Multilevel Interference-aware Scheduling on Modern GPUs

A Dissertation Presented
by

Leiming Yu

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Engineering

Northeastern University
Boston, Massachusetts

April 2019

Contents

List of Figures

List of Tables

Acknowledgments

I would thank my parents for the gracious and endless support along my journey of study. I would also thank my parents-in-law for their kind considerations. Especially, I want to thank my dear wife, Yang Chen, for her loving care.

My colleagues at NUCAR lab always share their innovative ideas and great suggestions when needed. Big thank you for their support. I would like to thank Dr. Qianqian Fang for supporting me on MCX project which has produced many high-quality publications. Besides, I want to thank Dr. Norman Rubin for his critical and constructive feedback.

This thesis would not be possible without the support and guidance of my research advisor Dr. David Kaeli! I really appreciate his leap of faith to trust in me when I was in a very difficult situation. I am very grateful to be his student.

Abstract

Multilevel Interference-aware Scheduling on Modern GPUs

by

Leiming Yu

Doctor of Philosophy in Electrical and Computer Engineering

Northeastern University, April 2019

Dr. David Kaeli, Advisor

Driven by their impressive parallel processing capabilities, Graphics Processing Units (GPUs) have become the accelerator of choice for high-performance computing. Many data-parallel applications have enjoyed significant speedups after being re-engineered to leverage the thousands of cores on the GPU. For instance, training a complex deep neural network model on a GPU can be done within hours, versus the weeks of time it might take on more traditional CPUs. While most deep neural networks are hungry for more and more computing resources, a number of application kernels only use a fraction of the available resources. To better utilize the massive resources on the GPU, device vendors have started to support Concurrent Kernel Execution (CKE). The Hyper-Q technology from NVIDIA allows up to 32 data-independent kernels to run concurrently, leveraging parallel hardware work queues. These hardware work queues can execute concurrent kernels from either a single GPU context or multiple GPU contexts. With support for concurrent kernel execution, multiple applications can be co-located and co-scheduled on the same GPU, significantly improving resource utilization.

The application throughput provided by CKE is subject to a number of factors, including the kernel configuration attributes, the dynamic behavior of each kernel (e.g., compute-intensive vs. memory-intensive), the kernel launch order and inter-kernel dependencies, etc. Launching more concurrent kernels does not always achieve better performance. It is challenging to predict the potential performance benefits of using CKE. Typically, a developer will have to compile and run their program many times to obtain the best performance. In addition, as multiple GPU applications co-scheduled on the device, the contentions for shared resources, such as memory bandwidth and computational pipelines, result in interference which can often impact the CKE performance.

In this thesis, we seek to optimize the execution efficiency for GPU workloads at a kernel granularity, as well as at an application granularity. We focus on providing a performance tuning

mechanism for concurrent kernel execution and develop an efficient GPU workload scheduler to achieve improved quality-of-service in a cloud environment. We have developed an empirical model named Moka, to estimate the performance benefits using concurrent kernel execution. The model analyzes a non-CKE application comprising multiple kernels, using the profiling information. It delivers an estimate of the performance ceiling by taking into account data transfers and GPU kernel execution behavior. Moka also provides guidance to find the best performing kernel-stream mapping, quickly identifying the best CKE configuration, resulting in improved performance and the highest utilization of the GPU. In addition, a machine-learning based interference-aware scheduler named Magic was developed to improve the system throughput for multitasking on GPUs. Magic framework implements offline short profiling analysis to study the important interference metrics and conducts interference sensitivity prediction for GPU workloads based on the selected machine learning models. Our scheduler outperforms a state-of-art similarity-based scheduler on a single GPU system and achieves a high system throughput compared to the least-loaded policy on a multi-GPU system.

Chapter 1

Introduction

Over the past decade, we have seen major changes in direction when designing next-generation computing hardware, moving from fast and complex uniprocessor designs, to simpler, yet explicitly parallel, multiprocessor/multicore designs. Central Processing Units (CPUs) have been the traditional workhorse for a range of platforms ranging from mobile devices to cloud servers. CPU vendors have elected to incorporate multiple computing cores in a single chip (e.g., up to 18 cores on the Intel Core i9 CPU, and up to 32 cores on the AMD Threadripper CPU).

While industry has been successful in delivering multi-core CPU designs, improving the throughput of CPU workloads requires rethinking the implementation of our applications. At a programming level, these changes involve parallelizing algorithms, redesigning data structures, and exploiting multi-threading techniques and Single Instruction Multiple Data (SIMD) instructions present on the hardware. Due to power constraints and resource contention present in the computation units and memory system on current multi-core architecture, the actual performance gains of multi-threaded CPU-based applications cannot fully meet the computational demands given the rapidly growing needs of data processing. Motivated by these issues, industry has explored the introduction of accelerators that can accelerate data-parallel computations.

Graphics Processing Units (GPUs) have become of the accelerator of choice on many systems. GPUs provide massively parallel computing cores and high memory bandwidth. With tens of thousands of parallel processing threads executing in a Single-Instruction-Multiple-Thread (SIMT) manner, modern GPUs can sustain much higher compute throughput than a CPU. By exploiting the inherent parallelism in applications, including linear algebra, molecular dynamics and game physics, GPUs can easily enjoy double-digit speedups over an optimized CPU design [79, 112, 113, 109, 111].

GPUs were originally designed for rendering 3-D computer graphics, where millions of pixels need to be processed concurrently. To take advantage of the parallel processing power

for non-graphics applications, programmers need to understand complex graphics programming interfaces (e.g., DirectX [33] and OpenGL [88]), and streaming languages (e.g., BrookGPU [14] and sH [60]). GPU programming has been more user-friendly since the release of Compute Unified Device Architecture (CUDA) [65] in 2007 and Open Computing Language (OpenCL) [35] in 2008. The CUDA framework is proprietary to NVIDIA GPUs, whereas OpenCL is an open source standard for parallel computing supported by a variety of hardware targets including CPUs, GPUs, field-programmable gate arrays (FPGAs), and digital signal processors (DSPs). While OpenCL is designed for scalability and portability, CUDA is targeted for performance.

Given the rapid pace of architectural evolution in the GPU market, and supported by a variety of customized libraries, GPUs have been able to achieve impressive speedups for a range of applications. NVIDIA's CUDA offers more custom library support than AMD does for OpenCL, mainly because CUDA is proprietary, while OpenCL is open. The range of supported programming features for each framework is presented in Table 1.1. Since the majority of this thesis will focus on acceleration on NVIDIA GPUs, we will use CUDA terminology throughout this thesis.

Table 1.1: Advanced Features for CUDA and OpenCL

Features	**CUDA**	**OpenCL**
Unified Memory	Yes(6.0+)	Yes(2.0+)
Dynamic Parallelism	Yes(5.0+)	Yes(2.0+)
C++	Yes(6.0+)	Yes(2.1+)
Stream Priority	Yes(5.5+)	Yes(2.1+)
Pipes	No	Yes(2.0+)
Thread Data Sharing	Yes(5.0+)	No
Mixed-Precision	Yes(7.5+)	Yes(1.0+)
Concurrent Kernel Execution	Yes(4.0+)	Yes

Leveraging the massive parallel processing power of the GPU, and equipped with a number of programming features that can exploit the parallelism provided by a GPU, we have seen a growing number of serial applications being ported to a GPU for acceleration. The performance of a GPU-based application heavily depends on the ability of the programmer to exploit concurrent execution on the device, fully exploiting the available computing resources. GPU vendors continue to increase the number of computational resources on each new hardware generation to meet the application demands.

As shown in Table 1.2, the number of available computing cores on today's GPUs has surged by 24 times, from 216 CUDA cores on early Tesla-based GPUs, to 5120 CUDA cores of the latest Volta-based GPUs. As a result, the number of transistors has increased from 1.4 to 21.1 billion (by a factor of 15), and the single precision throughput has been increased from 477 GFLOPs

to 14899 GFLOPs, improved by a factor of 31. With so many computational resources available, we need to enhance the effective orchestration of these parallel resources in order to maximize performance gains from GPU acceleration.

The need to efficiently exploit parallel execution on a GPU motivates us to revisit the different levels of concurrency present in GPU workloads. In this thesis, we evaluate how to best to achieve multi-level concurrency and provide novel approaches to increase execution efficiency.

Table 1.2: Device resources and computing capability for evolving architectures of NVIDIA GPUs.

GPU	GTX 260	GTX 480	GTX 680	GTX 980	GTX 1080	TITAN V
Architecture	Tesla	Fermi	Kepler	Maxwell	Pascal	Volta
Computing Cores	216	480	1536	2048	2560	5120
Transistors	1400 million	3000 million	3540 million	5.2 billion	7.2 billion	21.1 billion
FP32 Performance (GFLOPS)	477	1345	3090	5132	9216	14899
Release Year	2008	2010	2012	2014	2016	2017

1.1 GPU Computing

GPUs and CPUs present two different design points in terms of architectural philosophy. CPUs are designed to execute a small number of complex tasks, whereas GPUs are designed to execute a larger number of simpler tasks. CPUs have a small number of registers per core for a given task. Context switching between tasks on a CPU is relatively expensive since the live registers need to be stored and restored, accessing main memory. By comparison, GPUs have multiple banks of registers to support lightweight context switching, simply using a bank selector to switch between different registers sets and contexts, avoiding costly accesses to memory. CPUs use context switching to hide long stalls which are often caused by I/O and long latency operations, while GPUs work at a finer granularity and switch among different instruction streams to hide latencies [19].

In order to deploy a GPU for acceleration, the data initialized on the CPU (referred to as the host) needs to be copied to the GPU (referred to as the device). Data transfer between the host and device are often realized via the interconnect of a PCI-Express bus or an InfiniBand network. A typical GPU usually contains one or two DMA (Direct Memory Access) copy engines for data copy. For GPUs with one copy engine, only one-way transfer, either host-to-device or device-to-host, is allowed through the interconnect. For GPUs with two copy engines, concurrent bi-directional transfers are supported.

Read/write data that is transferred to the GPU is typically stored in global memory. For read-only objects, data can be stored in either constant memory or texture memory, supporting faster data access. These three different memory modules are all interconnected and shared with the

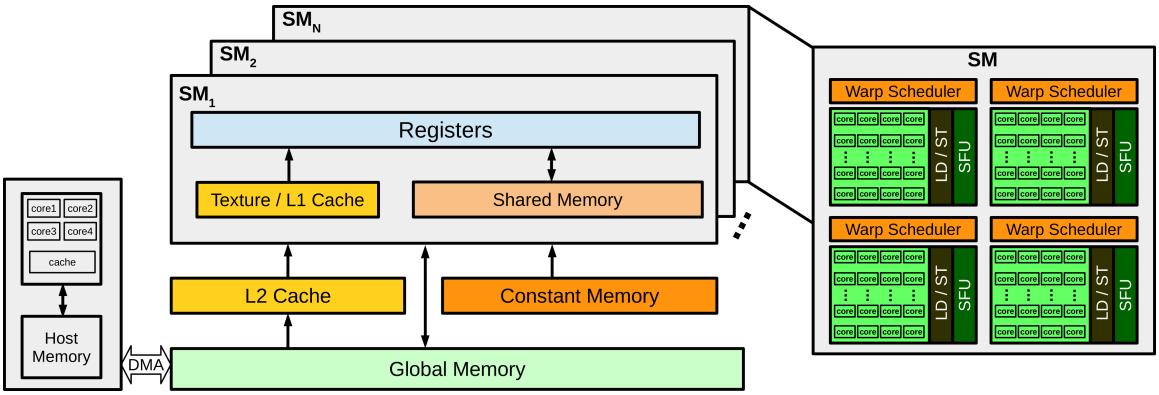


Figure 1.1: A typical GPU computing flow.

streaming multiprocessors (SMs). A single streaming multiprocessor on a GPU provides hundreds of CUDA cores that are designed to execute computations with different precisions (i.e., integer/single-precision/double-precision/half-precision). Each SM also contains special function units (SFUs) for transcendental functions (e.g., SIN, COS and natural logarithms). To efficiently utilize these computing units, each SM has a set of local registers for fast data access and storage. In addition, each SM has shared memory to cache the data from the global memory, reducing global memory traffic and increasing the effective memory bandwidth.

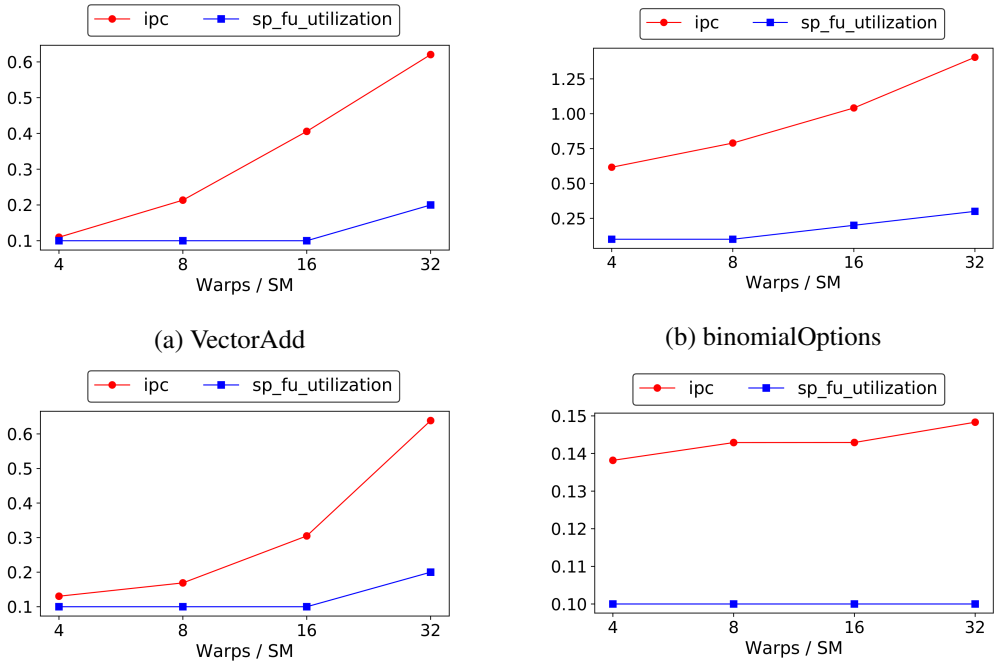
As hundreds of thousands of threads are launched on the GPU, they are grouped into warps (on an NVIDIA GPU, a group of 32 threads execute in a Single Instruction Multiple Data (SIMD) fashion) and dispatched by the warp scheduler on each streaming multiprocessor. Given multiple warp schedulers, warps executing different instruction streams can run in parallel to increase instruction-level parallelism. When a warp is stalled due to a long latency memory operation, the scheduler can quickly switch to another runnable warp in order to hide the latency. Equipped with fast context switching, the execution efficiency of GPU threads can be boosted.

GPUs are the ideal accelerator for data-parallel applications since the inherent independent executions can be mapped to many parallel threads for concurrent execution. To increase the utilization of the computing resources on the device, we need to consider how best to manage accesses to the GPU memory hierarchy. By exploring data parallelism and carefully managing data movement, GPU-accelerated applications often observe a significant speedup over the optimized CPU implementations.

1.2 Concurrent Kernel Execution

With the help of fabrication technology, modern GPUs accommodate more computing resources (see Table 1.2) with each new generation for high-performance computing. As applications with diverse source requirements have been ported to GPU for acceleration, resource underutilization is often observed for a single kernel.

Figure 1.2 shows the instructions per second (*IPC*) and single precision function unit utilization (*sp_fu_utilization*) for four distinctive GPU kernels. As we increase the number of warps (a warp is defined as a group of 32 threads on NVIDIA GPUs) per streaming multiprocessor, the *IPC* consistently increases for all four applications. For single precision function units utilize, we see an increase from 10% to 30%, leaving most of its resources under-utilized. In lavaMD, double precision computation is dominant, hence the single precision function unit utilization is consistently low. Since the ratio of double precision units to single precision units is 1 to 32 on the NVIDIA GTX 950 GPU, as more warps are dispatched to the streaming multiprocessor, the *IPC* performance is constrained by the double precision resources, which leads to a marginal increase in performance.



(a) VectorAdd

(b) binomialOptions

(c) b+tree

(d) lavaMD

Figure 1.2: Executing more warps of threads improves the throughput (i.e., instructions per second), whereas device resources (i.e., single precision function units) are still underutilization.

Here, we explore the impact of launching more GPU threads, which should improve the

performance, but may not result in high utilization of GPU resources. Memory-intensive kernels (e.g., VectorAdd and b+tree) often have high utilization of global memory bandwidth and a high cache hit rate, while the arithmetic function units remain underutilized, with less than 20% utilization of the single-precision function units. Even for compute-intensive kernels (e.g., binomialOptions), the arithmetic function units remain untapped.

To address the issue of resource under-utilization for a single kernel, modern GPUs support concurrent kernel execution (CKE), a feature that allows multiple kernels to run concurrently on the same device (see Table 1.1). Depending on the available compute resources (e.g., registers and shared memory), multiple kernels can be collocated on the device for concurrent execution. On NVIDIA Fermi GPUs, up to 16 concurrent kernels are supported [86], whereas up to 32 concurrent kernels are supported for NVIDIA Kepler GPUs [68]. The benefits of running CKE are two-fold:

1. we can maximize the total number of actively running threads from different kernels, which can improve resource utilization,

2. we can balance resource utilization for two distinctive kernels which are compute-intensive and memory-intensive, by running them concurrently.

Due to the inherent nature of GPU kernels, their associated resource demands can vary greatly, including the number of required processing threads. If a kernel has a small number of threads, occupying a fraction of the available hardware threads on the GPU, a second kernel can be dispatched to run concurrently and improve the occupancy on the GPU. The occupancy metric represents the ratio of active running threads to the maximum available hardware threads available on the device. High occupancy means a high degree of thread utilization.

When multiple kernels are collocated on the GPU, they can request different amounts of device resources for each of their computations. Some kernels stress the memory system (i.e., memory-intensive kernels), while others require more computation than memory (i.e., compute-intensive kernels). Running similar kernels (similar in terms of resource usage) concurrently often leads to resource competition and significant performance degradation. For instance, a memory-intensive kernel will spend more time waiting for data than computing the results. Running two memory-intensive kernels concurrently may lead to competition in the data cache. Since the data cache space and bandwidth are limited on the GPU, cache performance could be much lower than when running them separately. The latency of data accesses cannot be easily hidden due to memory delays and the fact that the next thread will also want to request memory. Therefore, to properly leverage CKE, the demands of memory bandwidth and compute resources need to be considered carefully before assigning kernels to run concurrently.

1.3 Challenges with Concurrency

1.3.1 Kernel-level concurrency

As concurrent kernel execution is supported on modern GPUs, different tasks can be implemented as kernels for parallel execution. Leveraging CKE, task-level parallelism can be exploited to maximize application throughput. However, launching more concurrent kernels does not always achieve better performance. Figure 1.3 shows the CKE performance when running the expectation step in the Hidden Markov Model (HMM) kernel. We run seven different test cases, where the number of hidden states is increased from 64 to 4096. We can observe a 2x speedup when running two concurrent kernels. The overall performance plateaus after attempting to run three concurrent kernels. Using more than four concurrent kernels can even decrease the performance in some cases.

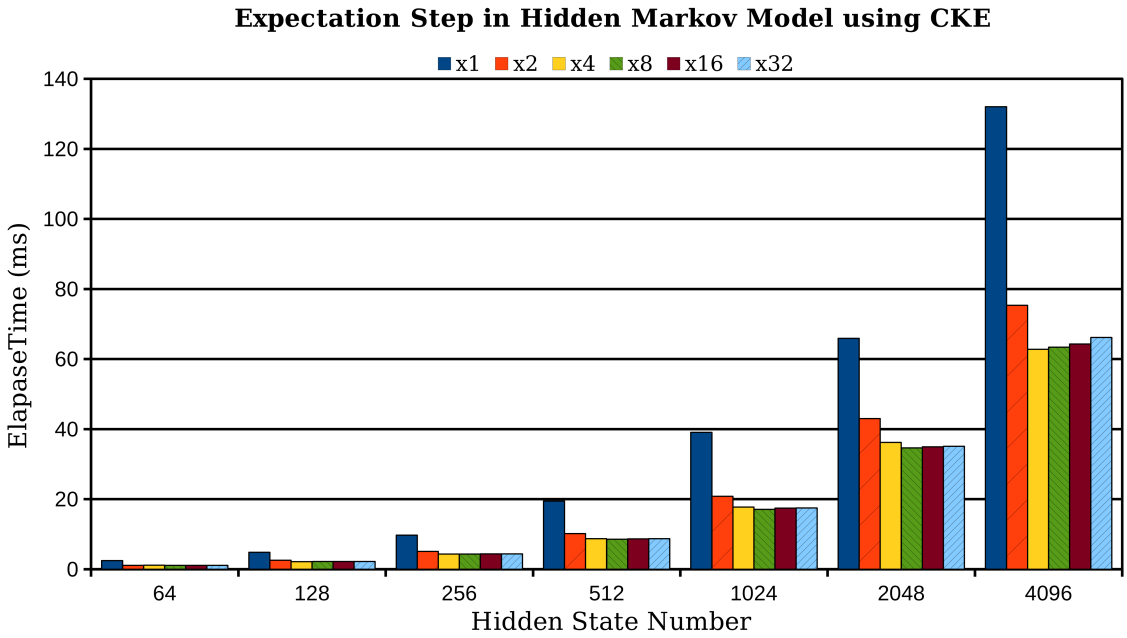


Figure 1.3: The performance of the expectation step in the Hidden Markov Model kernel after applying concurrent kernel execution, where we illustrate seven cases for different number of hidden states and vary the number of concurrent kernels. The analysis uses an NVIDIA GTX TITAN GPU.

Launching more kernels does not always translate to additional concurrency during execution. The amount of concurrency for kernel execution is determined by the availability of device resources. These resources often include registers, shared memory and the maximum number of

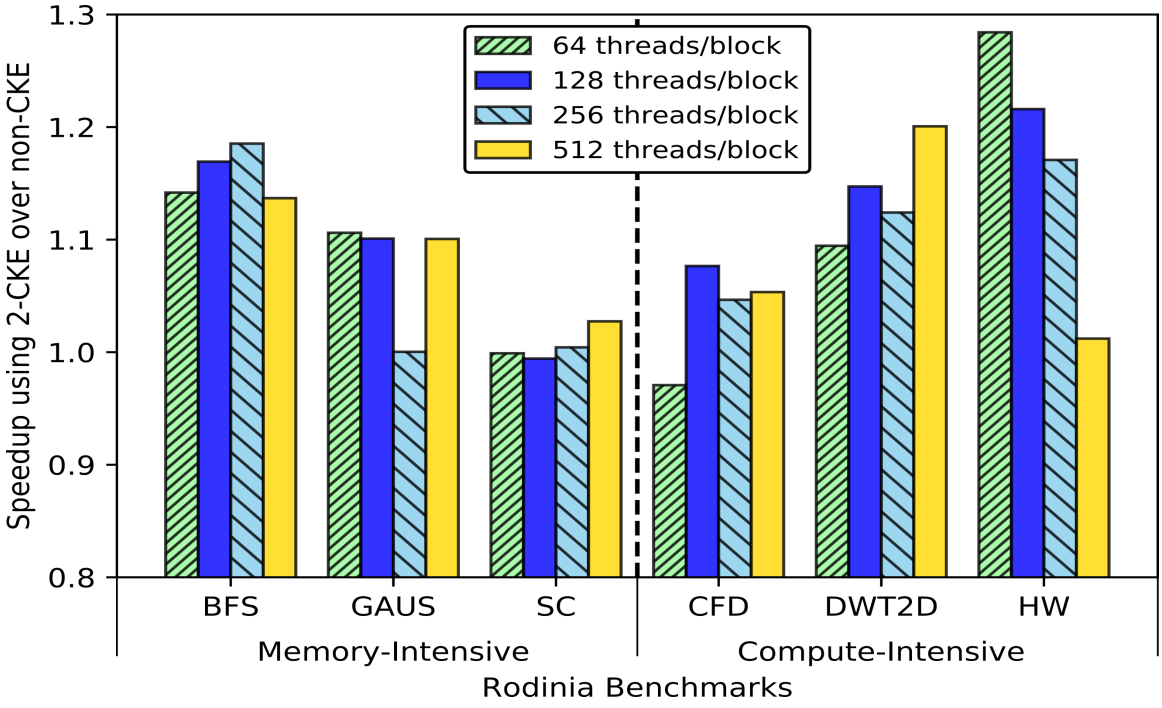


Figure 1.4: Speedup of Rodinia benchmarks using 2 concurrent kernels, versus a non-concurrent implementation, on a NVIDIA GTX 950 GPU. Four different block sizes are considered.

active threads supported on the streaming multiprocessors (SMs). For instance, assume we have a kernel running that has already exhausted device resources. A queued kernel cannot be scheduled until the active kernel releases some of its resources. The CKE performance for the HMM kernel is shown in Figure 1.3. The limiting factor in this kernel is the availability of the shared memory.

In addition to the occupancy metric, the configuration of a kernel configuration can have a major impact on concurrent kernel execution. Figure 1.4 shows the speedup achieved by using CKE for 6 applications from the Rodinia benchmark suite [15], as run on an NVIDIA GTX 950. Speedup is measured versus a non-CKE baseline. Applications are configured using four different block sizes. As observed in Figure 1.4, the applications exhibit variations in speedup due to changes in block size when processing the same dataset. In some cases, such as CFD with a block size 64, CKE can actually reduce performance. Figure 1.4 highlights the complex relationship between computational characteristics and the grid properties (block size), as they both impact application performance when using CKE. Hence, it is challenging to determine the best configuration of using concurrent kernel execution to maximize performance. Typically, a developer will have to compile and run their

program many times to obtain the best parameters. We attempt to tackle this problem by developing a model-based concurrent kernel analysis in this thesis.

1.3.2 Application-level concurrency

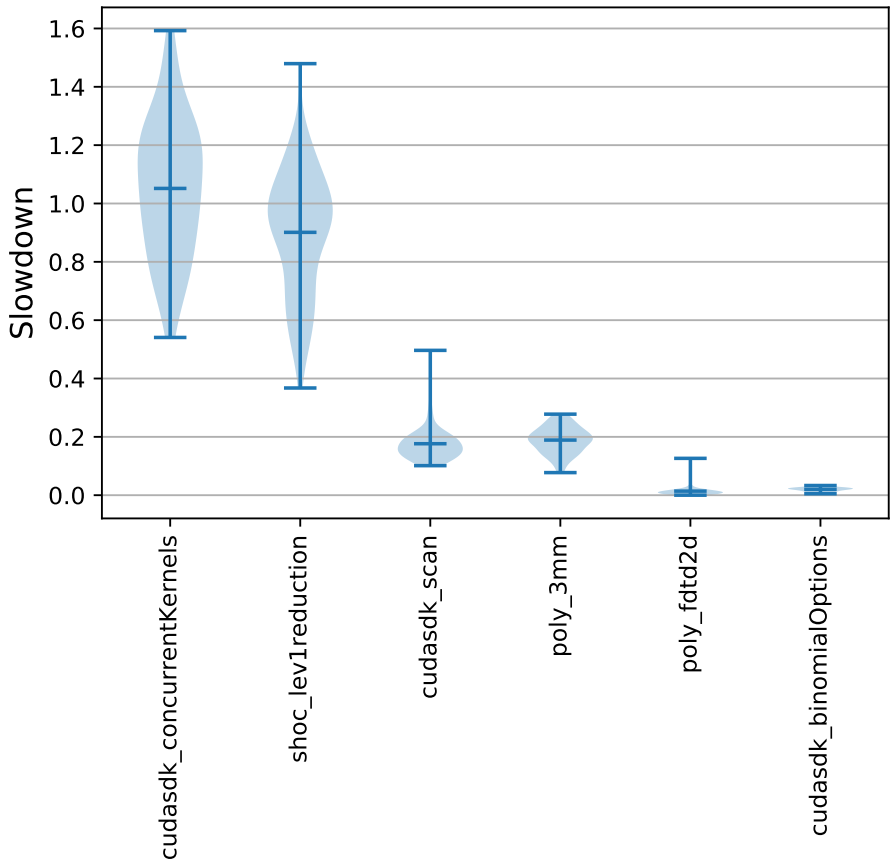


Figure 1.5: The slowdown profile (versus non-concurrent execution) observed for six different GPU applications when run concurrently with a second application (78 total applications) on the same NVIDIA GTX 1080 Ti GPU. The distribution (violin shape) is estimated using a gaussian kernel density. Each violin plot shows the worst-case slowdown (top bar), the best-case slowdown (bottom bar) and the average slowdown (middle bar).

With hardware support for concurrent kernel execution, kernels from both a single application, as well as multiple applications, can run concurrently on the same GPU. Thus, if we leverage multiple hardware work queues, multiple GPU applications can be dispatched to the GPU concurrently to fully utilize the compute and memory bandwidth capacity. As cloud providers (e.g., Amazon EC2 [8] and Microsoft Azure [63]) offer GPU instances for accelerated computing, schedul-

ing GPU workloads efficiently to maximize the system throughput and meet the quality-of-service (QoS) demands of these environment becomes paramount.

Due to the complex interaction and different resource requirements across concurrent GPU applications, supporting QoS requirements on a GPU-based cloud server environment presents new challenges. The benefits of running multiple applications concurrently can vary significantly from one workload to the next. In Figure 1.5, we show the slowdown for 6 selected GPU applications when co-executed with 78 other GPU applications - here we consider only that at most two application run concurrently, though more can be considered. The reported slowdown is the ratio of co-running runtime to the dedicated runtime for the target application. We have observed that, when simultaneously collocated with another application, the slowdown for the concurrent kernel benchmark from the CUDA SDK (cuda_concurrKerns) ranges from 0.5x to 1.6x, while the slowdown for binomialOptions from the CUDA SDK (cuda_binomOpts) ranges from 0.01x to 0.04x.

Both cuda_concurrKerns and shoc_reduction are more sensitive to interference than the other 4 applications, whereas the cuda_binomOpts is the most robust to interference. We can clearly see that each GPU application experiences a different degree of slowdown when running concurrently with the other GPU applications. Therefore, it is key to determine the potential for interference in advance of the actual concurrent execution if we do not want to sacrifice performance.

An efficient GPU workload scheduler should be aware of the characteristics of each workload that has been scheduled to run and select the best candidate for co-execution with the goal of avoiding interference, achieving the best co-executed throughput. In previous work on the *Mystic* framework[97], GPU workloads are scheduled based on their differences with one another, where the least similar applications are dispatched for co-execution. The similarity distance is measured by using the predicted performance metrics from a short profiling run of a GPU application. The metrics profiled are selected based on expert knowledge. We are motivated to develop an autonomous metric learning process that can capture the characteristics of a GPU application, and improve the quality of scheduling decisions driven by the similarity approach.

1.4 Contributions of this Thesis

The goal of this thesis is to optimize the execution efficiency for GPU workloads at a kernel granularity, as well as at an application granularity. We focus on providing a performance tuning mechanism for concurrent kernel execution and develop an efficient GPU workload scheduler to achieve improved quality-of-service in a cloud environment. The contributions of this thesis are summarized below:

- We have developed a kernel-matching algorithm to predict the best block size for GPU kernels.
- We have built an empirical model named *Moka* to predict the performance of concurrent kernel execution on a GPU.
- We have validated *Moka* using real-world GPU applications and achieved a maximum of 12% prediction error as compared to the actual runtime performance when using concurrent kernel execution.
- We have developed an autonomic feature selection method to identify the prominent metrics for scheduling GPU workloads based on resource usage patterns and improved QoS by 25%, as compared to using previously proposed interference metrics [97].
- We have developed *Magic*, a machine-learning based interference-aware scheduler for GPU workloads, to handle single-GPU and multi-GPU systems.
- We have demonstrated that *Magic* can improve the overall system throughput on a single-GPU system by 16% and 10% as compared to a first-come-first-serve policy and a state-of-art similarity-based policy, respectively. Compared to a least-loaded policy and a round-robin policy on a multi-GPU system, our scheduler outperforms them by 21% and 22%, respectively.

1.5 Organization of Thesis

The rest of this thesis proposal is organized as follows. Chapter 2 presents background information on general purpose computing on GPUs, concurrent kernel execution, and interference-ware scheduling for GPU workloads. Chapter 3 presents prior work that addresses concurrency issues at the kernel and application levels. Chapter 4 describes the proposed framework for model-based concurrent analysis and evaluates the performance of real-world GPU workloads. Chapter 5 presents an interference-aware workload scheduler for GPU clusters. Finally, we conclude by summarizing the major contributions of this thesis and present future work in Chapter 6.

Chapter 2

Background

2.1 General Purpose Computing on GPU

GPUs have become the device of choice for accelerated computing due to their high thread-level performance (floating-point operations per second) and attractive power efficiency (performance per watt). Compared to multi-core CPUs, many-core GPUs have much higher memory bandwidth and single-precision throughput. As shown in Table 2.1, GPUs offer 4x-10x more memory bandwidth than CPUs for server computing. We compare the Xeon series from Intel and the Tesla series from NVIDIA. For consumer-level devices, as shown in Table 2.2, the single-precision (FP32) throughput achieved by the NVIDIA GPUs is, on average, 30x higher than a contemporary CPU. Given their massive parallel computing capabilities, GPUs have been adopted across a wide range of compute-intensive application areas, including machine learning, computer vision, image processing, and physics simulation [79, 64, 113, 109, 111].

To take advantage of the massive computing power provided by the GPU, developers need to understand the underlying architecture and consider the best approach to fully utilize the available resources. In the following sections, we will review the NVIDIA GPU architecture, including GPU programming features and discuss GPU best practices.

Table 2.1: Comparison of the memory bandwidth available on various CPUs and GPUs, targeting server-class devices.

Year	2011	2012	2013	2014	2015	2016	2017
Intel CPU							
Device	Xeon X5690	Xeon E5-2690	Xeon E5-2697 V2	Xeon E5-2699 V3	Xeon E5-4699 V3	Xeon E5-2699 V4	Xeon Gold 6154
Code Name	Westmere	Sandy bridge	Ivy bridge	Haswell	Haswell	Broadwell	Skylake
Bandwidth	32 GB/s	51 GB/s	60 GB/s	68 GB/s	68 GB/s	77 GB/s	119 GB/s
NVIDIA GPU							
Device	Tesla C2050	Tesla K20	Tesla K40	Tesla K80	Tesla M40	Tesla P100	Tesla V100
Code Name	Fermi	Kepler	Kepler	Kepler	Maxwell	Pascal	Volta
Bandwidth	144 GB/s	208 GB/s	288 GB/s	2 x 240 GB/s	288 GB/s	732 GB/s	900 GB/s

Table 2.2: Comparison of single-precision floating-point throughput on various CPUs and GPUs, targeting consumer devices.

Year	2012	2013	2014	2015	2016	2017
Intel CPU						
Device	Intel Core i5 3570	Intel Core i7 4770K	Intel Core i7 4790K	Intel Core i7 6700K	Intel Core i5-7200U	Intel Core i7 8700K
Code Name	Ivy Bridge	Haswell	Haswell	Skylake	Kaby Lake	Coffee Lake
FP32 Throughput	105 GFLOPs	182 GFLOPs	234 GFLOPs	200 GFLOPs	128 GFLOPs	361 GFLOPs
NVIDIA GPU						
Device	GTX 680	GTX TITAN	GTX 980	GTX TITAN X	GTX 1080	TITAN V
Codename	Kepler	Kepler	Maxwell	Maxwell	Pascal	Volta
FP32 Throughput	3090 GFLOPs	4500 GFLOPs	4612 GFLOPs	6144 GFLOPs	8228 GFLOPs	12288 GLOPs

2.1.1 GPU Evolution

GPUs were originally designed to render 3-D graphics for displays and gaming. Prior to 2007, even though GPUs contained hundreds of parallel processing units, programmers could not easily tap their computational resources without using graphics Application Programming Interfaces (APIs) (e.g.,OpenGL [88] and DirectX [33]). With the introduction of programmable hardware shaders, and the the development of GPU programming languages such as CUDA C [65] and OpenCL [91], leveraging a GPU to accelerate execution becomes much easier. Since then, general purpose computing on GPU has gained significant momentum. Today, GPUs are the accelerator of choice, with more and more GPU-accelerated applications enjoying significant speedups.

To boost the performance of applications, GPU vendors have introduced new architectures to meet both the growing computational and energy-efficiency demands. The evolution of architectural design for NVIDIA GPUs is presented in Table 2.3.

Table 2.3: The architectural parameters for 5 generations of NVIDIA GPUs.

Architecture	Fermi	Kepler	Maxwell	Pascal	Volta
Release Year	2009	2012	2014	2016	2017
CUDA Cores (per SM)	32	192	128	128	64
Register File Size (per SM)	128 KB	256 KB	256 KB	256 KB	256 KB
Warp Schedulers (per SM)	2	4	4	4	4
Dispatch Units (per SM)	2	8	8	8	4
Load / Store Units (per SM)	16	32	32	32	32
Special Function Units (per SM)	4	32	32	32	4

For the Fermi architecture, each streaming multiprocessor (SM) has 32 CUDA cores and 2 warp schedulers, where each warp scheduler has a dispatch unit. The dual-warp scheduler allows scheduling and dispatching instructions from two independent warps (a group of 32 threads executing in SIMD lockstep) at the same time. Shared memory and L1 cache are unified and configurable, with a total size of 64 KB [27]. For the Kepler architecture, each SM has 192 CUDA cores and 4

warp schedulers. To increase the utilization of compute resources, each warp scheduler is capable of scheduling two instructions per warp every clock cycle. Kepler has twice the number of Load/Store (LD/ST) Units as compared to Fermi, and 8x more Special Function Units (SFUs) [67]. Based on Kepler, the Maxwell architecture has a new design scheme to improve power efficiency. Rather than having 192 CUDA cores per SM (a non-power-of-two organization) as in the Kepler architecture, each streaming multiprocessor of Maxwell is organized with 128 CUDA cores divided into four distinctive 32-CUDA core blocks, where each block has its own dedicated warp scheduler and dispatch unit. In the Maxwell SM design, aligning the physical CUDA core block with the warp size saves power by managing computation in a simplified datapath. Different from the Fermi and Kepler architecture, a Maxwell SM features 96KB of dedicated shared memory, instead of 64KB combined shared memory/L1 cache. On Maxwell GPUs, the L1 cache has been integrated with the texture cache. As a result, each CUDA core of the Maxwell can offer 1.4x more performance over a Kepler CUDA core, while delivering 2x the performance per watt [69]. Starting with Pascal, the Fin Field Effect Transistor (FinFET) technology was adopted to incorporate more transistors on chip, providing higher performance and power efficiency than previous generations. The microarchitecture layout of a Pascal SM continues to follow the design of Maxwell SM [73]. For Volta-based GPUs, each SM integrates 64 CUDA cores, with 8 additional tensor cores to accelerate deep learning workloads. New tensor cores can achieve up to 12x and 6x higher peak TFLOPS for the training and interference steps, as compared to GP-100 device. The Volta CUDA core separate the data path for integer and floating-point operations, delivering much improved execution efficiency for workloads with a mix of computations. The Volta SM also features a combined capacity of 128 KB for the L1 data cache and share memory, which significantly improves memory access performance. It is worth mentioning that a new L0 instruction cache is used in each CUDA core partition inside the SM, in order to increase the efficiency of the instruction buffer [75].

2.1.2 GPU Memory Hierarchy

A GPU is a massively parallel computing device with a complex memory hierarchy. Data needs to traverse through the memory system before being processed by thousands of parallel computing cores. Understanding the structure of the GPU memory system and its corresponding properties (e.g., whether it is readable or writable and what is the latency for read/write) is key, in order to map the input data to the right level of the hierarchy. If we can achieve the best mapping, memory bandwidth utilization can be significantly improved and the arithmetic pipelines can be kept busy with computation, boosting application throughput. The GPU memory model is illustrated in Figure 2.1.

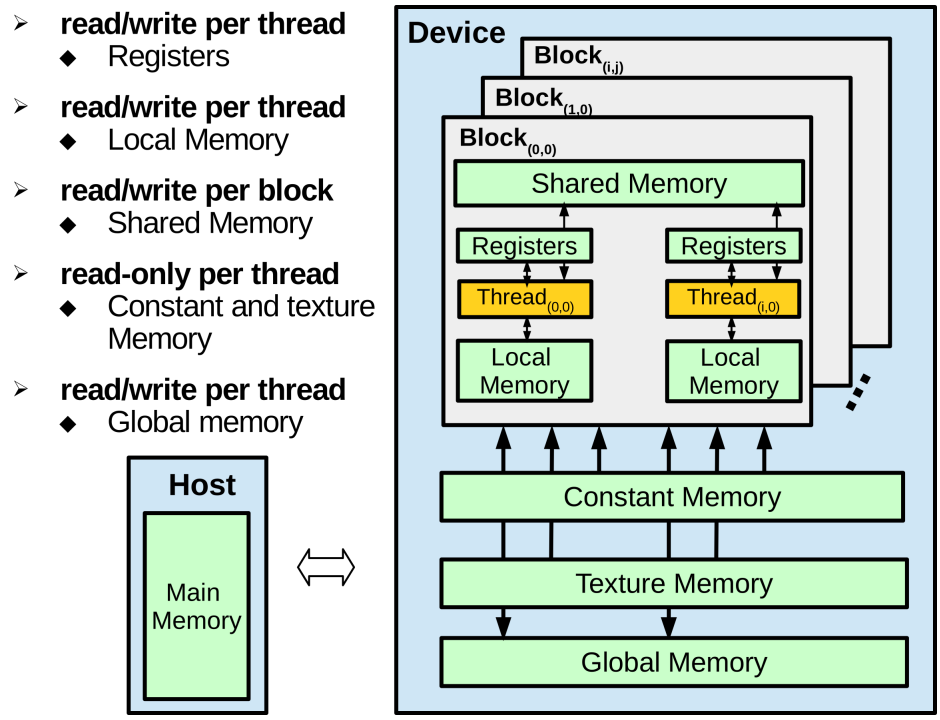


Figure 2.1: GPU memory model.

In GPU memory systems, registers provide the fastest access in the memory hierarchy. These on-chip registers are private to each GPU thread. Since the total number of registers per streaming multiprocessor is limited, given high register usage per thread, the total threads that can be launched can be significantly reduced. This limit can impact the achievable parallelism on the device, when there are not enough threads available to hide the data transfer latency. The register usage is determined by both the programmer and the compiler for NVIDIA GPUs. Adding an additional compiler option, *-Xptxas -v*, to the NVIDIA GPU compiler *nvcc* can report the register usage information for each GPU kernel.

When too many registers are consumed such that variables that can not fit in the available registers, data will be *spilled* to local memory, significantly impacting memory performance. Local memory is an abstraction of global memory, not a physical type of memory. Since local memory is off-chip, it is very expensive to access compared to the registers.

Besides the register file, shared memory is another on-chip memory which can be used for data caching and communication. Data stored in shared memory is visible to all threads within a thread block, in contrast to registers that are private to each thread. The data communication within a thread block can be realized through shared memory. To facilitate simultaneous data access for

parallel execution threads, shared memory is divided into 32 banks. Whenever multiple addresses are mapped to the same bank, memory contention can occur, resulting in contention in a memory bank that serializes memory requests. Bank conflicts can decrease the effective bandwidth by a factor equivalent to the number of colliding memory requests. An exception to this case is when all threads in a warp (a group of 32 threads) access the same shared memory address, which can be broadcast without a performance penalty. Threads in a thread block can use shared memory as a programmer-managed cache to reduce global memory traffic. The use of shared memory is commonly used to optimize high-performance cooperative parallel algorithms, such as parallel scan [39] and histogram [30].

Constant memory is an off-chip memory for storing data that does not change during the course of kernel execution. There is 64KB of constant memory on NVIDIA GPUs. Constant memory is cached and can broadcast a value when all the warp threads access the same address. Thanks to its low latency and high bandwidth, constant memory can reduce the required memory bandwidth significantly.

Similar to constant memory, texture memory is also read-only and off-chip. Texture memory was originally designed for graphics rendering pipelines. It is extremely useful when the memory access pattern shows lots of spatial locality. Though texture memory can be addressed for a 1-D, 2-D, or 3-D array, it works best when data is stored in 2-D in terms of data locality. On NVIDIA GPUs, texture memory is cached in a per-SM fashion, where there are multiple texure units for each streaming multiprocessor. Using the texture cache can reduce the memory traffic, alleviate coalescing constraints and improve the performance, as compared to working out of global memory.

Global memory is the largest memory on the device, with the highest latency for data reads and writes. Global memory is often an order of magnitude slower than shared memory and the register file. Because of this, accesses to global memory should be minimized, especially for I/O bound kernels.

Coalescing memory accesses is an optimization technique for efficient data access that can combine multiple memory requests into a single transaction. Leveraging this technique, the total number of global memory read/store operations can be reduced, and DRAM bandwidth can be beter managed.

Table 2.4 shows the properties of each aforementioned memory space. In the GPU memory hierarchy, registers are the fastest. For kernels using a lot of registers, shared memory can be utilized to stage the data in order to reduce register pressure. Register spilling should always be avoided due to the slow performance of the local memory. Both constant memory and texture memory are read-only. While constant memory is useful when all threads access the same address, texture

Table 2.4: GPU memory properties.

Memory	Scope	LifeTime	Read/Write	Speed
Register	Thread	Kernel	Both	Fastest
Local Memory	Thread	Kernel	Both	Slow
Shared Memory	Block	Kernel	Both	Fast
Constant Memory	Grid	Application	Read-only	Fast
Texture Memory	Grid	Application	Read-only	Fast
Global Memory	Grid	Application	Both	Slow

memory is great for exploring the spatial locality of the input data. Coalescing memory accesses is critical to improve global memory performance since it can significantly improve the effective bandwidth [87, 42].

2.2 Concurrent Kernel Execution

With the rapid growth of computing resources on the GPU, a single GPU kernel often cannot leverage all the available resources. GPU vendors support concurrent kernel execution (CKE) to improve resource utilization on modern GPUs. In this section, we present the architectural support enabling CKE, illustrate the benefits of using CKE, and show the factors impacting the CKE performance.

2.2.1 Architecture Support for CKE

Concurrent kernel execution (CKE) was first supported on NVIDIA Fermi GPUs [27]. It allows additional kernels to run on the device when a single kernel does not fully utilize the available resources. Up to 16 concurrent kernels are supported on the Fermi architecture. CKE exposes opportunities for small kernels to run concurrently in order to maximize resource utilization and improve the overall throughput of a GPU application. For a large kernel, CKE can also improve the throughput by mapping a single large kernel into smaller kernels, overlapping data transfers with kernel execution. As shown in Figure 2.2, data transfers and kernel computation are divided into two parts, where each part is implemented as a stream for concurrent kernel execution. The overlapped execution between kernel execution and data transfer can been observed. Eventually, leveraging 2 CKEs, the previous elapsed time is reduced by 30%, achieving a 1.3x performance improvement.

At the programming level, concurrent kernels are inserted into the software work queues that are associated with CUDA streams. At a hardware level, Fermi-based GPUs have only a single work queue for kernel execution. Unnecessary serialization (i.e., false serialization) can be triggered when multiple kernels from different streams are dispatched for concurrent execution. Figure 2.3

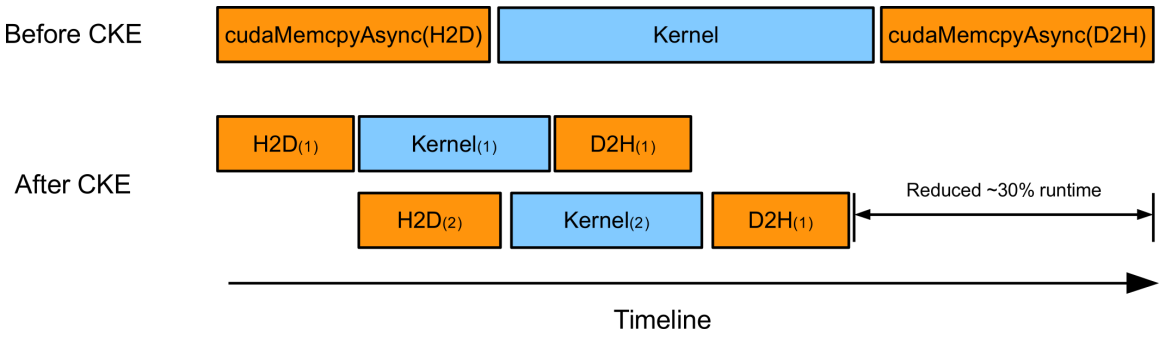


Figure 2.2: Overlap kernel computation with data transfer using CUDA streams for two concurrent kernel execution. The scheme halves the data transfer and kernel computation. H2D stands for host-to-device data transfer, D2H stands for device-to-host data transfer.

illustrates this class of serialization across kernels from two independent CUDA streams for GPUs with one hardware work queue. Given enough resources, kernels from stream 0 and stream 1 should be able to run concurrently. However, when stream 0 is scheduled before stream 1, kernel X (KernX) from stream 1 is forced to execute after kernel C (KernC) from stream 0. Provided with a single work queue, the performance benefits of using concurrent kernel execution can be limited.

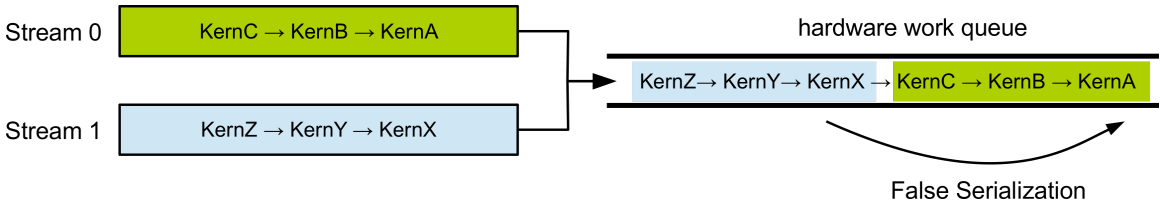


Figure 2.3: False serialization among kernels from two streams due to a single hardware work queue.

To address the false serialization issue, NVIDIA has engineered 32 parallel hardware work queues to parallelize concurrent kernel execution, also known as Hyper-Q [68]. With Hyper-Q, independent CUDA streams can be inserted into the parallel hardware queues without serialization. Figure 2.4 shows that as more hardware queues are supported, the dispatched CUDA streams can run simultaneously. By having 32 work queues available for concurrent kernel execution, higher device utilization can be achieved in many scenarios by dispatching streams on what would otherwise be an idle streaming multiprocessor.

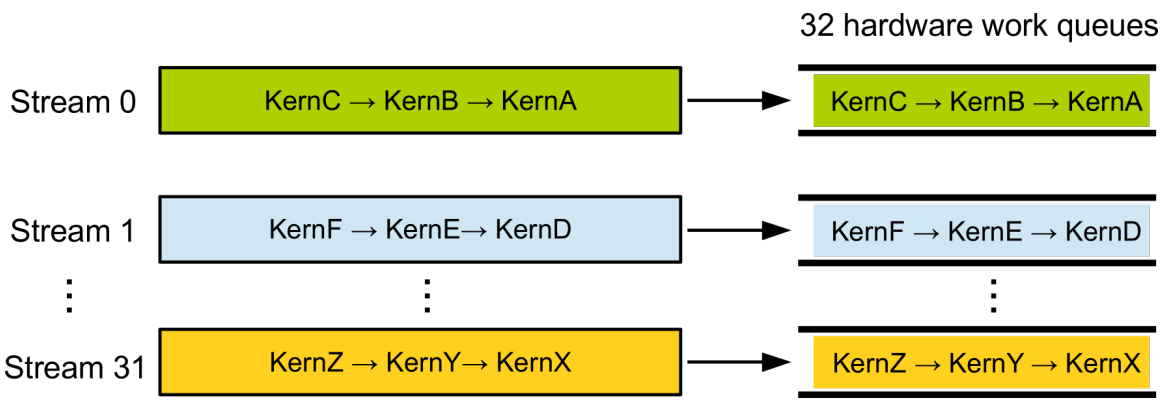


Figure 2.4: Hyper-Q supports for parallelizing the execution of CUDA streams.

2.2.2 Concurrency using CKE

Concurrent kernel execution can allow multiple kernels to share the GPU to maximize utilization and throughput. In addition to the concurrency of kernel execution, CKE can achieve concurrency between kernel execution and data transfer, and also enable concurrency between GPU and CPU computations. Figure 2.5 illustrates the scheme to breakdown serial execution into multiple independent execution streams, leveraging CKE to attain N-way concurrency, where N is increased from 2 to 4. In Figure 2.5 (b), 2-way concurrency is attained by overlapping D2H(1) and Kern(2), where Kern stands for the kernel, H2D stands for a host-to-device data transfer, and D2H stands for a device-to-host data transfer. Since concurrent bi-directional data transfers are supported on modern GPUs, a D2H transfer from stream 1 and a H2D transfer from stream 3 will not interfere with each other if scheduled concurrently. By having D2H(1), Kern(2), H2D(3) execute concurrently, 3-way concurrency is achieved, as shown in Figure 2.5 (c). Furthermore, the CPU can work collaboratively with the GPU, taking care of a portion of the computation. Figure 2.5 (d) highlights the achieved 4-way concurrency among D2H(1), Kern(2), H2D(3) and Kern(4) on a CPU. Overall, leveraging all of the inherent concurrency present in an application using CKE, a large portion of the communication overhead can be hidden and the application throughput can be increased significantly.

2.2.3 Performance Factors on CKE

Concurrent kernel execution can improve the overall system throughput by exploring four types of concurrency, including:

1. the concurrency between kernel executions,

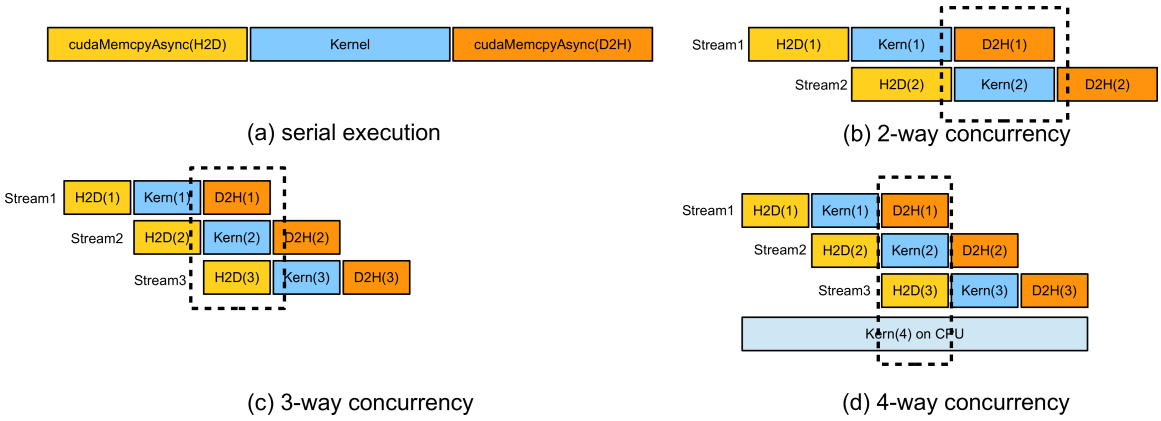


Figure 2.5: From serial execution to 4-way concurrency using concurrent kernel execution. H2D stands for host-to-device data transfer, D2H stands for device-to-host data transfer, Kern stands for Kernel and the number in the parenthesis stands for the stream id.

2. the concurrency between data transfers,
3. the concurrency between data transfer and kernel execution, and
4. the concurrency between GPU computation and CPU computation.

There are several performance factors that could constrain the achievable concurrency and limit the overall throughput using CKE, which include:

1. resource limits on the device,
2. pageable memory or pinned memory for host data allocation.
3. the number of copy engines on the GPU, and
4. the proper coding of CUDA streams when programming CKE.

GPUs provide high memory bandwidth and massively parallel computing cores for high-performance computing. But the limited number of registers and shared memory can constrain the maximum number of active running threads for a GPU kernel. For instance, if a kernel requires 100 registers per thread and there are 64K registers per streaming multiprocessor, only 640 threads per SM are allowed to run concurrently. Given 2 SMs on the GPU, and the kernel can launch 1280 threads, there is no extra resources to host another kernel for concurrent execution, unless some threads finish their computation early and release the occupied resources. Therefore, the relationship

between the kernel configuration and the available device resources plays an important role in tuning multi-kernel concurrent execution.

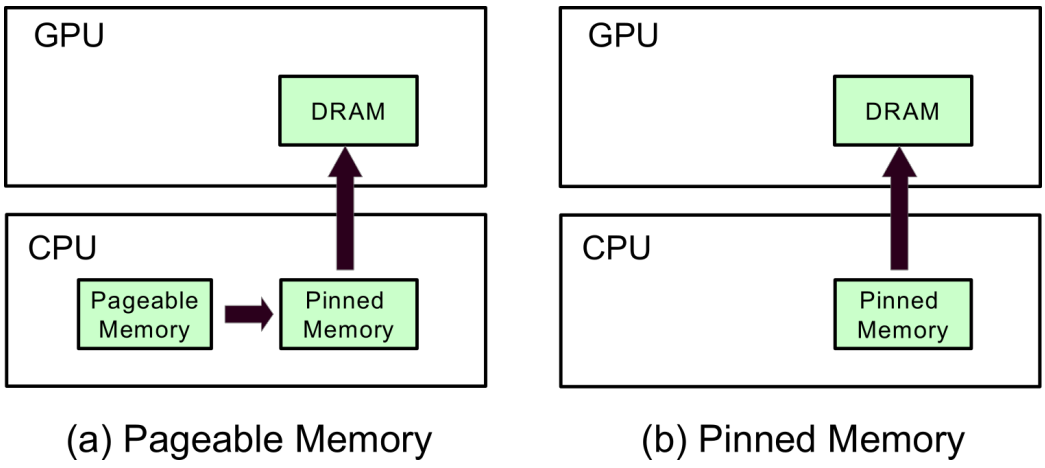


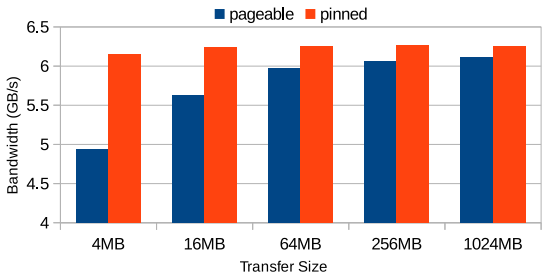
Figure 2.6: Compare pageable data transfer with pinned data transfer on GPUs.

To prepare the data for GPU kernel computation, there are two types of memory: i) pageable memory and ii) pinned memory. Data creation using $malloc$ allocates data in the pageable memory space. An alternative is using $cudaMallocHost$ or $cudaHostAlloc$, is to use pinned memory. The difference between $cudaMallocHost$ and $cudaHostAlloc$ is that $cudaHostAlloc$ API supports fine-grained control over the pinned memory by flagging different properties of the allocated space (e.g., the *cudaHostAllocMapped* option allows mapping the allocation into the CUDA address space) [77].

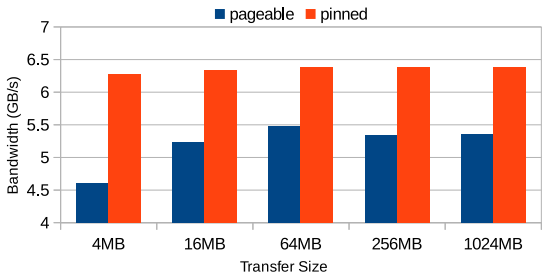
A comparison between pageable and pinned memory is shown in Figure 2.6. If we do not use pinned memory in CUDA, the CPU allocates data in pageable memory. When a data transfer is requested, the CUDA driver allocates a temporary pinned memory (or page-locked) space, copies data from pageable memory to pinned space, and then transfers the data to the GPU device memory. Due to the extra staging step, the data transfer using pageable memory takes more time than using pinned memory.

Figure 2.7 shows the achieved bandwidth when using pageable and pinned memory for data transfer on two different GPU platforms (NVIDIA GTX 950 GPU and Tesla K40c GPU). Though increasing the transfer size improves the achievable bandwidth when using pageable memory, the pinned memory consistently provides higher bandwidth than the pageable memory. The staging overhead is more significant for small data transfers, compared to using larger data transfers. Pinned memory provides a sustainable bandwidth, regardless of the transfer size. It improves the bandwidth

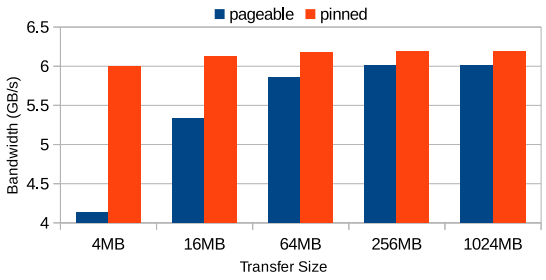
by 2%-45%, with an average of 14% improvement, over pageable memory.



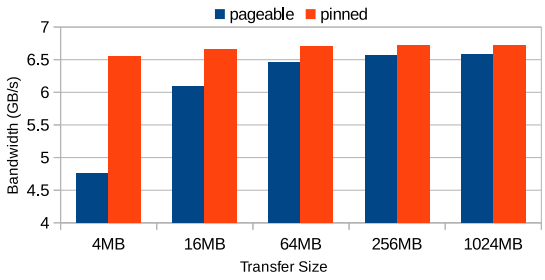
(a) Host-to-Device transfer on a GTX 950.



(b) Device-to-Host transfer on a GTX 950.



(c) Host-to-Device transfer on a K40c.



(d) Device-to-Host transfer on a K40c.

Figure 2.7: Comparion of the achievable PCIe bandwidth between pageable and pinned memory for data transfers on two GPUs. The NVIDIA GTX 950 is a commodity gaming GPU, whereas the Tesla K40c is a computing GPU for servers. The bi-directional, host-to-device and device-to-host transfers are evaluated.

We can see that transferring small data using pinned memory yields higher bandwidth than using pageable memory. However, oversubscribing the pinned memory can reduce the physical memory available for the operating system. To fine the best performance for an application, the programmer should verify the available pinned memory on the Linux system by running the following command:

```
$ulimit -a | grep "max locked memory"
```

To increase the pinned memory, users can edit the configuration file /etc/security/limits.conf (on a Ubuntu system), and add the following two lines.

```
*       hard    memlock         unlimited
*       soft    memlock         unlimited
```

In addition, coding styles and the number of copy engines can also impact the execution behavior when using concurrent kernel execution. In Figure 2.8 we show two coding styles for CKE, ***interleaved*** and ***batch***, based on the operation order across multiple streams. The interleaved style iterates through all operations, one stream at a time, whereas the batch style dispatches all the host-to-device (H2D) transfers first, then all the kernel computations, and finally all the device-to-host (D2H) transfers.

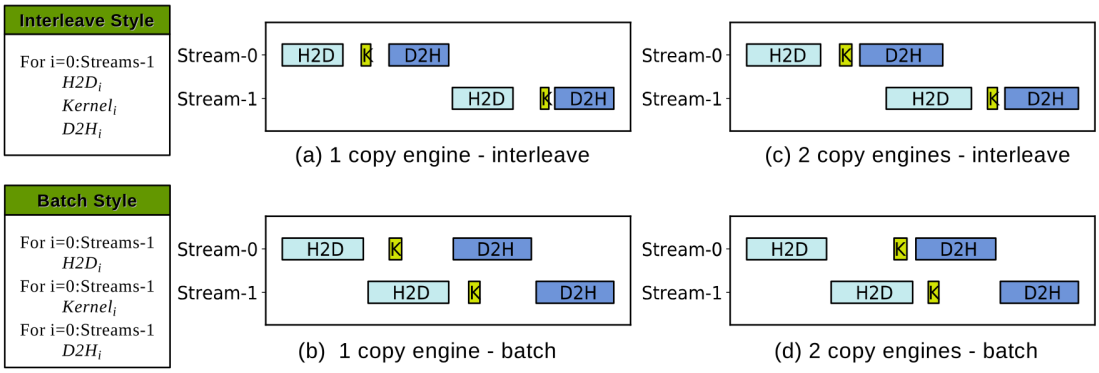


Figure 2.8: An example of concurrent kernel execution using two CUDA streams. Based on the coding style, the execution patterns are shown for NVIDIA GPUs with one or two copy engines. H2D stands for host-to-device transfer, K stands for kernel execution, and D2H stands for device-to-host transfer.

Figure 2.8 illustrates how the copy engine and coding styles affect the execution pattern when using CKE. In this experiment, the profiled kernel is a simple vector addition. For GPUs with a single copy engine, the PCIe bus will be utilized, one CUDA call at a time. If we interleave operations, as shown in Figure 2.8 (a), the H2D transfer in stream-1 starts after the D2H transfer in stream-0 ends, since stream-0 is dispatched before stream-1. If we consider a batch style of programming, the H2D transfer of stream-1 starts immediately after the H2D transfer of stream-0 ends. Overlapping data transfer and kernel execution is observed in Figure 2.8 (b).

For GPUs with dual copy engines, if we use interleaved dispatch, as shown in Figure 2.8 (c), the D2H transfer in stream-0 and the H2D transfer in stream-1 are overlapped. Note that the H2D in stream-1 did not start immediately after the D2H in stream-0, which is due to the API launch overhead. Batching the same operations together, two-way transfers can run concurrently on GPUs with two copy engines. The CUDA API launch overhead can be hidden by adopting the batch programming style, as can be seen in Figure 2.8 (d). In general, the batch programming style can achieve better performance than the interleaved style.

Chapter 3

Related Work

In this chapter, we first present an overview on GPU performance modeling. Next, we review related work on optimizing concurrent kernel execution. Finally, we present related work on GPU workload scheduling.

3.1 GPU Performance Modeling

To help software engineers understand the performance characteristics and bottlenecks of GPU-accelerated applications, Hong et al. designed an analytical model that can estimate the execution time of the parallel programs running on GPUs [41]. They defined the ***memory warp parallelism*** (MWP) by estimating the number of memory requests that can be executed concurrently, based on the running threads and their memory bandwidth consumption. They introduced ***computation warp parallelism*** (CWP) as a metric to measure how much computation can be done by other warps while one warp is serving a memory access request. Their analytical model considers a set of metrics, including the number of warps per SM, total execution cycles, number of dynamic instructions, cycles per instruction, coalesced/non-coalesced memory accesses, and synchronization effects. They explore memory-level parallelism and try to explain how a GPU can hide memory latency using computation warps. They perform their evaluation on an early NVIDIA Tesla architecture, and reported a 5.4% mean error for their model when running microbenchmarks and 13.3% mean error for full GPU applications.

Kothapalli et al. proposed a GPU performance prediction model which integrated three performance models, 1) bulk-synchronous parallel (BSP) model [98], 2) parallel random access machine (PRAM) [28], and 3) queue-read queue-write asynchronous PRAM (QRQW) model [29]. Their proposed model measured computation cycles (N_comp) and memory access cycles (N_memory),

and applied the max(sum) operation on the thread cycles to predict the GPU kernel runtime [54]. No intra-block synchronization nor cache performance were taken into consideration.

Kerr et al. characterized GPU applications at a PTX (a low-level parallel thread execution virtual machine) level and applied linear regression to predict the application runtime [53]. The metrics available at the PTX level are limited. For instance, kernel size and the underlying GPU configurations cannot be obtained. Their prediction accuracy varied significantly depending on the workload.

Baghsorkhi et al. proposed an analytical model to predict the runtime of a GPU kernel [10]. The model analyzes how a GPU kernel makes use of the GPU microarchitectural features and predicts the runtime based on the workflow graph of the kernel.

Luo and Suda used ***Ocelot*** [52] to analyze PTX GPU kernels and predict the overall execution time by summing up the estimated cost of memory instructions and the cost of different compute models [58]. Their proposed memory and compute models are based on Hong's work [41].

Zhang and Owens developed a throughput model based on characterizaing the: 1) instruction pipeline 2) shared memory access, and 3) global memory access. Based on the dynamic statistics collected from the Barra simulator [18], their model can predict application performance within 5-15% of the actual runtime on dense matrix multiplication, tridiagonal solver and sparse matrix-vector multiplication kernels.

Lai and Seznec proposed a timing estimation tool for GPUs named (***TEG***) to estimate the kernel runtime [56]. TEG uses kernel assembly, and an instruction trace collected using the Barra simulator, as input. Based on the instruction execution order colleced from the issue engine model and the functional units model, timing traces are collected. Their prediction accuracy for matrix multiplication on a GTX 280 (NVIDIA Tesla architecture) varies from -11% to 4%, depending on the number of warps dispatched on the streaming multiprocessor. Their work on TEG did not consider more detailed performance factors, such as shared memory bank conflicts and cache performance.

Instead of focusing on predicting kernel performance, Boyer et al. proposed a data transfer model to improve the overall GPU performance prediction [13]. The data transfer model is a simple linear regression model where the intercept (i.e., the transfer overhead) and the slope (the ratio of the elapsed time over the transfer size) are learned through benchmarking. Their model is able to predict data transfer overhead with an error of 8%. The inclusion of their model reduces the prediction error from 255% to 9%.

Werkhoven et al. developed a performance model to predict the benefits of overlapping data transfers and kernel computation for GPU applications [99]. Their data transfer model uses the ***LogGP*** model [7] which models large message transfer. which models large message transfer. The

proposed model can estimate application performance with an error ranging from 3.8% to 10.7% for the three different implementations.

Wu et al. proposed a GPU performance model using machine learning techniques [105]. To build the model, a set of training kernels are first executed on the based GPU configuration, where the performance counters for each kernel are collected. K-means is applied to form representative clusters based on the performance counter values obtained using training kernels. Each cluster captures a distinctive performance-scaling behavior. Then a three-layer fully-connected neural network model is trained to identify the best cluster, based on the performance counters of a GPU kernel. Their model can predict a kernel's performance across a range of different GPU configurations, with a prediction accuracy less than 15% as compared to the actual execution times.

Previous work has studied kernel behavior using simulators (e.g., Ocelot [52] and Barra [18]) or profiling tools (e.g., cuobjdump [76] and nvprof [71]). Based on the collected traces, three types models have been proposed to predict GPU performance: 1) analytical 2) empirical, and 3) machine learning based. Data transfer models have also been proposed to improve the prediction accuracy.

3.2 CKE Development and Exploration

As general purpose computing on GPUs becomes popular, applications with diverse source requirements have been ported to GPUs for acceleration. Since modern GPUs can accommodate more computing resources (see Table 1.2) with each new generation, GPU resources are frequently underutilized by a single kernel. To solve this problem, concurrent kernel execution (CKE) has been proposed to allow multiple kernels running concurrently on a GPU [27].

Guevara et al. pioneered the concept of GPU concurrent execution by proposing the concept of ***kernel merging*** on the NVIDIA Tesla architecture [36]. The Tesla family of GPUs does not support concurrent kernel execution [66]. They showed that by combining two kernels into a single kernel whenever GPU resources are underutilized, they could improve application throughput by 12-20% improvement versus serial execution.

Wang et al. proposed a ***kernel fusion*** approach to improve the power efficiency of GPU applications [100]. They identified three different fusion strategies: 1) inner-thread fusion, 2) inner thread-block fusion, and 3) inter thread-block fusion. Using dynamic programming to select the best fusion strategy, they demonstrated significant improvements in throughput and power efficiency. However, merging multiple kernels into a single kernel can leave some resources idle, since GPU resources can be reclaimed until all the merged kernels finish their computation.

The Fermi architecture was the first generation of NVIDIA GPUs to support concurrent kernel execution [27]. The main drawback of NVIDIA's Fermi implementation was that we can only run concurrent kernels from the same thread context. Multi-threaded programs have to be run sequentially, introducing context switching overhead. Wang et al. proposed ***manual context funneling***, allowing kernels from different threads to run concurrently, achieving better performance than the automatic context funneling in CUDA v4.0 [101].

Adriaens et al. proposed ***spatial multitasking*** for concurrent kernel execution, where each kernel uses a subset of the GPU resources [6]. They tested different partitioning heuristics using GPGPU-Sim [11]. They suggested using a Smart Even policy, which can achieve an average of 1.16x, 1.24x and 1.32x speedup over cooperative multitasking (i.e, sequential execution) for two, three, four concurrently running applications.

Wende et al. observed that CKE performance breaks down whenever multiple host threads dispatch kernels in order, without synchronizing their actions. To avoid this issue, they adopted a producer-consumer approach to mange GPU kernels by reordering them before the actual invocations [104]. Their reordering mechanism dispatches one kernel per queue per round-robin cycle. After applying the ***kernel reordering*** scheme on a thermodynamics simulation, they observed a 1.4x-2.6x speedup for a 1-GPU implementation and 1.3x-2.x speedup for a 2-GPU implementation, when compared to no kernel reordering.

Naively scheduling kernel pairs, without the knowledge of the kernel characteristics, can have a negative impact on concurrent kernel execution. Gregg et al. developed ***KernelMerge*** scheduler that can merge multiple OpenCL kernels into a single kernel call to achieve better CKE performance [34]. Two different scheduling algorithms are applied, round-robin work-stealing and fixed partitioning. Using KernelMerge, 39% of the merged kernels showed a speedup, which translated to an 18% maximum speedup. Their KernelMerge did not consider the characteristics of a kernel that could potentially better guide us of whether to exploit CKE for two kernels.

Wu et al. applied ***kernel fusion*** and ***kernel fission*** to optimize data warehousing applications [106]. Kernel fusion provides opportunities to reduce the GPU memory footprint and improve temporal locality, whereas kernel fission can overlap kernel computation and data transfer. By combining kernel fusion and fission, Wu et al. reported a 41.4% improvement over serial execution, 31.3% better than kernel fusion only, and 10.1% better than kernel fission only.

Pai et al. identified the issue of resource underutilization for GPU kernels and proposed an ***elastic kernel*** for fine-grained control over resource usage [80]. To use an elastic kernel, the native CUDA kernel needs to be transformed in such a way that the physical grid is mapped to a virtual grid which is similar to thread mapping. By incorporating elastic kernel-aware concurrency policies,

the proposed technique increased system throughput (STP) and the average normalized turnaround time (ANTT) by 1.21x and 3.73x, respectively.

Kernelet was developed by Zhong and He to augment the runtime support for concurrent kernel execution on GPUs [116]. It implements kernel slicing (i.e., kernel fission) at the granularity of thread blocks, where each kernel slice has tunable occupancy. By applying a greedy scheduling algorithm, they could improve performance by 31% and 23% on NVIDIA Tesla C2050 and GTX 680 GPUs, respectively.

Jog et al. proposed a memory scheduling policy named ***first-ready round-robin FCFS*** to improve the fairness and performance for concurrently executed GPU applications [45]. Based on the evaluation results using GPGPU-sim, their application-aware memory scheduler can preserve DRAM page hit rates and make sure the collocated memory-intensive applications do not starve other applications.

Liang et al. used a thread-block interleaving method to explore the ***spatial-temporal multitasking*** capability for concurrent kernel execution [57]. Their proposed spatial multitasking scheme searches the optimal SM allocation for concurrent kernels, whereas the temporal multitasking scheme finds the optimal scheduling order for a group of kernel sets. They have demonstrated performance improvements of 46% and 37% over sequential execution and default CKE execution, respectively. Their scheme allocated SMs for different kernels, without considering how best to interleave thread blocks from different kernels on the same SM to improve resource utilization.

Jiao et al. applied kernel slicing and leveraged concurrent kernel execution and DVFS (Dynamic Voltage and Frequency Scaling) to improve energy efficiency for GPUs [43]. Since running more than two kernels concurrently did not yield enough benefit, they experimented with two-kernel combinations in their work. A two-layer neural network model is trained to predict the optimal performance per watt by varying the frequency settings, given a kernel pair and a specified block ratio. By dynamically adjusting the block ratio and selecting different combinations of the clock and memory frequency for concurrent kernels, they obtained performance per watt improvements of up to 34.5% when compared to sequential execution.

Maestro was proposed by Park et al. to perform dynamic resource management for efficient utilization of multitasking GPUs [81]. By integrating three components: 1) dynamic resource management framework, 2) 2-way resource allocation, and 3) kernel-aware warp scheduling mechanisms, they proposed a framework that improved throughput by 20.2% over a spatial multitasking scheme, and 13.9% over using a simultaneous multikernel (SMK) approach [102].

Gong et al. developed ***Twinkernels*** as a compiler solution to optimize the performance of concurrent kernel execution [31]. Two kernels are analyzed at the binary level and the kernel

instructions are interleaved to increase the utilization of device resources. Gong et al. achieved a 25% performance improvement as compared to the default SIMT model.

Wen et al. proposed ***MaxPair*** for OpenCL to optimize the throughput of concurrent kernel execution [103]. They model the concurrent execution problem as a graph, where each kernel is represented as a vertex and the edge weight captures the speedup ratio for co-running the two adjacent vertices. They used a weighted max matching algorithm to find the best scheduling kernel pairs. Compared to the state-of-the-art schemes, ***MaxPair*** can improve performance by 8% and 4% on an AMD and an NVIDIA platform, respectively.

To reduce the corunning interference and minimize data cache trashing and memory pipeline stalls for CKE, Dai et al. suggested: 1) balancing memory accesses for concurrent kernels, and 2) limiting the in-flight memory instructions [20]. Their proposed schemes outperformed two state-of-art intra-SM sharing schemes, Warped-slicer [107] and SMK [102], by 24.6% and 27.2%, respectively.

3.3 GPU Workload Scheduling

For warehouse-scale computing, delivering the advertised quality-of-service (QoS) level is critical, especially for commercial clouds. Workloads scheduled on the same node will compete for shared resources, frequently leading to performance degradation. By characterizing shared resource utilization of each workload, a cloud system can minimize the degree of interference. In this section, we present previously published work on GPU workloads scheduling [40].

3.3.1 Standalone Environment

Bautin et al. proposed ***GERM***, a scheduler for GPU workloads, that utilizes a Deficit Round Robin scheduling policy [12]. By sampling the time of day register, the scheduler estimates the computation time of the current command group from previously completed command groups. Based on the timing information, GERM can adjust resource allocation for each process in the following cycles.

TimeGraph, proposed by Kato et al. , is a real-time GPU scheduler implemented at the device driver level to manage GPU resources in a responsive manner [50]. ***BifGraph*** supports two scheduling policies, Predictable Response Time (PRT) and High-Throughput (HT), to achieve a balance between response time and throughput. Compared to a tick-driven scheduler, TimeGraph improved system throughput by 30x, with only a 4-10% performance overhead. Later, Kato et al. developed a responsive GPGPU execution model (***RGEM***) to protect high-priority tasks from

competing workloads in multitasking environments [49]. Two scheduling policies are presented in their framework, 1) Memory-Copy Transaction scheduling (based on the default pageable memory) and 2) Kernel Launch scheduling. The Memory-Copy Transaction policy truncates a large memory copy operation into smaller pieces, providing preemption points at boundaries between the separate pieces. The Kernel Launch policy adopts the Predictable Response Time (PRT) policy in the user space. Their experiments showed that the response times of high-priority GPU workloads can be guaranteed under the proposed execution model. To address the resource-sharing problems in GPU-accelerated windowing systems, Kato et al. proposed 1) Priority Inheritance with a X-server (PIX) protocol, and 2) Reserve Inheritance with a X-server (RIX) protocol [48]. By evaluating the protocols on graphics rendering tasks, they observed that multiple GPU-accelerated graphics applications running concurrently can be correctly prioritized and isolated. Then, they developed ***Gdev*** at an operating system level to enhance GPU resource manangement [51]. They adopted the same concept of Memory-Copy Transaction scheduling to split transactions, such that the staging overhead of moving data from pageable memory to pinned memory can be hidden. To improve the bandwidth utilization for virtual GPUs, they proposed a bandwidth-aware non-preemptive device (BAND) scheduling algorithm. BAND does not reduce the priority when the resource budget is exhausted, and it adds ”time-buffering” for bursty workloads to achieve fairness.

Menychtas et al. proposed ***disengaged scheduling*** to achieve both fairness and good utilization for GPU resource management at the OS level [62]. By combining a token-based timeslice scheduler with their scheme, they can better control fairness for a multitasking environment. The disengaged scheduler limits idle time and incurs less than a 4% overhead on average.

3.3.2 Virtualization Environment

Virtualization technology allows us to abstract away the underlying hardware, improving system utilization and reducing expenditures on hardware and energy. As GPUs emerge as the first choice of accelerator in cloud environments, virtualized GPUs become important instances for high-performance computing. Due to the non-preemptive nature of GPUs, efficiently sharing resources while achieving a high degree of quality-of-service has emerged as a challenging task.

GViM, proposed by Gupta et al. , uses Round Robin and XenoCredict(XC)-based scheduling schemes to dispatch GPU requests at the driver level [37]. In their framework, each GPU request is assigned with credits which represent the given execution time. The higher the credits, the more time is allocated to the execution of guest requests on the GPU. XC processes the GPU call buffer for a period based on the credit obtained. This enables weighted fair-sharing between guest VMs.

Later, Gupta et al. proposed ***Pegasus*** to meet various requirements for GPU workloads

across virtual machines [38]. ***Pegasus*** includes five scheduling schemes, including: 1) first-come-first-served (FCFS), 2) proportional to the fair-share (AccCredit), 3) strict co-scheduling (CoSched), 4) augmented credit-based scheme (AugC) and 5) SLA feedback-based (SLAF) scheduling. With moderate extra overhead, they report a 18-140% performance improvement as compared to the default GPU-driver scheduling.

As GPU applications in VMs compete for shared resources, not every application has enough parallelism to fully utilize the available GPU resources. To address the issue, Ravi et al. proposed consolidating multiple GPU kernels from different VMs for space and time sharing [85]. Space sharing allows kernels to be assigned to different streaming multiprocessors (SMs), whereas time sharing supports sharing resources within the same SM. Their framework computes the affinity score between every two kernels to estimate the performance improvement. The affinity score is assigned depending on the total number of kernel threads, with the idea that lower the thread volume, the higher the affinity. The framework schedules n kernels for concurrent execution, depending on the set of affinity scores.

GPUvm, proposed by Suzuki et al. , adopted the BAND scheduler from Gdev and incorporated CPU time into the credit adjustment to improve decision making based on a credit scheduling policy [93]. Tian et al. proposed ***gVirt*** to control GPU commands in the guest ring buffer during the VM's time slice to minimize the wait period [84]. It ensures that the total number of submitted commands is within the time slice.

To optimize gaming applications in the cloud, Qi et al. proposed the ***VGRIS*** scheduler to address a range of performance requirements [84]. Three scheduling policies are included in the scheduler: 1) SLA-aware, 2) Proportional Share and 3) Hybrid. SLA-aware scheduling provides the minimum amount of GPU resources to each VM, while Proportional-Share scheduling distributes resources based on the priority. Hybrid scheduling applies SLA-aware first, and then switches to Proportional-Share scheduling if additional resources are available. Zhang advanced VGRIS by proposing a ***vGASA*** scheduler for gaming applications run in the cloud [114]. It deploys a dynamic feedback control loop using a proportional-integral controller, to calculate the SLA requirements during runtime to improve SLA-aware scheduling.

gScale, proposed by Xue et al. , advances the gVirt scheduler by introducing a private shadow graphics translation table (GTT) [108]. GTTs requires page table copying on every context switch, except for idle vGPUs. The scheduler implements slot sharing, which divides graphics memory into several slots and dedicates a single slot to each vGPU. With low runtime overhead, gScale can achieve a 4x improvement in scalability compared to gVirt.

3.3.3 Interference Analysis

As more and more applications are streamed onto the GPU for concurrent execution, delivering guaranteed performance is a non-trial problem. A number of interference factors can lead to performance degradation. Jog et al. showed that competing for DRAM bandwidth can severely impact concurrent kernel execution performance [45]. They proposed a first-ready round-robin policy, versus a strict first-come-first-serve policy, to improve the fairness between collocated applications. Phull et al. observed that a major source of GPU interference is due the kernel runtime and the kernel launch frequency, where GPUs are time-shared among jobs [82]. Later, Chen et al. pointed out four major factors that could lead to long tail latency: 1) the duration and occupancy of GPU kernels, 2) the kernel scheduling order, 3) the number of kernels, and 4) contention on the PCIe bandwidth [17]. Based on these observations, they further developed a performance prediction model for interference-aware scheduling [16]. ***Mystic***, proposed by Y. Ukidave et al. , utilizes a short profile of a GPU application and applies Collaborative Filtering to predict the full profile information [97]. After computing the similarity distance based on the predictive features, the least similar application will be dispatched to the target GPU node to minimize the concurrent running interference.

Previous work has suggested interference factors covering the temporal and spatial behavior of kernel execution and resource utilization for GPU workloads. To obtain these interference factors, workloads are required to be profiled using vendor-specific profiling tools. Profiling all the performance metrics is a time-consuming process. Short profile for targeted interference factors is preferred to minimize the overhead. In this thesis, we are interested in applying machine-learning techniques to identify a few prominent interference factors while reducing the profiling overhead. Since most of the developed schedulers focus on resource availability and few works have been proposed to study the interference, we are motivated to develop an efficient interference-aware scheduler for GPU workloads.

Chapter 4

Model-based Concurrent Kernel Analysis

As discussed in Chapter 2.2.2, leveraging concurrent kernel execution can improve multi-kernel concurrency and improve the overall throughput of a GPU application. However, it is challenging to maximize the potential performance benefits of using CKE. Chapter 2.2.3 shows a list of factors that can impact the CKE performance for a GPU application. Typically, a developer will have to compile and tune the program many times to obtain the best performance. To tackle this problem, we present *Moka*, an empirical model to estimate the performance benefits using concurrent kernel execution. The model analyzes a non-CKE application comprising multiple kernels and delivers an estimate of the performance ceiling that can be achieved by leveraging CKE on a particular GPU. ***Moka* attempts to answer the question of whether a programmer should attempt to employ CKE for an application.** The model also provides guidance to find the best performing kernel-stream mapping, quickly identifying the best CKE configuration, resulting in improved performance and improved utilization of the GPU. *Moka* accounts for many runtime factors that could potentially impact performance, including the host-to-device data transfer overhead, the stream-launch overhead, the block scheduling mechanism and other potential resource-associated bottlenecks. In addition, we present an effective block size tuning technique that can further benefit CKE.

An overview of *Moka* framework is presented in Figure 4.1. At first, GPU applications are profiled to collect application characteristics. Based on the profiled metrics, *Moka* classifies a GPU kernel as either compute-intensive and memory-intensive, and suggests the best block size accordingly. After tuning kernel performance, *Moka* starts modeling the execution of concurrent kernels encapsulated in the CUDA streams, taking into account data transfers and kernel execution

behavior. Equipped with knowledge of the dynamic runtime characteristics, as well as the static properties of the kernels, *Moka* includes a data transfer model and a kernel execution model to estimate the CKE performance for an entire application. Since our work is based on the NVIDIA GPU architecture, CUDA terminology is used, but our work can easily be adapted to other GPU standards.

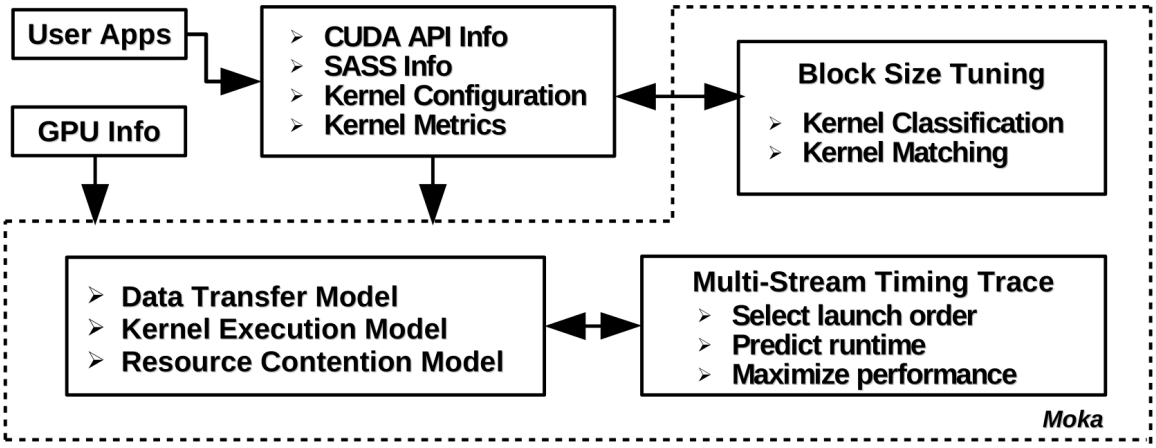


Figure 4.1: Overview of Moka.

4.1 Block Size Tuning

A kernel can launch hundreds of thousands of threads on a GPU. These threads are organized in blocks, which run concurrently in a streaming multiprocessor (SM). If we perform computation using small blocks, we may not fully utilize the available memory bus bandwidth. Utilizing large blocks can oversubscribe device resources, such as registers and shared memory, limiting the number of concurrent blocks run per SM. Finding the right thread block size can be crucial to achieving good performance, especially for stencil computations [22, 47, 95]. Here, we propose the *Similar Kernel Method* (SimK) for tuning the block size. Our *SimK* approach utilizes the profiling information obtained from NVIDIA's *nvprof* profiling tool, capturing information on resource requirements and runtime characteristics of the application [72]. Based on the ratio of memory accesses to compute operation cycles, each GPU kernel is categorized into one of two groups: i) memory-intensive, or ii) compute-intensive. *SimK* computes the Euclidean distance between the input kernel and training kernels to search for the most similar kernel. The block size of the most similar kernel is then selected by *SimK*. Figure 4.2 shows the workflow of *SimK*.

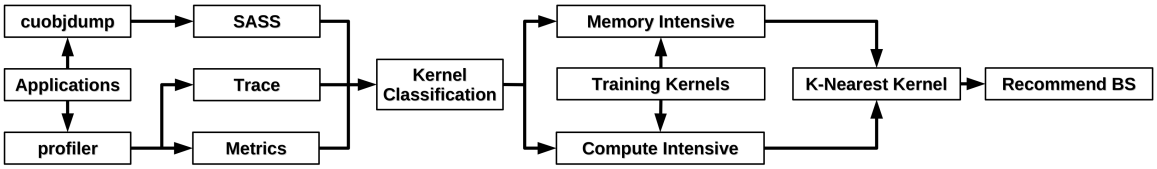


Figure 4.2: Similar Kernel Method for block size tuning.

4.1.1 Kernel Classification

To determine whether a kernel is memory intensive or compute intensive, Yang et al. evaluated the memory-transaction intensity of an input kernel, computing the ratio of the number of global memory transactions versus the number of compute instructions [44]. If we ignore the cycles per memory transaction and other memory system usage, Yang's method results in too much variance to provide accurate guidance. To obtain a more accurate estimate, GPU simulators and PTX emulators can be used [11, 96, 52]. The GPU low-level assembly instrumentation tool *SASSI*, can also provide accurate traces on NVIDIA platforms, which are typically more accurate than using a simulator [90]. However, the effectiveness of these tools can be limited, given that most open source GPU simulators lag behind the latest hardware features. Even the SASSI toolset, which is developed by NVIDIA, is presently only supported on the CUDA 7 toolchain [89].

In *SimK*, we propose a warp-based scheme to classify each kernel. A *warp*, which is a group of 32 threads on NVIDIA architectures, is the smallest non-divisible execution unit of a GPU kernel, executing in Single Instruction Multiple Data (SIMD) mode. A single warp's execution characteristics should be representative of the execution for the entire kernel. Therefore, a kernel can be considered memory-intensive if the warp is memory-intensive. To generate accurate cycle counts for memory and arithmetic instructions in a warp, we need the frequency of each instruction in the warp, and the cycles per instruction for those operationsd. From the kernel metrics generated by nvprof, the frequency of executed GPU assembly instructions (SASS) can be obtained (see Table 4.1). The distribution of different SASS instructions is approximated based on the SASS histogram in the GPU kernel binary [76]. Meanwhile, we obtain the number of clock cycles for each SASS instruction through benchmarking. In Tables 4.2 and 4.3, we provide cycle counts for the SASS instructions obtained on a GTX 950 platform, an NVIDIA Maxwell GPU. During microbenchmarking, the compiler option *-Xptxas -O0* is used to short-circuit optimization by the PTX assembler. This is only used during profiling.

Table 4.1: Metrics used for characterizing SASS instruction execution.

Metric Name	Metric Type	Description
inst_fp_32	Compute	FP32 instructions
inst_fp_64	Compute	FP64 instructions
inst_integer	Compute	Integer instructions
inst_compute_ld_st	Compute	Compute load store instructions
gld_transactions	Memory	Global memory load transactions
gst_transactions	Memory	Global memory store transactions
shared_load_transactions	Memory	Shared memory load transactions
shared_store_transactions	Memory	Shared memory store transactions

To produce instruction frequencies for each warp, we apply the following equations:

$$warp\_fp32 = inst\_fp\_32/total\_threads \tag{4.1}$$

$$warp\_fp64 = inst\_fp\_64/total\_threads \tag{4.2}$$

$$warp\_int = inst\_integer/total\_threads \tag{4.3}$$

$$warp\_ldst = inst\_compute\_ld\_st/total\_threads \tag{4.4}$$

$$warp\_ldg = gld\_transactions/total\_threads \tag{4.5}$$

$$warp\_stg = gst\_transactions/total\_threads \tag{4.6}$$

$$warp\_lds = smld\_transactions/total\_threads \tag{4.7}$$

$$warp\_sts = smst\_transactions/total\_threads \tag{4.8}$$

To measure the compute instruction cycles per warp, we apply the following equations:

$$\begin{aligned} warp\_cmp = {} & warp\_fp32 \times sass\_fp32\_clks \\ & + warp\_fp64 \times sass\_fp64\_clks \\ & + warp\_int \times sass\_int\_clks \\ & + warp\_ldst \times sass\_ldst\_clks \end{aligned} \tag{4.9}$$

To measure the memory instruction cycles per warp, we apply the following equations:

$$\begin{aligned} warp\_mem = {} & warp\_ldg \times sass\_ldg\_clks \\ & + warp\_stg \times sass\_stg\_clks \\ & + warp\_lds \times sass\_sts\_clks \\ & + warp\_sts \times sass\_sts\_clks \end{aligned} \tag{4.10}$$

Each GPU kernel is categorized, as shown below:

$$\text{Compute-intensive} : warp\_cmp > warp\_mem$$

$$\text{Memory-intensive} : warp\_mem > warp\_cmp$$

Table 4.2: Arithmetic SASS instructions on the GTX 950.

Inst. Type	Opcode	Clocks	Inst. Type	Opcode	Clocks
Integer	IADD	15	Single	FADD	15
	ISUB	15		FMUL	15
	IMNMX	15		FMNMX	15
	ISAD	15		FSET	15
	IMUL	86		FFMA	15
	IMAD	101	Double	DADD	48
	ISET	15		DMUL	48
	SHL	15		DMNMX	48
	SHR	15		DFMA	51

Table 4.3: Memory SASS Instructions on the GTX 950.

Access Type	SASS	Clocks
Global Load	LDG	650
Global Store	STG	19
Shared Load	LDS	26
Shared Store	STS	19

In our warp-based scheme, we compute the SASS histogram after dumping the kernel binary using the NVIDIA binary tool *cuobjdump*. The total clocks for a particular data type are computed as the sum of the clocks from the SASS instructions belonging to this data type. For instance, assuming there are 4 *IADDs* and 2 *IMULs* in the SASS histogram, and that the number of integer instructions per warp is 6, the number of integer clocks for the warp is 232.

$$warp\_int \times sass\_int\_clks = 4 \times 15 + 2 \times 86 = 232$$

4.1.2 Data Set

For each kernel, *SimK* helps to find the best match in a pool of GPU kernels. The CUDA SDK suite is used here, where OpenGL applications are omitted since we are focusing on compute [74]. Applying the aforementioned kernel classification method, we identify 20 compute-intensive and 21 memory-intensive kernels, as shown in Tables 4.4 and 4.5. The block size of the

selected kernels is adjustable. We benchmark the kernel's performance on a GTX 950 and record the best block size for each kernel. For 2-D kernels, the block size will be the overall thread block size of the 2-D grid. For a 2-D kernel with 8 threads per block along each dimension, 64 is chosen for the block size. The same rule applies to the 3-D case.

Table 4.4: Compute-intensive Kernels

Kernel Dims	Kernel Name	Application
1D	fwtbatch2kernel	fastWalshTransform
	mergeranksandindiceskernel	mergeSort
	mergesortsharedkernel	mergeSort
	dwthaar1d	dwtHaar1D
	blackscholesgpu	BlackScholes
	bitonicsortshared	sortingNetworks
	sppreprocess2d_kernel	convolutionFFT2D
	sppostprocess2d_kernel	convolutionFFT2D
	spprocess2d_kernel	convolutionFFT2D
	test_interval_newton	interval
	computeangles_kernel	lineOfSight
	binomialoptionskernel	binomialOptions
	inverseCNDKernel	quasirandomGenerator
2D	transposeDiagonal	transpose
	shfl_vertical_shfl	shfl_scan
	matrixMulCUDA	matMul
	padKernel_kernel	convolutionFFT2D
	paddataclamptoborder_kernel	convolutionFFT2D
	stereoDisparityKernel	stereoDisparity
	quasirandomGeneratorKernel	quasirandomGenerator

Table 4.5: Memory-intensive kernels.

Kernel Dims	Kernel Name	Application
1D	scalarProdGPU	scalarProd
	reduce6	reduction
	vectoradd	vectorAdd
	modulateAndNormalize_kernel	convolutionFFT2D
	modulatekernel	fastWalshTransform
	generatesamplerankskernel	mergeSort
	scanExclusiveShared2	scan
	scanExclusiveShared	scan
	uniformupdate	scan
	mergeHistogram256Kernel	histogram
	histogram64Kernel	histogram
	mergeHistogram64Kernel	histogram
	reduceMultiPass	threadFenceReduction
	reduceSinglePass	threadFenceReduction
	uniform_add	shfl_scan
	computeVisibilities_kernel	lineOfSight
2D	transposeCoarseGrained	transpose
	transposeNoBankConflicts	transpose
	transposeCoalesced	transpose
	transposeFineGrained	transpose
	transposeNaive	transpose

4.1.3 Feature Metrics

To find the best kernel match in terms of block size, we collect performance counters values for each kernel (see Table 4.6). Both compute-intensive and memory-intensive kernels share metrics, including the *IPC* (instructions per cycle), the *achieved_occupancy*, the *issue_slot_utilization* and the *eligible_warps_per_cycle*, as shown in Table 4.6. For memory-intensive kernels, *c2m_ratio* is the ratio of *warp_cmp* over *warp_mem*, which measures the intensity of compute operations. The ratios range between 0 and 1. Metrics related to loads and stores from/to global memory (*gld_ratio, gst_ratio*) and shared memory (*sld_ratio*, *sst_ratio*) are computed based on the bandwidth utilization of the memory system [61].

For compute-intensive kernels, the *m2c_ratio*, which quantifies the intensity of memory operations, and the utilization of single and double-precision units (*sp_fu_util, dp_fu_util*), are also included. Note that the *sp_fu_util* is the normalized utilization of the special function units (*special_fu_utilization*) and single precision units (*single_precision_fu_utilization*), derived from the

default profiling metrics.

Table 4.6: Performance counters used to judge kernel similarity.

Memory-Intensive		**Compute_intensive**	
c2m_ratio	gld_raito	m2c_ratio	dp_fu_util
IPC	gst_ratio	IPC	sp_fu_util
achieved_occupancy	sld_ratio	achieved_occupancy	-
issue_slot_utilization	sst_ratio	issue_slot_utilization	-
eligible_warps_per_cycle	shared_utilization	eligible_warps_per_cycle	-

4.1.4 t-SNE Analysis

In our evaluation, each kernel is represented by a high-dimensional feature vector (see Table 4.6), a 10-D vector for memory-intensive kernels, and a 7-D vector for compute-intensive kernels. Identifying the most similar kernel in this high-dimensional space is challenging. Using the *Similar Kernel Method*, the *t-SNE* technique is applied to visualize the high-dimensional feature data and measure the kernel distances in a lower dimensional space [59]. As opposed to Principal Component Analysis, which produces a linear mapping to perform dimensionality reduction, *t-SNE* uses a nonlinear transformation to preserve the local structure of the high-dimensional data. During our analysis, the Euclidean distance is used to compute the distances between kernels in 2-D space. *SimK* selects three most similar kernels, whose block size will be recommended to the target kernel. *SimK* uses a majority voting scheme when evaluating the recommendations of the top-3 most similar kernels and makes a suggestion accordingly. In the case where all three kernels suggest different block sizes, the block size, matching the number of CUDA cores per streaming multiprocessor, is recommended.

4.2 Concurrent Kernel Execution Modeling

Besides tuning kernel performance, *Moka* models concurrent kernel execution by taking into account both data transfers and kernel execution. In the following sections, we will discuss a PCIe-based data transfer model to predict the data transfer time, and a GPU performance model to estimate concurrent kernel execution time.

4.2.1 Data Transfer Model

For concurrent kernel execution, each CUDA stream is responsible for copying data to the device, as well as for the kernel computation. There are two types of data transfers: i) blocking

calls using pageable memory, and ii) asynchronous calls using pinned memory. Figure 4.3 shows the performance impact of different types of data transfer when running a vector addition operation on 1M floats. Using pageable memory for data transfers introduces significant kernel launch overhead in stream-0, see Figure 4.3(a). Using pinned memory can avoid much of this overhead, reducing the overall runtime from 4.75ms to 3.59ms on the GTX 950, achieving a 1.3x speedup. Our data transfer model is based on using pinned memory over the PCIe bus.

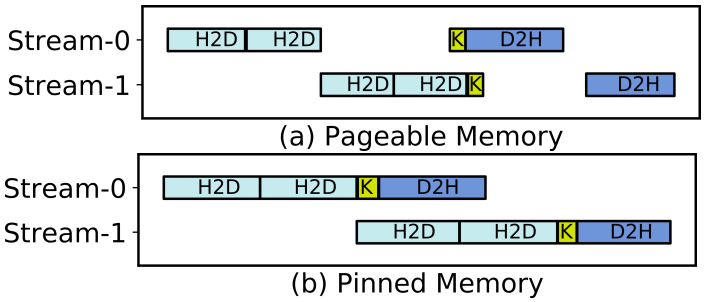


Figure 4.3: Comparing pageable and pinned memory performance impact on two CUDA streams on a GTX 950.

Van Werkhoven et al. proposed the *LogGP* model for CPU-GPU data transfers [99]. Their model is based on linear interpolation for sending a long message, assuming a fixed communication overhead. To quantify the long message transfer speed, we increased the data size from 2 to 1024 bytes. We compared their *LogGP* prediction with linear regression. To train the linear regression model, we measured the data transfer time for data sizes ranging from 1 to 10K floats, with a step size of 4. We use 4 floats as the step size since 128 bits are encoded for each lane of a PCIe 3.0 bus. Three linear regression models are trained, *LR-256, LR-1K, LR-10K*, by using the first 256, 1K and 10K floats from the training set. Figure 4.4 compares performance by predicting the elapsed time to transfer 1K, 1M, 10M and 100M floats.

As shown in Figure 4.4, *LogGP* overestimates the performance by 17% to 49%. Using the same 1K bytes for training, *LR-256* performs better than *LogGP*, predicting the performance of transfers within 0.3%-26%. As the training set grows, the prediction accuracy increases. *LR-1K* and *LR-10K* can predict within 8% and 3%, respectively, as compared to the actual runtime. Hence, linear regression is employed for modeling data transfers.

However, the current model does not consider any PCIe contention that can occur when multiple CUDA streams compete for the same bandwidth. PCIe supports full-duplex communication, where data transfers in opposite directions can be handled simultaneously. Therefore, there is no contention over the PCIe bus since two transfers can be serviced when they are in opposite directions,

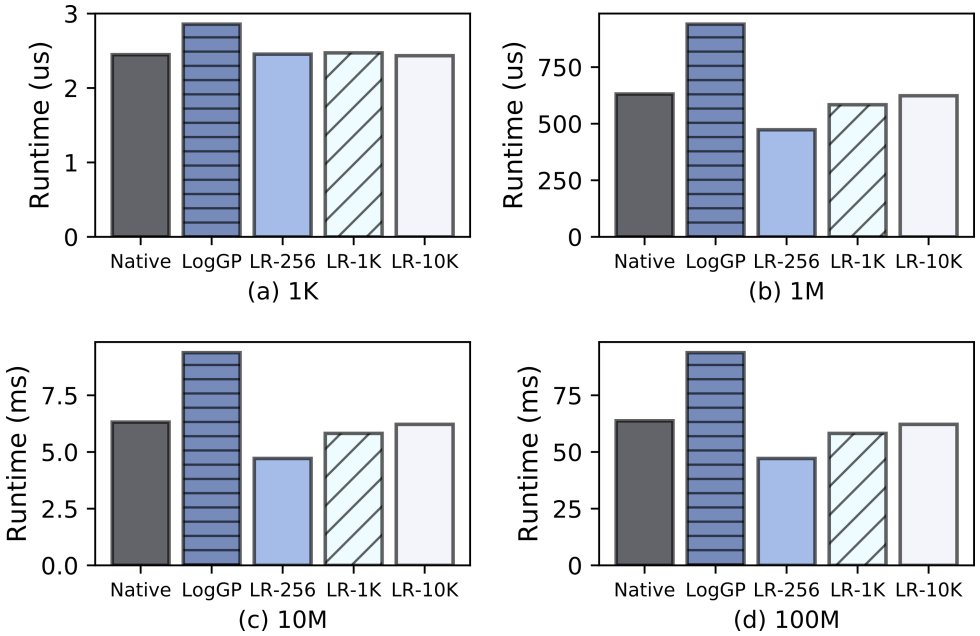


Figure 4.4: Comparison of LogGP and linear regression to characterize the host-to-device data transfer on a GTX 950.

whereas the bandwidth is shared when transferring data in the same direction. Bandwidth contention is considered in our data transfer model, as shown in Algorithm 1.

Algorithm 1 The PCIe Bandwdith Contention for CKE.

if H2D/D2H Contention **then**

$BW(S_i) = BW_{PCIe} / \sum S_i$

else if No Contention **then**

$BW(S_i) = BW_{PCIe}$

end if

$TransTime(S_i) = TransBytes(S_i)/BW(S_i)$

S_i : CUDA stream

4.2.2 Average Block Execution Model

When a kernel is dispatched on a device, the GigaThread Engine on the NVIDIA GPU is in charge of scheduling thread blocks on each streaming multiprocessor (SM). The thread blocks are issued in a round-robin fashion based on the leftover policy [43][80]. Given the independent nature of thread blocks on a GPU, each block contains the same instructions, so we propose using the *Average Block Execution* (AvgBlkExe) to model kernel execution.

Before dispatching thread blocks, the GigaThread Engine will ensure the availability of resources on the target SM. These resources, including registers, shared memory and busy threads, will limit the maximum number of blocks that can be allocated per SM, as expressed in Equation 4.11.

$$
\begin{aligned}
BlkLmt\_Reg &= SM\_Reg / BLK\_Reg \\
BlkLmt\_Shared &= SM\_Shared / BLK\_Shared \\
BlkLmt\_Thread &= SM\_maxThread / BLKSize \\
BlkLmt\_Dev &= SM\_maxBLK \\
\\
maxBlkPerSM &= min\{BlkLmt\_Reg, \\
&\qquad BlkLmt\_Shared, \\
&\qquad BlkLmt\_Thread, \\
&\qquad BlkLmt\_Dev\}
\end{aligned}
\tag{4.11}
$$

The *AvgBlkExe* assumes each block has an identical lifetime. In a round-robin fashion, batches of threads blocks are dispatched to each SM. When device resources are fully occupied, the rest of the blocks will be launched in future iterations. The *AvgBlkExe* metric measures the average block execution time from the kernel execution time, using Equation 4.12.

$$
\begin{aligned}
KernBlksPerIter &= GPU\_SMs \times maxBlkPerSM \\
Iters &= KernelBlks / KernBlksPerIter \\
AvgBlkTime &= KernelTime / Iters
\end{aligned}
\tag{4.12}
$$

Based on the block trace on each SM, the overall kernel time is the runtime of the most time-consuming SM, see Equation 4.13.

$$
\begin{aligned}
SM\_Time(K_i) &= BLK\_Start(K_i) - BLK\_End(K_i) \\
Kernel\_Time(K_i) &= MAX\ \{SM\_Time(K_i)\}
\end{aligned}
\tag{4.13}
$$

For multiple kernel instances, concurrent kernel execution on the GPU can be modeled based on the *AvgBlkExe* pattern. In Figure 4.5, we assume that a maximum of three blocks can be allocated per SM for Kernels A and B, due to resource constraints. Kernels A and B have 8 and 5

blocks, respectively. Kernel A is dispatched ahead of Kernel B, whose average block execution time is half of the Kernel B's. It takes two iterations for Kernel A to execute all blocks. During the second iteration, each SM has enough resources to host two additional blocks from Kernel B. On SM-0, as soon as BLK 6 from Kernel A ends, BLK4 (which is waiting) from Kernel B can start immediately due to the availability of resources.

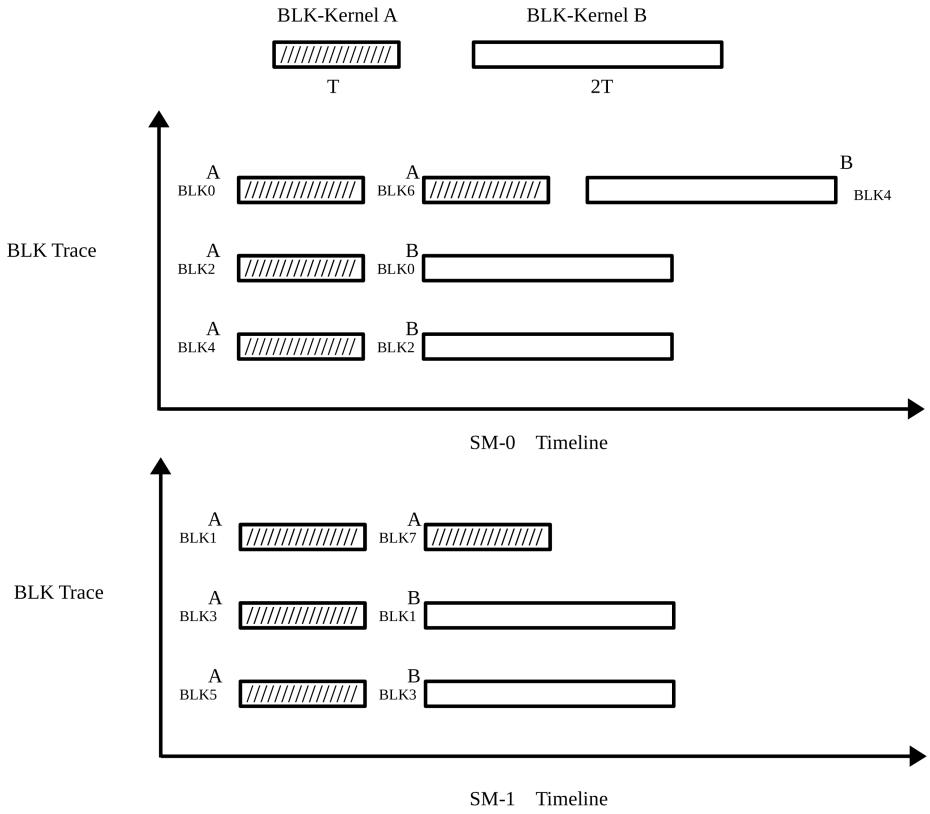


Figure 4.5: The average block execution pattern for two kernels on a GPU with two streaming multiprocessors.

4.2.3 Resource Contention Model

As multiple kernels share the GPU simultaneously, blocks from different kernels some interference will be generated between blocks [97, 31]. If blocks contend for the same resource, the block execution time of the current kernel will probably be extended. To quantify the interference factor, we apply the *Max Method* to the performance metrics shown in Table 4.7.

For a performance metric i, the corresponding contention factor C_i is the sum of the metric utilization from all concurrent kernels. If the combined utilization is less than 1, C_i becomes 1, as

Table 4.7: Performance metrics used for contention nalysis.

Metrics Name	Resource Type	Metrics Name	Resource Type
eligible_warps_per_cycle	occupancy	dram_utilization	memory
cf_fu_utilization	function unit	tex_utilization	memory
tex_fu_utilization	function unit	shared_utilization	memory
ldst_fu_utilization	function unit	l2_utilization	memory
single_precision_fu_utilization	function unit	sysmem_utilization	memory
double_precision_fu_utilization	function unit	flop_sp_efficiency	compute
special_fu_utilization	function unit	flop_dp_efficiency	compute

shown in Equation 4.14.

$$C_i = (\sum_{j=0}^{kern} Util_i^j > 1)? \sum_{j=0}^{kern} Util_i^j : 1 \tag{4.14}$$

For each resource type, the maximum utilization across all metrics is used as the contention ratio, since that resource will end up being the limiting factor, as expressed in Equation 4.15.

$$C_{type} = MAX \left\{ C_i^{type} \right\} \tag{4.15}$$

The overall contention factor of running multiple concurrent kernels $C^{'}$ is the max contention ratio among all the resource types C_{type}.

$$C^{'} = MAX \left\{ C_{occupancy}, C_{fu}, C_{mem}, C_{cmp} \right\} \tag{4.16}$$

For concurrent blocks from different kernels, the average block execution time is adjusted using Equation 4.17.

$$AvgBlkTime^{'}(K_i) = AvgBlkTime(K_i) \times C^{'} \tag{4.17}$$

4.2.4 Stream Launch

Initializing a CUDA stream when leveraging CKE comes with some cost. As more and more streams will be present in the scheduling queue, the launch overhead will increase accordingly. Equation 4.18 describes our stream launch model.

$$StreamStart_i = StreamStart_{i-1} + StreamLaunchOvhd \tag{4.18}$$

On NVIDIA GPUs, the cost of issuing a CUDA call is around 5-10μs[94]. On our experimental platform, the measured *StreamLaunchOvhd* is 7μs. Figure 4.7 compares our model

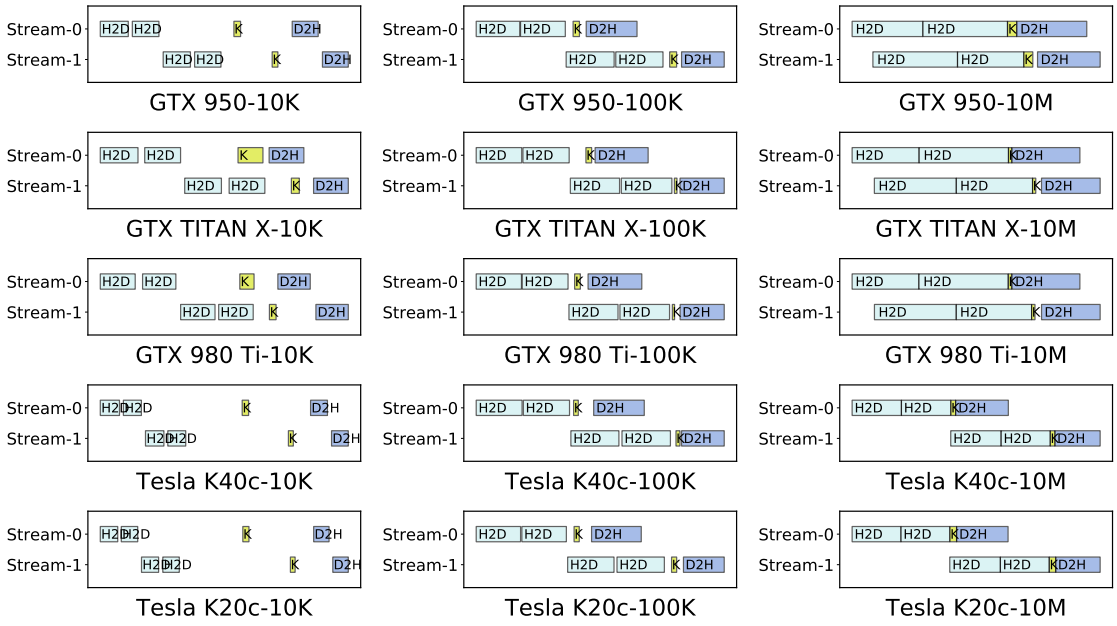


Figure 4.6: Overlapped data transfers using two CUDA streams, where the transfer size includes 10K, 100K and 10M floats. Three gaming GPUs (GTX 950, TITAN X and GTX 980 Ti) and two computing GPUs (Tesla K40c and K20c) are benchmarked.

prediction with the actual stream launch overhead. As the number of streams grows larger than 6, the model prediction tends to overestimate the overhead.

On the GTX 950, there are two copy engines: i) host-to-device (H2D) and ii) device-to-host (H2D). Previously, we assumed that the host-to-device transfers between CUDA streams have no overlap since there was only one H2D engine. However, this assumption only applies to the Tesla series for NVIDIA GPUs. As shown in Figure 4.6, there is no H2D overlap for the 10K and 100K cases on the gaming GPUs. But for the 10M case, the H2D transfers for two concurrent CUDA streams conflict with each other. The specific starting point for the observed H2D overlap, *H2D_Ovlp_Threshold*, is quantified through benchmarking. If the transfer size is larger than the threshold, the start time will be adjusted accordingly, as captured in Equation 4.19. The scheme for launching a CUDA stream is described in Algorithm 2.

$$StreamStart_i = StreamStart_{i-1} + \textit{H2D\_Ovlp\_Threshold} \tag{4.19}$$

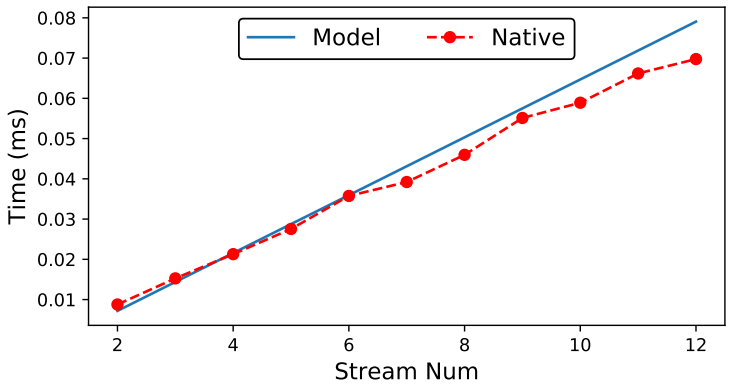


Figure 4.7: Comparison of the stream launch model prediction with actual timings on the GTX 950.

Algorithm 2 Stream Launch Model

Initialize CUDA calls for the first stream S_1
for Next Stream S_i **do** ▷ i = 2 : StreamNum
  Check the first call type with previous stream S_{i-1}
  if H2D for S_{i-1}, H2D for S_i **then**
    Run *H2D_Ovlp_Threshold* analysis
  else if H2D for S_{i-1}, Kernel for S_i **then**
    Apply $StreamStart_i$
  else if Kernel for S_{i-1}, H2D for S_i **then**
    Apply $StreamStart_i$
  else if Kernel for S_{i-1}, Kernel for S_i **then**
    Run *AvgBlkExe* to determine $StreamStart_i$
  end if
  Update the following calls in the stream
end for

4.2.5 Model Consolidation

Leveraging the aforementioned models, concurrent kernel performance can be predcited based on Algorithm 3. In our model, there are three states for each CUDA call: i) *sleep*, ii) *active* and iii) *done*. *Moka* utilizes a trace table to keep track of all the CUDA calls. Once a CUDA call is selected, it transitions from the *sleep* state to the *active* state. Then the algorithm returns back to the *done* state once it finishes.

Given multiple CUDA streams, we can customize their dispatched order. Then the *Stream Launch Model* is applied to initialize the starting point. The following CUDA calls in the current stream will have an offset based on the updated starting point. After all CUDA calls are configured in the trace table, we sort them based on their start time. *Moka* goes through each call in the sorted trace table and checks for the possibility of leveraging concurrency. The data transfer model is applied to quantify transfer contention, whereas the block execution and contention models are used to quantify kernel concurrency. After all the CUDA calls are completed, the modeling for concurrent kernel execution ends. The timing trace can be obtained from the trace table.

Algorithm 3 Modeling Concurrent Kernel Execution

```
Initialize CUDA call for each stream
Initialize trace table based on Stream Launch Model
Sort trace table by the starting time of CUDA calls
while not all CUDA calls are done do
    wake a call from sleep
    check concurrency among the wake list
    if concurrent data transfer then
        Apply Data Transfer Model
    else if concurrent kernel execution then
        Apply AvgBlkExe and Contention Model
    end if
    Update wake list and trace table
end while
```

4.3 Evaluation Platform

To carry out our evaluation of CKE on an NVIDIA the same GTX 950 GPU as used earlier, which is based on the Maxwell architecture. On the Maxwell, each streaming multiprocessor (SM) has 128 cores and 4 warp schedulers, as shown in Figure 4.8. Each warp scheduler issues instructions on 32 CUDA cores. Shared memory becomes independent from the L1 cache, and a unified L1 and texture cache is engineered in the SM design.

The GTX 950 is equipped with two copy engines and supports concurrent copy and kernel execution. We use the CUDA 8.0 driver version and CUDA 8.0 runtime version in our experiments. A more detailed list of features supported on the GTX 950 is provided in Table 4.8.

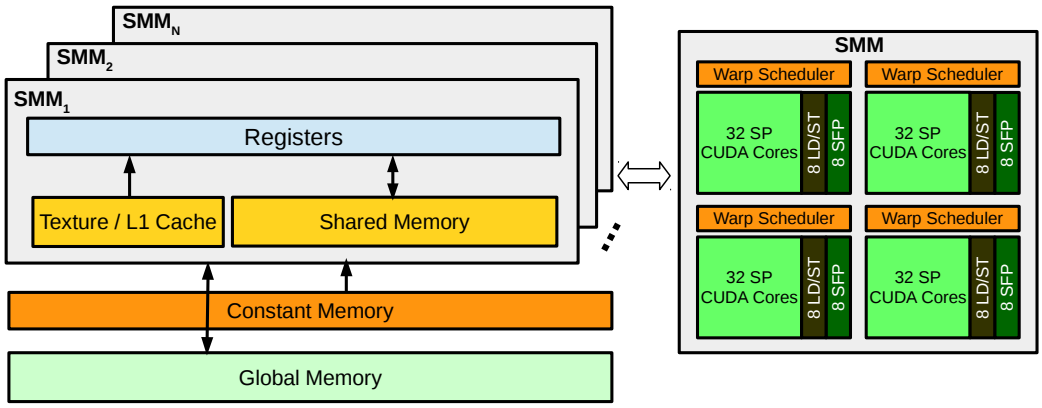


Figure 4.8: The NVIDIA's Maxwell GPU architecture.

Table 4.8: GTX 950 Specifications

CUDA Cores	768	**Global Memory**	1997 MB
Multiprocessors	6	**L2 Cache**	1 MB
CUDA Cores / SMM	128	**Constant Memory**	64 KB
Core Clock	1393 MHz	**Shared Memory per Block**	48 KB
Memory Clock	3305 Mhz	**Register per Block**	65536
Memory Bus Width	128-bit	**Integrated GPU sharing Host Memory**	No
Copy engines	2	**Unified Virtual Addressing**	Yes

4.4 Evaluation on Real-world Applications

4.4.1 Monte Carlo eXtreme (MCX)

MCX is a GPU-accelerated Monte Carlo method for modeling light propagation inside complex media such as human tissue [26]. The MCX kernel is classified as compute-intensive since the memory-to-compute ratio (*m2c_ratio*) is 0.27. Given that the kernel runs in 1-D, all of the 1-D compute-intensive kernels in Table 4.4 are included for block size tuning. If we record the compute-intensive performance counters in Table 4.6, we record 14 samples across the 7 different features, which are input for t-SNE analysis. These samples are then mapped to a 2-D embedded space using t-SNE, where principal component analysis is applied for initialization. After mapping the sample data to a lower dimension space, the three most similar kernels (*blacks, dwt, interval*) are identified, as shown in Figure 4.10. For *blacks* and *interval*, their best block size is 32, whereas *dwt's* best block size is 128 threads per block. Based on a majority vote, a block size of 32 is selected for MCX. Compared to the default block size of 64, either configuration will produce the best performance.

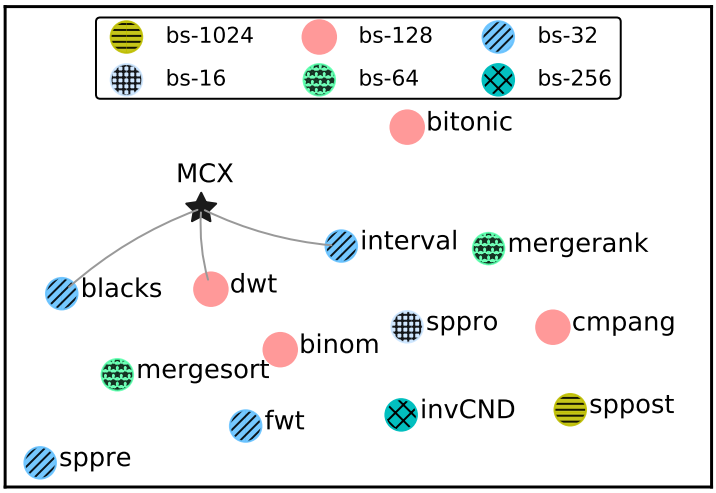


Figure 4.9: Block size tuning for MCX using t-SNE.

Next, we apply *Moka* to predict the concurrent kernel execution performance for MCX. Figure 4.10 shows that *Moka* can accurately capture the kernel's execution runtime, producing estimates within 5% of the actual runtime.

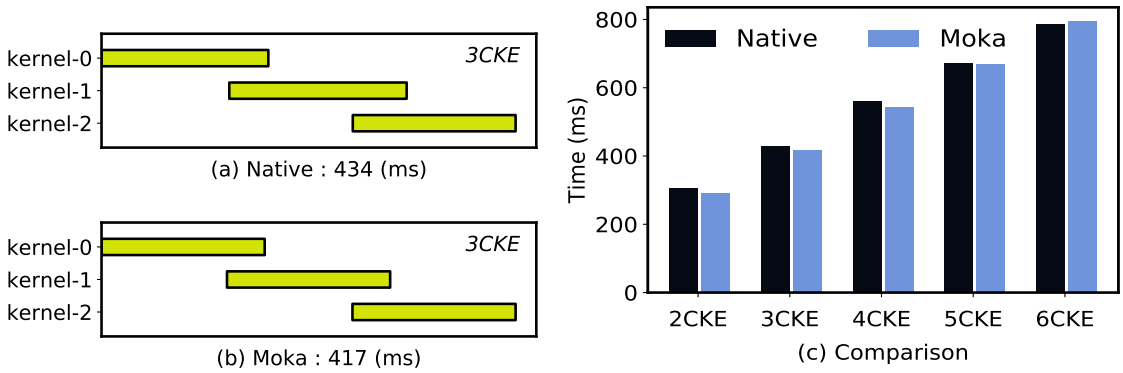


Figure 4.10: CKE performance prediction using *Moka* for MCX.

4.4.2 Hidden Markov Model (HMM)

Hidden Markov Model is a popular machine learning algorithm used in speech recognition [112]. We utilize concurrent kernel execution in the Expectation-Maximization stage to accelerate the recognition speed. During each iterative training step, the mean and covariance of the multivariate Gaussian density for each hidden state need to be updated. Since the update to each state is independent, CKE can be applied to improve performance. There are two kernels (*em_gammaobs* and *em_expectmu*) that run in a 2-D grid, each possessing different memory-to-compute intensity ratios, as reported in Table 4.9. For either the memory-intensive or compute-intensive group, we use the corresponding 2-D training kernels in Tables 4.4 and 4.5. Applying t-SNE analysis, a block size of 64 is recommended for *em_gammaobs* by *tranCoalesced* and *tranNaive*, whereas a block size of 256 is suggested for running *em_expectmu* by all 3 of the nearest kernels. If we use an 8×8 grid for *em_gammaobs*, and a 16×16 grid for *em_expectmu*, we achieve the best performance on the GTX 950.

Table 4.9: Characteristics of the HMM kernels.

Memory-intensive	**Kernel Dims**	**c2m_ratio**
em_gammaobs	2D	0.45
Compute-intensive	**Kernel Dims**	**m2c_ratio**
em_expectmu	2D	0.92

For HMM kernels, *Moka* can predict runtime performance within 12% of the actual runtime, as shown in Figure 4.12. When evaluating the concurrent execution pattern, *Moka* tends to overpredict the runtime. This is because the aforementioned *StreamLaunchOvhd* introduces a fixed interval between the CUDA calls, whereas the actual overhead can vary. For the small kernels

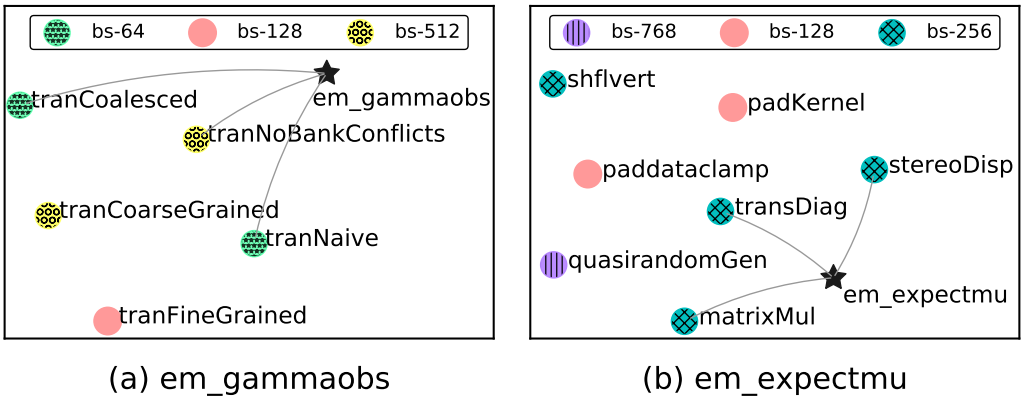


Figure 4.11: Block size tuning for HMM kernels using t-SNE.

present in the HMM application, this fixed launch overhead leads to a longer predicted runtime.

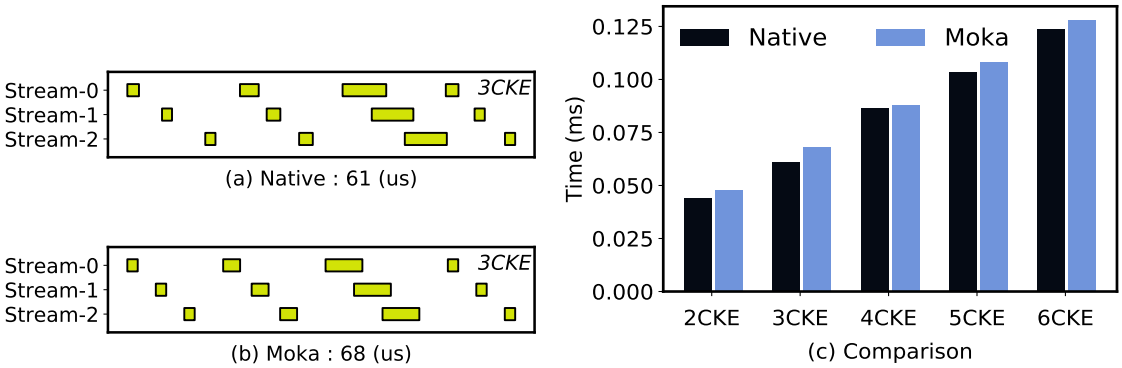


Figure 4.12: CKE performance prediction using *Moka* for HMM.

4.4.3 Workload Scheduling

Given the performance estimation capabilities of *Moka* for concurrent kernel execution, we can also use *Moka* to guide workload scheduling by adjusting the stream launch order. Here, we simulate the concurrent execution of three GPU applications in a shared context, which includes *vectorAdd* (V) and *matrixMul* (M) from the CUDA SDK, and *pathfinder* (P) from the Rodinia benchmark suite. The batch coding style (as mentioned in Chapter 2.2.3) is applied, and pinned memory is used to replace the default pageable memory.

The native and *Moka* predictions are plotted for each launch combination in Figure 4.13. For VMP, there is no contention for data transfer and kernel execution. A significant degree of

concurrency between compute-intensive (*matrixMul*) and memory-intensive (*pathfinder*) kernels is observed. For *VPM*, the actual H2D starting point for *matrixMul* occurs earlier than the model predicted. The H2D contention between *pathfinder* and *matrixMul* is also observed. *Moka* tends to overestimate the contention for data transfers, see *PVM* and *PMV*. For *MVP*, the concurrent kernel execution between *matrixMul* and *vectroAdd* is hidden by the H2D transfer of *pathfinder*. H2D contention is also observed for *MPV*.

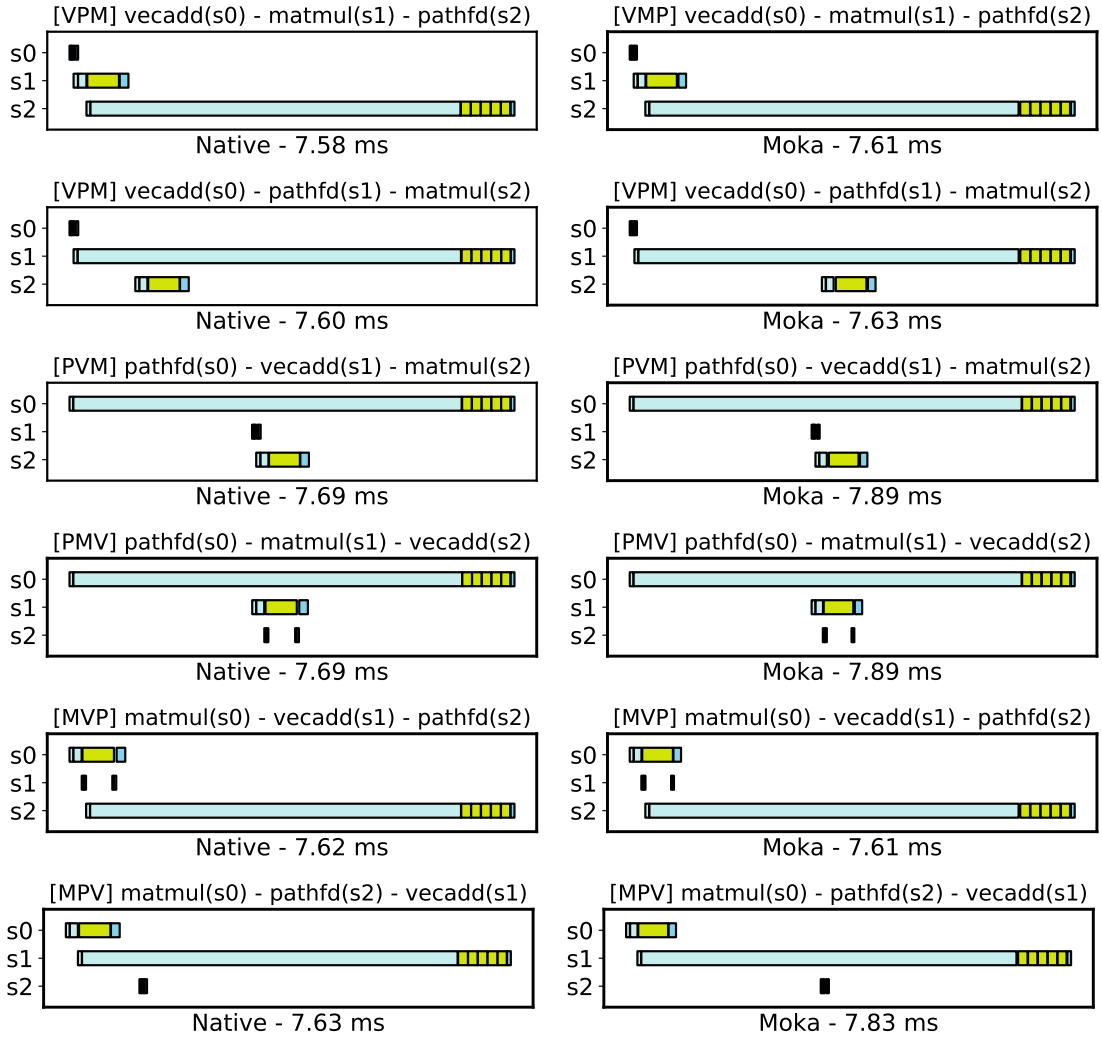


Figure 4.13: Modeling concurrent kernel execution of *vector addition*, *matrix multiplication* and *pathfinder* using Moka. Prediction results on six different combinations of launch order are illustrated.

According to the prediction, *Moka* will recommend both *VMP* and *MVP* as the best launch order. On the native device, *VPM* performs best. One of the shortcomings of *Moka* is that linear

modeling schemes are applied to capture dynamic runtime behavior. The performance difference between *MVP*, as suggested by *Moka*, and the best *VPM* is less than 1%. In all, *Moka* can capture the CKE performance trend and provides a highly accurate recommendation for workload scheduling.

4.5 Summary

In this chapter, we proposed an empirical model for concurrent kernel execution on the NVIDIA Maxwell GPU architecture. The major contributions of our modeling scheme are to tune GPU kernels by predicting the best block size and estimate CKE performance of multi-kernel applications using profiled performance counters. Our proposed CKE performance model *Moka* uses average block execution to model kernel execution and includes multiple performance factors to estimate the impact of resource competition on the device.

Our evaluation shows that the average estimation error is within 12% versus the measured runtime, with a geometric mean less than 5%. For multi-kernel applications, our model can suggest a close-to-optimal solution to drive dispatching orders for concurrent kernel execution. We believe that our model provides a convenient tool for GPU programmers to estimate the performance ceiling of concurrent kernel execution on modern GPUs.

Chapter 5

Interference-aware scheduling for GPU workloads

As modern GPUs increase device resources with every generation, the computational and memory resources may not be fully utilized by a single application. With the introduction of concurrent kernel execution, applications can be co-located and co-scheduled on the same GPU, significantly improving resource utilization. However, contention for shared resources, such as memory bandwidth and the computational pipeline, results in interference and often leads to performance degradation for co-located GPU workloads. Scheduling workloads to minimize interference and satisfy quality-of-service requirements, becomes a challenging issue, especially for cloud-based GPUs.

If we try to run two applications concurrently without considering the underlying characteristics of the two workloads, the resulting interference is difficult to estimate. We need to characterize the applications if we want to minimize interference and improve system throughput. In Figure 1.5, we show the performance impact on 6 target GPU applications when co-executed with another application on an NVIDIA GTX 1080 Ti GPU. Tests include a total of 79 GPU applications including the 6 target applications. We can see that the concurrent kernel benchmark from the CUDA SDK [74] (*cuda_concurrKerns*) and the reduction application from the SHOC benchmark suite [21] (*shoc_reduction*) are more sensitive to interference than the other 4 applications, showing more that a 90% slowdown on average. We can see that each GPU application has its own pattern of sensitivity when co-executed with a second application. Thus, it is key to characterize and accurately predict the interference impact before launching the concurrent execution. An efficient GPU workload scheduler should be aware of the characteristics of queued workloads and select the appropriate pair for co-execution with the goal of avoiding interference and achieving the best overall throughput.

In this chapter, we present *Magic*, a machine learning based interference-aware scheduler for GPU workloads. *Magic* utilizes profiling metrics, provides automated feature extraction to characterize GPU workloads, and predicts the sensitivity in terms of potential interference for concurrently scheduled applications. In addition, we also add support for clusters equipped with multiple GPUs from different GPU generations and configurations. On a single GPU system, our proposed scheduler improves performance as compared to a first-come-first-serve policy by 16%, and achieves 10% better throughput than a state-of-art similarity-based scheduler. On a multi-GPU system, our proposed scheduler outperforms a least-loaded policy by 21%, and increases performance over a round-robin policy by 22%.

5.1 Magic Framework

The *Magic* framework implements interference-aware scheduling using two stages of analysis. The first stage involves workload analysis, where incoming workloads from the clients pass through a short profiling process. During this process, the workloads are profiled to obtain a set of prominent features to represent each workload. These targeted features are preprocessed offline using the proposed *Principal Feature Analysis* method. We refer to the set of selected features as the *Short Profiling* features. The second stage applies our interference prediction model to estimate the sensitivity in terms of the degree of interference for the targeted workload. Leveraging our predicted interference outcome and interference-aware scheduling policy, the scheduler rearranges the dispatch order for the workloads in the job queue, improving throughput while minimizing the impact of interference. An application's status and the GPU's status are constantly updated as soon as dispatching decisions are made.

5.2 Offline Analysis for Short Profiling

A GPU has a limited supply of resources to share across concurrent workloads. These resources include registers, shared memory and copy engines for data transfer. As multiple GPU workloads are streamed onto a device, contention for these shared resources become inevitable. Depending on the availability of the resource and the degree of contention, the performance impact for co-located workloads can be significant. Therefore, it is key to understand the resource requirements of each pending workload so the GPU workload scheduler can make better decisions when it comes to minimizing interference. In this section, we utilize *Principle Feature Analysis* to select features

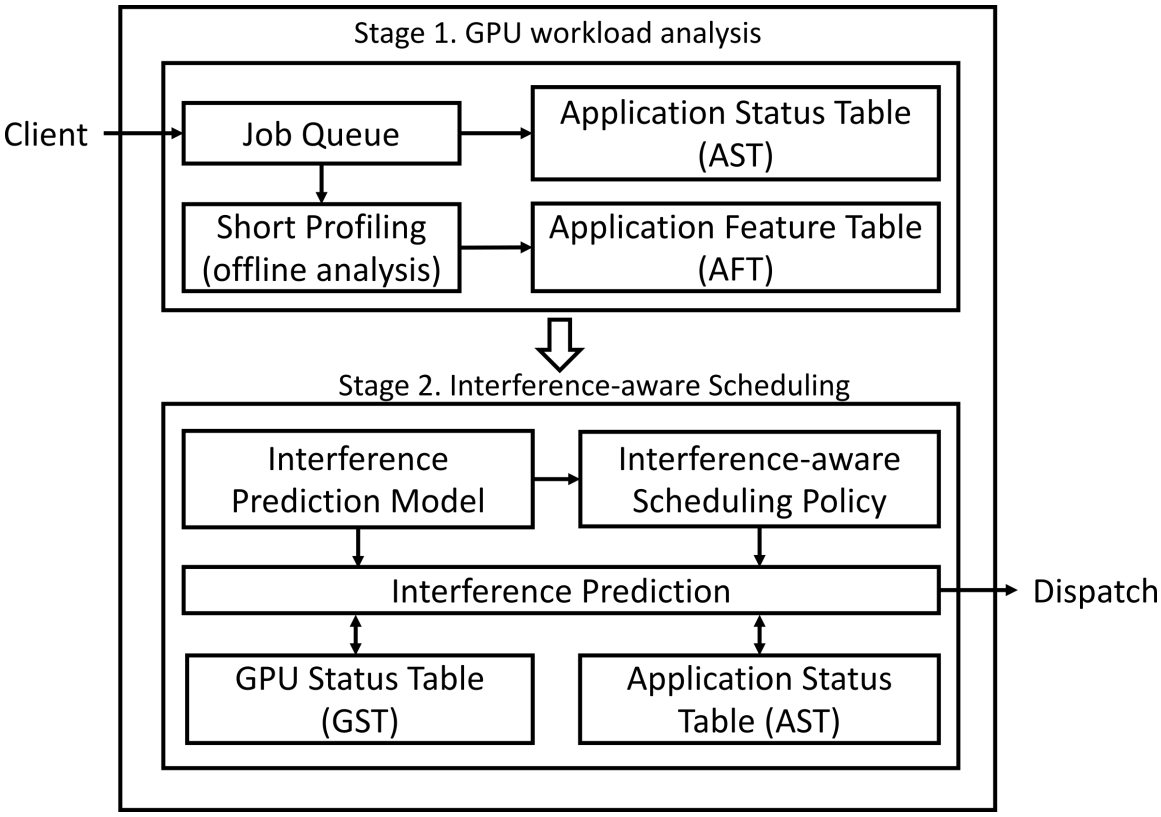


Figure 5.1: Magic framework workflow.

that can best characterize resource requirements of GPU workloads. We run a similarity-based scheduler to demonstrate the effectiveness of the chosen parameters.

5.2.1 Feature Selection Method

Profiling tools for GPU workloads, such as the NVIDIA command profiler *nvprof* [70] and AMD's *CodeXL*[9], profile a wide spectrum of performance counters. However, not all of the profiled metrics are useful for interference analysis. In this study, we have implemented a feature selection method to choose interference-related metrics, as shown in Figure 5.2.

To provide a rich data set for our analysis, we begin by profiling 56 compute applications from the NVIDIA CUDA SDK 8.0, where the OpenGL-related and long-running applications are excluded. The resulting profile metrics represent execution in a total of 287 distinct kernels that are present in these 56 GPU applications. For each kernel, there are 120 performance counter values collected using *nvprof* on an NVIDIA Pascal-based GTX 1080 Ti GPU [70]. In all, a feature matrix

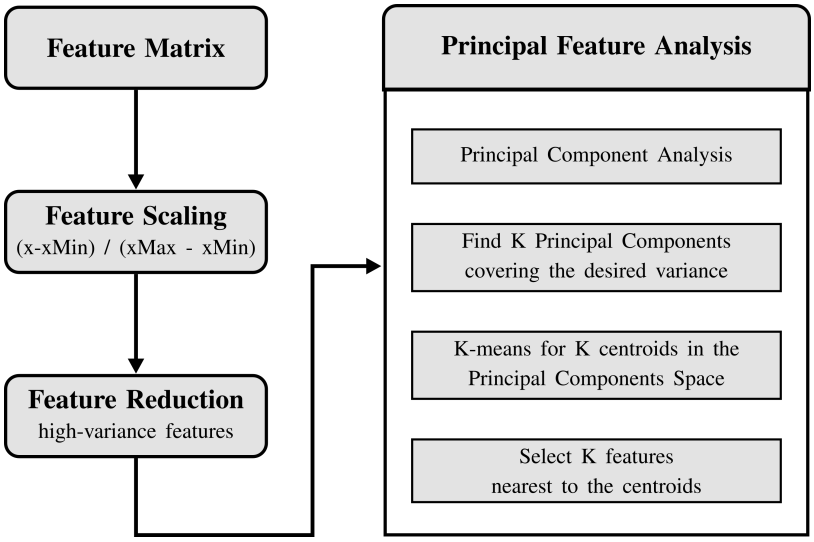


Figure 5.2: Feature selection method for offline analysis.

of 287 x 120 is generated. Next, *Feature Scaling* is applied to normalize these profiling metrics, where the min and max values are used to scale the data between 0 and 1. The MinMaxScaler is applied due to the fact that it is robust when working with features with very small standard deviations, and can preserve zero entries when working with sparse data [4]. For the *Feature Reduction* step, features with low variance are removed since they provide less useful information about a workload's characteristics versus features with higher variance. In our case, this step results in the original 120 performance metrics being reduced to 64. Note that the number of features after *Feature Reduction* may not be exactly 64, depending on the feature matrix and configurations used during the preprocessing steps. Furthermore, among the selected 64 features, *Principal Feature Analysis* (PFA) is used to identify a subset that can be representative for interference analysis.

5.2.2 Cases for Feature Selection

In this section, we explore whether a reduced set of features can perform as well as the full set of profiling features. Here, we label the case where we consider the entire set of performance counters reported by the NVIDIA profiler as *FeatAll*. We label the case where we used the 64 features after applying the *Feature Reduction* step as *Feat64*. Then, we set up a few more cases with learned features using PFA, varying the hand-picked variance coverage on *Feat64*, as shown in Table 5.1. Each feature set is labeled as *Feat*$\{N\}$, where N is in the range of 9-42 features. In addition, we also include the feature set used in the *Mystic* framework [97], labeled *FeatMystic*, in order to compare with the PFA selected features.

Table 5.1: Variance coverage of Feat64 by selecting different number of features using PCA.

Num. of Feature	Variance Coverage	Num. of Feature	Variance Coverage
42	99%	14	85%
26	95%	12	80%
18	90%	9	75%

Table 5.2: GPU applications for feature set evaluation.

GPU Applications	Features	GPU Applications	Features
interval	Recursion	MC_SingleAsianOptionP	Computational Finance
transpose	Linear Algebra		CURAND Library
matrixMul	Linear Algebra	convolutionFFT2D	Image Processing
scan	Data Parallel		CUFFT Library
reduction	Data Parallel	mergeSort	Sorting, Data Parallel
binomialOptions	Computational Finance	sortingNetworks	Sorting, Data Parallel
SobolQRNG	Computational Finance	radixSortThrust	Sorting, Data Parallel
quasirandomGenerator	Computational Finance		Thrust Library

5.2.3 Evaluation Method

To compare the benefits of using different feature sets, we use 13 GPU applications from the CUDA SDK, as shown in Table 5.2. This subset includes a range of domains, including linear algebra, computational finance, image processing and popular GPU libraries. Here, we use a state-of-art similarity-based approach, the same as used in the *Mystic* framework, to dispatch these workloads[97]. Similarity-based scheduling analyzes the resource usage patterns (i.e., similarities) among GPU applications and co-locates workloads with the least similar usage pattern to minimize the potential interference.

To measure the similarity, two popular distance metrics are compared: i.) *euclidean* distance, and ii.) *cosine* distance. The *euclidean* distance focuses on the magnitude of the data, whereas the *cosine* distance emphasizes the direction. After generating the pairwise distance matrix, Agglomerative Clustering is performed [23]. We compare 7 methods to compute the cluster distance, which include: *single*, *complete*, *average*, *weighted*, *centroid*, *median* and *ward* methods. To find the best combination of methods and metrics, we use the *Cophenetic Correlation Coefficient* to measure the relationship between the clustering result and the pairwise distance matrix [46]. The closer the coefficient is to 1, the better the clustering preserves the distance. After testing on the *FeatAll* case, which includes the entire set of profiling metrics, the *euclidean* metric for pairwise distance and the *centroid* method for the Agglomerative Clustering are chosen, where the highest Cophenetic Correlation is achieved. In this thesis, the similarity between GPU workloads is measured based on the *euclidean* distance.

We have visualized the clustering patterns using a dendrogram for each case, as shown in Figure 5.3. We can see that *FeatMystic* generates a distinct clustering pattern compared to the *FeatAll* case. For instance, two sorting applications, *mergeSort* and *sortingNetworks*, are more distant from each other using *FeatMystic* versus when using *FeatAll*. In addition, we also notice the clustering pattern changes when using fewer features. For instance, *Feat18* and *Feat14* show a dissimilar clustering structure.

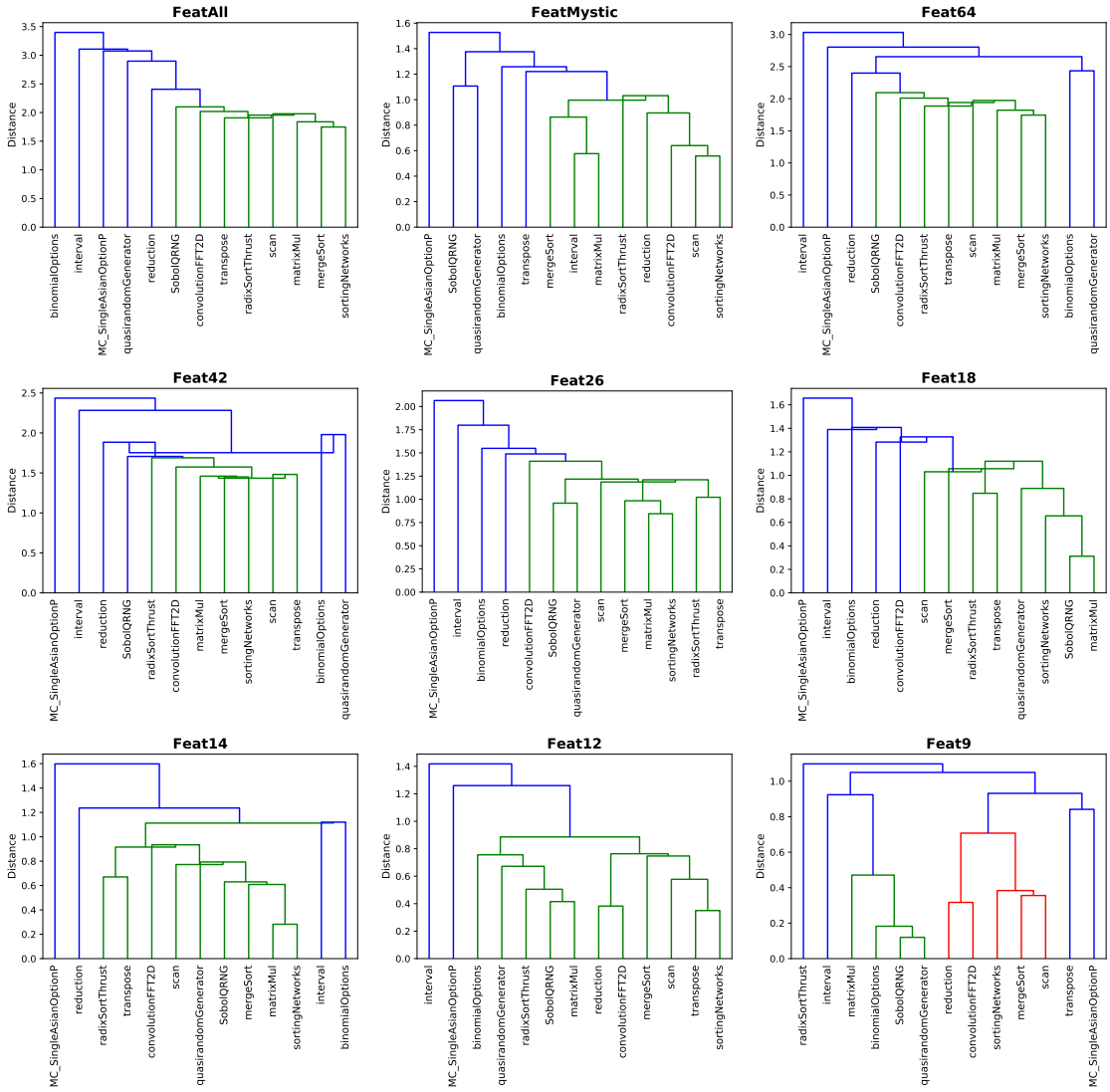


Figure 5.3: Dendrograms of 13 GPU applications using different feature sets. The distance option uses euclidean distance and the linkage option, applying the centroid method.

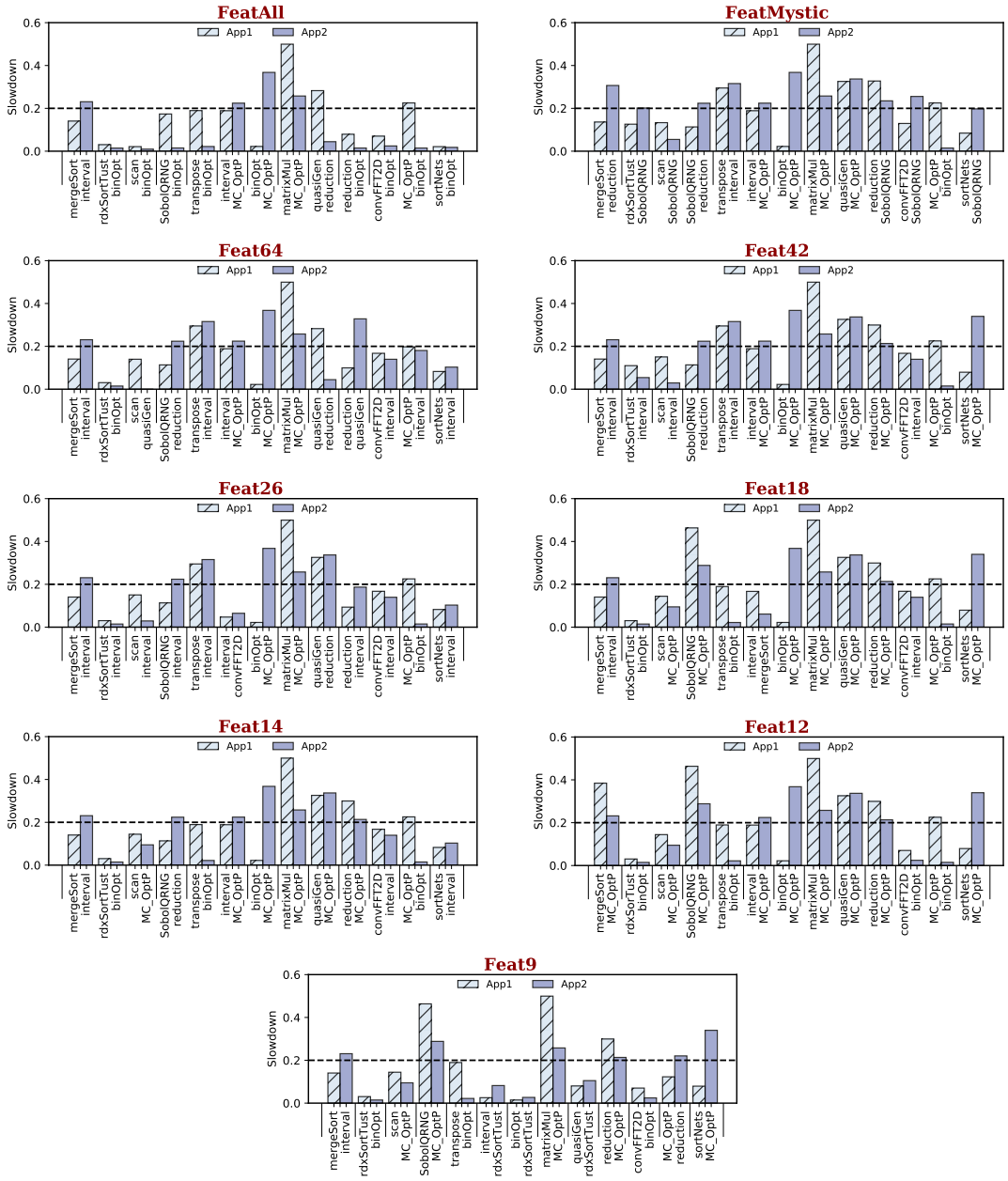


Figure 5.4: Performance impact when co-executing two GPU applications on a single NVIDIA GTX 1080Ti GPU. For each test, App2 is selected as the least similar application to co-run with App1. The dashed line shows the QoS. The higher the speedup, the lower the interference, and vice versa.

To dispatch workloads and obtain the elapsed time, an in-house scheduler implementation is used, as described in Section 5.6.1. The software allows us to launch independent processes at the same time. During our experiments, we select the least similar application (*App2*) to co-locate with the target application (*App1*). Therefore, there are 13 cases for each feature set, as shown in Figure 5.4.

To measure the performance, given that co-execution involves some imprecision in terms of when exactly an application starts execution, we report on the co-execution runtime (T_Corun) for each application (App_i) by taking the average of 100 runs for each test case. We also measure the dedicated execution runtime ($T\_Dedicate$) for each application. The performance slowdown is calculated using Equation (5.1).

$$Slowdown(App_i) = T\_Corun(App_i)/T\_Dedicate(App_i) - 1 \tag{5.1}$$

Here, we consider a 20% slowdown for co-located kernels as our Quality-of-Service (QoS) target, which is shown as the dashed line in Figure 5.4. *FeatAll* performs the best, where 7 out of 26 cases violate QoS, observing a 12% slowdown on average. *FeatMystic* produces 14 out of 26 cases below the QoS threshold, with an average slowdown of 22%. Surprisingly, *Feat9* achieves an average slowdown of 16%, where 9 out of 26 cases missed the 20% QoS target. When using fewer than 10% of the metrics from *FeatAll*, *Feat9* achieves a comparable performance, which ranks as the *second best* in terms of the average slowdown. In Table 5.3, we list the subset of 9 prominent features learned. These metrics provide coverage of several important device resources, including streaming multiprocessors, system memory, global memory, shared memory and cache.

Table 5.3: The Feat9 metrics identified using PFA.

Metric Name	**Resources**
warp_execution_efficiency	Streaming Multiprocessor
branch_efficiency	Streaming Multiprocessor
issue_slot_utilization	Streaming Multiprocessor
global_hit_rate	Global Memory
gst_transactions_per_request	Global Memory
tex_cache_hit_rate	Cache
local_store_transactions_per_request	Cache
shared_load_transactions	Shared Memory
sysmem_write_throughput	System Memory

Furthermore, we analyze the profiling overhead when using the full set of performance metrics (*FeatAll*), versus the case of only profiling a subset (*Feat9*). We observed that profiling 9 performance metrics can reduce the time spent on collecting all the performance counters by 1.1x-2.4x, as shown in Figure 5.5. We also found the profiling overhead reduction in *sdk_binominalOpt* is

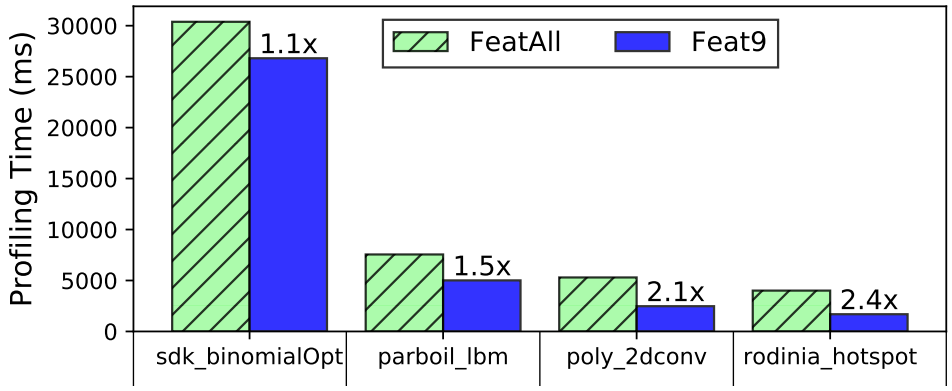


Figure 5.5: Compare the profiling overhead between analyzing all the performance counters (FeatAll) and a selected 9 performance counters (Feat9).

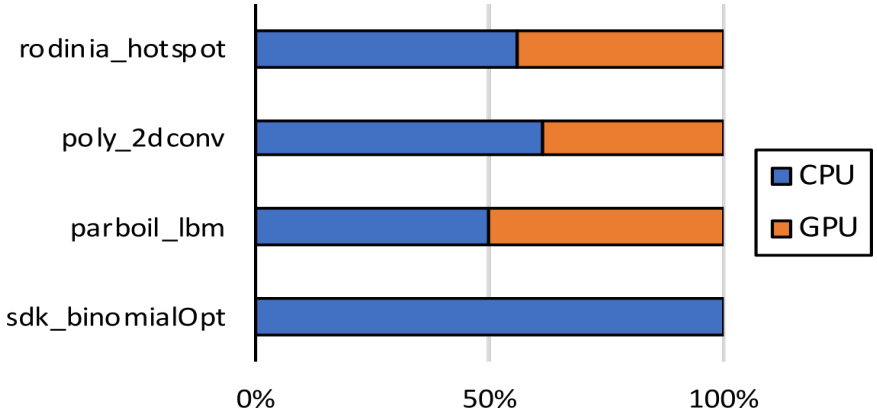


Figure 5.6: The runtime breakdown of CPU versus GPU execution for the selected applications.

much less than the other three applications. To understand what is causing this, for each application, we can look at the portion of time spent on the CPU versus the GPU, which is presented in Figure 5.6. For *sdk_binominalOpt*, CPU computation dominates the entire execution, thus only 10% of the profiling overhead is reduced. For applications which spend more time executing on the GPU, the runtime benefits of selecting to use shorter profiling runs is more significant. Note that the NVIDIA profiling tool replays kernels multiple times during the profiling process to collect the required metrics [70]. Depending on the portion of portion of time execution uses the GPU, the benefits of reduced profiling time can vary.

5.2.4 Other Observations

We have found that the feature metrics contained in *Feat9* subset can achieve a high level of QoS. However, there are still cases where the two least similar GPU applications do not run harmoniously. For instance, running *matrixMul* with *MC_OptP* achieves a 30% slowdown on average, which violates the targeted 20% threshold (see Figure 5.4). We consider that the similarity-based approach can provide a good estimate of the co-execution interference, while achieving a sub-optimal scheduling outcome. In the following section, we introduce our interference analysis to estimate the sensitivity level to the interference for each GPU application. We will target integrating our analysis model with the proposed scheduling policy to build an efficient interference-aware scheduler.

5.3 Interference Analysis

A similarity-based scheduler considers the relative distance in terms of resource usage of two kernels in order to estimate the potential interference. As shown in Figure 1.5, some GPU workloads (e.g., the concurrent kernel application from the CUDA SDK, and the reduction application from SHOC benchmark suite) are extremely sensitive to interference, so no matter what type of workload is co-run, the kernel will be impacted significantly. Therefore, rather than calculating the relative similarity, we consider how to characterize the sensitivity level to any interference. This factor could potentially guide the scheduler to make better decisions when dispatching GPU workloads so that the overall interference can be minimized. In this section, we will go through the process to characterize each workload's sensitivity level.

5.3.1 Methodology

To quantify the sensitively level of a GPU workload to interference, one approach is to build a regression model to predict the slowdown ratio. However, since the standard deviation of the slowdown ratio is high, according to the experiments in Figure 1.5, regression models could result in high variance in terms of prediction accuracy. In our analysis, we transform the regression task into a classification task by using two classes (i.e., sensitive and insensitive to interference) to label the targeted GPU workloads. If the average slowdown ratio is below a predefined threshold, the workload is considered to be insensitive to the co-running interference, and vice versa. Classifying incoming workloads into two distinct classes provides a better generalization of the sensitivity level of each kernel.

To build classification models, we use a total of 79 distinct workloads (see Section 5.6.3) for training and testing. To generate the ground truth for interference sensitivity, we measure the

slowdown for the targeted workload when running concurrently with a different workload. Each combination of two co-located workloads is run and timed 3 times. The best performance across the 3 runs is used to compute the performance (see Equation 5.1). The average slowdown ratio for the targeted workload is calculated using Equation 5.2, where K stands for the number of tests.

$$AvgSlowDownRatio(App_i) = \frac{1}{K}\sum_{test=0}^{K} Slowdown(App_i) \tag{5.2}$$

Using a predefined threshold $Thld$ for interference sensitivity, each GPU workload is assigned to a class using Equation 5.3, where i represents the workload, 1 is an interference-insensitive class and 0 is an interference-sensitive class.

$$InterferSensitivity_i = (AvgSlowDownRatio_i < Thld)?1 : 0 \tag{5.3}$$

So far we have built a dataset for our interference analysis, where each application is represented by a feature set and the classes (i.e., interference-sensitive or interference-insensitive) are labeled. Since our dataset is limited to 79 workloads, a stratified k-fold cross validation is used. The difference between a stratified k-fold and a traditional k-fold is that stratification preserves the percentage of samples of each class for the folds [2].

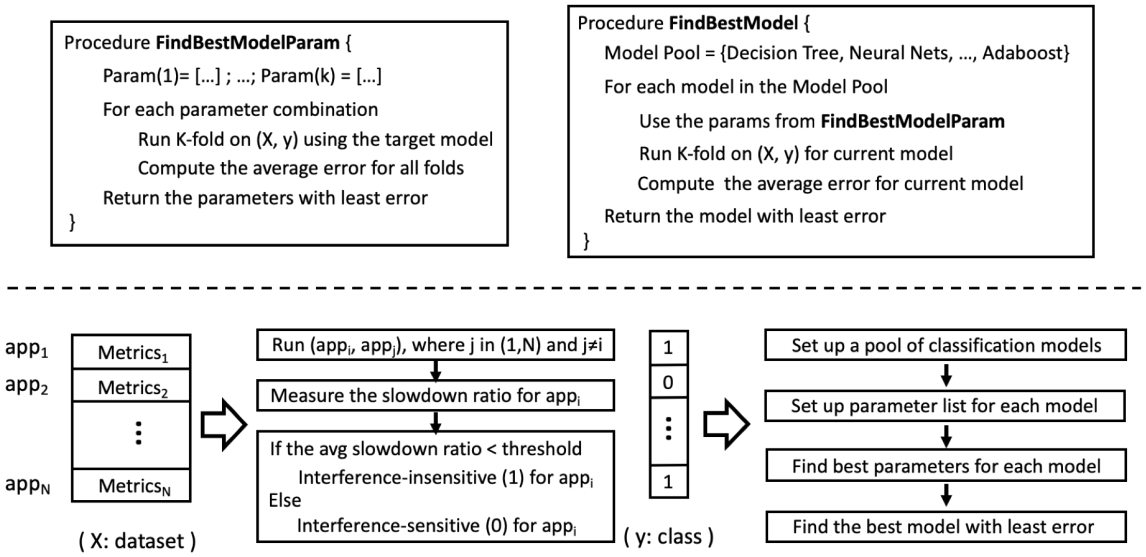


Figure 5.7: Interference analysis workflow.

Eight popular classification models are introduced to the model pool Decision Trees, K-Nearest Neighbor, Support Vector Machines, Random Forest, Neural Networks, Adaboost, Gaussian

Naive Bayes and Quadratic Discriminant Analysis [1]. Greedy search is applied to find the best configuration for each model (see ***FindBestModelParam*** in Figure 5.7). To identify the candidate, we choose the model that generates the lowest average error over all k folds. Next, we run k-fold cross validation again for all of the models in the model pool, using their best configurations (see ***FindBestModel*** in Figure 5.7). Finally, the model with the lowest error is selected for our interference-aware scheduler. The workflow for our interference analysis is summarized in Figure 5.7.

For our interference analysis, two separate feature sets are analyzed: 1) all the performance metrics (*FeatAll*), and 2) PFA selected metrics (*Feat9*). The reason to select *Feat9* is that it achieves a balance between quality during interference estimation and limited profiling overhead, as demonstrated in Section 5.2.3. Compared with using all of the profiling metrics, we want to see how well the reduced feature set (*Feat9*) can perform in terms of classification accuracy and scheduling outcome.

5.3.2 Interference-aware Scheduling Policy

Based on the interference characteristics of GPU workloads, an efficient scheduling policy is needed to guide scheduling of incoming GPU workloads to minimize interference and improve system throughput. To develop best practices for interference-aware scheduling, we conduct experiments on six distinctive GPU workloads mentioned at the beginning of current Chapter 5, which include concurrent kernel, scan, binomialOptions, reduction, 3mm and fdtd2d. We consider the concurrent kernel and reduction kernels belong to the interference-sensitive class, and the rest of the applications belong to the interference-insensitive class. Here, we seek to select the best job to be co-located with binomialOptions. ***Weighted Speedup*** is used to measure the system throughput, as described in Equation 5.4.

$$Weighted\ Speedup(App_i) = \sum_i Slowdown(App_i) \tag{5.4}$$

The performance of running different workloads with binomialOptions is presented in Figure 5.8. Note that binomialOptions belongs to the interference-insensitive class. We can see that co-locating two interference-insensitive jobs achieves a weighted speedup ranging from 1.81 to 1.98. Meanwhile, co-locating interference-sensitive applications (e.g., reduction and concurrent kernel) with the interference-insensitive applications (e.g., binomialOptions), results in an average weighted speedup of 1.46. Since co-locating interference-insensitive jobs achieves a better weighted speedup than co-locating interference-sensitive jobs, we consider prioritizing co-location of interference-insensitive applications for GPU multitasking in our proposed scheduling policy.

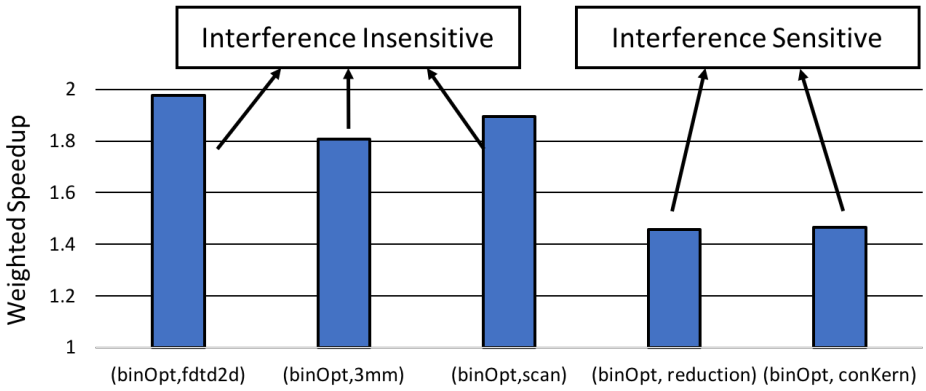


Figure 5.8: Performance measured by weighted speedup, when multitasking two GPU workloads on a GTX 1080 Ti GPU.

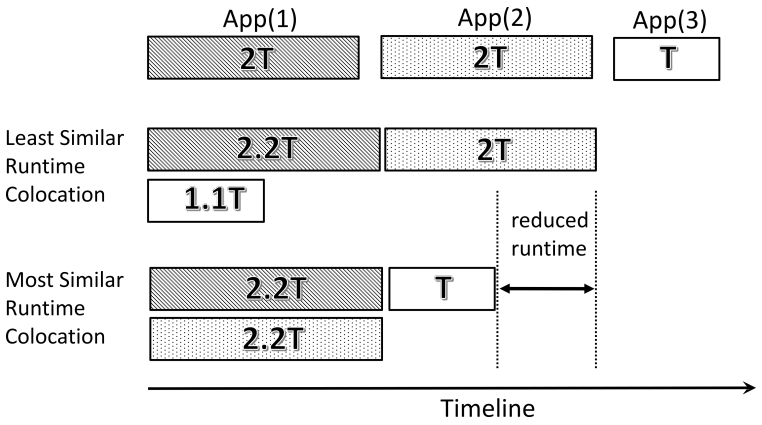


Figure 5.9: Performance comparison of selecting workloads with the least similar and least similar resource requirements for GPU multitasking, where we assume that a maximum of only two processes can run concurrently.

In addition to interference sensitivity, it is helpful to know how long a GPU workload may run in order to improve system throughput. In Figure 5.9, we consider how to schedule three workloads on the same GPU, where only two jobs maximum are allowed to run at the same time. Here, we assume the slowdown ratio for all 3 applications is 10%. In the case of co-locating jobs with the least similar runtime, the overall elapsed time is 4.2T, where App(1) is extended to 2.2T due to interference and App(2) remains 2T. On the other hand, by scheduling jobs with similar runtime requirements, the overall runtime is reduced to 3.2T, where the runtime of App(2) is hidden by App(1) and the overall runtime is reduced. Currently, we use the binary size of a workload as a coarse-grained estimate of the runtime. Instead of accurately predicting the runtime, we focus on the relative runtime ratios between jobs in our scheduler.

Overall, we have developed an interference-aware scheduling policy *InferBin*, as described in Algorithm 4. To begin, *InferBin* loads the interference prediction model. Then it obtains the feature metrics for the workloads in the waiting list and predicts whether a GPU workload is interference-sensitive or not using the model. For workloads belonging to either interference classes, we sort them according to their binary size. Similar to selecting jobs by the most similar runtime for GPU multitasking, we arrange the workload order by selecting the most similar binary size for jobs in both classes. Next, we place the sorted jobs belonging to the interference-insensitive class ahead of jobs belonging to the interference-sensitive class, which is recommended according to the previous analysis (see Figure 5.8). *InferBin* starts scheduling the rearranged workloads in the job queue until the job queue is empty. It keeps monitoring the status of actively running jobs. When the number of concurrently running jobs reaches the limit, *InferBin* will wait until one of the active jobs ends before dispatching the next job.

Algorithm 4 The InferBin scheduling policy.

```
1:  procedure INFERBIN_SCHEDULE( )
2:      load the interference prediction model
3:      obtain the feature metrics for waiting jobs
4:      predict the interference sensitivity for waiting jobs
5:      sort interference-insensitive jobs by the binary size
6:      sort interference-sensitive jobs by the binary size
7:      modify job order using most similiar runtime method
8:      prioritize interference-insensitive jobs in the queue
9:      while job queue is not empty do
10:         if activeJobs < maxCoRun then
11:             dispatch next job
12:         else
13:             spin for an available slot to corun
14:             dispatch next job
15:         end if
16:     end while
17: end procedure
```

5.4 Support for Heterogeneous GPU clusters

The interference analysis presented earlier is specifically tailored for homogeneous GPUs (multiple identical GPUs). To tune GPU sharing on heterogeneous GPU clusters, where GPUs are present representing different microarchitectures or from different device generations, the *Magic* framework would need to run interference analysis once for each different GPU microarchitecture/-generation. As shown in Figure 5.2, the performance for the target workload on different GPU devices could vary significantly. For instance, the gaussian kernels run on a GTX 760 executes 1.8x faster than the same implementation on a GTX 950. For the three memory-intensive applications (backprop, bfs and gaussian) in Rodinia benchmark suite [15], the speedup using GTX 760 over GTX 950 ranges from 0.9x to 1.8x. For the three compute-intensive applications (lavaMD, dtw2d and heartwall), the speedup achieved using a GTX 760 can vary wildly as compared to execution on a GTX 950, ranging from 0.6x to 1.4x. Performance will depend upon both the resource requirements of the workload (e.g., the memory and compute intensity) and the microarchitecture of the targeted GPU devices (e.g., the number of computing cores, the memory bandwidth and latency, the number of registers and the shared memory size) could impact the performance. Due to the complex interaction between a workload and targeted device microarchitectures, it becomes quite challenging to predict which device will produce the best performance for the GPU workload.

In previous work, analytical models [41, 10, 56], empirical models [115, 110] and machine learning models [53, 105] have been used to predict GPU performance. These models are designed to estimate the runtime performance on real hardware. Distinct from these prior approaches, we study the relative performance benefits among all the GPU devices using neural networks. We utilize a trained neural network model to estimate the probability of producing the best performance and select the candidate with the highest probability.

To train the neural network model, we use the metrics generated from the NVIDIA command line profiler [71]. Two metrics sets, *FeatAll* and *Feat9*, are included in our analysis so that the benefits of using the reduced feature sets for interference-aware scheduler can be evaluated. GPU workloads from six popular open source benchmark suites are studied (described in Chapter 5.6.3). For each workload, we run on real (versus simulated) GPU hardware to obtain a ground truth execution.

To identify the best neural network model structure for the modeling task at hand [5], we explored the sensitivity for the hidden layer sizes, activation functions, optimization solvers and L2 regularization penalty. For the hidden layer sizes, we tested 3 fully-connected hidden layers, where each hidden layer can have 30 / 60 / 100 neurons. Therefore, we have 3 different hidden layer structures (30,30,30), (60,60,60) and (100,100,100). In our experiments, we found that 3 hidden

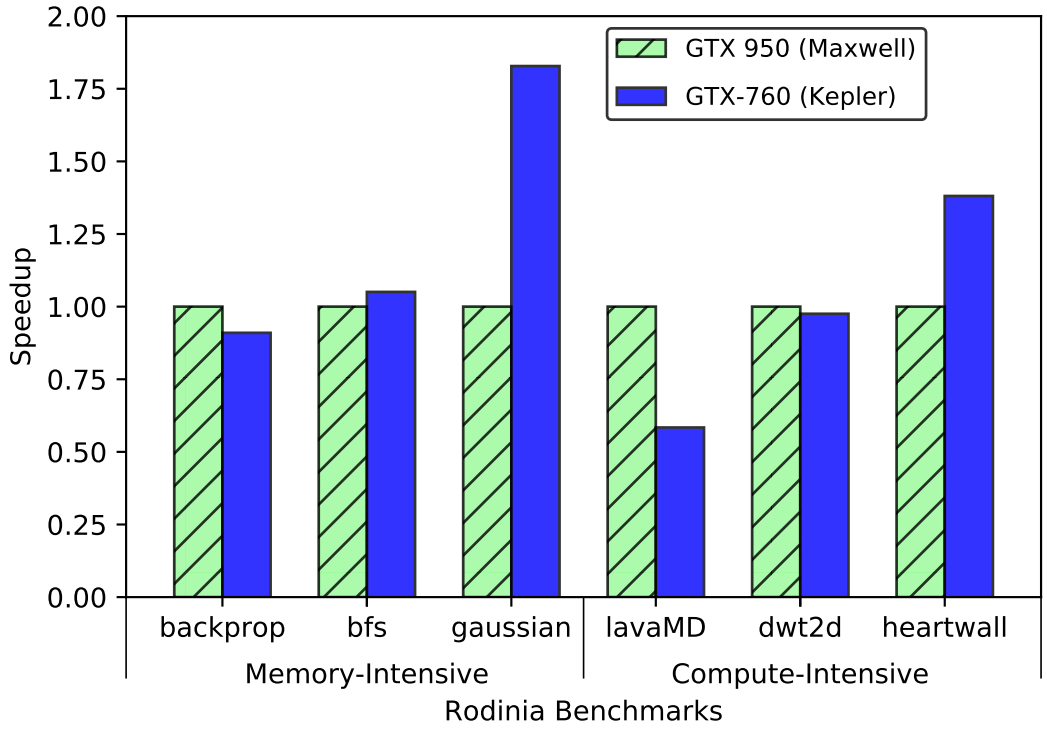


Figure 5.10: Relative performance comparing a GTX 760 and a GTX 950 for six Rodinia GPU benchmarks. The GTX 950 performance is used as the baseline, which is shown as the 1x speedup.

layers generate good prediction accuracy without increasing the computational complexity. For activation functions, we compare results for *identify* (no activation), *logistic*(the logistic sigmoid function), *tanh* (the hyperbolic tan function) and *relu* (the rectified linear unit function). We also take different optimizing solvers into consideration, which include *sgd*, *adam* and *lbfgs*. We consider four different values for the L2 regularization parameter (1/0.1/0.01/0.001). In all, our search space comprises 144 combinations of these four model parameters.

For each combination, we perform a stratified k-fold cross validation for training and testing, which can maintain the sample ratio for each class among the folds [2]. For example, given 5-fold cross validation, there will be 80% samples for training and 20% samples for testing. When splitting the total data set, random shuffling is enabled. After testing, the average error is recorded for each combination. We identify the combination of model parameters that produce the lowest error rate for predicting the fastest device/workload pair.

5.5 Magic Scheduling Policy

Our *Magic* framework schedules GPU workloads according to the proposed interference-aware analysis for both homogeneous and heterogeneous GPU clusters. For homogeneous GPU clusters, since GPUs have identical configurations, the offline interference analysis only needs to be run once to develop a prediction model. To dipatch a GPU workload, we first identify the least-loaded GPU in the cluster using the *GPU Status Table* (see Figure 5.1). For the selected GPU, the *InferBin* scheduling (see Algorithm 4) is applied, where we first run our interference prediction model to analyze the interference sensitivity of the workload. The dispatching sequence in the queue is reordered by prioritizing the interference-insensitive workloads and sorting them according to job size. For heterogeneous GPU clusters, a trained neural network model is used to predict the fastest device for each workload. Each GPU has its own job queue, where job dispatching order is maintained using the *InferBin* scheduling policy. Due to imbalance across GPUs in terms of the number of jobs and job runtime, some GPUs may finish earlier than others. Thus, work stealing is employed to increase cluster utilization. The mechanics of the *Magic* scheduling policy is presented in Algorithm 5.

5.6 Experimental Setup

5.6.1 Implementation of Magic Framework

We implement the *Magic* framework in Python. As illustrated in Figure 5.11, communication between clients and the server is handled through the INET socket using IPv4 as the TCP/IP protocol. As soon as the server receives a client's message, it collects a short profile of the workload and enters the workload into the job queue. Guided by the scheduling policy, the scheduler decides which job to be dispatched. A Python *multiprocessing* package [83] is used to launch jobs, where each job is encapsulated in an independent process. A cluster status table is created to monitor workload activity on the GPU. The job status table is maintained to keep track of the job id, GPU id, job status (dispatched or done), start time and end time. The job status table can be saved and is used to evaluate system performance. Users can define the maximum number of co-located jobs to explore the multitasking capabilities of the GPU.

5.6.2 Platform

We evaluate *Magic* using a platform equipped with a NVIDIA GTX 1080Ti GPU. This Pascal-based GPU has 3586 CUDA cores and 28 streaming multiprocessors, and supports concurrent

Algorithm 5 Magic Scheduling Policy for GPU Clusters

```
1:  procedure MAGIC_SCHEDULE( )
2:      if GPU cluster is homogeneous then
3:          select the least loaded GPU
4:          apply InferBin_Schedule() on the selected GPU
5:      end if
6:      if GPU cluster is heterogeneous then
7:          load trained neural network model
8:          predict the best GPU for waiting jobs
9:          set up the job queue for each GPU
10:         sort the interference-insensitive and interference-sensitive jobs by the binary size
11:         prioritize interference-insensitive jobs for each job queue
12:         while job queue is not empty do
13:             select the least loaded GPU considering the maxCoRun constraint
14:             if queue is not empty for the selected GPU then
15:                 dispatch next job in the queue
16:             else
17:                 steal jobs from the next least loaded GPU
18:                 dispatch the new job
19:             end if
20:         end while
21:     end if
22: end procedure
```

kernel execution and concurrent copy operations. The details of our software and hardware evaluation system are summarized in Table 5.4. Persistent mode is turned on during our experiments, allowing the driver to remain loaded, even when there are no clients running [78]. The number of concurrent compute and copy engine connections is set to 8, as the default [3].

Table 5.4: Platform specifications for a single GPU system.

Hardware	Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz
	NVIDIA GeForce GTX 1080Ti (Pascal Architecture)
Software	Ubuntu 14.04 x86_64 with kernel 4.4.0-87-generic
	CUDA Driver Version: 375.88

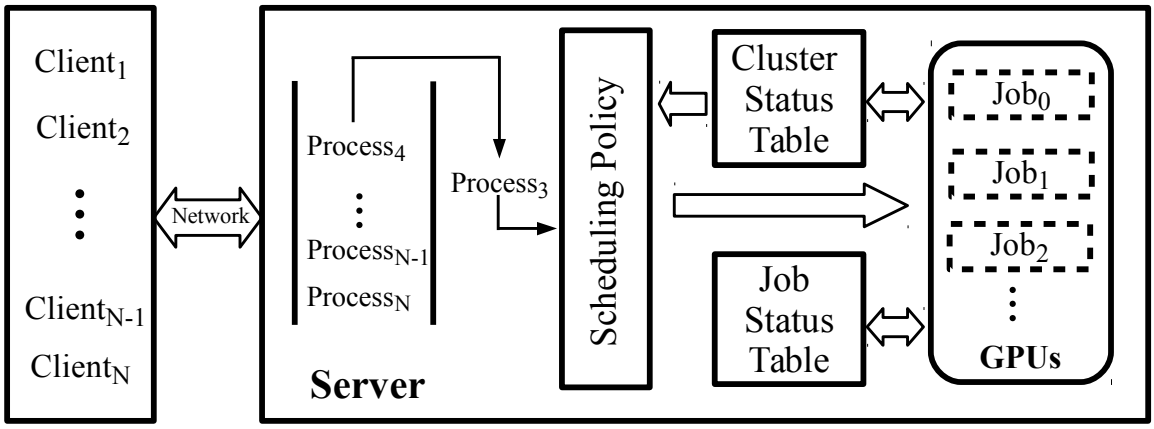


Figure 5.11: Implementation of the Magic framework.

For multi-GPU systems, we use a platform installed with two NVIDIA GPUs: 1) a Kepler-based GTX 760 and 2) a Maxwell-based GTX 950. Both GPUs have 6 streaming multiprocessors (SM). The GTX 760 has 192 CUDA cores per SM, with a total of 1152 CUDA cores. The GTX 950 has 128 CUDA cores per SM, with a total of 768 CUDA cores. Both GPUs support concurrent kernel execution. The GTX 760 has 1 copy engine, whereas the GTX 950 has 2 copy engines. Persistent mode is also turned on during all experiments. The details of the heterogeneous GPU system are described in Table 5.5.

Table 5.5: Platform specifications for a multi-GPU system.

Hardware	Intel(R) Core(TM) i7-4790K CPU @ 4GHz
	NVIDIA GeForce GTX 760 Ti (Kepler Architecture)
	NVIDIA GeForce GTX 950 Ti (Maxwell Architecture)
Software	Ubuntu 16.04 x86_64 with kernel 4.4.0-124-generic
	CUDA Driver Version: 367.48

5.6.3 Workloads

For our analysis and performance evaluation of *Magic*, we use 79 GPU applications selected from the CUDA SDK, Lonestar, Parboil, PolyBench, Rodinia and SHOC benchmark suites. The CUDA SDK includes applications from a broad range of fields, including finance, image processing, and linear algebra [74]. Lonestar includes several irregular parallel GPU applications [55]. Parboil covers throughput-oriented applications, including image processing, biomolecular simulation, fluid dynamics and astronomy [92]. Polybench/GPU is a collection of GPU-based benchmarks targeted for linear algebra, data-mining and stencil computations [32]. Rodinia is a benchmark suite developed to

evaluate heterogeneous platforms [15]. SHOC is designed to test the performance and stability of a heterogeneous computing system [21]. Among these 79 workloads, 24% run in less than 1 second, 43% take 1 to 2 seconds to complete, 21% take 2 to 5 seconds to complete, and 3% take 5 to 10 seconds to complete. 9% run longer than 10 seconds.

5.6.4 Evaluation Metrics

We use two popular system-level metrics for performance evaluation, proposed by Eyerman and Eeckhout, to evaluate multiprogram workloads [25].

System Throughput (STP) measures the work completed per unit of time, with values ranging between 1 to N, as described in Equation 5.5. STP is a the-higher-the-better metric.

$$STP = \sum_{i=1}^{N} \frac{T\_Dedicate_i}{T\_Corun_i} \tag{5.5}$$

Average Normalized Turnaround Time (ANTT) measures the arithmetic average of the turn-around time slowdown due to multitasking, as described in Equation 5.6. ANTT is a the-lower-the-better metric.

$$ANTT = \frac{1}{N} \sum_{i=1}^{N} \frac{T\_Corun_i}{T\_Dedicate_i} \tag{5.6}$$

T_Corun_i is the runtime for i^{th} application when it is co-scheduled with other applications. $T\_Dedicate_i$ is the runtime for i^{th} application when it is the only application executing on the GPU. N stands for the total number of applications.

5.6.5 Scheduling Policies

For the single GPU system, three scheduling policies are evaluated: 1) a *first-come-first-serve* (fcfs) policy, 2) a *similarity-based* policy and 3) our proposed *InferBin* policy. The *first-come-first-serve* policy schedules workloads based on the order in which they arrive. The *similarity-based* policy schedules workloads with distinctive resource utilization patterns. For the *similarity-based* policy, *FeatAll* is used to calculate the similarity distance. For the *InferBin* policy, both *FeatAll* and *Feat9* are used for comparison. Given the number of scheduling policies and feature sets developed in this thesis, we evaluate four different schedulers: 1) fcfs, 2) sim-FeatAll, 3) InferBin-Feat9, and 4) InferBin-FeatAll.

For the multi-GPU system, we also studied three scheduling policies: 1) the *least-loaded* (ll) policy, 2) the *round-robin* (rr) policy and 3) the proposed *InferBin* policy. For the *least-loaded* policy, a least-loaded GPU is selected for queuing jobs. For the *round-robin* policy, workloads take turns to be scheduled on each GPU in the round-robin fashion. When the total number of co-running workloads reaches the limit, the scheuler will wait until one of the running jobs completes. Both the *FeatAll* and *Feat9* metrics are used for the *InferBin* policy. For performance comparison, we have four schedulers to evaluate: 1) ll, 2) rr, 3) InferBin-Feat9, and 4) InferBin-FeatAll. Next, we compare the performance of these schedulers.

5.7 Evaluation of the Benefits of Magic

While the 79 workloads can be dispatched in 1 of 79! different sequence, we select 3 random launch sequences to evaluate different schedulers. We explore the multitasking capabilities of the GPU by varying the maximum number of co-located running jobs (*MaxCoRun*) from 2 to 8. For the single-GPU system, workloads are executed on a NVIDIA GTX 1080 Ti GPU. For the heterogeneous multi-GPU system, we use two NVIDIA GPUs (GTX 750 and GTX 950) from Kepler and Maxwell generations, respectively.

5.7.1 Single-GPU System

According to the interference analysis presented in Section 5.3, we need to train the interference prediction model based on a pool of different machine learning models. We have identified the best models for *FeatAll* and *Feat9* feature sets, where an Adaboost model is used for *FeatAll* and a K-Nearest Neighbor model is used for *Feat9*. Both models are able to achieve better than a 70% prediction accuracy on average.

The total elapsed time of running all the jobs using four different schedulers (*fcfs*, *sim-FeatAll*, *InferBin-FeatAll*, and *InferBin-Feat9*) are shown in Figure 5.12. For the case where the *MaxCoRun* is 2, *sim-FeatAll* and *InferBin-FeatAll* did not outperform *fcfs*, except for *InferBin-Feat9*, which outperformed *fcfs* by 6%. For the case where *MaxCoRun* is 4, *InferBin-FeatAll* and *InferBin-Feat9* clearly outperforms *fcfs* by 10% and 15%, respectively. For the case where *MaxCoRun* is 6, *sim-FeatAll*, *InferBin-FeatAll* and *InferBin-Feat9* outperform *fcfs* by 14%, 18% and 22%, respectively. For the case where *MaxCoRun* is 8, the performance of *fcfs* was outpaced by the other three schedulers by more than 17% on average. We have observed that both simliarity-based and our proposed *InferBin* policies perform well when there are more than 4 jobs allowed to run concurrently

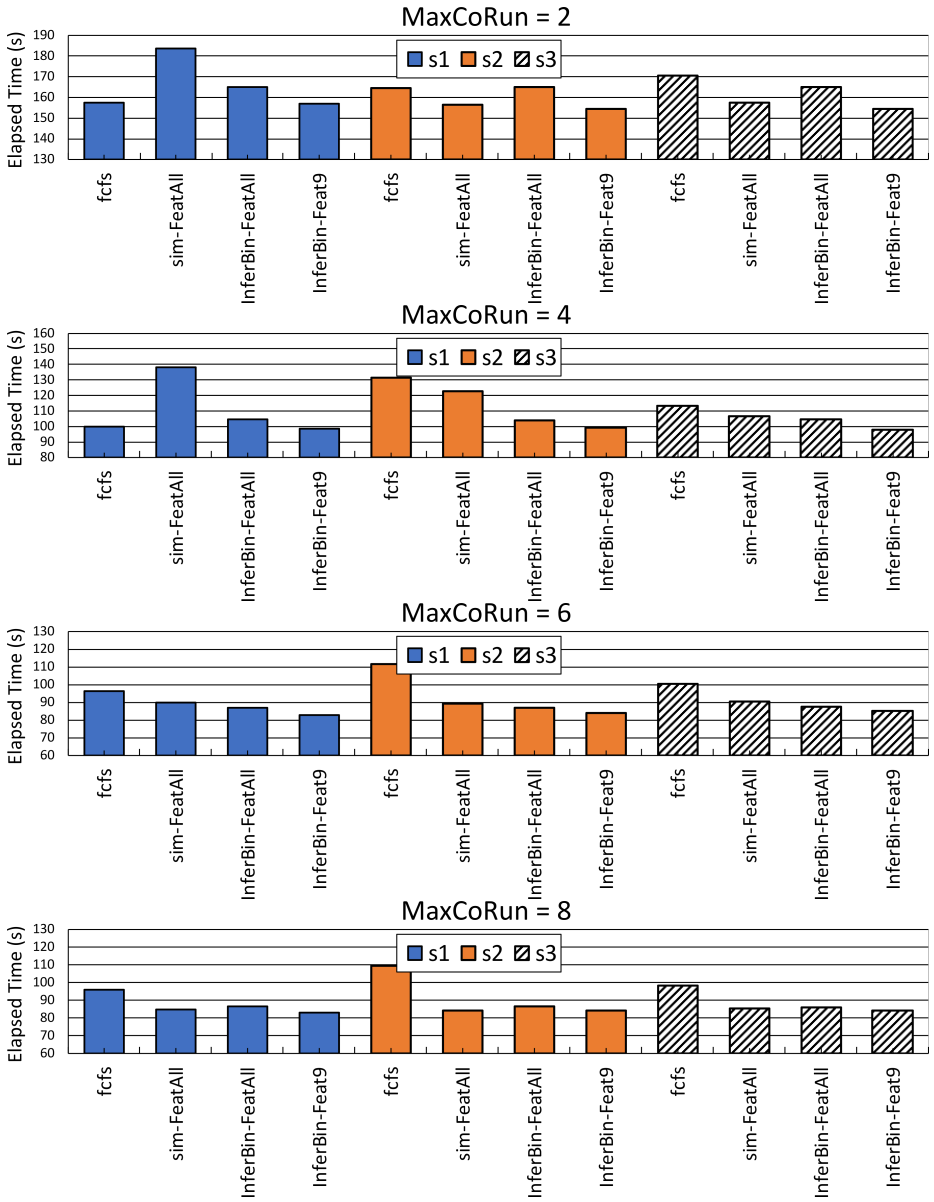


Figure 5.12: Performance of four schedulers (fcfs, sim-FeatAll, InferBin-FeatAll and -Feat9) for three different application launch sequences (s1/s2/s3), where the maximum number of co-located jobs on the GTX 1080 Ti GPU was increased from 2 to 8.

on the GPU. In all, *InferBin-Feat9* performed the best. It outperformed *fcfs* and *sim-FeatAll* by 16% and 10% on average, respectively.

The System Throughput (STP) and Average Normalized Turnaround Time (ANTT) are presented in Figures 5.13 and 5.14. Our proposed *InferBin-Feat9* scheduler improves STP by 10%-36% for *fcfs*, 5%-37% for *sim-FeatAll* and 2%-9% for *InferBin-FeatAll*. Meanwhile, *InferBin-Feat9*

reduced ANTT by 13%-56% for *fcfs*, 6%-68% for *sim-FeatAll* and 6%-17% for *InferBin-FeatAll*.

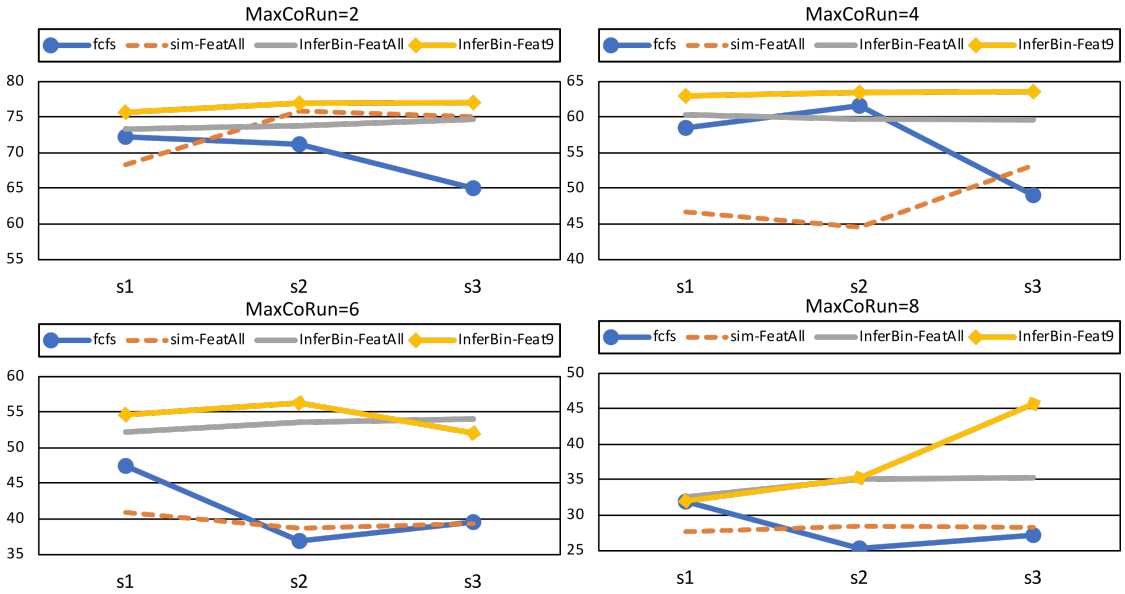


Figure 5.13: The system throughput using different schedulers.

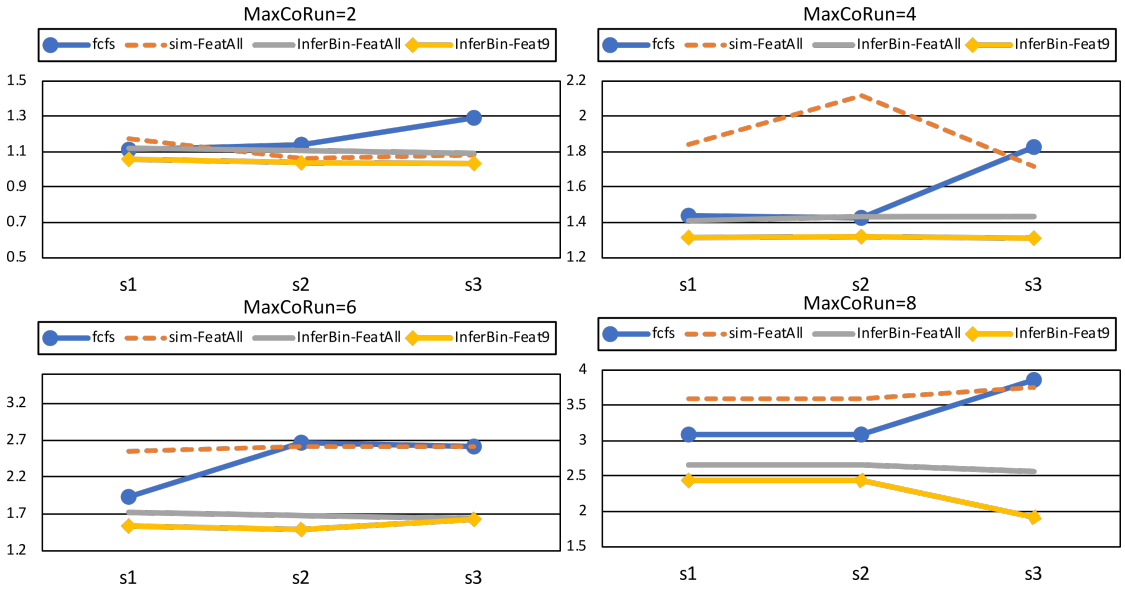


Figure 5.14: The Average Normalized Turn-around Time using different schedulers.

5.7.2 Multi-GPU System

In Section 5.4, we proposed using a neural network model to predict the best device to run for the GPU workloads in a multi-GPU system. After defining the model parameter search space and training the model with stratified 5-fold cross validation, our finalized neural network models for *FeatAll* and *Feat9* are able to achieve better than a 97% prediction accuracy on average.

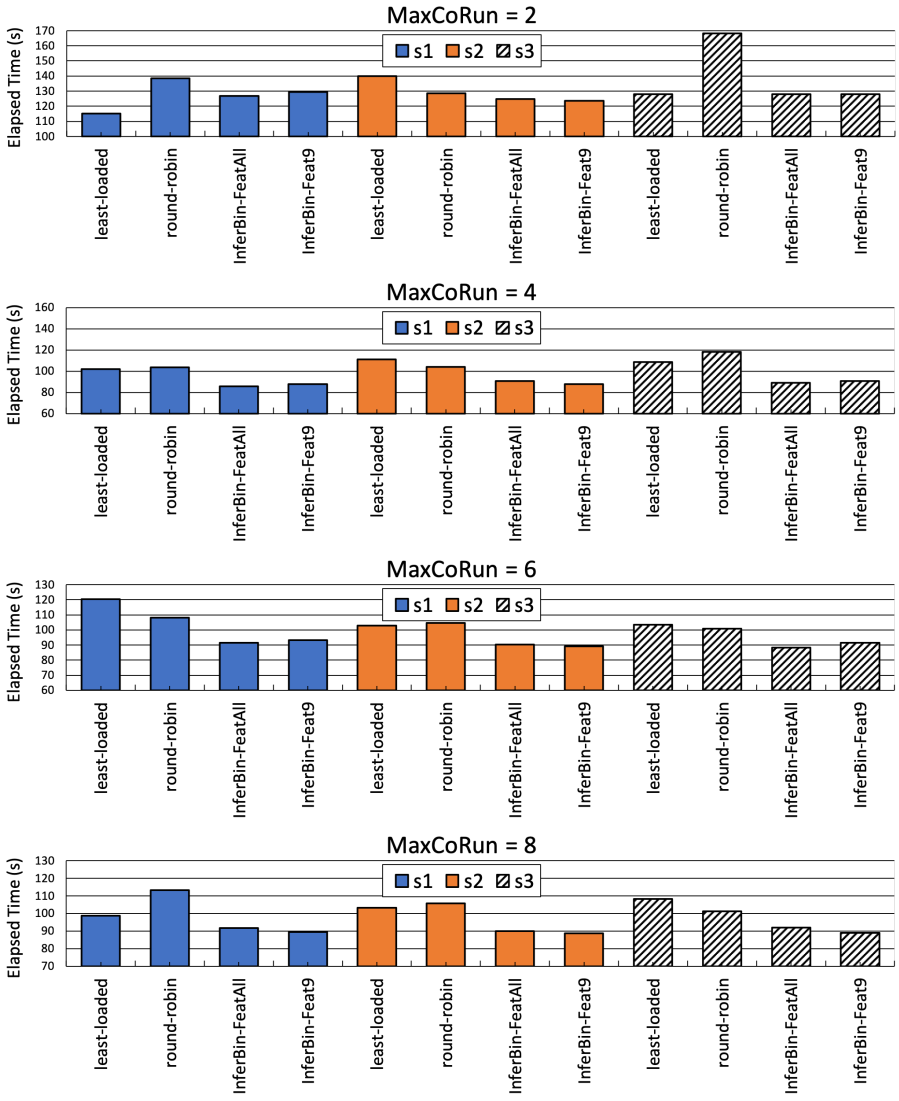


Figure 5.15: Performance of four schedulers (least-loaded, round-robin, InferBin-FeatAll and -Feat9) for three different application launch sequences (s1/s2/s3), where the maximum number of co-located jobs on two GPUs (GTX 760 and GTX 950) was increased from 2 to 8.

The total elapsed time of running all the jobs using four different schedulers (*least-loaded*,

round-robin, *InferBin-FeatAll*, *InferBin-Feat9*) is presented in Figure 5.15. For the case where the *MaxCoRun* is 2, *InferBin-FeatAll* and *InferBin-Feat9* perform as good as *least-loaded*. Compared to the *round-robin* policy, *InferBin-FeatAll* and *InferBin-Feat9* achieve a 15% and 14% average improvement, respectively. For the case where *MaxCoRun* is 4, *InferBin-FeatAll* and *InferBin-Feat9* outperform *least-loaded* by 21%. On average, *InferBin-FeatAll* and *InferBin-Feat9* perform 22% better than *round-robin*. For the case where *MaxCoRun* is 6, *InferBin-FeatAll* and *InferBin-Feat9* outperform the *least-loaded* policy by 21% and 19%, respectively. *InferBin-FeatAll* achieves a 15% performance improvement over *round-robin*, whereas *InferBin-Feat9* obtains a 14% improvement. For the case where *MaxCoRun* is 8, *InferBin-FeatAll* and *InferBin-Feat9* outpace the *least-loaded* policy by 13% and 16%, respectively. Similarly, the performance of *round-robin* was outperformed by *InferBin-FeatAll* and *InferBin-Feat9*, by more than 17%.

We have noticed that our proposed *InferBin* policies perform best when maximum concurrent running jobs (*MaxCoRun*) is set at 4. As we increase *MaxCoRun*, the performance benefits of adopting our *InferBin* policies drops. Since the CPU is a 4-core Intel chip with the hyper-threading enabled, running more than 8 threads could introduce more contention on the CPU resources. This could lead to the observation that when more than 8 jobs are enabled for concurrent execution on the 2-GPU system, the overall performance does not improve. In all, *InferBin-Feat9* can perform as good as *InferBin-FeatAll*. Our proposed *InferBin* policies have outperformed the *least-loaded* and *round-robin* policies by 14% and 18% on average, respectively.

The System Throughput (STP) and Average Normalized Turnaround Time (ANTT) are presented in Figure 5.16 and Figure 5.17. On average, the proposed *InferBin-FeatAll* and *InferBin-Feat9* schedulers improve STP by 36% for *least-loaded* and 24% for *round-robin*. Overall, *InferBin-FeatAll* can reduced ANTT by 1.8x for *least-loaded* and 1.5x for *round-robin*, whereas *InferBin-Feat9* can reduce ANTT by 2.6x for *least-loaded* and 1.7x for *round-robin*.

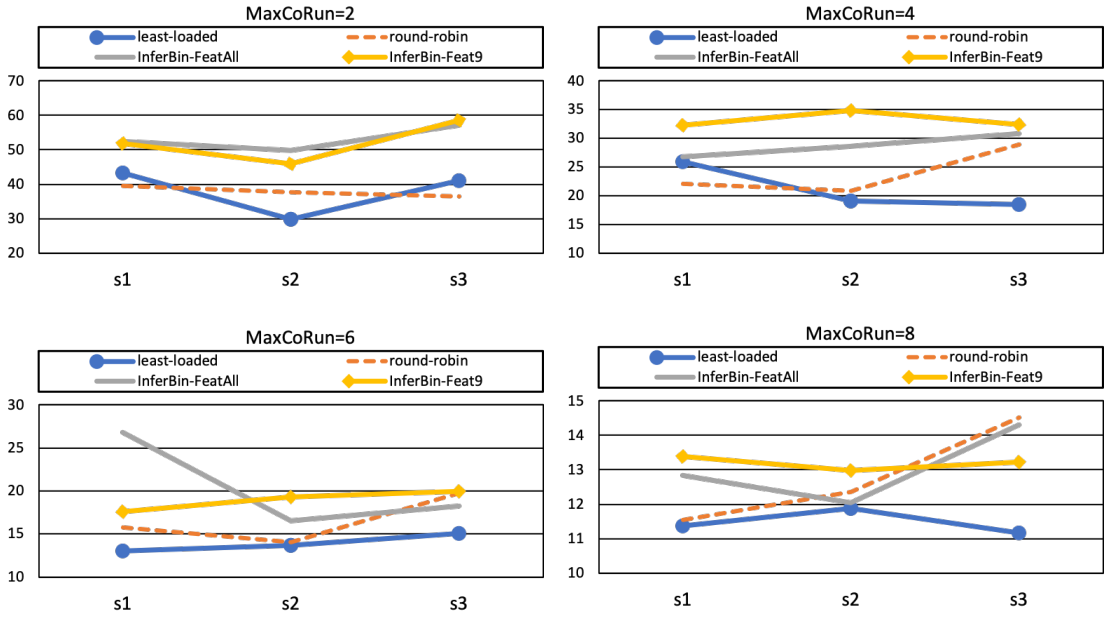


Figure 5.16: The system throughput using different schedulers on a 2-GPU (GTX 760 and GTX 950) system.

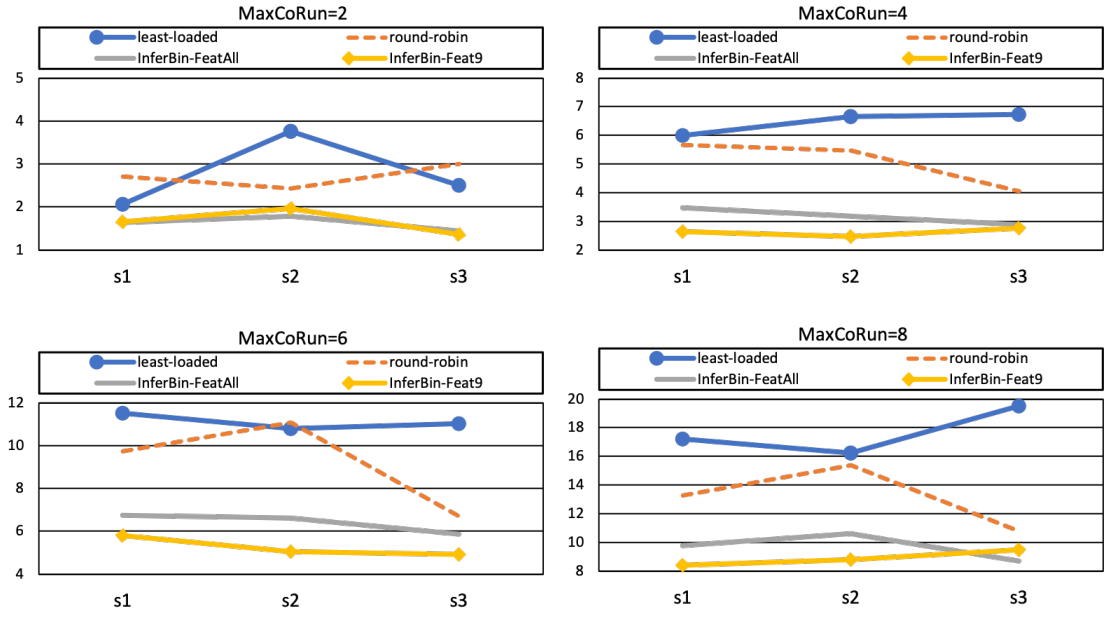


Figure 5.17: The Average Normalized Turn-around Time using different schedulers on a 2-GPU (GTX 760 and GTX 950) system.

5.8 Summary of Magic

In this chapter, we have presented *Magic*, an interference-aware scheduler for GPU workloads. *Magic* adopts an automatic feature selection algorithm, based on *Principle Feature Analysis*, to identify prominent metrics for interference analysis. *Magic* requires a small amount of profile information from a queued workload to estimate the potential interference with other workloads. We proposed a greedy search method to build an effective interference prediction model for GPU workloads. Guided by our *InferBin* scheduling policy, we achieved an average speedup of 16% versus a first-come-first-serve policy and 10% versus a *Similarity* scheduling policy. We have found that using a few prominent metrics (*Feat9*) can many times result in better scheduling decisions than using the full set of profiling metrics (*FeatAll*). In a best-case scenario, our scheduler can improve STP by 37% and reduce ANTT by 68%.

For a heterogeneous GPU cluster, the *Magic* framework uses a neural network model to predict the best device for an incoming workload before applying the *InferBin* scheduling policy. We have observed that our proposed *InferBin* can outperform a *least-loaded* policy by 15% and outperform a *round-robin* policy by 17%, on average. In addition, our scheduler can improve STP on average, providing a 36% benefit over a *least-loaded* scheduler and 24% over a *round-robin* scheduler. On average, our scheduler can reduce ANTT by 1.9x over *least-loaded* and 1.6x over *round-robin*. We observed that using a reduced feature set (*Feat9*) can achieve similar performance benefits as using a full profiling metric set (*FeatAll*).

Chapter 6

Conclusion

Modern GPUs support concurrent execution for kernels and applications to improve device resource utilization and boot system throughput. Running kernels and applications concurrently without considering the interference often lead to a suboptimal outcome. As we see new cloud-based GPU instances being offered to the public, it becomes increasingly important to reduce the interference from co-located kernels and jobs, while maintaining a good level of quality-of-service. We explored how to properly design interference-aware scheduling support at the kernel and application level on modern GPUs to meet the demands of future cloud-based GPU computing.

In this thesis, we presented the challenges to achieve a high degree of multilevel concurrency on modern GPUs. For kernel-level concurrent execution, we characterized occupancy metrics, kernel configurations and computational characteristics, showing the how each can impact performance. Leveraging profiling information and instruction cycles measured via microbenchmarks, we can categorize a GPU kernel into two groups, memory-intensive and compute-intensive. Based on the kernel similarity, we developed a *Similar Kernel Method* to recommend the best block size for the target kernel configuration. Furthermore, we developed an empirical performance model called *Moka* to explore the design space for concurrent kernel execution (CKE), which takes into consideration data transfer patterns, GPU execution patterns, resource contention and stream launch overhead. Using real-world GPU applications, our study shows *Moka* can estimate the CKE performance within 12% of the actual CKE runtime. In addition, *Moka* can suggest a close-to-optimal solution to determine the dispatch order when funneling different kernels for concurrent kernel execution.

In this thesis we explored optimizing execution efficiency for GPU workloads not only at the kernel level, but also at the application level. We have found that considering GPU resources used by a workload by concurrently executing application can benefit performance significantly. We can achieve better utilization of the available GPU resources. However, GPU workloads can

exhibit varying levels of performance degradation due to concurrent execution. It is challenging to improve system throughput and maintain a high level of fairness, especially for cloud providers. Little prior work has studied the factors contributing to concurrent execution interference. We applied *Principal Feature Analysis* to identify prominent metrics associated with interference, rather than relying on the programmer's domain expertise. We demonstrated that trained interference metrics can be integrated into a model that can produce optimized scheduling decisions and reduce runtime overhead. We proposed training an interference prediction model to categorize GPU workloads into two groups, interference-sensitive and interference-insensitive, based on a pool of different machine learning algorithms. Our best model can achieve higher than 70% prediction accuracy. By incorporating an interference prediction model, our proposed interference-aware scheduling policy named *InferBin* improves system performance by 16% as compared to a first-come-first-serve policy, and outperforms a state-of-art similarity-based scheduler by 10%. Furthermore, we added support for heterogeneous GPU clusters for the interference-aware scheduler. Leveraging a neural network model for cross-GPU performance prediction, our scheduler selects the fastest device for each workload and applies our *InferBin* policy to mitigate interference. On a heterogeneous multi-GPU system, our proposed scheduler outperformed a least-loaded policy and a round-robin policy by 21% and 22%, respectively.

6.1 Future Work

Our proposed *Moka* framework explores the design space for concurrent kernel execution. If we have the ability to predict the CKE performance accurately, *Moka* can be integrated into the GPU compiler to automatically funnel different kernels into the same context for higher computational throughput.

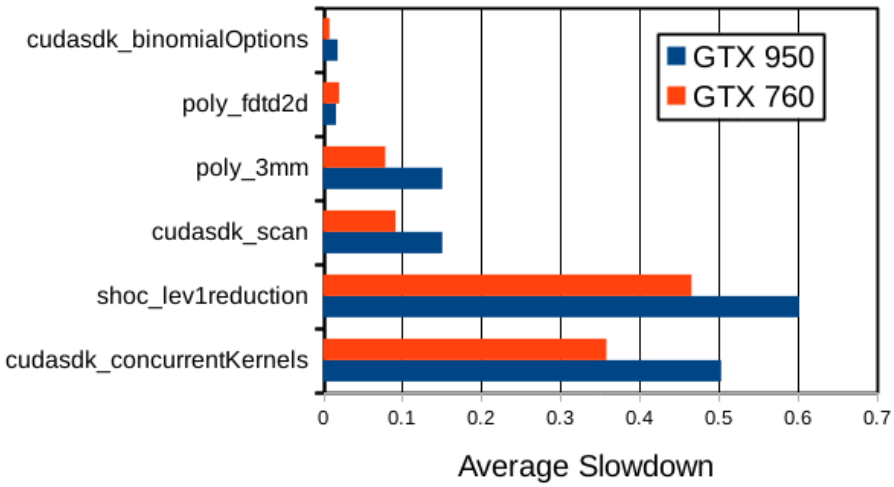


Figure 6.1: Average execution slowdown comparing a GTX 950 and a GTX 760.

In Section 1.3.2, we demonstrated how each workload introduces a specific level of interference when co-located with a second workload on a specific GPU (i.e., a GTX 1080 Ti GPU). In Figure 6.1, we plot the average slowdown for the same six workloads on two different NVIDIA GPUs: 1) a GTX 950 and 2) a GTX 760. While we can observe that the sensitivity classification varies for each workload, the general trend observed across sensitivity classes remain the same. The threshold used to decide whether a GPU workload is sensitive or insensitive should be set on a per GPU architecture basis. For our interference-aware scheduling exploration applied at an application level, our proposed scheduling policy (*InferBin*) used the binary size as an indicator of workload execution time, which introduces some inaccuracy in our work. A more robust performance prediction model is needed for GPU workloads to guide the scheduler to mitigate the interference of concurrently executed GPU workloads. Our proposed interference prediction model estimates the sensitivity level of an application with the default data size. Revisiting sensitivity analysis, using different application input sizes, should be a fruitful direction for future work. Our framework can also be extended to support GPU virtualization technology such as rCUDA [24] to further optimize the concurrent execution efficiency for cloud computing.

Bibliography

[1] Classifier comparison, 2018.

[2] Cross-validation: evaluating estimator performance, 2018.

[3] CUDA C Programming Guide. *NVIDIA Corporation, Oct*, 2018.

[4] Preprocessing data, 2018.

[5] Multi-layer perceptron classifier, 2019.

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

[7] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. Loggp: Incorporating long messages into the logp model. *Journal of parallel and distributed computing*, 44(1):71–79, 1995.

[8] Amazon. Amazon EC2 Elastic GPU. `https://aws.amazon.com/ec2/elastic-gpus/`, 2018.

[9] AMD. AMD CodeXL-Powerful Debugging, Profiling and Analysis Tool, 2014.

[10] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An adaptive performance modeling tool for gpu architectures. In *ACM Sigplan Notices*, volume 45, pages 105–114. ACM, 2010.

[11] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009.

[12] M. Bautin, A. Dwarakinath, and T.-c. Chiueh. Graphic engine resource management. In *Multimedia Computing and Networking 2008*, volume 6818, page 68180O. International Society for Optics and Photonics, 2008.

[13] M. Boyer, J. Meng, and K. Kumaran. Improving gpu performance prediction with data transfer modeling. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1097–1106. IEEE, 2013.

[14] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM transactions on graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

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

[16] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 17–32. ACM, 2017.

[17] Q. Chen, H. Yang, J. Mars, and L. Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. In *ACM SIGPLAN Notices*, volume 51, pages 681–696. ACM, 2016.

[18] S. Collange, D. Defour, and D. Parello. Barra, a parallel functional gpgpu simulator, 2009.

[19] S. Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[20] H. Dai, Z. Lin, C. Li, C. Zhao, F. Wang, N. Zheng, and H. Zhou. Accelerate gpu concurrent kernel execution by mitigating memory pipeline stalls. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 208–220. IEEE, 2018.

[21] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[22] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.

[23] I. Davidson and S. Ravi. Agglomerative hierarchical clustering with constraints: Theoretical and empirical results. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 59–70. Springer, 2005.

[24] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 224–231. IEEE, 2010.

[25] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE micro*, 28(3), 2008.

[26] Q. Fang and D. A. Boas. Monte carlo simulation of photon migration in 3d turbid media accelerated by graphics processing units. *Optics express*, 17(22):20178–20190, 2009.

[27] N. Fermi. Nvidias next generation cuda tm compute architecture, fermi, 2009.

[28] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978.

[29] P. B. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write asynchronous pram model. In *European Conference on Parallel Processing*, pages 277–292. Springer, 1996.

[30] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides, and N. Guil. An optimized approach to histogram computation on gpu. *Machine Vision and Applications*, 24(5):899–908, 2013.

[31] X. Gong, Z. Chen, A. K. Ziabari, R. Ubal, and D. Kaeli. Twinkernels: an execution model to improve gpu hardware scheduling at compile time. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pages 39–49. IEEE Press, 2017.

[32] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10. IEEE, 2012.

[33] K. Gray. *Microsoft DirectX 9 programmable graphics pipeline*. Microsoft Press, 2003.

[34] C. Gregg, J. Dorn, K. M. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent gpgpu kernels. In *HotPar*, 2012.

[35] K. O. W. Group et al. The opencl specification, version 1.0. 29, 8 december 2008.

[36] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. Enabling task parallelism in the cuda scheduler. In *Workshop on Programming Models for Emerging Architectures*, volume 9. Citeseer, 2009.

[37] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24. ACM, 2009.

[38] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *2011 USENIX Annual Technical Conference (USENIX ATC11)*, page 31, 2011.

[39] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[40] C.-H. Hong, I. Spence, and D. S. Nikolopoulos. Gpu virtualization and scheduling methods: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 50:35, 2017.

[41] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

[42] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, 2011.

[43] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra. Improving gpgpu energy-efficiency through concurrent kernel execution and dvfs. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, pages 1–11. IEEE, 2015.

[44] Y. Jiao, H. Lin, P. Balaji, and W.-c. Feng. Power and performance characterization of computational kernels on the gpu. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 221–228. IEEE, 2010.

[45] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das. Application-aware memory system for fair and efficient execution of concurrent gpgpu applications. In *Proceedings of Workshop on General Purpose Processing Using GPUs*. ACM, 2014.

[46] E. Jones, T. Oliphant, and P. Peterson. {SciPy}: open source scientific tools for {Python}, 2014.

[47] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[48] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar. Resource sharing in gpu-accelerated windowing systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 191–200. IEEE, 2011.

[49] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. Rgem: A responsive gpgpu execution model for runtime engines. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 57–66. IEEE, 2011.

[50] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011.

[51] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt. Gdev: First-class gpu resource management in the operating system. In *USENIX Annual Technical Conference*, pages 401–412. Boston, MA;, 2012.

[52] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of ptx kernels. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 3–12. IEEE, 2009.

[53] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling gpu-cpu workloads and systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 31–42. ACM, 2010.

[54] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. Narayanan, and K. Srinathan. A performance prediction model for the cuda gpgpu platform. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 463–472. IEEE, 2009.

[55] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 65–76. IEEE, 2009.

[56] J. Lai and A. Seznec. Break down gpu execution time with an analytical method. In *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, pages 33–39. ACM, 2012.

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

[58] C. Luo and R. Suda. A performance and energy consumption analytical model for gpu. In *Dependable, autonomic and secure computing (dasc), 2011 ieee ninth international conference on*, pages 658–665. IEEE, 2011.

[59] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[60] M. McCool and S. D. Toit. *Metaprogramming GPUs with Sh*. AK Peters Ltd, 2004.

[61] X. Mei and X. Chu. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[62] K. Menychtas, K. Shen, and M. L. Scott. Disengaged scheduling for fair, protected access to fast computational accelerators. In *ACM SIGPLAN Notices*, volume 49, pages 301–316. ACM, 2014.

[63] Microsoft. Windows Virtual Machines Documentation. `https://docs.microsoft.com/en-us/azure/virtual-machines/windows/`, 2018.

[64] J. Nickolls and W. J. Dally. The GPU Computing Era. *Micro, IEEE*, 30(2):56–69, 2010.

[65] NVIDIA. Compute unified device architecture programming guide, 2007.

[66] NVIDIA. Technical brief: Nvidia geforce gtx 200 gpu architectural overview, 2008.

[67] NVIDIA. NVIDIA GeForce GTX 680: The fastest, most efficient GPU ever built, 2012.

[68] NVIDIA. Nvidia's Next Generation CUDA™ Compute Architecture, Kepler™ GK110, 2012.

[69] NVIDIA. NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made, 2014.

[70] NVIDIA. NVIDIA Visal Profiler, 2014.

[71] NVIDIA. Command-line Profiler, 2016.

[72] NVIDIA. CUDA Profiler Users Guide (Version 8.0), 2016.

[73] NVIDIA. NVIDIA GeForce GTX 1080 Whitepaper, 2016.

[74] NVIDIA. CUDA Toolkit, 2017.

[75] NVIDIA. NVIDIA TESLA V100 GPU ARCHITECTURE THE WORLDS MOST ADVANCED DATA CENTER GPU, 2017.

[76] NVIDIA. CUDA Binary Utilities, 2018.

[77] NVIDIA. CUDA Runtime API. *NVIDIA Corporation, May*, 2018.

[78] NVIDIA Corporation. nvidia-smi - NVIDIA System Management Interface program.

[79] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

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

[81] J. J. K. Park, Y. Park, and S. Mahlke. Dynamic resource management for efficient utilization of multitasking gpus. *ACM SIGOPS Operating Systems Review*, 51(2):527–540, 2017.

[82] R. Phull, C.-H. Li, K. Rao, H. Cadambi, and S. Chakradhar. Interference-driven resource management for gpu-based heterogeneous clusters. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 109–120. ACM, 2012.

[83] Python Software Foundation. Python language reference, version 3.7, 2018.

[84] Z. Qi, J. Yao, C. Zhang, M. Yu, Z. Yang, and H. Guan. Vgris: Virtualized gpu resource isolation and scheduling in cloud gaming. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11:17, 2014.

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

[86] S. Rennich. Cuda C/C++ Streams and Concurrency. *NVIDIA GPU Computing Webinars*, 2012.

[87] G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in cuda. *Nvidia CUDA SDK Application Note*, 18, 2009.

[88] D. Shreiner, M. Woo, and J. Neider. Opengl: programming guide: the official guide to learning opengl, version 1.4. *Addison-Wesley*, 2004.

[89] M. Stephenson and D. R. Johnson. SASSI. `https://github.com/NVlabs/SASSI`, 2015.

[90] M. Stephenson, S. K. Sastry Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler. Flexible software profiling of gpu architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 185–197. ACM, 2015.

[91] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12:66–73, 2010.

[92] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

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

[94] D. Tarjan, K. Skadron, and P. Micikevicius. The art of performance tuning for cuda and manycore architectures. *Birds-of-a-feather session at SC*, 9, 2009.

[95] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos. Understanding the impact of cuda tuning techniques for fermi. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 631–639. IEEE, 2011.

[96] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2sim: a simulation framework for cpu-gpu computing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 335–344. ACM, 2012.

[97] Y. Ukidave, X. Li, and D. Kaeli. Mystic: Predictive scheduling for gpu based cloud servers using machine learning. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 353–362. IEEE, 2016.

[98] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

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

[100] G. Wang, Y. Lin, and W. Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, pages 344–350. IEEE, 2010.

[101] L. Wang, M. Huang, and T. El-Ghazawi. Exploiting concurrent kernel execution on graphic processing units. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 24–32. IEEE, 2011.

[102] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 358–369. IEEE, 2016.

[103] Y. Wen, M. F. O'Boyle, and C. Fensch. Maxpair: Enhance opencl concurrent kernel execution by weighted maximum matching. In *Proceedings of the 11th Workshop on General Purpose GPUs*, pages 40–49. ACM, 2018.

[104] F. Wende, F. Cordes, and T. Steinke. 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*, pages 74–83. IEEE, 2012.

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

[106] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing data warehousing applications for gpus using kernel fusion/fission. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2433–2442. IEEE, 2012.

[107] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 230–242. IEEE, 2016.

[108] M. Xue, K. Tian, Y. Dong, J. Ma, J. Wang, Z. Qi, B. He, and H. Guan. gscale: Scaling up gpu virtualization with dynamic sharing of graphics memory space. In *USENIX Annual Technical Conference*, pages 579–590, 2016.

[109] L. Yu, A. Goldsmith, and S. Di Cairano. Efficient convex optimization on gpus for embedded model predictive control. In *Proceedings of the General Purpose GPUs*, pages 12–21. ACM, 2017.

[110] L. Yu, X. Gong, Y. Sun, Q. Fang, N. Rubin, and D. Kaeli. Moka: Model-based concurrent kernel analysis. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 197–206. IEEE, 2017.

[111] L. Yu, F. Nina-Paravecino, D. Kaeli, and Q. Fang. Scalable and massively parallel monte carlo photon transport simulations for heterogeneous computing platforms. *Journal of biomedical optics*, 23(1):010504, 2018.

[112] L. Yu, Y. Ukidave, and D. Kaeli. Gpu-accelerated hmm for speech recognition. In *Parallel Processing Workshops (ICCPW), 2014 43rd International Conference on*, pages 395–402. IEEE, 2014.

[113] L. Yu, Y. Zhang, X. Gong, N. Roy, L. Makowski, and D. Kaeli. High performance computing of fiber scattering simulation. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*, pages 90–98. ACM, 2015.

[114] C. Zhang, J. Yao, Z. Qi, M. Yu, and H. Guan. vgasa: Adaptive scheduling algorithm of virtualized gpu resource in cloud gaming. *IEEE Transactions on Parallel and Distributed Systems*, 25:3036–3045, 2014.

[115] Y. Zhang and J. D. Owens. A quantitative performance analysis model for gpu architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382–393. IEEE, 2011.

[116] J. Zhong and B. He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1522–1532, 2014.