3.018	0.61%
S5-S6	2.00	1.998	0.12%
S7-S8	4.00	3.937	1.59%
S9-S10	1.30	1.333	2.49%
S11-S12	1.50	1.494	0.39%

We can observe in Table 3 that the combinations of S1-S2 and S7-S8 provided the highest slowdowns. This occurs because, in these cases, k_2 has a great number of blocks, but k_1 left almost no space for their execution. In the K40 GPU, the maximum number of active blocks per SM is 16.

Comparing the estimated and the actual slowdown obtained, we can observe that the percentage relative error is small, at most 2.49%. For the whole experiment, with the 100 kernels, we obtained an average of 3.49% of error and a standard deviation of 6.43%.

6.4. Real-World Applications

In order to evaluate real-world scenarios, we used 8 applications from the Rodinia benchmark suite: 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 4 summarizes these applications resource requirements. We used the most relevant kernel (in terms of percentage of execution time) for each application in the experiments.

Table 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

6.4.1. Results

Among all possible pair combinations of these applications, we present the results of the pairs in which k_1 is the PFL application. PFL allows concurrency from the beginning (Case A), since $Blocks_{PFL} < (ActiveBlocks_{PFL} \times nSM)$.

Table 5 shows the comparison between the estimated and the actual slowdown for the combinations of PFL with all the other applications. We observed an average error of 10.6%. The highest error was produced by the combination PFL-HS3. In this case, our model estimated a smaller slowdown than the real achieved value. HS3 is the application with the highest percentage of usage of the memory bandwidth, around 60%. The experiments with the kernel HS3 suggest that the slowdown was increased by the memory contention. 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.

Table 5. Estimated vs Actual slowdown (Rodinia kernels)

Kernel Pair	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%

7. Conclusions

This work presented an in-depth study on the concurrent kernel execution in the GPU. Modern GPU architectures support concurrent sharing of the GPU resources among multiple kernels, which can unleash the power of the GPU for dynamic and highly virtualized environments. However, the GPU does not have an operating system and the hardware implements a *leftover* policy that assigns as many resources as possible for one kernel and then assigns the remaining resources to another kernel. Under this policy, a resource-hungry kernel can prevent the concurrent execution of other small kernels. Based on this, we studied the necessary conditions for simultaneous execution, and proposed an algorithm that describes when actual concurrency can occur. We also proposed a model for slowdown estimation. Our algorithm and slowdown model do not require simulation instrumentation. They use only the resource requirement data of the kernels.

We validated our slowdown model with synthetic and real-world applications. Our model was able to predict the slowdown in different resource requirements scenarios. For the synthetic applications, that do not present memory interference, our model was able to predict the slowdown with an average of 3.49% of error. For real-world applications, the average error was higher, around 10.6%. The application with the greatest memory bandwidth requirements was the one that provided the greatest error. Our results show that the GPU resources can be shared among the kernels, but the hardware does not provide a fair scheduling policy. In this sense, it is important to identify the kernel characteristics to co-schedule applications with complementary resource requirements. Further studies are necessary to evaluate the impact of memory interference in our model.

For future work, we intend to investigate memory contention in real-world applications and include in our model this study. We also intend to study the effects of distinct GPU architectures in our model.

References

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

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

Ausavarungnirun, R. (2017). *Techniques for Shared Resource Management in Systems with Throughput Processors*. PhD thesis, Carnegie Mellon University.

Breder, B., Charles, E., Cruz, R., Clua, E., Bentes, C., and Drummond, L. (2016). 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*, pages 98–109. Editora da Sociedade Brasileira de Computação (SBC).

Che, S., Sheaffer, J. W., Boyer, M., Szafaryn, L. G., Wang, L., and Skadron, K. (2010). A characterization of the rodinia benchmark suite with comparison to contemporary CMP workloads. In *IEEE International Symposium on Workload Characterization (IISWC), 2010*, pages 1–11.

Goswami, N., Shankar, R., Joshi, M., and Li, T. (2010). Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications. In *IEEE International Symposium on Workload Characterization (IISWC), 2010*, pages 1–10.

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

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

Jeong, M. K., Erez, M., Sudanthi, C., and Paver, N. (2012). A qos-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an mpsoc. In *49th Annual Design Automation Conference*, pages 850–855.

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

Lal, S., Lucas, J., Andersch, M., Alvarez-Mesa, M., Elhossini, A., and Juurlink, B. (2014). GPGPU workload characteristics and performance analysis. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014*, pages 115–124.

Li, T., Narayana, V. K., El-Araby, E., and El-Ghazawi, T. (2011). GPU resource sharing and virtualization on high performance computing systems. In *International Conference on Parallel Processing (ICPP), 2011*, pages 733–742.

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

Pai, S., Thazhuthaveetil, M. J., and Govindarajan, R. (2013). Improving GPGPU concurrency with elastic kernels. In *ACM SIGPLAN Notices*, volume 48, pages 407–418.

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

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

Suzuki, Y., Kato, S., Yamada, H., and Kono, K. (2014). Gpuvm: Why not virtualizing GPUs at the hypervisor? In *USENIX Annual Technical Conference*, pages 109–120.

Tanasic, I., Gelado, I., Cabezas, J., Ramirez, A., Navarro, N., and Valero, M. (2014). Enabling preemptive multiprogramming on GPUs. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 193–204.

Ukidave, Y., Paravecino, F. N., Yu, L., Kalra, C., Momeni, A., Chen, Z., Materise, N., Daley, B., Mistry, P., and Kaeli, D. (2015). Nupar: A benchmark suite for modern GPU architectures. In *6th ACM/SPEC International Conference on Performance Engineering*, pages 253–264.

Wende, F., Cordes, F., and Steinke, T. (2012). On improving the performance of multithreaded CUDA applications with concurrent kernel execution by kernel reordering. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC), 2012*, pages 74–83.