August 2020

GPU Resource Optimization and Scheduling for Shared Execution Environments

Ryan Seamus Luley
Syracuse University

Abstract

General purpose graphics processing units have become a computing workhorse for a variety of data- and compute-intensive applications, from large supercomputing systems for massive data analytics to small, mobile embedded devices for autonomous vehicles. Making effective and efficient use of these processors traditionally relies on extensive programmer expertise to design and develop kernel methods which simultaneously trade off task decomposition and resource exploitation. Often, new architecture designs force code refinements in order to continue to achieve optimal performance. At the same time, not all applications require full utilization of the system to achieve that optimal performance. In this case, the increased capability of new architectures introduces an ever-widening gap between the level of resources necessary for optimal performance and the level necessary to maintain system efficiency.

The ability to schedule and execute multiple independent tasks on a GPU, known generally as concurrent kernel execution, enables application programmers and system developers to balance application performance and system efficiency. Various approaches to develop both coarse- and fine-grained scheduling mechanisms to achieve a high degree of resource utilization and improved application performance have been studied. Most of these works focus on mechanisms for the management of compute resources, while a small percentage consider the data transfer channels. In this dissertation, we propose a pragmatic approach to scheduling and managing both types of resources – data transfer and compute – that is transparent to an application programmer and capable of providing near-optimal system performance.

Furthermore, the approaches described herein rely on reinforcement learning methods, which enable the scheduling solutions to be flexible to a variety of factors, such as transient

application behaviors, changing system designs, and tunable objective functions. Finally, we describe a framework for the practical implementation of learned scheduling policies to achieve high resource utilization and efficient system performance.

GPU RESOURCE OPTIMIZATION AND SCHEDULING FOR SHARED EXECUTION ENVIRONMENTS

by

Ryan Luley

B.S., St. Lawrence University, 2001

M.S., Syracuse University, 2008

Dissertation

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer and Information Science and Engineering

Syracuse University

August 2020

DISCLAIMER

The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

Acknowledgements

I owe a great debt of gratitude to many people who have helped, guided, and encouraged me in the completion of this endeavor.

First, to my advisor, Prof. Qinru Qiu – I am incredibly grateful for your patience and guidance throughout my studies. I have learned a great deal about conducting research and performing critical analysis over these years. I am appreciative of your support and encouragement, and particularly your understanding of the demands of my professional career and family life.

To my defense committee, Prof. Lixin Shen, Prof. Roger Chen, Prof. Yuzhe Tang, Prof. Fanxin Kong, and Prof. Ehat Ercanli – thank you for taking the time to read my dissertation, provide valuable feedback, and serve on the committee.

To the men and women of the Air Force Research Laboratory Information Directorate who have helped make this possible. Though the names and faces have changed over time, my sincere gratitude goes to all of those who encourage and support professional development and career broadening educational opportunities. I do not know that I would have undertaken this rewarding experience without the push. Many friends and colleagues have helped me along the way, but I would like to offer specific appreciation to a few people. To my immediate supervisors over the past several years, Joseph Caroli and Donald Telesca, both of whom encouraged and accommodated this academic pursuit. Though at times it may have seemed interminable, they never stopped encouraging me, and also continued to push me professionally. I am also indebted to the Rome "HPC staff" – Cameron, Chris, Denise, Bryan, Alex, and others – who have always ensured that I had access to the computer systems and software needed to perform this research.

To current and former students in Prof. Qiu's research group, particularly Dr. Qiuwen Chen, Dr. Khadeer Ahmed, Dr. Zhe Li, and Amar Shrestha, though our interactions were atypical of laboratory mates, I was always impressed by your technical skills and collaborative spirit. I wish you all the best in your future pursuits.

To my parents, Frankie and Jim Luley, who taught me the value of hard work and made sacrifices to give me every opportunity for success. To my siblings, Kate and Sean, for a sibling rivalry built on love that continues to push us each in positive ways to be better.

Finally, there is an adage attributed to Confucius: *It does not matter how slowly you go as long as you do not stop*. There is no doubt that this endeavor took longer than it should have, but that was a sacrifice worth making to never miss a moment of being a husband and father. The completion is doubly rewarding in that sense. To my wife Jennifer and our three children, Troy, Aidan, and Cara – you are and will always be my greatest accomplishments. Thank you for the enduring and unconditional love, support, and encouragement. Lastly, I love you all and now you know about computers.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Over the past two decades there has been a dramatic increase in the widespread adoption of general purpose graphics processing units (GPGPUs, commonly shortened to GPUs), across both computing platform varieties and application domains. Initially designed as an accelerator for rendering high definition, three-dimensional graphics, the addition of programmability quickly led to the scientific computing community finding ways to exploit the high parallelism and single-instruction-multiple-data (SIMD) architecture to significantly accelerate computational calculations which were traditionally executed on central processing units (CPUs) in distributed or grid computing systems [1] [2]. The development of higher-level application programmer interfaces (APIs) and software development kits (SDKs) more suited for the scientific programmer than the graphics programmer lagged behind the increased demand for these types of applications, but hardware vendors such as NVIDIA and AMD eventually began to adapt their architectures to suit such needs. Thus, implementations of Compute Unified Device Architecture (CUDA), introduced in 2007 [3], and the Open Computing Language (OpenCL), introduced in 2009 [4], have supplanted Open Graphics Library (OpenGL) as the primary means to program GPUs.

Enabling a wider audience through these more generally-applicable frameworks, GPUs have increasingly been adopted for numerous different applications across many different domains, to include medical, financial, and cybersecurity. With this increase in potential applications, the range and scale of hardware architectures has grown to match. Now, the largest high performance computers (i.e., supercomputers) contain hundreds of GPU accelerators for the highly parallel scientific codes [5], but the same architectures can be scaled down to meet the real-time and low-power constraints necessary for the control of autonomous vehicles [6].

Regardless of what end of this spectrum is considered, two of the most important system-level metrics are performance and power-efficiency. For supercomputing systems, which often exhibit power consumption on the order of megawatts, efficient and effective utilization is imperative, because it directly relates to the cost of operation. Furthermore, at the application level, performance is critical as it pertains to job throughput, which can often be viewed through the lens of resource contention and utilization. In both cases, we can even put a monetary value on such performance. In the case of power efficiency, it has been reported that 1 megawatt costs approximately 1 million USD [7]. Meanwhile, on the throughput side, in 2015 a new trans-Atlantic cable was laid to connect stock traders between London and New York. This 300 million USD cable reduced latency between the global markets by 5 milliseconds [8], showing the value of seemingly small performance improvements.

The other end of the spectrum, the mobile, edge system, clearly shares the same critical metrics despite different reasoning. In particular, the power efficiency concern is not due to the cost but rather a hard system constraint. Most often these types of systems have a limited power supply, and therefore computing implementations have a hard budget. As a result, efficient use becomes a critical concern. At the same time, such systems are often real-time systems and must meet certain performance guarantees, e.g., for safety reasons like collision avoidance in autonomous vehicles. From this perspective, it is obvious that throughput is an equally critical metric.

With similar end objectives in mind, but different constraints in the architectures which can enable those objectives, it becomes a challenge to utilize general-purpose techniques. The method in which application tasks are scheduled on a GPU are agnostic to the application or the operating environment, instead choosing general-purpose, best-effort types of algorithms such as

the first-come, first-served (FCFS) approach. We will show in this research why such a general-purpose approach is insufficient in scenarios which demand high task throughput and power efficiency, and advocate for an approach that can adapt to a dynamic environment and achieve optimal resource utilization to achieve the system objectives.

1.1 Thesis and Contributions

The subject of this dissertation is the job scheduling problem for enabling optimal performance and utilization of GPUs within shared execution environments. The thesis of this work is that it is possible to design and implement a practical framework for GPU task scheduling where the following conditions are met:

1. System throughput and resource utilization are improved over established baseline scheduling practices;
2. Task characterization relies primarily on development-time information, rather than detailed profiling techniques; and
3. The resulting approach requires no user code transformations nor does it impose modifications upon the GPU workload scheduler.

We define a novel approach to GPU job scheduling which is minimally obtrusive to a programmer and does not require low-level device driver modifications. Furthermore, our methods are flexible to dynamic application scenarios and are not constrained by preprogrammed rules or heuristics which are aimed at achieving best average-case performance.

Compared to other methods proposed in related works, we claim two additional advantages and contributions to the approach described in this dissertation. First, the scheduling framework is transparent to the programmer. This implies that a new API or extension of existing APIs is not

necessary, in order to leverage the scheduling capability. Instead, we propose to utilize a method call intercept approach [9] to provide our proposed capability. Second, our techniques are constrained to abide by the existing capability in today's NVIDIA hardware drivers. We do not assume that we can effect changes in the low-level schedulers, i.e., thread block scheduler. While the performance benefits we report are encouraging, there is insufficient evidence to suggest that we can significantly influence the architecture design of a vendor such as NVIDIA. Therefore, we believe our framework is possibly the most pragmatic and complete solution among related works in this space.

1.2 Dissertation Organization

The remainder of this dissertation is organized as follows:

- Chapter 2 includes a background of GPU computing and scheduling considerations, and provides motivation for improvements to performance and energy efficiency;
- Chapter 3 evaluates the performance of the NVIDIA Hyper-Q technology towards enabling greater shared resource scheduling, and demonstrates certain limitations to the current approach with respect to data transfer dependencies and order of execution. We propose solutions to improve the efficiency and effectiveness of Hyper-Q, which become the foundation of our work in subsequent chapters;
- Chapter 4 describes the implementation of a reinforcement learning approach to managing the data transfer resources for moving data to and from the GPU. Our work shows that a shared transfer mechanism can be learned which maximizes the bandwidth utilization and reduces transfer waiting times, to enable higher levels of concurrent execution on the GPU. Our method is compared against the baseline approach, as well as a heuristic approach we

developed. The reinforcement learning approach represents an improvement over our heuristic because it is flexible to the characteristics of the scenario;

- Chapter 5 describes the implementation and evaluation of a deep Q-learning approach for scheduling jobs to the GPU. Our approach is formulated as a multi-objective optimization problem in which we are simultaneously attempting to minimize the average normalized turnaround time and maximize the GPU resource utilization. Our method is compared to various scheduling algorithms and shows competitive performance under both objectives;
- Chapter 6 discusses previous research and relates it to the contributions claimed in this dissertation; and
- Chapter 7 concludes the dissertation with a review and summary of the key concepts and contributions of our research, and presents ideas for future work and application of the proposed methods summarized in this thesis.

Chapter 2 Background and Motivation

This chapter provides necessary background on GPU computing and scheduling, as a basis for understanding the key concepts in the overall body of research. In addition, we highlight motivating themes towards pursuing the solutions proposed herein.

2.1 A Brief Introduction to GPU Architectures and Computing

An understanding of the utilization of a GPU device begins with understanding the architecture design of a representative system. For the purposes of this research, we have restricted our study to NVIDIA architectures. Through the generations of architecture designs – named for famous scientists such as Fermi, Kepler, Maxwell, Pascal, Volta, and Turing – there have been many changes in the capability while the underlying architectural design has remained relatively the same. Figure 1 shows the basic design of these architectures, which can be primarily decomposed into the computational unit called a symmetric multiprocessor (SMX). Each SMX consists of a number of CUDA cores, with each core capable of executing a warp, i.e., a group of threads. The SMX also contains a register file and shared memory, while the entire device contains a large global memory for loading data from the host (i.e., CPU).

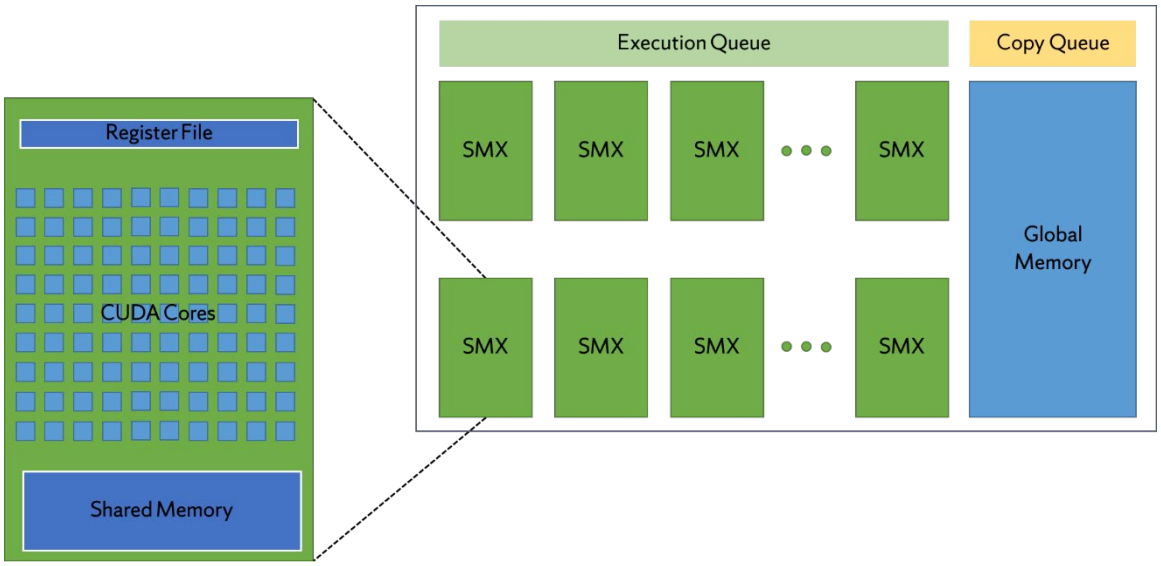

Figure 1. Simplified depiction of NVIDIA GPU and Streaming Multiprocessor (SMX) architecture.

Among the biggest changes that have occurred over the course of the many generations of NVIDIA GPU designs is in the number of CUDA cores within a SMX, as well as the number of SMX units contained on a device, as seen in Table I. The most recent generations, ceding to the increasing popularity and demand from the artificial intelligence (AI) and machine learning (ML) communities, have introduced the Tensor core as another functional unit of the SMX. These units perform a specialized function for accelerating AI/ML calculations, but are not a consideration in this research.

It is also imperative to understand how applications are instantiated and executed on the GPU device. We define an application as a group of smaller tasks, working in coordination to achieve an overarching objective, e.g., to calculate the k-nearest neighbors of a data point. The tasks which make up this application can be classified as either data transfer tasks or computation tasks. Data transfer tasks move data from the host (CPU) to the device (GPU) for computation, or in reverse when computation has completed (for readout or storage). Computation tasks, named kernels, perform SIMD calculations on the data transferred to the device. Kernels are decomposed into a number of thread blocks, each consisting of some uniform number of threads. This hierarchy

CUDA Memory Transfer Methods

```
cudaMemcpy(dst, src, size, kind)
cudaMemcpyAsync(dst, src, size, kind, stream)
```

CUDA Kernel Method

```
kernelMethod<<<gridSize, blockSize, shMemBytes, stream>>>(var1, var2, …)
```

Figure 2. Basic signatures for CUDA memory transfer and kernel method calls.

defines certain data sharing limitations, as well as the basic schedulable unit for execution, which is the thread block. The basic signatures of a memory transfer call and kernel method call are shown in Figure 2.

Table I. Comparison of NVIDIA GPU Architecture Specifications [10].

	GPU Architecture			
Technical Specifications	**Kepler**	**Maxwell**	**Pascal**	**Volta**
Compute Capability	3.5	5.2	6.0	7.0
Number of SMXs	15	24	56	80
FP32 Cores / SMX	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SMX	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SMX	n/a	n/a	n/a	8
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
GPU Die Size	551 mm^2	601 mm^2	610 mm^2	815 mm^2

For the memory copy method, there are two forms depending on whether we wish to invoke a synchronous or asynchronous copy. The first method, `cudaMemcpy`, is implicitly synchronous and will be executed on the default stream, while the second method, `cudaMemcpyAsync`, is executed on the specified stream and asynchronously with the host. Data of `size` bytes from memory address `src` is copied to memory address `dst`. The transfer direction is specified by kind, and can be one of: `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, or `cudaMemcpyDeviceToDevice`.

For the kernel method, `gridSize` refers to the number of thread blocks, and can be a 3-dimensional configuration, while `blockSize` refers to the number of threads per thread block and can likewise be 3-dimensional. The shared memory per block (`shMemPerBlock`) and `stream` arguments are optional in general, though `stream` is required for concurrent execution. The method arguments refer to the addresses in global memory for data on which the method operates.

As we will discuss in subsequent chapters, nearly all of the information required to enable our proposed optimization techniques is contained within these method calls. We will contrast this with alternative approaches proposed by previous works relevant to this problem.

2.2 GPU Scheduling

To begin to understand the implications and difficulties of sharing a single GPU among multiple independent applications, we first must understand how the GPU handles task scheduling in general, for a single application. Scheduling is considered across both temporal and spatial dimensions, i.e., deciding when and where the task should execute. Temporal scheduling is determined by the aforementioned FCFS algorithm; when a task reaches the head of the queue, it will be serviced (if sufficient resources are available). This scheduling is also non-preemptive – once a block is dispatched to the device, execution cannot be pre-empted for another block. Furthermore, as will be discussed later, this non-preemption applies at the task level as well. Spatial scheduling determines on which of the SMX units a task can execute, based upon the dual concerns of the task resource requirements and the resource availability on the SMX. Furthermore, spatial scheduling is broken down to the thread block level as mentioned in the previous section.

At the hardware scheduling level, the smallest schedulable unit is the thread block. This means that an entire block's resource requirements must be capable of being satisfied by some

SMX on the device in order for it to be dispatched for execution. Furthermore, it is not necessary that all thread blocks of a kernel are able to be handled at one time. This introduces the concept of an execution wave. If we view block scheduling as an instantaneous process, each wave schedules the maximum allowable number of blocks based on the block requirements and device resource availability. Once a wave has been scheduled, any remaining blocks are held in the execution queue until some other execution completes on the device, freeing resources.

There are many factors that go into deciding how many blocks can compose a complete execution wave. The configuration of the kernel's thread blocks is a significant factor, because this determines the amount of each resource that is required. Another key limiting factor is a built-in constraint on the number of resident thread blocks allowed per SMX. Such a constraint is important to note, because it is often possible that other resources will not be completely exhausted before the maximum number of resident blocks is reached. This is particularly likely in a concurrent execution scenario consisting of kernels with different configurations and resource requirements. Table II lists a number of the resource constraints per SMX over multiple generations of NVIDIA GPUs.

Table II. NVIDIA GPU Technical Specifications by Compute Capability Generation [11].

Compute Capability	**3.5**	**5.2**	**6.0**	**7.0**
Maximum Threads / SMX	2048	2048	2048	2048
Maximum Thread Blocks / SMX	16	32	32	32
Maximum Threads per Block	1024	1024	1024	1024
Maximum Registers / SMX	65536	65536	65536	65536
Shared Memory per SMX	48 KB	96 KB	64 KB	96 KB

Though details of the spatial scheduling algorithm are unpublished, it is widely believed that thread blocks are assigned to SMX units in a round-robin fashion [12] [13]. However, this claim has been empirically refuted by other work [14]. In either case, there are multiple

considerations that this uncertainty presents, which we will address throughout this research. First, the distribution of thread blocks across all SMX units ensures that certain techniques, e.g., power gating, cannot be reliably applied to increase power efficiency. Second, the spatial scheduling approach results in a nondeterministic pattern, which eliminates possible methods to increase utilization and/or efficiency by mapping certain blocks to specific SMXs. Finally, under concurrent scenarios, the disparate resource requirements of independent kernels result in asymmetric utilization across the set of SMX units. In the following section, we will discuss this point further as it pertains to a broader understanding of the hardware scheduling algorithm of NVIDIA GPUs.

2.3 Multi-stream, Multi-kernel Scheduling

Applying the aforementioned scheduling policy to a scenario in which multiple independent applications share the GPU device creates a necessity to deconflict resource contentions and ensure certain properties, such as fairness. One way that this is enabled in GPU computing is through the construct of streams.

A GPU stream represents an independent flow of execution between the CPU and GPU, similar to a CPU thread. Within a stream, operations are performed sequentially in the order which they are called (i.e., FCFS). Between distinct streams, operations may be performed concurrently or interleaved, depending on the availability of resources. Figure 3 shows an abstract representation of the multi-stream scenario. The operations within the stream are scheduled into

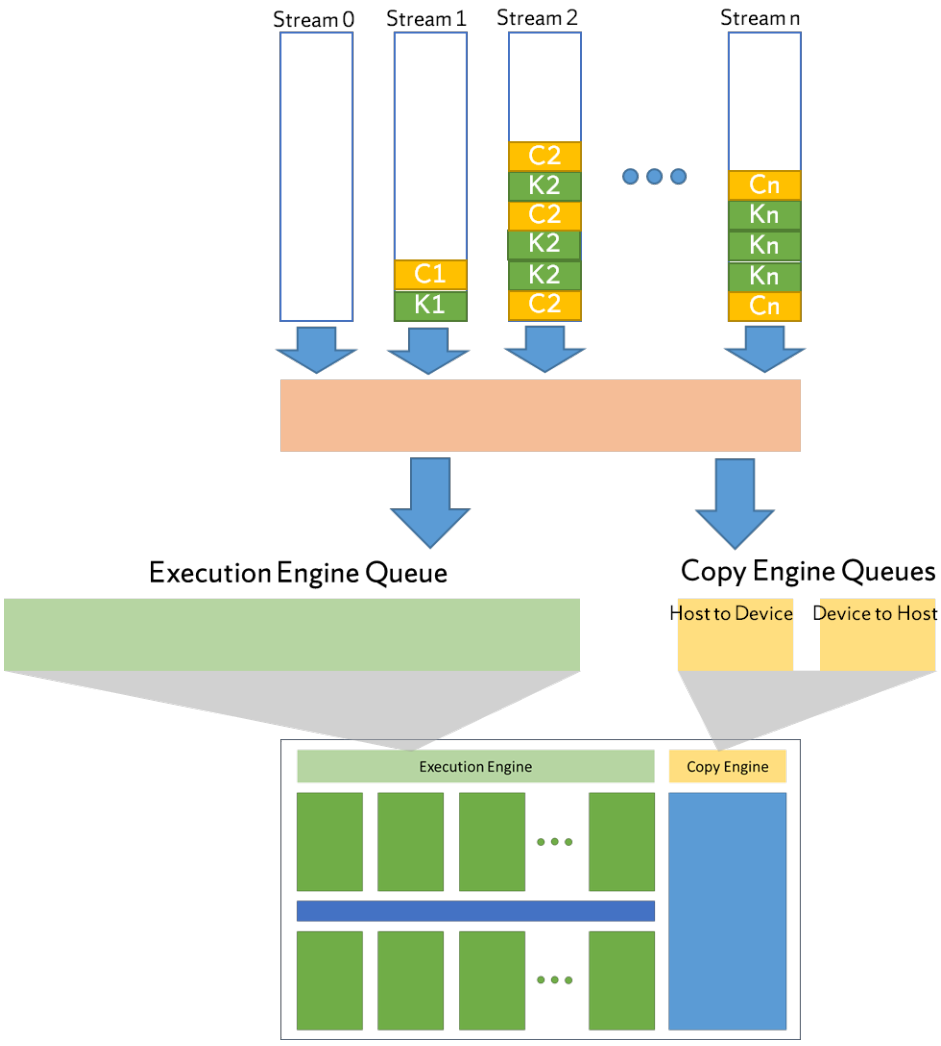

Figure 3. Multi-kernel scheduling hierarchy.

a single work queue, with an operation only being dispatched to the next level device scheduler once it has reached the front of the stream queue.

The next level scheduler represents either the execution queue (for kernels) or the copy queue (for memory transfers). Again, operations within these queues are scheduled in a FCFS manner, but with a number of caveats arising from certain device conditions. These caveats have been empirically derived as a set of scheduling rules by [15] for the NVIDIA Jetson TX2 architecture, and validated for several other NVIDIA architectures [16]. We consolidate these rules for reference in Appendix A.

The importance of understanding these scheduling rules is critical to enabling maximum resource utilization and high system throughput. As an example, we discussed earlier how GPU scheduling is non-preemptive at the kernel level. This is reflected by a combination of rules G3 and X1. Together, these rules state that once the blocks of a kernel have started to be dispatched to the GPU (by X1), no other kernel can be scheduled until all of the first kernel's blocks have been dispatched (G3). This condition is most likely to occur when the first kernel is stalled in the execution queue due to one of the resource rules not being satisfied (R1-R3). However, it is entirely possible that the configuration of a subsequent kernel in the execution queue would satisfy these resource rules. Thus an opportunity for increasing device utilization would be missed, motivating the need for a more refined approach to execution queue scheduling. Without going into similar explanation, we claim that these considerations also exist for the copy queue, necessitating an analogous examination of alternative scheduling policies.

2.4 Opportunities and Benefits to Increasing Utilization and Throughput

Prior research [17] [18] provides motivation for further analysis of the utilization of GPU resources for the purposes of achieving high performance and energy efficiency. In [17], the authors categorize applications along two primary axes – those which are compute-bound and those which are memory-bound – and show that the relationship between performance, utilization, and energy efficiency varies relative to this characterization. Compute-bound applications exhibit the property of strong scaling. For a fixed problem size, if the application is given more processing cores, it will exhibit speedup proportional to the number of cores (i.e., complete same amount of work in less time). Memory-bound applications, on the other hand, have the property of weak scaling. In this case, performance is capped for a fixed problem size, and that the addition of more

processors can only boost performance if given a proportional increase in the problem size (i.e., complete more work in the same amount of time).

Hong and Kim [17] show the relationship of these scaling properties to the performance and power consumption of various GPU kernels, in an effort to derive the optimal number of cores to utilize in an application. The claimed benefit of this approach is that optimal energy efficiency is achieved at different utilizations based on the application type. For compute-bound applications, the optimal point is at full utilization, while for memory-bound applications it is somewhere less than that. The obvious conclusion is that many applications exist which do not require the full complement of GPU resources to achieve optimal performance. Furthermore, the objective of the approach in [17] is to develop a model to identify the optimal number of cores for a single kernel, and is successful in demonstrating improvements in energy efficiency as a result. However, the approach does not address concurrent execution scenarios or propose a method to exploit the potential of underutilized GPU resources.

The research presented in this dissertation builds upon both the insights on performance demonstrated by [17], as well as the omission of techniques to mitigate the underutilization of the GPU resources. To illustrate why this is an important consideration, consider the simple scenarios presented in Figure 4. On the left we show a notional scenario of single-kernel execution (i.e., no device sharing between independent kernels) and the resulting performance and utilization. In particular, the device is only 62.5% utilized and the makespan of scheduled kernels is 4 time units. However when we consider concurrent execution, as shown on the right, utilization increases to 83.3% and makespan decreases to 3 time units. While the illustrated scenario is an oversimplification of a complex scheduling problem, particularly given the scheduling rules and

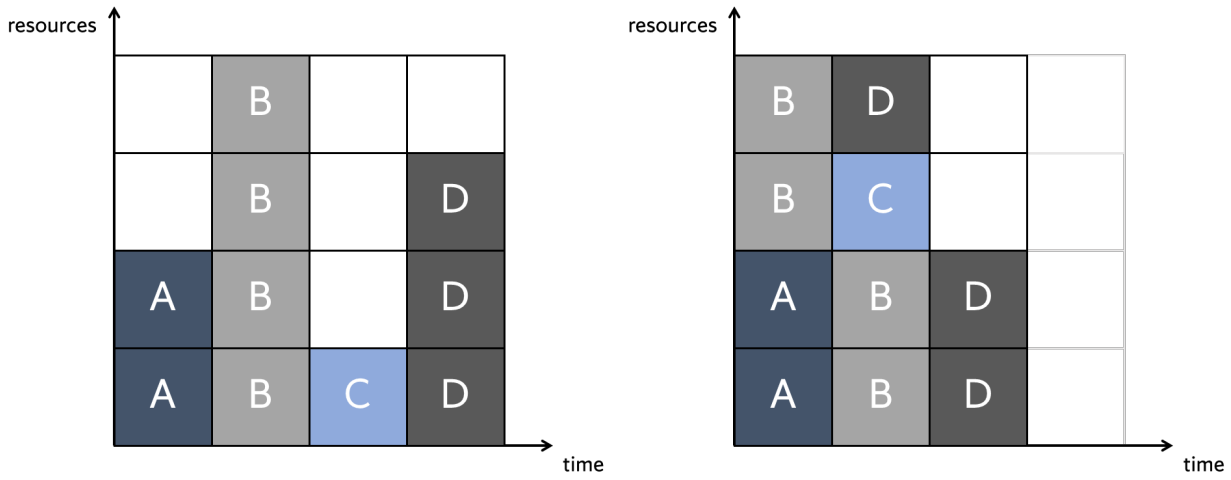

Figure 4. Performance and utilization comparison of serial (left) and concurrent (right) execution.

constraints discussed above, it provides the initial motivation to analyze and develop novel techniques to maximally exploit the capability of GPU computing architectures.

The benefit of exploiting concurrency and maximizing device utilization is further validated when considering the power consumption characteristics of a typical GPU device. As discussed above, work is assigned to the SMX units on the device in a round-robin or semi-round-robin manner. As a result, the entirety of the device is powered during execution, and strategies such as power-gating are generally not applicable. Certainly, there are a variety of factors that determine the peak power consumption for kernel applications, to include thread operations, memory accesses, etc., but we observed that the peak is often independent of the overall percentage of the primary device resources being utilized. Figure 5 illustrates such an example.

In this example, the peak performance of our kernel is achieved with less than full GPU utilization. Similarly, the peak power consumption increases negligibly, approximately 5%, from the lowest utilization level to the maximum utilization possible. This serves to empirically validate the observations from [17], and allow us to make the claim that methods to increase system throughput, i.e., completing more tasks in less time, will have a positive impact on system energy

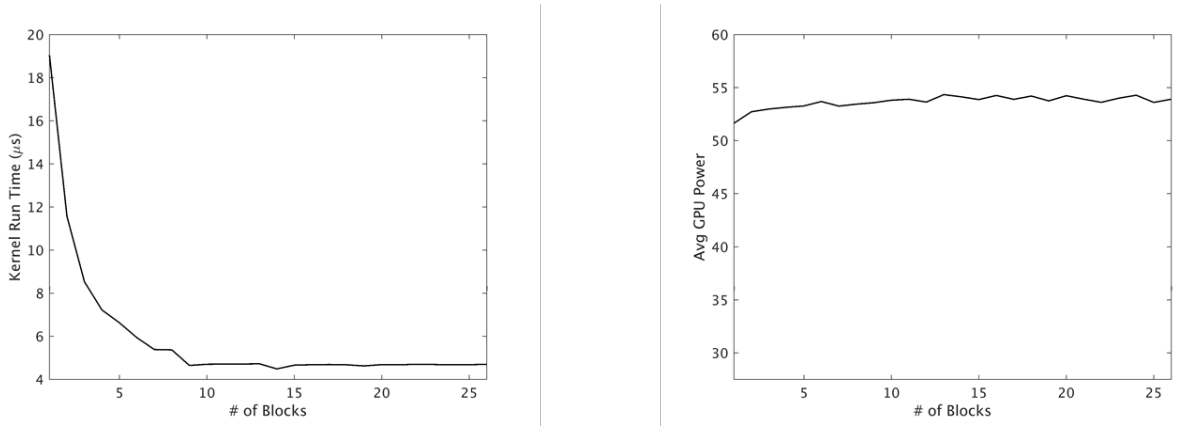

Figure 5. Kernel run time (left) and GPU power consumption (right) for increasing number of blocks and device utilization.

efficiency without introducing any techniques that explicit manage power consumption. Given these opportunities, the primary focus of this research is on the optimal scheduling of independent GPU tasks for the purposes of increasing system throughput and resource utilization. In addition to the aforementioned energy efficiency, benefits to such an approach also include improving respective task performance as measured by the average normalized turnaround time. Finally, though the focus of this research is on a single-GPU system, we discuss an expected benefit to adopting this approach in a multi-GPU system for the purpose of reducing the amount of required hardware, i.e., the total number of GPUs needed in the system to meet a performance objective.

2.5 Metrics

Throughout this dissertation, we will discuss and measure performance of the GPU system and our proposed solutions with regard to certain standardized metrics. Most of these metrics have been similarly adopted by previous research in this area, and provide a meaningful comparison of our approach to that of the other works.

Scheduling performance for GPU tasks can primarily be determined by understanding the composition of the segments of task execution that factor into the overall execution time.

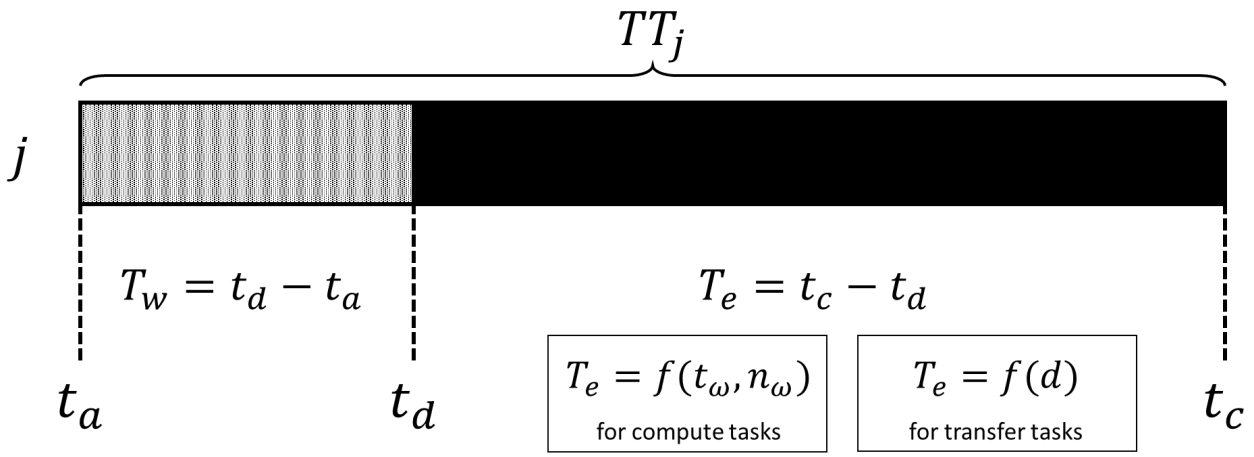

Figure 6. Illustration of task turnaround time components.

Illustrated in Figure 6, we can determine the basic measure of turnaround time, which is simply calculated as the difference between task completion time and task arrival time, see (2-1).

$$TT_j = t_{completion} - t_{arrival} = t_c - t_a \tag{2-1}$$

Task arrival time, t_a, is an immutable property; there is no scheduling policy that can affect the time at which new tasks arrive into the queue. Task completion time, t_c, on the other hand, is particularly affected by not only how long the task executes but also the length of time that the task is required to wait for access to resources, i.e., to be scheduled, or dispatched. Task execution time, T_e, is measured as the difference between completion time (t_c) and dispatch time (t_d). For compute tasks, this difference is a function of the execution time per wave, t_ω, and the number of waves, n_ω. An execution wave is the minimum number of thread blocks of the GPU kernel that can be launched simultaneously on the compute device. For data transfer tasks, T_e is a function of the size of the data transfer. We will discuss this in detail in Chapter 4.

Given the objective of achieving optimal task performance, we can frame the problem as one of simultaneously minimizing T_w and T_e. Furthermore, we assume that we can accurately

measure or model the minimum value of T_e – this is the performance observed given unfettered access to the resources, i.e., no contention with other tasks. Therefore, the primary objective should be to minimize T_w; certainly, as T_w approaches 0, we would expect TT_j to be approximately equivalent to T_e. However, resource sharing may increase actual run time, therefore it is imperative to develop solutions which balance the minimization of both segments.

We will take particular note of T_w as a key metric of task-level performance, since it is most directly tied to scheduling algorithm policy. The following subsections also highlight system-level metrics that are of interest in this research.

2.5.1 Average Normalized Turnaround Time

The first metric we introduce is the average normalized turnaround time (ANTT) defined by Eeckhout [19]. ANTT is particularly useful for measuring performance across a set of tasks, rather than merely considering best- or worst-case performance, as it measures the average effect of processor sharing across the set relative to the expected best-case performance per task when each has exclusive access.

The first step in calculating ANTT is to be able to calculate the normalized turnaround time for each task in the set. Normalized turnaround time (NTT) for a task or program is calculated as the ratio between the single-program turnaround time and the multi-program turnaround time. The single-program (SP) scenario occurs when the task or program has exclusive access to the processing system, while multi-program (MP) occurs when the task or program shares the processing system with one or more other tasks or programs. The equation for NTT is given in (2-2).

$$NTT = \frac{TT_{MP}}{TT_{SP}} \tag{2-2}$$

Given *n* tasks or programs, ANTT can be thus calculated as shown in (2-3). ANTT is a "lower-is-better" metric where the optimal value is 1.0, indicating that MP performance is equivalent to SP performance.

$$ANTT = \frac{1}{n}\sum_{i=1}^{n} NTT_i \tag{2-3}$$

2.5.1 System Throughput

The second metric of interest for evaluating the system performance is System Throughput (STP) defined by [19]. This metric is closely related to ANTT described above, and is therefore calculated similarly. The system throughput defines the amount of progress that a set of tasks or programs makes under processor sharing relative to exclusive execution.

STP is calculated by first determining the normalized progress (NP) of each task or program. NP is calculated as in (2-4), while STP for a set of *n* tasks or programs can subsequently be calculated by (2-5).

$$NP = \frac{TT_{SP}}{TT_{MP}} \tag{2-4}$$

$$STP = \sum_{i=1}^{n} NP_i \tag{2-5}$$

STP is a "higher-is-better" metric, and indicates the amount of single-program progress that is achieved in a given amount of MP time.

2.5.3 Transfer Bandwidth Utilization

With specific respect to the data transfer problem, we define a simple metric to measure the transfer bandwidth utilization. First, we must distinguish that our utilization metric is relative to the peak achieved bandwidth capacity. This value is often different, specifically less than, the peak theoretical bandwidth capacity.

Therefore, given the peak achieved bandwidth, B_{max}, we define the utilization, B_{util}, as shown in (2-6), where B_{obs} gives the observed bandwidth.

$$B_{util} = \frac{B_{obs}}{B_{max}} \tag{2-6}$$

2.5.4 Compute Resource Utilization

The final metric of consideration in this research is the GPU compute resource utilization. In particular, we focus on maximum residency of the four main computing resources noted in Table II: blocks, threads, registers, and shared memory. The selection of this set of resources is both consistent with related works in this area, as well as the GPU scheduling rules described in [15].

For a given GPU architecture, we can calculate the respective maximum resident values for each resource *r* by simply multiplying the per SMX specification, r^i_{smx}, by the total number of SMX units, N_{smx}, as in (2-7) where $i = \{blocks, threads, registers, shared\ memory\}$.

$$r^i_{max} = r^i_{smx} * N_{smx} \tag{2-7}$$

Then, if we can determine the instantaneous value of the resource that is currently in use at time *t*, given as $r^i_{in\_use}$, we can calculate a percentage of resource r^i that is utilized by (2-8).

$$Util_{r^*}(t) = \frac{r^*_{in\_use}}{r^*_{max}} \tag{2-8}$$

Finally, the total device utilization is determined by summing the individual utilization percentages and dividing by 4, shown by (2-9).

$$Util_D(t) = \frac{\sum Util_{r^*}(t)}{4} \tag{2-9}$$

Contrast this definition with the methods supported by hardware, in particular using the NVIDIA System Management Interface, nvidia-smi [20] and the NVIDIA Profiler, nvprof [21]. With nvidia-smi, we can query device utilization at a sampling rate between 166 milliseconds to 1 second. However, the query reports only the percentage of time in the past sampling window when one or more kernels is executing on the GPU. The nvprof utility, on the other hand, reports an `achieved_occupancy` metric. This value also does not suffice for our purposes, as it only provides a ratio of the average active warps to the maximum active warps. Therefore, occupancy is only measure in terms of thread resources. Neither metric measures precisely how much of the device is being utilized in the sampling window, while our metric provides greater insight into the total resource utilization performance.

Chapter 3 Effective Utilization of CUDA Hyper-Q

In this chapter, we evaluate the effective use of CUDA Hyper-Q Technology and describe the foundation of the methodology which we will further develop in subsequent chapters. A key understanding from this dissertation is that the primary limiting factors to optimal GPU performance are due to implicit or explicit synchronizations. We make the claim that Hyper-Q provides many improvements for certain explicit synchronizations, but lacks fine-grained control which can inhibit optimal performance. Furthermore, we also propose a framework which aims to mitigate, if not completely eliminate, the implicit synchronizations which may severely degrade performance and are difficult to uncover. As a result, we advocate for a seamless integration approach to enable complex behaviors of our methodology without requiring significant code modification and rewriting. The research described in this chapter introduces the following contributions:

- It demonstrates that Hyper-Q technology effectively manages the fragments of GPU compute resources to increase overall system utilization and throughput.
- It shows the impact of memory transfer on concurrent kernel execution. First of all, due to the contention for the single direct memory access (DMA) engine for each transfer direction, false serialization may occur despite the availability of compute resources. Furthermore, the interleaving of DMA transactions among different streams extends the effective memory transfer time, and further delays kernel execution.
- It exploits other system level concurrency opportunities to improve overall throughput. Our analysis of application behaviors show that by methodically pairing applications with

different execution patterns, we can significantly improve concurrency due to overlapping of host-to-device and device-to-host memory transfers, or either memory transfer with kernel execution.

- Finally, we propose a Hyper-Q management framework which abstracts many key API methods which cause implicit synchronizations and reduces object creation or destruction overheads through the use of pre-allocated object pools.

This work reveals the potential to further improve GPU resource utilization compared to previous works by efficiently and effectively managing memory transfers and manipulating application ordering to increase overlap potential. First, we focus on the limitations inherent in the memory transfer architecture. With this as our underlying motivation, we develop a method for mitigating these limitations and improving overall concurrency. Using standard benchmark applications from the Rodinia suite [22] [23], we implement a modular testing application infrastructure to evaluate the performance improvement of each part of our approach. In addition, we examine the concurrent execution of heterogeneous applications with different GPU resource requirements, and allow for scheduling of kernels which oversubscribe the GPU. While those applications will be recommended for serial execution based on previous scheduling algorithms [24] [25], our experimental results demonstrate that we can achieve better performance than serialization by carefully scheduling the applications for concurrent execution. Improvement under our strategy is measured relative to serialized execution. We show that each part of our strategy is additive to overall performance improvement, attaining up to 59% improvement over serial execution. Furthermore, the methods discussed below provide the foundational elements of the memory transfer and compute optimization algorithms which we introduce in Chapters 4 and 5 respectively.

3.1 An Approach to Effectively Utilize CUDA Hyper-Q Technology

In this section, we describe the underlying tenets of our approach, which focuses on exploiting the capabilities inherent in the GPU hardware and management API and manipulates execution flow to produce more favorable conditions for device-level concurrency.

3.1.1 Lazy Resource Utilization Policy

Our initial approach follows a "lazy" method and exploits what is called Leftover policy in [26] to implicitly manage GPU utilization, rather than by actively modifying kernel configurations or virtually partitioning resources. Under this policy, the hardware will begin scheduling thread blocks for an execution round, or wave, in the order in which they are received, until one or more GPU resources is completely exhausted. In some cases, this may mean only scheduling thread blocks from one application kernel. However, in the cases where an application requires fewer resources, the GPU automatically schedules some thread blocks from another stream in to the unutilized resource space. This behavior is consistent with the scheduling rules outlined in [15], in particular rules X1 and R1.

This approach allows two or more kernels to be concurrently scheduled regardless of the sum total of their resource requirements. We find that this provides some additional concurrency improvement, while doing no worse than serialization, as might result from resource sharing techniques such as that defined in [25].

Furthermore, we eliminate the need to develop highly detailed schedulers, which micromanage GPU execution down to the thread block level as in [27]. Rather, we focus on methods to eliminate the implicit and explicit synchronizations which inhibit concurrency between independent threads and streams, and allow the Hyper-Q technology to manage block scheduling

at the lowest level. The effect of the synchronizations we have identified were also evaluated around the same time as our initial work by Butler, et al. [28]. Explicit synchronizations are intuitive to a programmer, typically because they are invoked in software code. Implicit synchronizations on the other hand, are not obvious to a program and can cause particular anguish when trying to determine why performance is limited under certain conditions. These type of synchronizations often deal with software-level resource creation or destruction, e.g., memory, streams, and events. Examples of both types of synchronizations are shown in Table III.

Table III. CUDA API calls resulting in synchronization effects.

Type of Synchronization	**CUDA API Call**
Explicit	cudaDeviceSynchronize, cudaStreamSynchronize, cudaEventSynchronize, cudaEventRecord, cudaStreamWaitEvent, cudaEventQuery
Implicit	cudaMalloc, cudaMallocHost, cudaStreamCreate, cudaStreamDestroy, cudaEventCreate, cudaEventDestroy

3.1.2 Effective Memory Transfer Latency

Current GPUs typically have two DMA engines, one for each transfer direction – host-to-device (HtoD) and device-to-host (DtoH). While Hyper-Q mostly solves false serialization among independent kernels with the creation of independent work queues, the constraint of only two DMA engines can severely limit concurrency. This is due to the effect of data dependencies on kernel execution. As demonstrated in Figure 7, representative of output from the NVIDIA Visual Profiler, the serialization and interleaving of independent memory transfers stalls progress among the work queues, in which the kernels must wait for required data to be fully transferred before executing. In the figure we can see that small HtoD transfers serialize in a single copy queue, despite having multiple execution streams and the availability of computing resources. Furthermore, control of the copy queue is interleaved between memory transfers from different

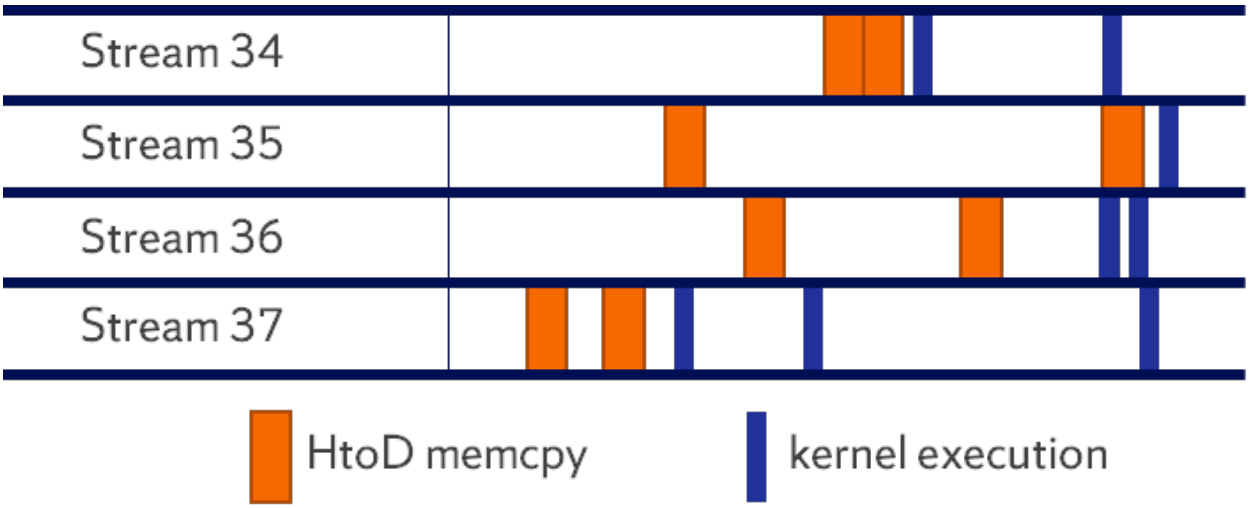

Figure 7. Representative profiler information showing interleaving of memory transfers executions among independent streams.

threads and streams, which prevent any one of the applications from making progress to the point that kernel execution could begin, i.e. all of the required data for that kernel has been transferred to the device. However, once all HtoD transfers complete, kernel execution is significantly parallelized across these execution streams.

This inherent limitation must be addressed in conjunction with, or prior to, applying resource sharing techniques from the existing research. It is not sufficient to simply create separate streams for memory copies, since they will still be serialized due to the contention for the DMA engine at hardware level. To mitigate this constraint, we introduce a host-side synchronization mechanism between application threads. We do this by using a simple mutual exclusion (mutex) object around the HtoD memory transfer stage of each application thread.

The mutex object provides a pseudo-burst transfer mechanism, in which all of the memory transfers for an application are completed before an application on another stream can take control of the copy queue. We believe this to be functionally equivalent to batched memory transfers, which have been presented as a possible mechanism for optimizing memory transfers [29]. In

addition, it has been shown that memory transfer time begins to scale linearly at transfer sizes of 8 KB [30]. We verified similar memory transfer performance for the Tesla K20. The benchmark applications evaluated under this research each have total memory transfer requirements that exceed 8 KB.

An alternative memory transfer approach, chunking, is proposed in [26]. Under this technique, large memory transfers are broken into many smaller transfers, and actually take advantage of the copy queue interleaving behavior. Under such conditions, applications with smaller total memory transfers are allowed to proceed sooner and overall concurrency should be improved. Pai, et al., [26] also focuses on applications with significantly larger, and mostly single, memory transfers (e.g. up to 100 MB) than we explore here. In contrast, since the transfers we experiment with are multiple and smaller in size, we seek to eliminate the interleaving behavior.

To measure the effectiveness of our approach, we introduce a metric called effective memory transfer latency. Given an application A_i, consisting of the set of operations given by (3-1), where m_{HD} is a HtoD memory transfer, k is a kernel, and m_{DH} is a DtoH memory transfer, we define this effective memory transfer latency, L_e, to be the total latency from the start time (T_{start}) of the first memory transfer to the completion (T_{end}) of the last memory transfer, as shown in (3-2), where * can be replaced with either HD to DH.

$$A_i = \{m_{HD}(1), \dots, m_{HD}(K), k_1(1), \dots, k_Z(n), m_{DH}(1), \dots, m_{DH}(K')\} \quad (3\text{-}1)$$

$$L_e(A_i) = T_{end}[m_*(K)] - T_{start}[m_*(1)] \quad (3\text{-}2)$$

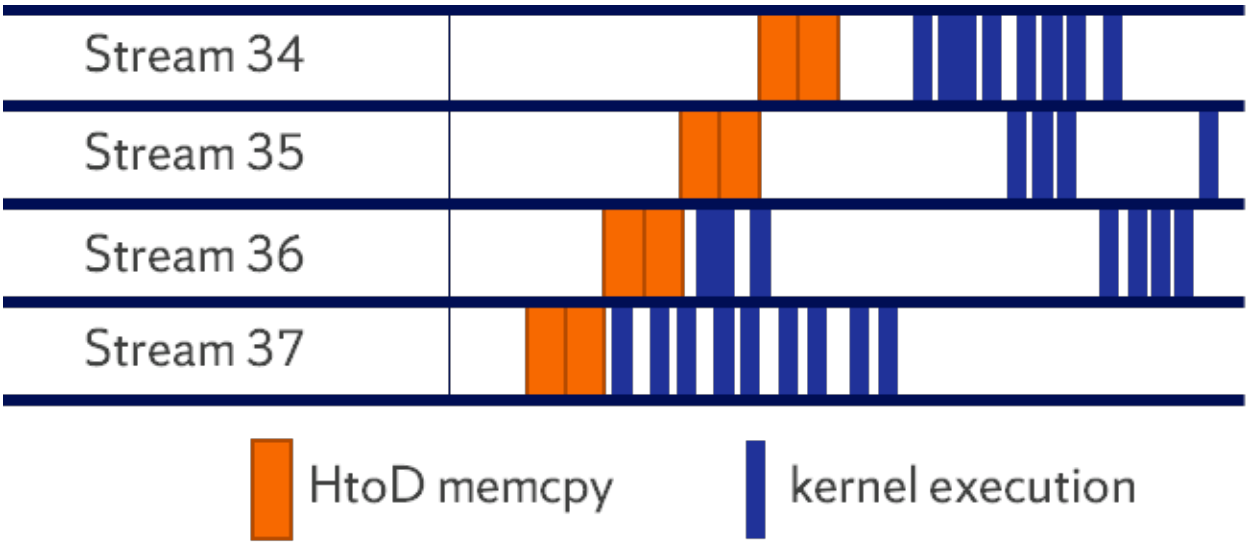

Figure 8. Representative profiler information showing effect of mutex mechanism on memory transfers executions among independent streams.

We calculate the average effective memory transfer latency, by summing L_e for each application A_i on stream s_j, and dividing by the number of applications executed on that stream. The overall average is then taken across all N_S streams.

Figure 8 demonstrates the improved concurrency due to memory transfer synchronization. Transfers for applications from different streams are no longer interleaved, which reduces the effective memory transfer latency and allows some applications to start kernel execution sooner than in the previous case (Figure 7).

3.1.3 Application Reordering

Given the scheduling policies introduced in Chapter 2, we determine that another effective way to improve system throughput without significant overhead is to address the order in which independent streams are scheduled. First, we assume in general that tasks from two independent streams arrive at the scheduler simultaneously. Then, the reordering approach should allow for concurrency by overlapping task calls among the independent streams in the following ways:

- Kernel execution and memory transfer
- Independent kernel executions (assuming available resources)
- Memory transfers in opposite directions, i.e. HtoD with DtoH

For example, among applications we evaluate in this work, we observed an execution pattern which consists of an iteration over a sequence of kernels, with HtoD and DtoH memory transfers inside the iteration loop. This type of application would be well-suited for concurrent execution with an application that consists of kernels that might oversubscribe GPU resources, since each application can overlap GPU accesses without directly competing for compute resources.

We examine the effects of the GPU's inherent FCFS scheduling policy within pairs of applications with different execution patterns by evaluating five distinct launch orderings. Each experimental launch order is simulated through a test harness method to force the desired order of selection within the hardware work queue, though each application executes from an independent stream. The experimental launch orders we test are described below:

- Naïve FIFO, in which applications are scheduled straightforwardly in a first-in, first-out (FIFO) manner. Given a set of applications Ω consisting of m copies of application A_X and n copies of A_Y, the Naïve FIFO scheduling approach results in the queue order given by Figure 9 (a).
- Round-Robin, in which applications are queued by type, and then launched on child threads in a round-robin fashion. Given the same set of applications Ω from above, the Round-Robin scheduling approach results in the order shown in Figure 9 (b).
- Random Shuffle randomly rearranges the set of applications Ω in schedule S_K. For this technique, we start with the Naïve FIFO scheduling order, and then apply a random

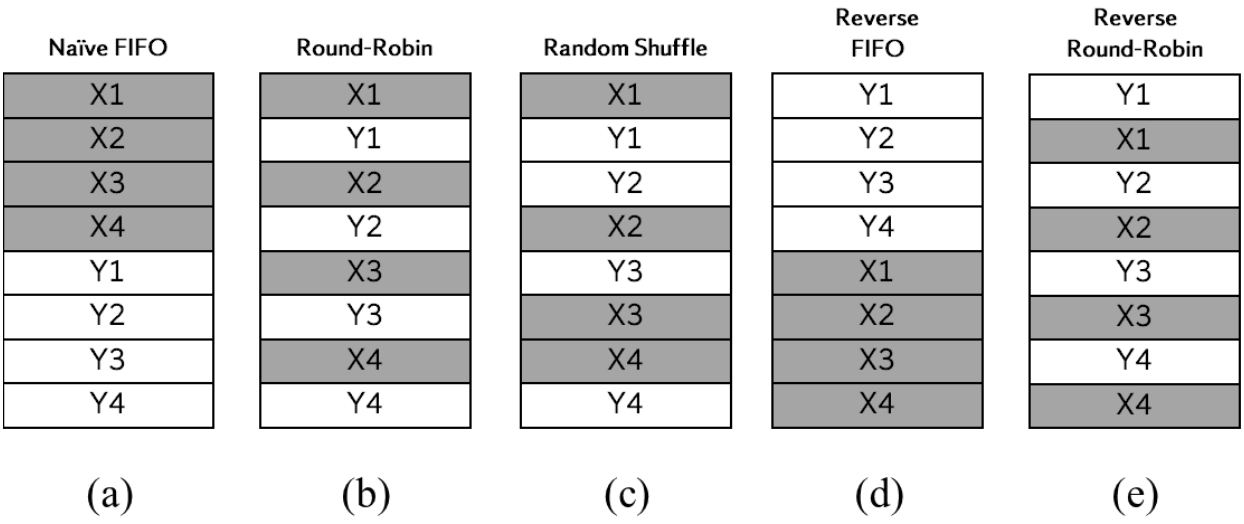

Figure 9. Application orderings under test.

permutation operation to the order to generate a randomly shuffled order. The respective number of applications for A_X and A_Y remains the same, with only the ordering affected. A representative result of the shuffling is given by Figure 9 (c).

- Our final two scheduling approaches are Reverse FIFO and Reverse Round-Robin. In these approaches, we take the results from FIFO and Round-Robin respectively, and reverse the order of the pairs in the schedule. The resulting queue orders are showing in Figure 9 (d) and (e), respectively.

The motivation for varying the queuing order is two-fold. First, this order represents the order in which our framework allocates CUDA streams to the independent applications. In the case where the number of applications scheduled is equal to the number of streams available, then stream allocation order is trivial. However, when there exist fewer execution streams (N_S) than applications to be scheduled (N_A), the scheduling mechanism enables us to control the order in which applications are executed. This is due to the serialization dependency of application tasks within a particular hardware execution queue, whether the individual tasks are independent

from each other or not. Also, while feasible for small numbers of applications and streams, an exhaustive analysis of all orderings is generally not possible and we prefer to evaluate an average-case performance given by the proposed ordering methods.

While host-side thread execution order is non-deterministic, by varying the order in which child threads are launched we aim to prejudice the execution order to follow the thread launch order, and consequently provide greater opportunity for concurrent execution on the GPU. By affecting the order in which applications are launched on child threads, we expect that we can improve overlap potential by interleaving applications with differing execution patterns. In addition, we expect that we could converge on an optimal ordering without exhaustively searching all possible orderings.

Table IV. Encapsulated CUDA methods in our software framework.

CUDA API Method	**HQ Framework**	**HQ Runtime Environment**
cudaMallocHost	allocateHostMemory	HostMemoryPool
cudaFreeHost	freeHostMemory	
cudaMalloc	allocateDeviceMemory	DeviceMemoryPool
cudaFree	freeDeviceMemory	
cudaStreamCreate	StreamManager, Stream classes	StreamPool
cudaStreamDestroy		

3.2 Hyper-Q Management Framework

To efficiently implement the techniques described above, we develop a C++ management framework which encapsulates CUDA API functionality and eliminates the implicit synchronizations caused by certain API methods. This approach is relatively seamless to integrate with existing code (i.e., does not require significant refactoring effort) and allows for pre-allocation of certain object types to eliminate synchronizations and reduce overheads.

The initial implementation of this framework was used for the experiments described in this chapter, and provided abstract method interfaces to the primary CUDA objects that can cause implicit synchronizations [11]. A mapping of framework methods to these CUDA objects is given in Table IV. Subsequently, we have matured this framework into what we call the HQ Runtime Environment, shown in Figure 10.

Our framework expands on the PreAllocator concept introduced by [28]. As noted in the work, the pre-allocation of certain resource pools enables the dynamic "creation" and "destruction" of those resources in runtime, without causing synchronizations which would inhibit concurrent execution on the GPU.

We ported a selection of benchmark applications from the Rodinia 3.0 suite, as shown in Table V, and observed that the amount of programming effort is minimal, since we are not modifying the actual functionality of the benchmark in a significant way.

The implementation of Rodinia benchmarks into our framework is technically distinct from the source code transformation techniques, such as proposed by [26]. In each instance, our implementation of a Rodinia application performs equivalently to its reference implementation. We do not modify the .cu source code file in any significant way. Furthermore, the benchmark kernel applications are logically decoupled from each other, and not merged into a super kernel to manage execution.

Table V. Selected kernels applications from Rodinia 3.0 suite.

Benchmark Name	**CUDA Name**
Gaussian Elimination	gaussian
k-Nearest Neighbors	nn
Needleman-Wunsch	nw
Speckle reducing anisotropic diffusion	srad_v2

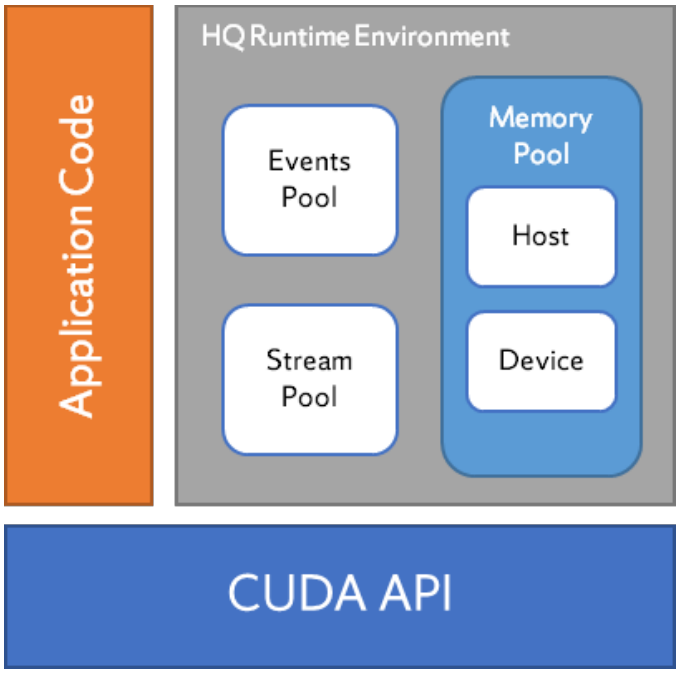

Figure 10. HQ Runtime Environment.

The HQ Runtime Environment is also scalable, as each of the following pools can be defined on a per-device basis. Therefore, in a multiple GPU scenario, independent object pools for each device can be created to support execution across each GPU, without requiring the user to keep track of the object-device relationships. In addition to the objects referenced in Table IV, the HQ Runtime Environment also implements an object pool for CUDA events, which will be required for managing aspects of our data transfer algorithm discussed in the following chapter.

Events Pool. The Events Pool creates a user-defined number of `cudaEvent_t` objects, which can be used for tracking application progress. CUDA events also enable fine-grained synchronization between independent streams without forcing device-wide synchronizations, such as `cudaDeviceSynchronize`. As will be discussed in the transfer queue algorithm in the following chapter, memory transfers are processed on a single stream (per direction), which is separate from the independent streams which the applications execute upon. Therefore, inter-stream

synchronization is critical to enforce the proper order of operations within an application's main thread of execution. We will use events to handle the inter-stream synchronization.

Stream Pool. The Stream Pool allocates the maximum number of `cudaStream_t` objects allowed by the hardware, less the number of streams required to serve the transfer queues (i.e., two, one per transfer direction), for application execution. On the Tesla K20 used in this work, the maximum number of hardware streams is 32 [31], thus the Stream Pool consists of 30 possible application streams in our experiments.

Memory Pool. Possibly the most important aspect of the runtime environment is the memory pool concept. In concurrent, shared execution environments, different applications are expected to come in and out of service dynamically at varying times. Thus, as the application begins execution the allocation of memory is necessary. Conversely, once the application ends the deallocation of memory is required to free up resources. However, since such operations cause implicit synchronization, it was critical to find a solution to eliminate that potential performance constraint.

It is worth noting that the memory pool solution developed here is not fully optimized for performance. In particular, we do not address possible fragmentation, though we do implement merging for contiguous free memory blocks. Furthermore, we noted during testing that the overhead of allocation and deallocation using the memory pool method was not significant. On the other hand, one advantage that is afforded through the use of the memory pool is to enable memory resetting after deallocation, to avoid potential memory leaks as described in [32]. Considering the intended scenario for the proposed algorithm, i.e. shared execution environments, we find this to be an additional incentive to utilize the memory pool mechanism.

3.3 Experiment Methodology

We evaluate our proposed approach using the Hyper-Q Management Framework defined in the previous section. While we believe that the framework provides for cleaner management of the various API function calls, and overall reduces the complexity of the test harness, we would expect that our techniques would exhibit equivalent performance improvements on similarly structured test applications that do not utilize our framework.

In order to realize the potential of GPU concurrency using streams, it is necessary to provide host multithreading. Our test harness uses C++ `std::thread`'s, which encapsulates Pthread functionality on our test workstation. All experiments are conducted on a Tesla K20 series GPU, with compute capability 3.5. In addition, our framework utilizes the C++11 standard and is compiled using gcc 4.8.1.

3.3.1 Benchmark Applications

We selected a subset of applications from the Rodinia benchmark suite [22] for our experiments. The applications were selected to represent various application domains, execution patterns, and resource requirements.

- *Gaussian Elimination*: Linear algebra algorithm for solving a system of linear equations.
- *k-Nearest Neighbors*: Machine learning algorithm for clustering unstructured data.
- *Needleman-Wunsch*: Dynamic programming algorithm for protein alignment in DNA sequencing applications.
- *Speckle Reducing Anisotropic Diffusion (SRAD)*: Image processing algorithm for removing speckle noise without destroying important image features.

Table VI. Application kernel grid and block dimensions, thread and threads per block requirements.

Application	Kernel	Data dim	Calls	Grid dim (x, y, z)	Block dim (x, y, z)	#TB	#TPB
Gaussian	Fan1	512 x 512	511	(1, 1, 1)	(512, 1, 1)	1	512
	Fan2		511	(32, 32, 1)	(16, 16, 1)	1024	256
Needle	needle_cuda_shared_1	512 x 512	16	(1, 1, 1) … (16, 1, 1)	(32, 1, 1)	16	32
	needle_cuda_shared_2		15	(15, 1, 1) … (1, 1, 1)	(32, 1, 1)	15	32
Srad	srad_cuda_1	512 x 512	10	(32, 32, 1)	(16, 16, 1)	1024	256
	srad_cuda_2		10	(32, 32, 1)	(16, 16, 1)	1024	256
Knearest	euclid	42764	1	(168, 1, 1)	(256, 1, 1)	168	256

3.3.2 Test Harness

The execution flow of our test harness begins with loading an application scheduling order to execute, instantiating a new class object for each separate application, allocating all host and device memory, and initializing host memory. Then the parent thread launches each application class instance on its own independent child thread. Within the child thread, each instance runs its particular execution pattern. After all child threads have completed, the host parent thread frees all host and device memory and destroys all stream objects.

Figure 11 shows the profiles of two generic execution patterns, from which most CUDA applications can be abstracted. In general, all applications consist of h_t transfers from host (CPU) to device (GPU), K kernel method calls, and d_t transfers from device to host, where h_t, K, and d_t are all greater than or equal to 1, and h_t and d_t need not be equal. Furthermore, certain applications may consist of inner looping over the kernel method calls and/or outer looping over the entire pattern.

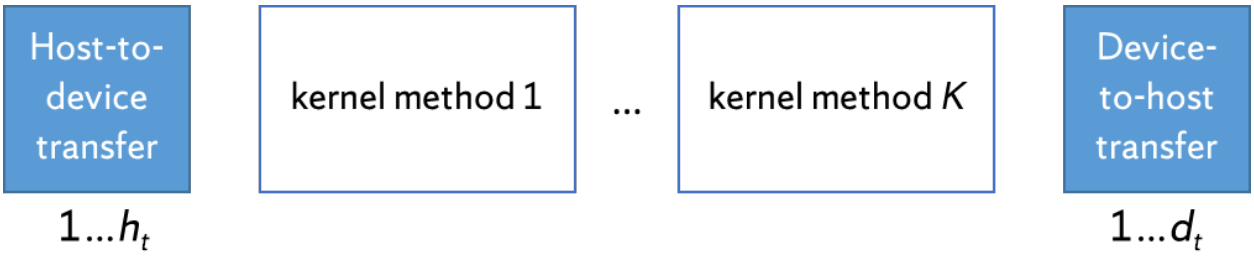

Figure 11. GPU Kernel Execution Pattern.

We examine both homogeneous workloads and heterogeneous workloads. A homogeneous workload is defined as a set of applications in which each application thread executes the same kernel functions on the same size data, and with the same grid/block configuration. A heterogeneous workload is defined as a set of applications in which the application threads take one or more different types. These types may have different data sizes and grid/block configurations. However, among applications of the same type, the workload should be considered homogeneous. For simplicity, we only look at heterogeneous workloads with two different application types, but our framework supports the ability to test workloads with a higher degree of application heterogeneity. The data sizes, grid and block dimensions used for the set of applications we have ported, is given in Table VI.

The test harness iteratively executes all possible pairs of applications with an increasing schedule length with N_A applications, over a similarly increasing number of GPU streams, N_S. This varies execution from fully serialized (i.e. N_A applications on a single stream) to fully parallelized (i.e. N_A applications on 32 streams). For brevity, we do not examine the scenarios where $N_S > N_A$, since we expect no performance improvement to be observed from the addition of idle streams. In the case of heterogeneous workloads, the number of applications is evenly split between the different types in the test pair.

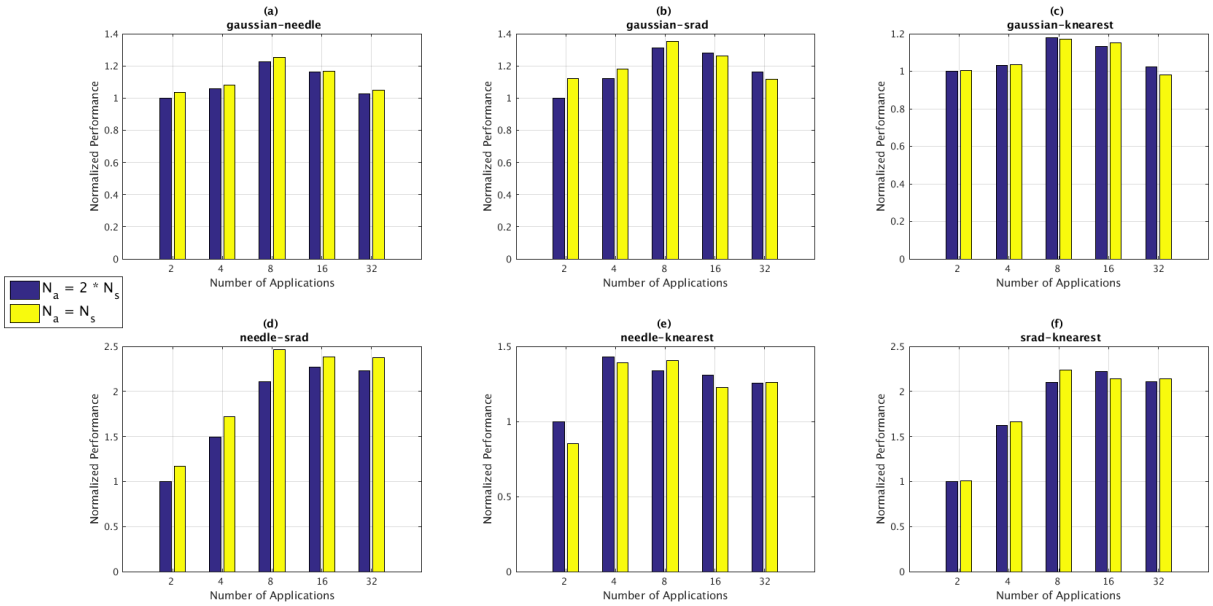

Figure 12. Performance improvement of heterogeneous workloads as compared to serialized execution under our lazy resource utilization policy. Performance is relative to serial execution, and compares half-concurrent and full-concurrent scenarios.

3.4 Experimental Results

Our results demonstrate that we can effectively utilize the built-in features of Hyper-Q technology to achieve significant concurrency improvement with minimal algorithmic complexity and without the need to transform kernel source code.

3.4.1 Lazy Resource Utilization

Figure 12 shows the performance improvements for each heterogeneous pairing under increasing workload scenarios, when compared to the serial case. The normalized performance is measured as the ratio of serial workload makespan to the multi-stream makespan. The results demonstrate that we can achieve significant speedup without implementing specific resource sharing techniques, because the Hyper-Q technology and GPU thread block scheduler work to efficiently utilize the device resources when possible.

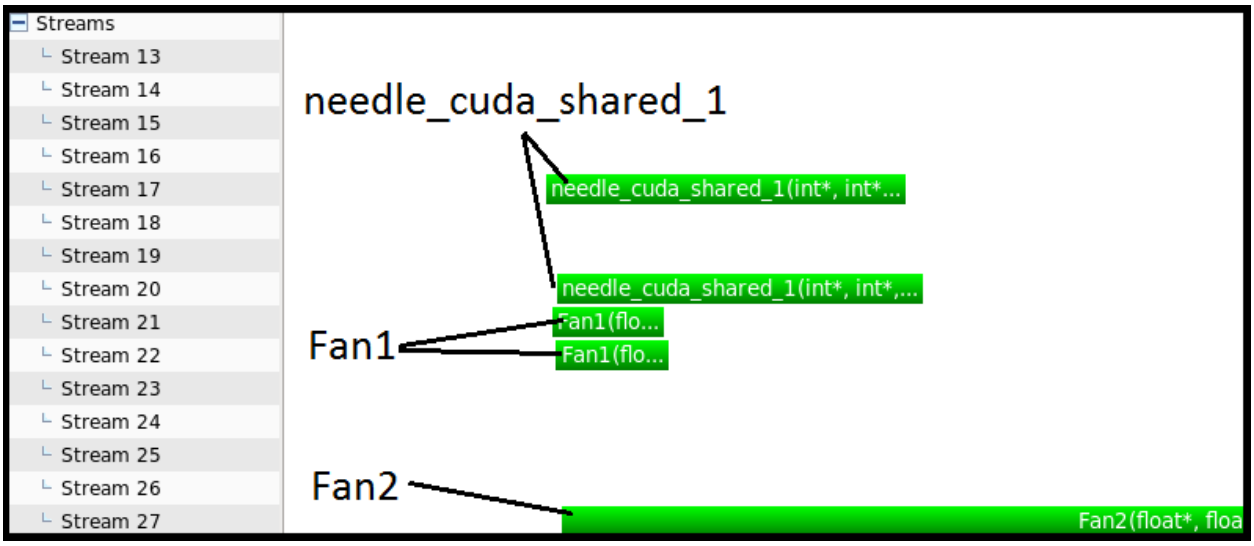

Figure 13. Screen capture from NVIDIA Visual Profiler demonstrating overlap of five kernels on five independent streams, despite total requests exceeding GPU resource limitations.

This can be seen in Figure 13, which shows overlapping kernel execution on five independent streams. At the point of execution shown in the snapshot, Stream 17 launches 89 thread blocks of `needle_cuda_shared_1`, Stream 20 launches 88 thread blocks of `needle_cuda_shared_2`, Streams 21 and 22 each launch one thread block of `Fan1`, and Stream 27 launches 1024 thread blocks of `Fan2`, for a total of 1203 thread blocks.

Under this scenario, and utilizing resource sharing techniques discussed above, these five streams would not be scheduled to execute concurrently because they are requesting more than the theoretical maximum number of thread blocks of 208. However by leveraging the Leftover policy, we allow the device to pack as many thread blocks onto the SMX units as possible, increasing the effective utilization to near 100%. In addition, the kernels `Fan2`, `srad_cuda_1`, `srad_cuda_2`, and `euclid` all require more than one execution round to fully complete, since individually they execute more thread blocks than can simultaneously occupy the GPU. In each case, the final execution round does not fully occupy the GPU and the Leftover policy employed by the scheduler

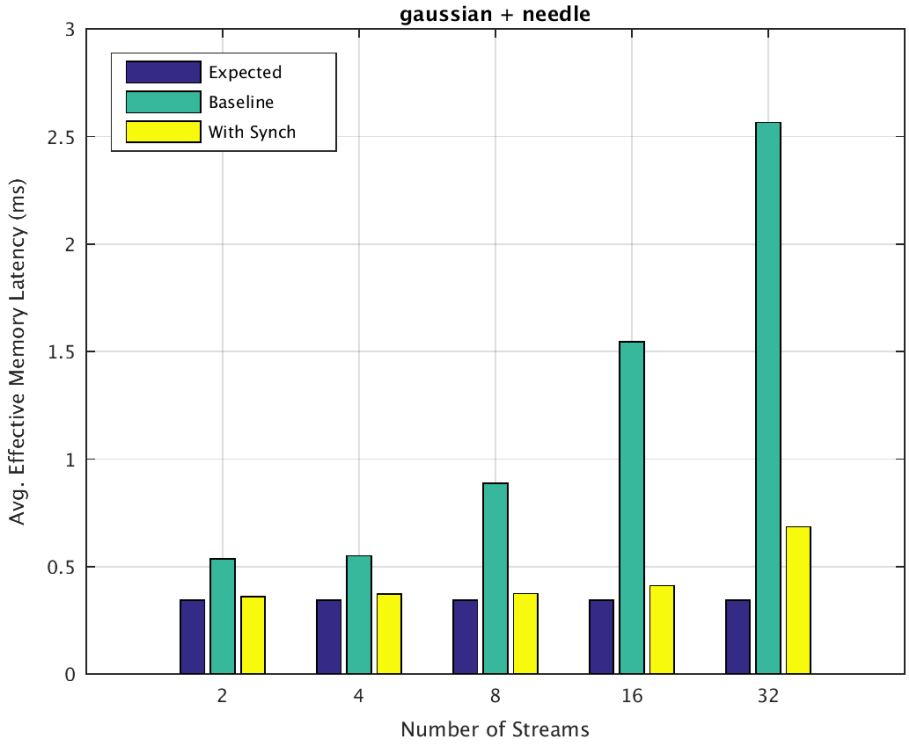

Figure 14. Effective memory transfer latency results.

detects unutilized resources and launches waiting thread blocks from the device's thread block scheduler queue.

Our results from Figure 12 show that we achieve up to 56% improvement (23.6% average) for the half-concurrent scenario, i.e. $N_A = 2 * N_S$, and up to 59% (24.8% average) for the full-concurrent scenario, i.e. $N_A = N_S$. We note that this compares favorably to the average performance improvements demonstrated by other techniques, without the additional overhead or complexity of such techniques.

3.4.2 Effective Memory Transfer Latency

Figure 14 demonstrates the effectiveness of our HtoD memory synchronization approach. We calculate the expected effective memory latency by taking the average performance of memory

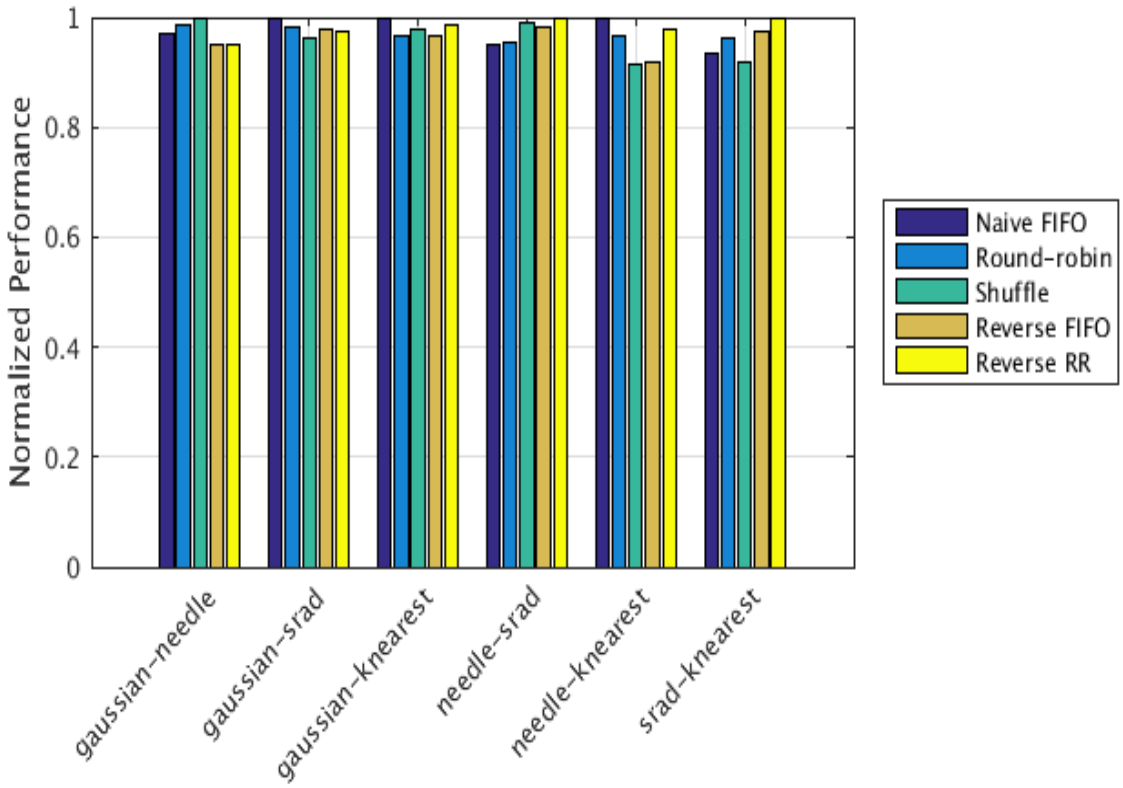

Figure 15. Performance comparison of different scheduling orders for each heterogeneous workload pair.

transfers from the homogeneous case for each application. To calculate the expected effective memory transfer latency for heterogeneous workloads, we average the homogeneous results for each application in the pairing. The results show that the average effective memory latency per application increases up to 8 times over expectation in the baseline case. However, using our approach we reduce the effective latency to be equivalent to the expected estimate.

The benefit of this reduction in effective latency is that applications can begin kernel execution sooner (i.e., reduces waiting time), since kernel execution for an application is dependent on all required memory transfers being completed. In addition, as the degree of concurrency increases, this effectively improves overlap potential, as will be discussed in the next section. Memory transfers and kernel executions from different streams can overlap in GPU execution,

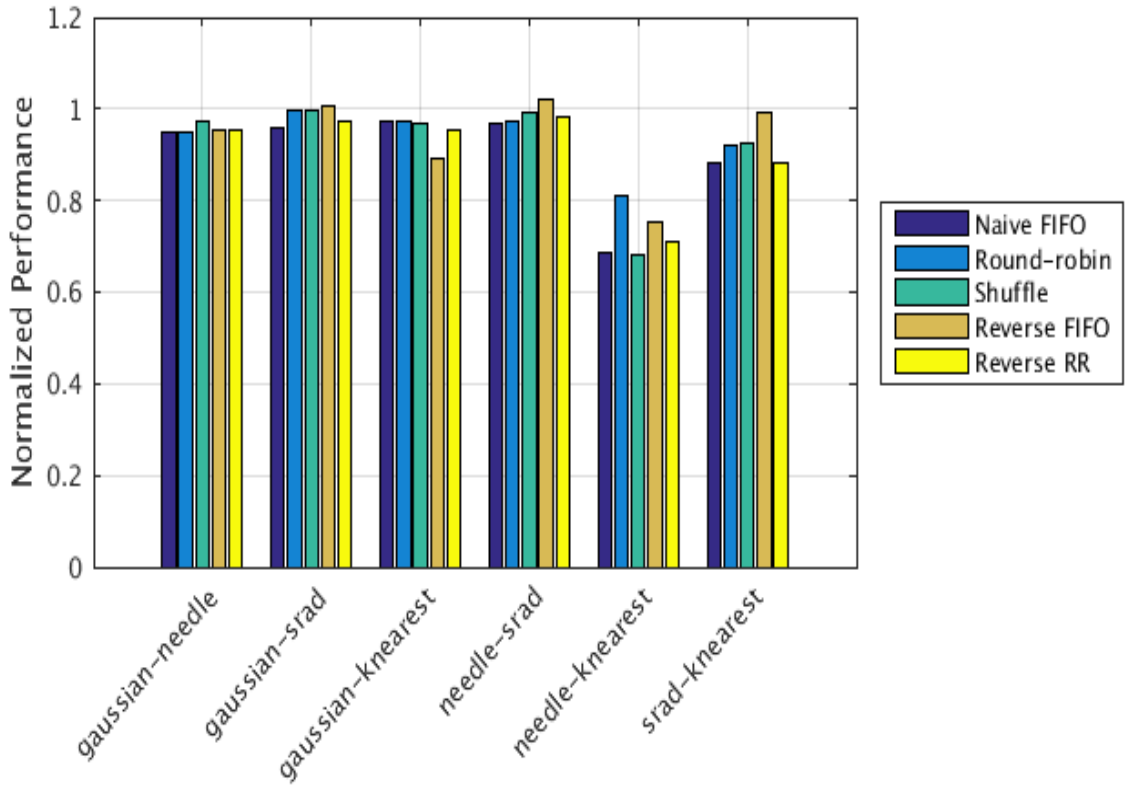

Figure 16. Performance comparison of different scheduling orders when adding memory synchronization.

therefore once an application's entire memory requirements have been transferred, execution of that application's kernels can overlap with subsequent HtoD transfers from other streams.

3.4.3 Application Reordering

Figure 15 compares the performance impacts of the different scheduling techniques introduced above for each workload pairing where $N_S = N_A = 32$. In addition, we use the default memory transfer behavior. In this case, normalized performance is relative to the worst-case schedule order for each particular pairing, to demonstrate the relative impacts of kernel ordering. We observe that schedule order can affect up to 9.4% performance improvement and 3.8% on average.

Figure 16 compares the performance of our different scheduling techniques when adding the memory synchronization technique defined in section 3.1.2. Under this scenario, we can achieve up to 31.8% performance improvement and 7.8% improvement on average.

It is important to reiterate that these performance improvements are relative to the worst-case scheduling performance among the orders we test, for a given pair of applications. We do not exhaustively search all possible orderings, but demonstrate that order does play a significant factor in concurrency performance, particularly as it relates to presenting greater opportunities for operation overlap. A more exhaustive experiment could easily be conducted by providing many more distinct random shuffle schedules. Furthermore, we conclude similarly to [33] that ordering can provide even greater performance gains than Hyper-Q alone, while also examining more potential orderings than in that work.

3.5 Discussion

We have presented a method for more effectively utilizing the CUDA Hyper-Q to improve overall performance and power efficiency on a set of applications. Our results demonstrate that leveraging the default concurrent techniques on the GPU can achieve greater performance when coupled with effective methods for reducing memory transfer contention and manipulating execution order through host-side scheduling approaches.

Our technique requires limited insight into the resource requirements of a particular application or application kernel, and relies on Hyper-Q to handle most of the execution concurrency. With these relatively simple adaptations, we demonstrated that we can achieve up to 59.1% improvement in performance over serialized execution. By utilizing our memory transfer

synchronization method, we found that performance could be improved by an additional 31.8% in certain cases.

Finally, we defined a management framework which is readily extensible for additional applications, and would be expected to demonstrate similar performance improvements. Because our technique does not rely on source code modification for effective resource sharing, there is less effort required to enable concurrency with new applications.

In the following chapters we will extend beyond the existing framework we have implemented and explore more robust methods for dynamically scheduling applications, i.e., managing the explicit synchronizations in GPU applications. These approaches are envisioned to support dynamic execution of streaming workloads, rather than a finite set, and are capable of learning specific system level objectives, such as greater throughput and lower power consumption.

Chapter 4 Optimizing Data Transfer Bandwidth using Reinforcement Learning

This chapter focuses on one of the often overlooked aspects of GPU sharing and utilization, the memory transfer engines. In most architectures, there exists only a single engine for each transfer direction, i.e., host-to-device or device-to-host. Furthermore, these engines are limited to processing a single request per direction at a time. For scenarios in which more than one thread and/or stream is present, this resource constraint can quickly become a limiting factor to optimal performance, as demonstrated in the previous chapter. We believe that by focusing on optimal utilization of the memory transfer engines, we can increase the degree of GPU compute resource utilization and execution overlap, ultimately leading to greater overall system-level performance. We begin with the motivation and goal of memory transfer optimization, and show that by simple transfer aggregation and division, we can improve the utilization of transfer bandwidth and achieve better performance at the application and system level. We then demonstrate that although the proposed memory transfer optimization technique is effective in many cases of concurrent kernel execution, it is sensitive to the setting of some hyperparameters, which would be determined by memory subsystem characteristics. To design an effective memory transfer optimization scheme usually requires detailed system characterization, which is time consuming. To improve its flexibility, we extend the technique by utilizing reinforcement learning methods to learn a dynamic policy that minimizes waiting time of the memory transfer tasks and increases bandwidth utilization.

4.1 Motivation

CUDA streams enable concurrent execution under three scenarios: (i) overlap of data transfer and kernel execution, (ii) overlap of independent kernel executions, and (iii) overlap of data transfers in opposite directions [11]. The level of concurrency possible in each case is further limited by the types or amount of resources that are currently available, compared to the type and amount that are necessary to support execution of the concurrent operations. Resources can be grouped in two main categories – compute and data transfer resources. The types of resources that are considered compute resources include threads, registers, and shared memory. The amount of each compute resource varies with the hardware generation. For data transfer resources, there is only one type – the direct memory access (DMA) engine – of which there are typically two per GPU. Each DMA engine supports transfer of data in only one direction, i.e. from host (CPU) to device (GPU), or vice versa.

Compute resources have varying levels of granularity, and every kernel requires a different proportion of the total resources based on the design of the algorithm. As a result, significant attention has been given to managing this granularity to maximize resource utilization [24] [27] [34]. On the other hand, the data transfer resources are either available or not. Due to fixed overhead, PCIe transactions with different payload size have different bandwidth utilization. However, there is no distinction between transactions that utilize the full PCIe bandwidth and those that utilize only a fraction. Certainly, this constraint can limit the ability to full exploit overlap opportunities, and thereby lead to underutilization of device resources. Yet, data transfer aware scheduling on GPU has only recently received attention and it is still somewhat limited [35].

Finally, even with the concurrency capabilities provided through CUDA streams, CKE is still largely affected by order-of-arrival since each of the underlying work queues is inherently

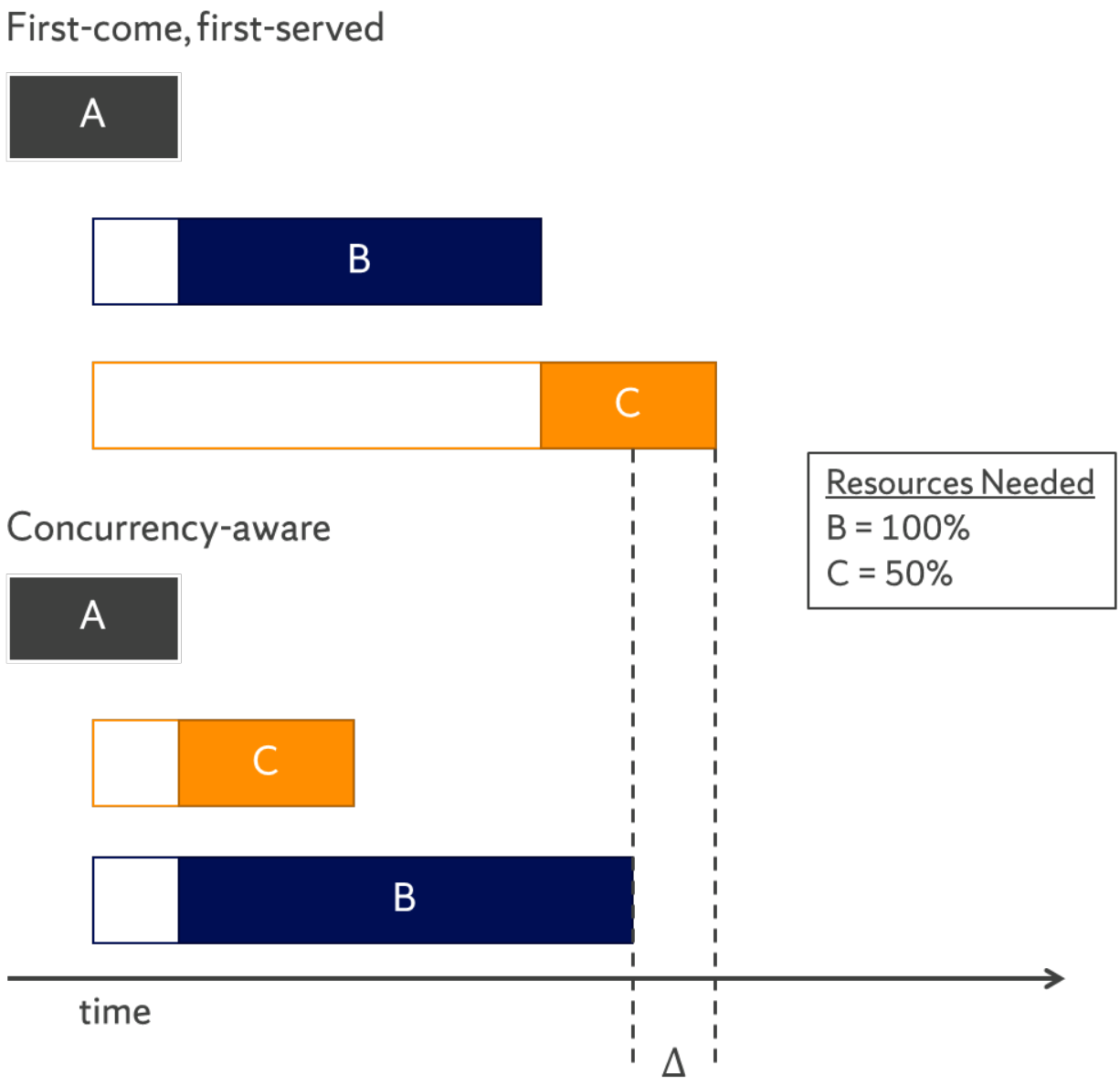

Figure 17. Example of non-commutative concurrency.

FCFS. This leads to the property of "non-commutative concurrency" described in [26], in which the order of execution of concurrent GPU operations affects the total execution time. Figure 17 gives an example of this situation. In the example, task A is already executing when tasks B and C arrive. We make the assumption that although both tasks are ready for execution, B arrives "before" C, i.e., B is enqueued on the ready queue before C. One potential factor that can contribute to the order precedence would be the respective data dependencies of each task – if data

for B is ready on the device before data for C. Thus, by the FCFS algorithm task B is served first. Since B consumes all compute resources, task C must wait until B completes before executing. Under a concurrency-aware approach, task C could be served first, consuming 50% of the resources and allowing B to begin executing with the remaining 50% of resources. Once task C completes the free resources will be assigned to B. The difference in schedules, Δ, reflects the non-commutative nature of concurrency behavior.

In this work, we address data transfer resource contention in order to mitigate explicit synchronizations on the copy engine. We seek for techniques that increase the concurrency of data transfer by better utilizing the memory transfer bandwidth, and at the same time exploit the non-commutative concurrency among the transfers. Our claim is that effective utilization of transfer bandwidth and reduction of task waiting time due to serialization on the copy queue will enable the additional measures to mitigate explicit synchronizations within the execution queue. The algorithm we propose is refined through the use of reinforcement learning techniques to achieve a dynamic approach which would be capable of adapting to workload characteristics on-the-fly.

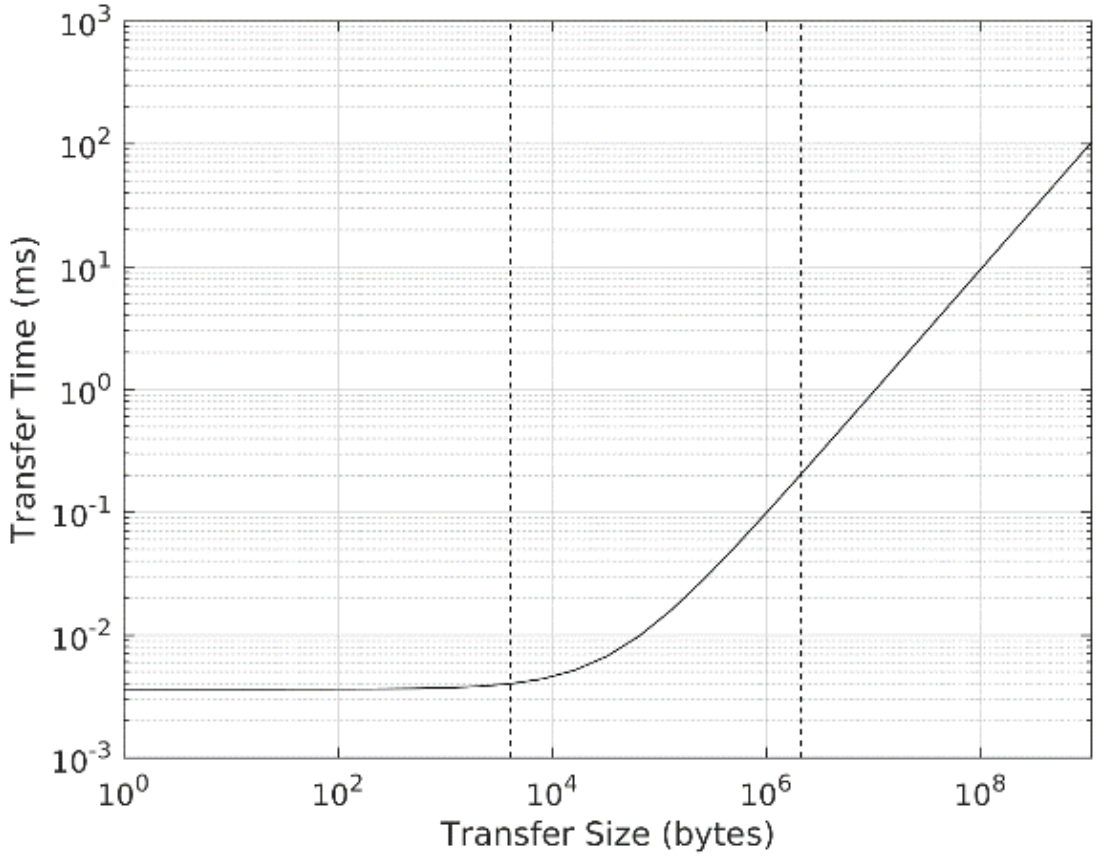

Figure 18. Memory transfer time relative to size of the transfer, for NVIDIA K40m GPU.

4.2 Bandwidth-Optimized Transfer Algorithm

The proposed memory transfer optimization algorithm is derived from the following observations of CUDA memory transfer performance in [36] which we have empirically validated for the NVIDIA K40, shown in Figure 18 and Figure 19. First, as noted in [36], data transfer execution time consists of two parts – a constant transfer overhead component (α) and a linear component dependent on the transfer size d of the transfer and the PCIe bandwidth (β). In Figure 18, we note that for small transfers, where $d \leq 4$ KB, the initial overhead to transfer data dominates the execution time and performance is constant in this region. Conversely for large transfers, i.e. $d \geq 2$ MB, the execution time is dominated by the bandwidth dependent component, when the

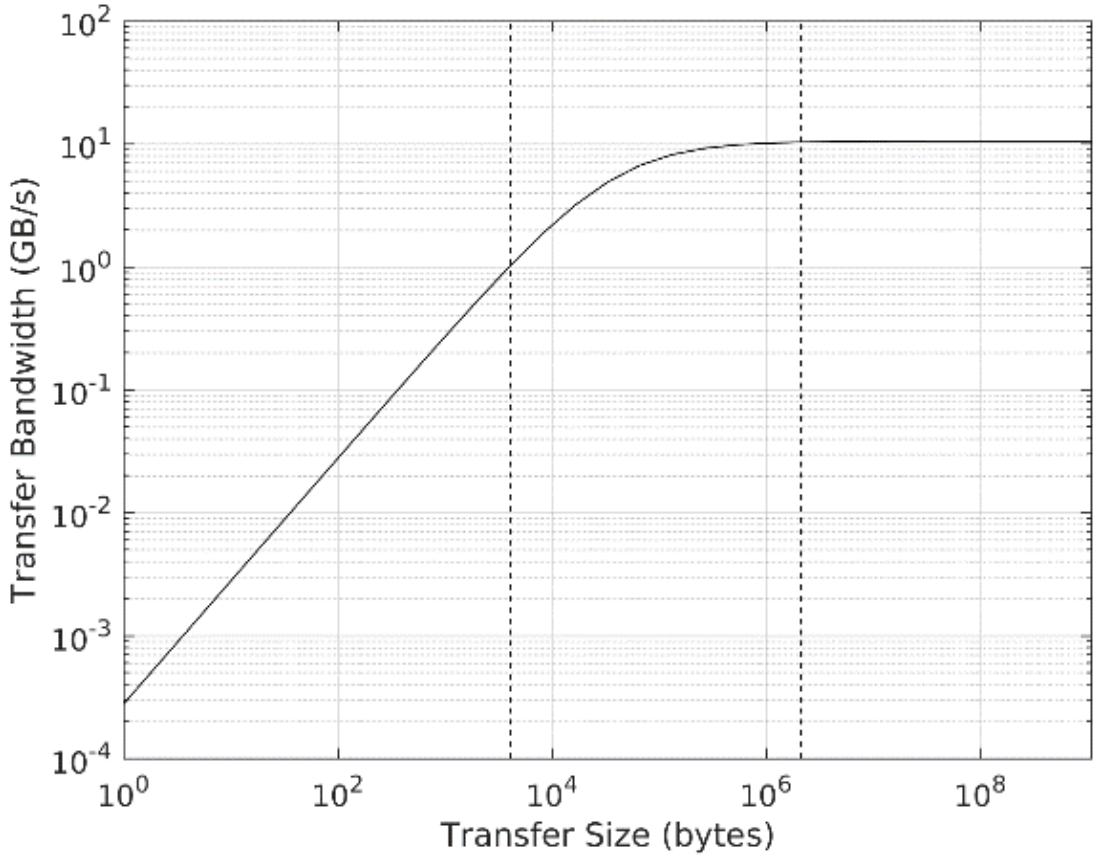

Figure 19. Memory transfer bandwidth relative to transfer size, for NVIDIA K40m GPU.

available bandwidth is saturated, resulting in transfer time scaling linearly with increasing *d* value. For the region between these points, the execution time scales slightly sublinearly and the available bandwidth is underutilized.

4.2.1 Algorithm Description

Listing 1 describes the proposed heuristic which exploits these observations. We categorize data transfers into one of the three regions mentioned above and execute the transfer accordingly to achieve (i) maximum transfer bandwidth performance and (ii) shorter overall response time, where response time is measured as the difference between completion time and arrival time.

Listing 1. Bandwidth-optimized Transfer Policy Algorithm.

```
Given: Qs, the send queue; Qa, the aggregate queue
Input: Transfer T, with property size (in bytes)
1   τ = 1 MB
2   ε = 0.1
3   if T.size ≥ (τ – ε) && T.size ≤ (τ + ε)
4      Qs ← T
5   else if T.size ≥ (τ + ε)
6      disassemble(T)
7   else
8      Qa ← T
9      aggregate()
10
11  method disassemble
12  nchildren = T.size / τ
13  for k = 1 to nchildren
14     create child transfer CT(k), CT(k).size = τ
15     Qs ← CT(k)
16
17  method aggregate
18  aggregateSize = ∑ q.size, for q ϵ Qa
19  if aggregateSize ≥ (τ – ε)
20     create aggregate transfer AT, AT.size = aggregateSize
21     while Qa is not empty
22        copy Qa.front to AT
23        Qa.pop
```

For each incoming data transfer, we inspect the size *d* and prescribe an action to take. Action selection is based on comparison of *d* to a threshold value, τ. Best practice guidance from Cook [37] suggests a threshold of 2 MB, however we set τ = 1 MB based on empirical analysis

which determined that transfers of this size achieve near-peak bandwidth at the lowest execution time. If $d = \tau \pm \varepsilon$, where ε gives a window around the threshold (we use $\varepsilon = 0.1 * \tau$), then the data transfer is scheduled for immediate execution. If rather, $d < \tau - \varepsilon$, the transfer is added to an aggregation queue. Once the sum of transfer sizes on the aggregation queue exceeds $\tau - \varepsilon$, the waiting transfers are combined into a single message of size D and scheduled for execution. Finally, if $d > \tau + \varepsilon$, we schedule the data transfer to be disassembled from a parent transfer of size d into k smaller child transfers of size (d/τ).

The foundation of these actions is based on the aforementioned properties of data transfer performance. The data transfer time can be modeled by a linear equation described in [36] and shown in (4-1), where $T(d)$ is the time to transfer d bytes, and α and β are hardware parameters. On the system we examine in this work, α is 4 μs and $1/\beta$ (i.e. the bandwidth) is 10.5 GB/s.

$$T(d) = \alpha + \beta d \tag{4-1}$$

The sum of the execution time of n individual transfers (4-2) is strictly greater than the total transfer time of an aggregate transfer (4-3).

$$\sum T(d_i) = (\alpha + \beta d_1) + (\alpha + \beta d_2) + \cdots + (\alpha + \beta d_n) = n\alpha + \beta \Sigma d_i \tag{4-2}$$

$$T(\Sigma d_i) = \alpha + \beta(\Sigma d_i) \tag{4-3}$$

Comparing the execution time $T(\Sigma d_i)$ of the aggregated transfer with the execution time $\Sigma\, T(d_i)$ of the non-aggregated transfer, the speed up ratio f can be calculated as in (4-4).

$$f = \frac{\sum T(d_i)}{T(\sum d_i)} = n - \frac{(n-1)\beta \sum d_i}{\alpha + \beta \sum d_i} \quad (4\text{-}4)$$

For significantly large transfers where d dominates α, the sum of execution times of disassembled child transfers will approximately equal the total execution time of the larger parent transfer. For all other cases, the ratio $\frac{\beta \sum d_i}{\alpha + \beta \sum d_i}$ is less than 1 and the speed up ratio f is greater than 1. The more significant the overhead α is compared to the transfer time $\sum d_i$, the higher the speed up ratio f will be. This motivates us to aggregate transactions with smaller transfer size, i.e. $d < \tau - \varepsilon$.

On the other hand, the disassembly approach breaks long transfers into smaller ones, to allow interleaving between different transfer requests initiated from different streams. It acts as a pseudo-preemption method to time-slice between transfers on different streams, which can reduce serialization effects [26]. The disassembly approach is only applied to transactions with large transfer size, i.e. $d > \tau + \varepsilon$, where the speed up effect of aggregation is negligible.

Preliminary analysis of the BWOPT policy demonstrated effectiveness under a broad range of execution conditions. However, it also highlighted areas where the policy caused degradation in performance relative to the default FCFS policy. First of all, its effectiveness depends on the selection of τ, as shown in Figure 20. In this example, we modeled two independent and concurrently executing applications and varied the size of the memory transfers for each application separately. We originally set τ = 2 MB, based on the bandwidth performance observations discussed and in keeping with CUDA programming best practices described by [37]. However, for workload combinations in which the independent applications had similar memory requirements consisting of transfers with d = 1 MB, this caused significant slowdown. In such a case, the aggregation approach causes the transfer queue to combine two 1 MB transfers into a single 2 MB transfer, which takes twice as long to transmit. This results in each program observing

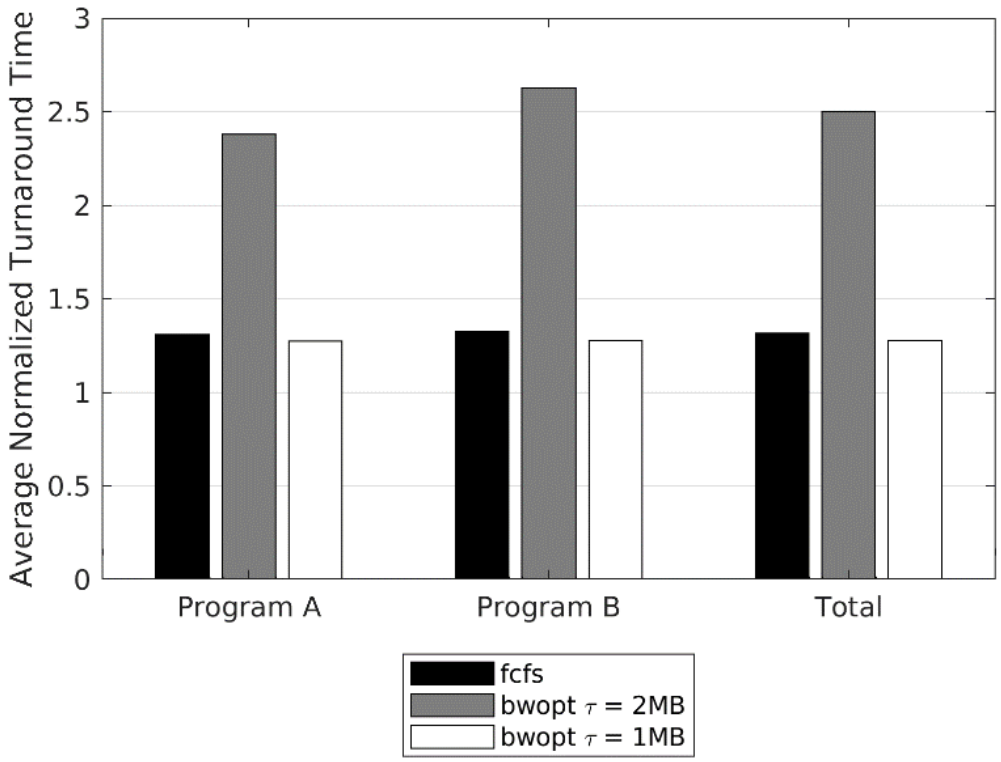

Figure 20. Average normalized turnaround time of two independent and concurrently executing applications under different transfer policies, including multiple values for BWOPT threshold, τ. Program A and Program B values represent the NTT for respective applications and Total represents the combined ANTT for both applications.

a turnaround time that is approximately doubled, as compared to the FCFS performance. In this case, the additional overhead of aggregation outweighs the potential gain, since there is very little additional bandwidth to be consumed. Note that when we lowered the threshold τ to 1 MB, we observed that the BWOPT outperformed FCFS. Full examination of the sensitivity of the threshold to workload memory demands was not conducted, however we believe that the above example provides sufficient motivation for exploring a dynamic approach.

We also note that the approach is particularly susceptible to the arrival rate of incoming transfer requests, as well as the size distribution of those transfers. The hidden assumption of aggregation method is that the extra latency for aggregation can be hidden by the large number of

available threads in GPU and its quick context switching. However, if the arrival rate of tasks is not sufficiently large, i.e. the period of time between transfer tasks exceeds the execution time of the waiting threads, then the aggregation overhead will become problematic. Furthermore, the size of transfers is an important factor, because in a limited period we may never receive enough transfer requests to reach the threshold for sending an aggregation. In either case, we need to early terminate the aggregation process and execute the memory transfer even though the transfer size is not yet large enough to reach optimal bandwidth.

To address these limitations and avoid hand-tuning of the algorithm for all possible scenarios, we apply the BWOPT algorithm within a reinforcement learning context using Monte Carlo methods for policy evaluation and iteration, which will be described in the following section.

4.2.2 Algorithm Implementation

The practical implementation of the BWOPT algorithm is enabled by the use of CUDA events. In particular, we use events as a mechanism to transparently synchronize execution from an application. For example, the code snippet in Figure 21 shows an example application code and our proposed HQ Runtime Environment method. In the original code, a programmer would utilize the standard CUDA API methods for handling data transfers and kernel execution. Within our environment, using a call intercept method which we will describe in greater detail later, we implement additional handling and synchronization to ensure valid execution. Specifically, the `cudaStreamWaitEvent` ensures that execution on the application stream does not continue until the data transfer has been completed.

In our implementation, the host data is copied into a separate aggregate copy buffer (`h_agg`). The aggregated buffer is eventually transferred once the size exceeds the packet

Application Code	HQ Runtime Environment
... cudaMemcpyAsync(d_a, a, num_bytes, cudaMemcpyHostToDevice, s1) my_kernel<<< g, b, 0, s1 >>>(d_a) cudaMemcpyAsync(a, d_a, num_bytes, cudaMemcpyDeviceToHost, s1) ...	*cudaMemcpyAsyncIntercept(cudaMemcpyHostToDevice)* cudaMemcpyAsync(h_agg, a, num_bytes, cudaMemcpyHostToHost, s_hd) cudaStreamWaitEvent(s1, e_agg_hd_complete) *cudaMemcpyAsyncIntercept(cudaMemcpyDeviceToHost)* cudaMemcpyAsync(d_agg, d_a, num_bytes, cudaMemcpyDeviceToDevice, s_dh) cudaStreamWaitEvent(s1, e_agg_dh_complete)

Figure 21. Memory transfer aggregation implementation.

threshold, at which point a local copy is made to transfer segments of the buffer to the appropriate device address (`d_a`). Once the aggregated transfer is copied from one side to the other, our implementation involves an additional call to `cudaMemcpyAsync` to make a local copy of data to the proper address location, where the cudaMemcpyKind is either `cudaMemcpyDeviceToDevice` or `cudaMemcpyHostToHost` as appropriate. The overhead of these copies is negligible for the size of transfers we evaluate, due to local memory bandwidth on the order of 200 GB/s.

The disassemble implementation is handled in a similar fashion, but does not require the additional memory buffer. Instead, the intercept method creates *m* `cudaMemcpyAsync` calls, where the source and destination addresses are relative to the original addresses `a` and `d_a`, respectively, and the `size` value is `num_bytes` / *m*.

4.3 Transfer Queue Model

In queuing theory, there are multiple system parameters which must be considered to reason about the performance of the queue design. Such parameters include the service policy (e.g. FCFS, earliest deadline first, etc.), the server capacity (e.g. single server, multiple servers) and the number of queues.

In this work, we focus on queue service policies, which is a software controllable knob, for queue optimization. Server capacity (i.e. the number of transfer channels) is not considered because it represents a hardware constraint and cannot be managed by software. Furthermore, this work focuses on single-GPU system, but we believe our technique can be applied in multi-GPU scenarios with some extensions.

There are many different types of service policies in queuing theory, including the FCFS approach used by GPU transfer queues. Other techniques include shortest-job-first (SJF) and processor sharing (PS). The former incorporates reordering of tasks based on shortest execution, while the latter serves multiple tasks simultaneously with each task receiving an equal proportion of the server capacity. In addition to introducing certain risks (i.e. starvation), SJF also causes non-commutative concurrency, hence we do not consider it as a plausible service policy for GPU data transfers.

The approach we consider is an extension of the PS technique. While the transfer engine constraint limits execution to a single data transfer per direction at a given time, there is no theoretical limit to the size of the data transfer (there is a practical limit given the memory capacities of host and device). Instead, we redefine capacity in this case to be in terms of the available memory transfer bandwidth. Note, that while PS techniques typically give equal shares

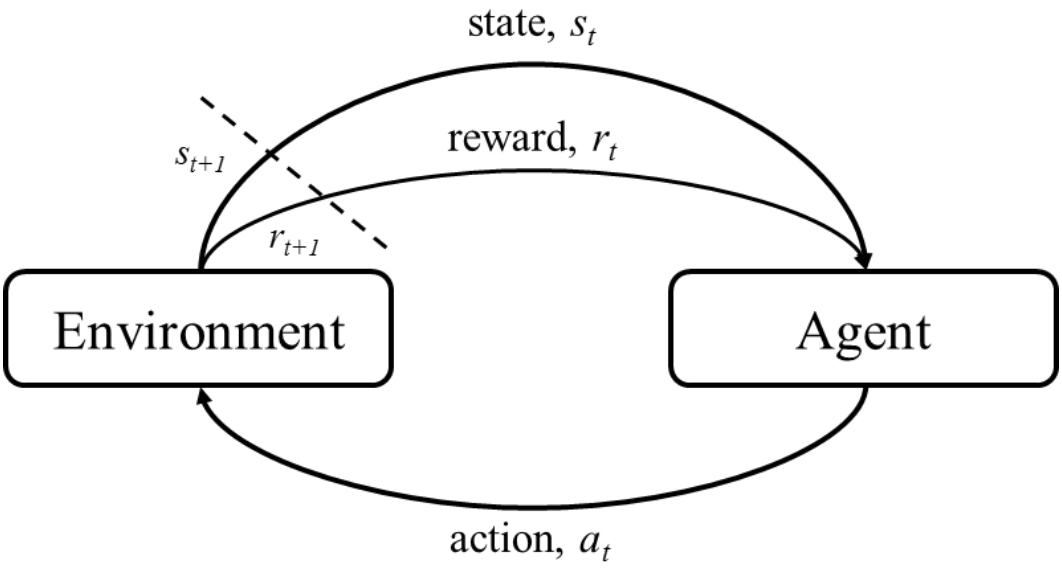

Figure 22. Reinforcement learning problem.

to each task, our approach does not necessarily allot an equal share of the total packet size to each task.

4.4 Q-Learning

All reinforcement learning problems have a basic structure as represented by Figure 22 [38], where an agent interacts with an environment to achieve some objective or goal. In particular, at time t, given some representation of the environment state, s_t, the agent chooses an action, a_t, in order to meet or make progress towards the goal. A measure of that progress is given through a reward value, r_t. Upon taking the action, the environment transitions to a new state, s_{t+1}, which is communicated to the agent. This process continues for subsequent time steps, with the agent receiving updated state information and reward feedback from the environment, until the agent has met its objective or some other terminating condition is met.

The method by which an agent chooses actions is called the policy, π. Using a trial-and-error approach indicative of model-free approaches, the agent seeks to learn the optimal policy,

π*, and thus the optimal action-value for a given state. This is referred to as Q-learning [39], where the objective is to estimate the function *Q(s,a)* for π. *Q* represents the value (or, quality) of taking action *a* from state *s*, and following π from then on. This estimate is given as the expected return under those conditions, as shown in (4-5) from [38].

$$Q^{\pi}(s,a) = E_{\pi}\{R_t | s_t = s, a_t = a\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma r_{t+k+1} | s_t = s, a_t = a\} \quad (4\text{-}5)$$

Q-learning provides an approximation of the optimal policy, π*, without explicitly following the policy and provides faster convergence. The additional benefit to a model-free approach is that it does not require an explicit model of the environment and is therefore flexible under many conditions.

4.4.1 Monte Carlo Policy Evaluation and Iteration

Monte Carlo (MC) reinforcement learning methods provide a model-free technique through which we can learn state-action transitions and subsequent rewards through experience rather than a known Markov Decision Process [38]. MC methods are episodic in nature, meaning that an episode of experience must be completed before the rewards associated with each state-action transition can be computed. This is helpful in our technique, because there is no guarantee of an immediate reward for each action.

State: Each transfer request arrival is considered to be a new state (i.e. decision epoch) in the set of states, *S*. We describe the state as a triple of (d_t, d_q, ρ) where d_t is the data transfer size in bytes, d_q is the sum of the size of transfers on the aggregation queue in bytes, and ρ is the status of the transfer channel resource (i.e. 0 = free, 1 = busy). For both d_t and d_q, we discretize the values by *log2*, in order to avoid state-space explosion.

Actions: At each state, we select from a set of four actions, derived from the BWOPT algorithm described above. Let the set of actions, $A = \{send, hold, aggregate, disassemble\}$. The *send* action means to schedule the data transfer for immediate transfer. The *hold* action adds the data transfer to the aggregation queue, while the *aggregate* action adds the transfer to the queue and then pops the queue on to a larger aggregate transfer which is then sent. Finally, the *disassemble* action splits the data transfer into k smaller transfers and schedules each smaller packet for immediate transfer.

Reward: We define the reward for a data transfer to be equal to $1/T_w$, where T_w is the waiting time of the transfer. Waiting time is defined as the difference between the service time, t_s, and the arrival time, t_a, i.e. $T_w = t_s - t_a$. In the case when $T_w = 0$, we penalize the system by setting the reward value to -1, since we seek to discourage immediate sending in case it does not maximize bandwidth. When calculating the return at each state, we use a discount factor $\gamma = 0.9$, which has the effect of weighting long-term rewards more heavily and results in returns which seek to minimize the waiting time. From a system perspective, this results in reduced average waiting time and increased bandwidth utilization due to few larger transfers, whereas nearsighted returns favor smaller immediate transfers and behavior more equivalent to FCFS.

***Policy*:** Our initial policy, π^0, is a deterministic policy, such that $\pi(s) = a$, for some $s \in S$, $a \in A$, and is based on the BWOPT policy described above. However, we use ε-soft MC methods in order to explore new policies and escape local optima. Using the ε-soft approach allows for more exploration of states to find optimal action values and maximize total reward. With this approach, we will take the greedy action ($a = argmax_a\ Q(s,a)$) with probability $1 - \varepsilon + \varepsilon / |A|$ and some other action with probability $\varepsilon / |A|$.

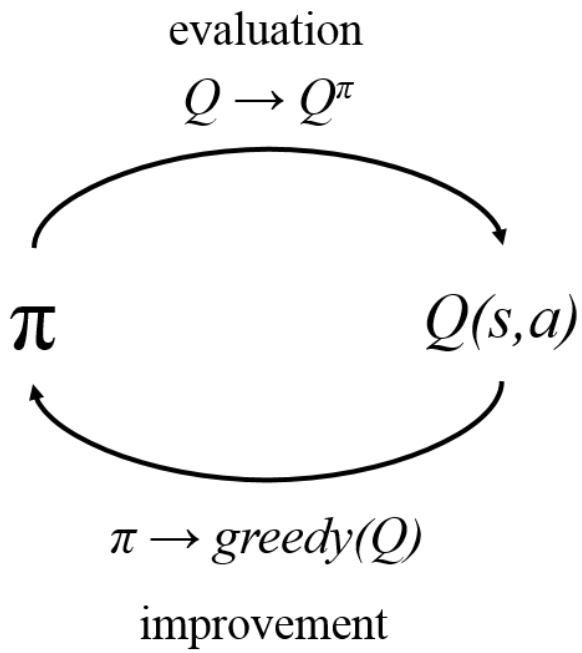

Figure 23. Generalized policy iteration and evaluation.

The objective of policy evaluation and iteration (Figure 23) is to continuously update our estimate of *Q(s,a)* using (4-6) based on experience and update π accordingly until we reach the optimal policy, π^*.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a_t) - Q(s_t, a_t)) \quad (4\text{-}6)$$

4.5 Evaluation

In this section we describe the experimental setup used to evaluate our proposed MC-based approach and then present the results. We conclude with a demonstration of the use of our derived optimal policy on a real workload.

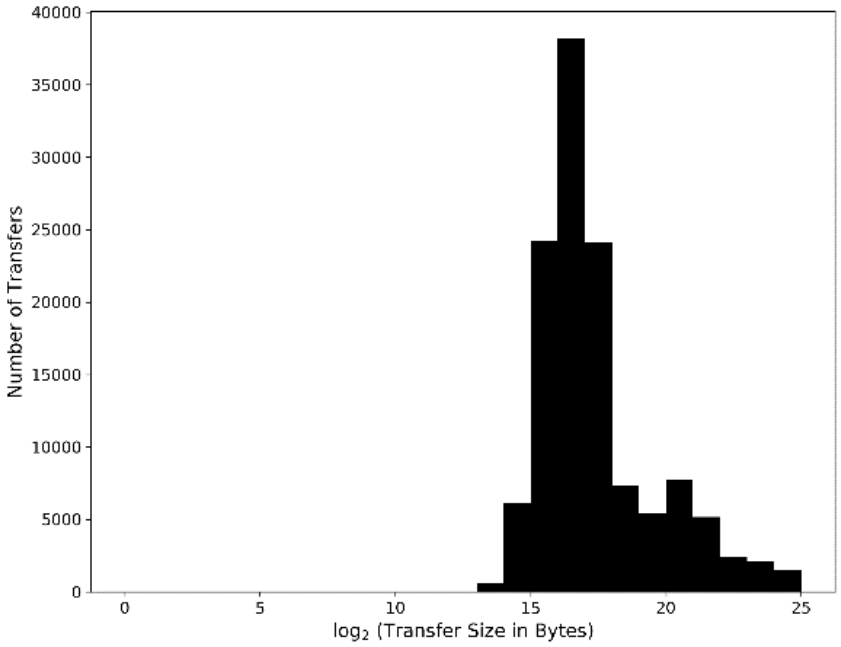

Figure 24. Distribution of simulated transfer sizes for all episodes in simulation.

4.5.1 Simulation Setup

Our simulation models only the host-to-device (*HtoD*) transfer engine. For simplicity, and without loss of generality, we do not consider the device-to-host (*DtoH*) transfer engine in this work. However, since the *DtoH* engine exhibits the same performance properties as *HtoD*, and because data transfers in opposite directions is a condition under which concurrent execution performs, we expect the results demonstrated for *HtoD* to be likewise applicable to *DtoH*.

We began by emulating real application traces as independent Poisson processes. Using parameters of the application profiles listed in Table VII, we generated a workload trace of each type, i.e., three independent traces. Within each type trace, the transfer task size is drawn from a Gaussian distribution, where we are randomly generating the exponent value and all sizes are powers of 2. The three traces are merged and sorted (by arrival time) into a single workload trace

consisting of 125 transfer requests per episode of our simulation, and we generated 1000 such episodes. The resulting distribution of transfer sizes for all episodes is shown in Figure 24.

Table VII. Workload Trace Generation Parameters.

Type	**Arrival Rate**	**Transfer Size ($d = 2^y$)**
I	$\lambda = 100$	$\mu = 13, \sigma = 1$
II	$\lambda = 20$	$\mu = 20, \sigma = 1$
III	$\lambda = 5$	$\mu = 24, \sigma = 1$

4.5.2 Policy Evaluation and Iteration

The FCFS and BWOP policies can be categorized as fully deterministic policies, so a single iteration of each trace is sufficient to characterize the respective performance. On the other hand, we require multiple iterations of the Monte Carlo approach in order to benefit from the learning on previous traces.

We begin with a semi-deterministic policy, which is based on the BWOP policy, however we add ε-soft control to the algorithm to allow exploration of state-action pairs in the early stages of evaluation. We simulate a new episode given an initial workload trace and apply the current policy. After we have completed evaluation of the episode, we update our estimate of $Q(s,a)$ and update π. This procedure repeats for all 1,000 traces to obtain a reasonable exploration of potential state-action transitions.

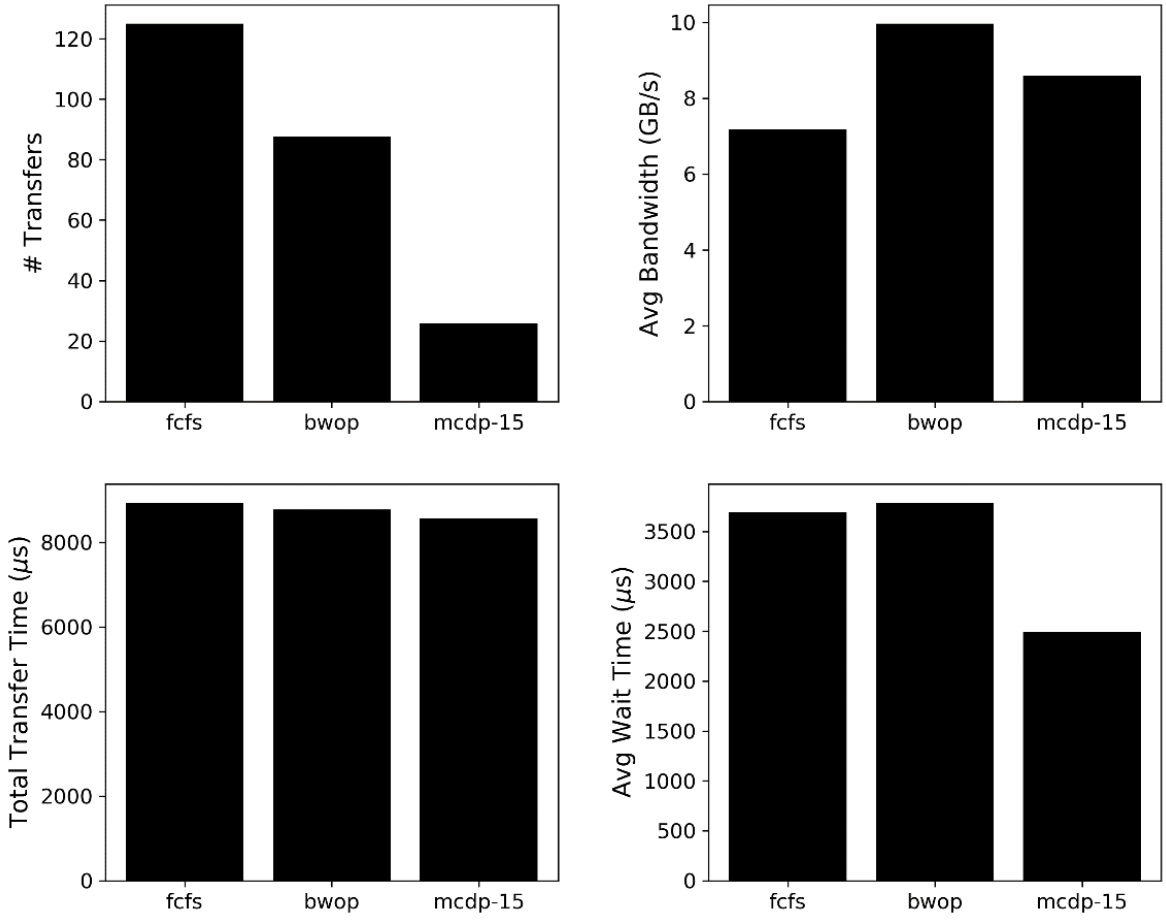

Figure 25. Comparison of transfer scheduling algorithm performance. Deterministic algorithms *fcfs* and *bwop* are compared to the result after 15 iterations of our Monte Carlo reinforcement learning method, *mcdp*. (top left) Total number of transfers sent. (top right) Average achieved transfer bandwidth. (bottom left) Sum of transfer durations for actual transfers sent. (bottom right) Average per-task waiting time.

Each set of 1,000 episodes constitutes one iteration. On the first iteration, $\varepsilon = 0.2$ and we reduce this value by 10% on each subsequent iteration, in order to balance exploration with exploitation as we converge on the optimal policy. We initially ran this procedure out to 50 iterations, but found that the performance converged on an optimal solution after 15 iterations and present our results to this point. Additional iterations exhibited signs of overfitting to prior experience.

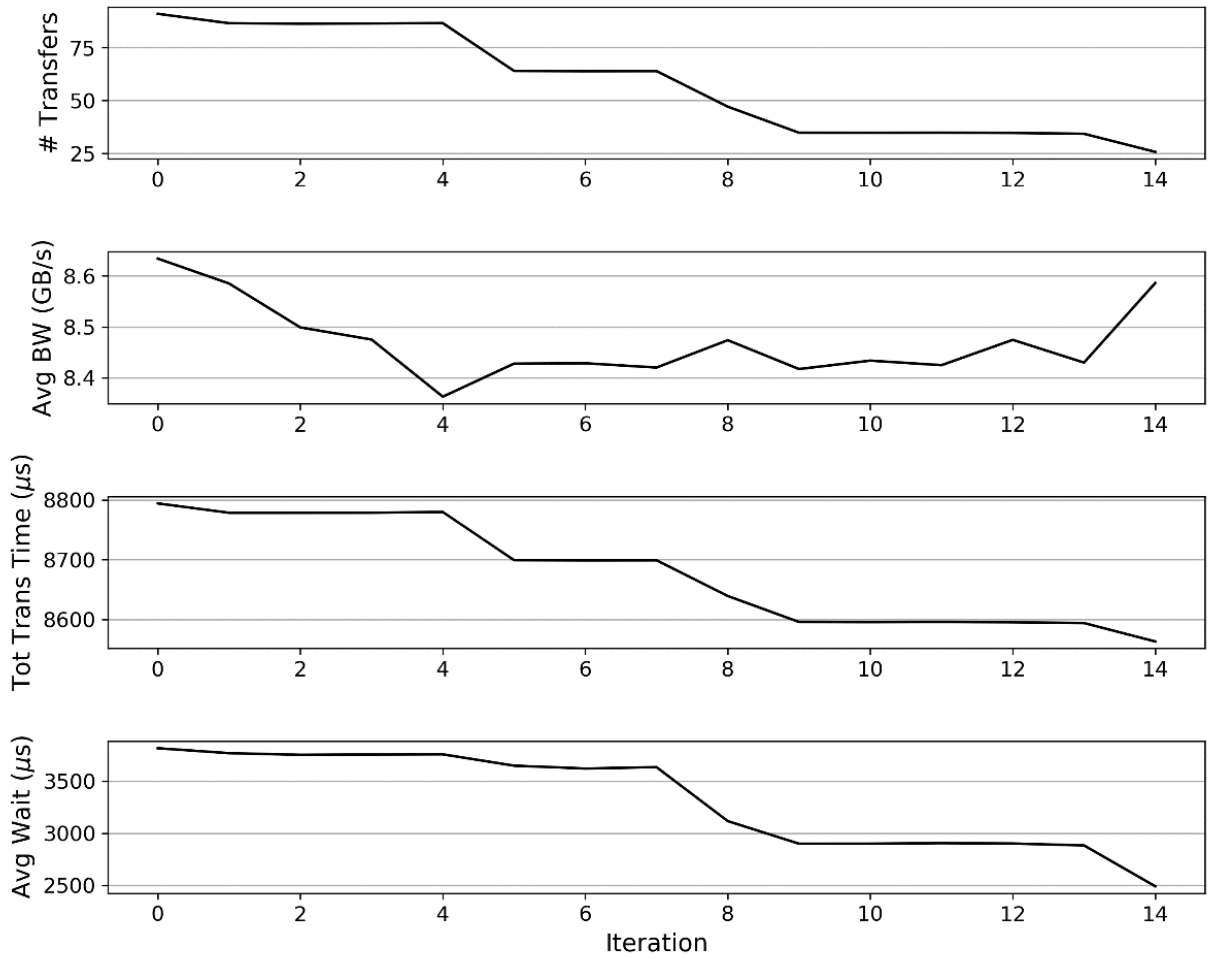

Figure 26. Evolution of performance metrics during MC learning process.

In Figure 25, we compare the performance of the three tested transfer policies across multiple various system and task metrics. The first observation is shown by the plot in the upper left, which indicates that the number of transfers sent is significantly reduced under the MC-DP policy. Each episode contains 125 transfer requests, and the FCFS policy serves each request separately. The aggregation and division methods in the BWOP and MC-DP policies allow for fewer, larger transfers to be sent overall which simultaneously results in near-optimal utilization of the transfer bandwidth, as shown in the upper right plot. The MC-DP policy provides 19.6% improvement on bandwidth utilization over the FCFS policy.

The lower left plot of Figure 25 provides an empirical validation of our earlier claim in (4-4) that serves as the basis of our transfer policy. Specifically, the aggregation of many smaller transfers into few larger transfers results in an overall reduction of 4.1% of the total time to complete the transfers in the episode. More precisely, the FCFS total transfer time can be determined using (4-2), while the MC-DP policy is approximately equivalent to (4-3).

Finally, the lower right plot of Figure 25 demonstrates perhaps the most critical improvement, where we compare the average waiting time. Recall, waiting time is the difference between the arrival time of the task and the time it is served. For FCFS, waiting time is a measure of resource-occupancy. On the other hand, for the BWOP and MC-DP policies waiting time also measures the time spent waiting for aggregation. In BWOP, this could approach an infinitely long time since we provide no guarantee that the aggregation queue will ever reach the threshold. We can mitigate this with a timeout factor, though we do not implement that in simulation. Rather, at the end of each episode, we immediately send any waiting aggregation transfer. MC-DP offers the advantage that we can learn a different threshold for sending the aggregation depending on prior experience and the dynamics of the workload. As a result, we observe 32.6% reduction in the average waiting time for MC-DP policy. This is significant for shared execution scenarios, since it can allow us to overcome non-commutative concurrency by enabling multiple independent kernels access to the GPU compute resources simultaneously. Whereas in other techniques which manage compute resource-sharing, data transfer dependencies could limit the effectiveness of the approach, application of the MC-DP transfer policy in combination with those techniques could lead to significant system efficiency. Figure 26 shows the evolution of the performance metrics discussed above over the 15 iterations of our MC-DP learning process.

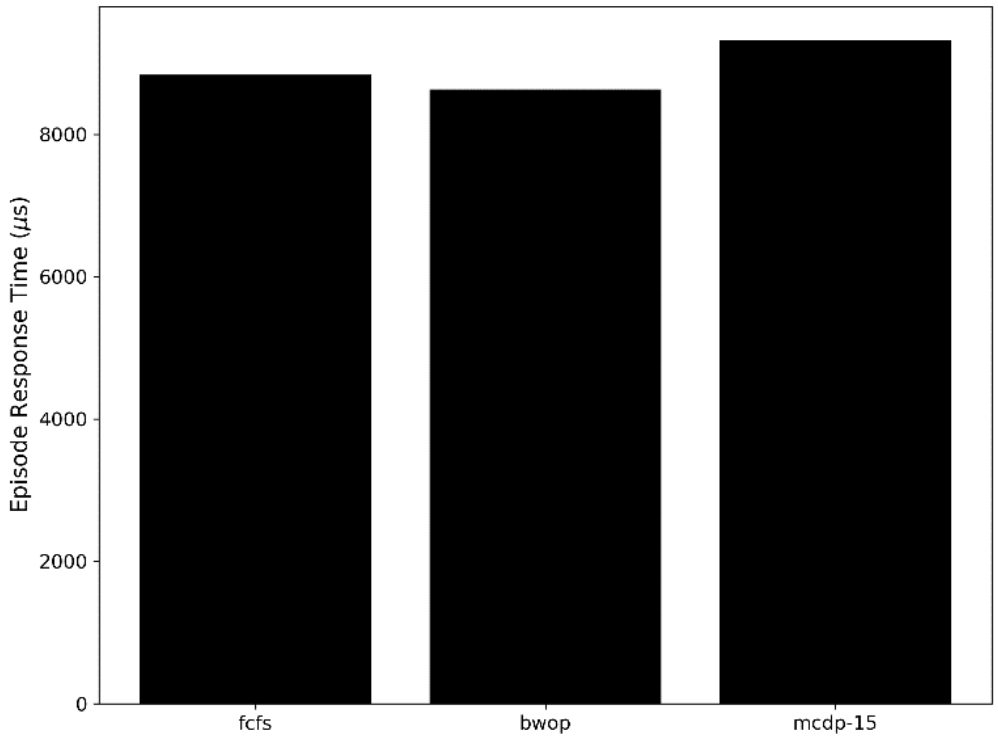

Figure 27. Episode response time for tested transfer policies.

The gains discussed above do not come without some cost to system performance. In particular, as shown in Figure 27, the effect of our MC-DP policy, in particular the aggregation of transfers resulting in significant reduction of total transfers served, is to increase the episode response time. Response time is a measure of the difference between the time of completion of the tail task and the time of arrival of the leading task. However, we believe the increase in response time of 5.5% is outweighed by the improved bandwidth utilization and, more significantly, the reduced per-task waiting time.

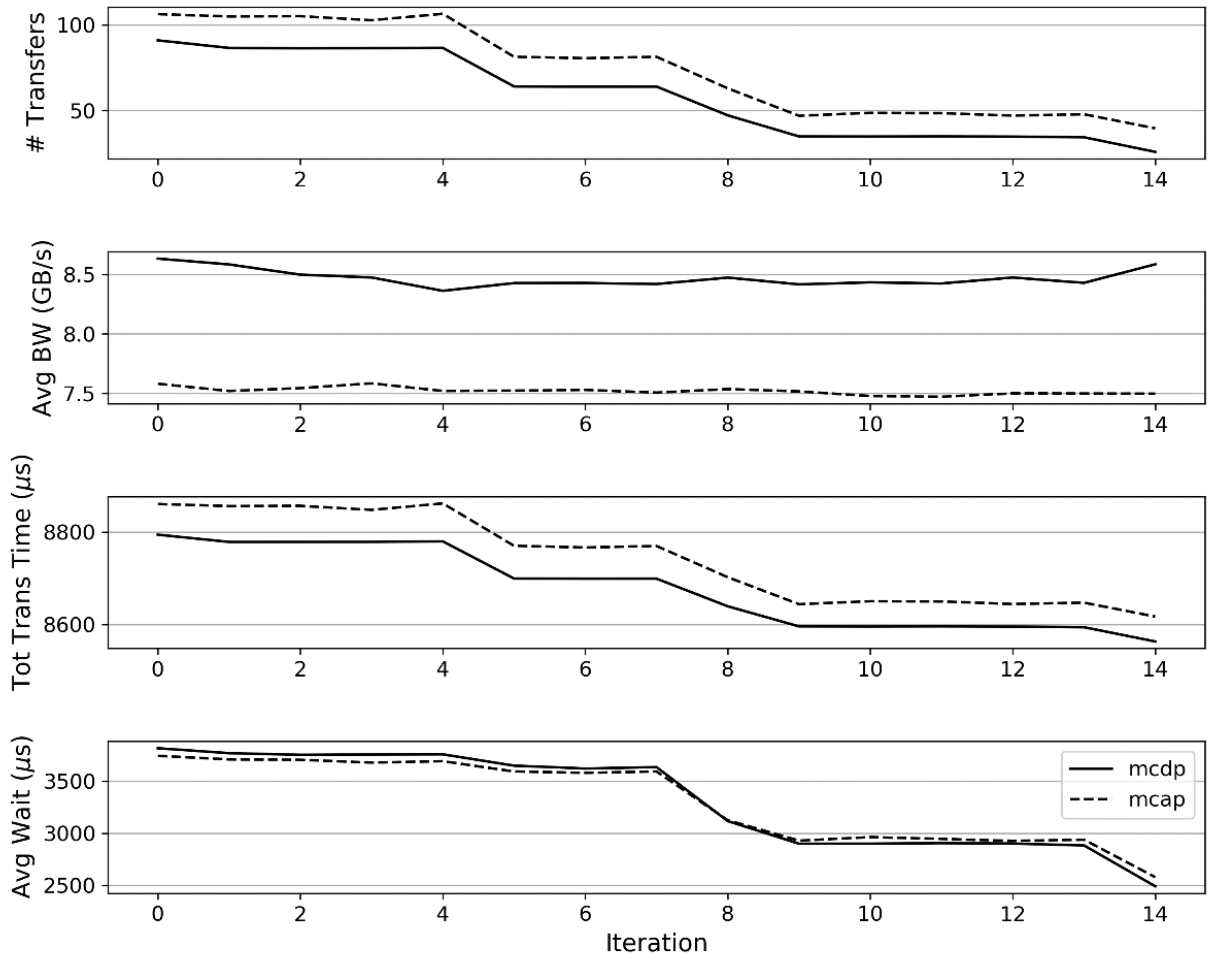

Figure 28. Comparison of performance metrics during MC learning process using initial policy definitions of deterministic (mcdp) or arbitrary (mcap).

4.5.3 Policy Improvement

To demonstrate the general effectiveness of the Monte Carlo approach to transfer optimization, we repeated the policy evaluation and iteration experiments using an arbitrary initial policy. Unlike MC-DP, we did not precondition the policy with the BWOP heuristic. Rather the Monte Carlo – Arbitrary Policy (MC-AP) was initialized with each action having equal probability of being selected at each state.

Similar to MC-DP evaluation, we found that MC-AP converged on optimal performance at 15 iterations. Figure 28 shows the comparison of MC-DP and MC-AP metrics and show that

while the MC-DP performance exceeds MC-AP in each case, the arbitrary case follows the same trend in learning with each iteration. These results suggest that we can apply the MC-AP approach on any new system that has not been previously characterized, as we have done in order to initialize MC-DP. This saves significant time, while offering similar performance gains relative to the other policies.

4.6 Conclusion

In this work we describe a method for improving the system throughput of GPUs under shared execution scenarios by optimizing the data transfer methods. In particular, we describe a heuristic which seeks to optimize the peak bandwidth of a known system, in an effort to improve performance relative to the traditional FCFS approach to reduce the waiting time of pending data transfers. We then improve on this approach using reinforcement learning techniques to overcome limitations in the heuristic. Our results demonstrate that the MC-DP approach increases the bandwidth utilization while significantly reducing the average waiting time for incoming transfers. As a result, our transfer optimization approach would complement previously studied GPU compute-resource sharing techniques.

Finally, we demonstrate that the absence of system characterization is not a limitation when using the Q-learning approach, as we are able to achieve comparable performance improvement starting from an arbitrarily-initialized policy.

We believe that this is the one of the few proposed techniques for increasing GPU shared-execution efficiency through data transfer-aware methods, and is likely the first which applies reinforcement learning techniques to optimize bandwidth utilization through aggregation strategies.

Chapter 5 A Deep Q-Learning Approach to GPU Task Scheduling

The field of high performance energy-efficient embedded computing (HPEEC) brings the consideration of power consumption onto a level playing field with application performance [40]. HPEEC dictates that power consumption receive equivalent priority as an evaluation metric through a variety of approaches, including hardware- and software-based techniques. The objective of such techniques is often more nuanced than simply reducing the overall power consumption; indeed, that is but one possible solution. However, to be energy-efficient need not mean consuming less power overall, but rather to be judicious about that power which is consumed, i.e., to ensure that none is wasted. HPEEC is critical to operational success across a variety of domains, including in defense applications for both air [41] [42] and space [43] [44] platforms. Such applications have both (near) real-time throughput requirements and hard power limitations.

The introduction of concurrent kernel execution (CKE) enables the simultaneous use of GPU resources by multiple independent application kernels to ensure higher resource utilization [45] [46]. However, the current CKE task scheduling approach is simplistic and quite often suboptimal. In this work, we propose a novel CKE task scheduling algorithm that simultaneously optimizes the system throughput and GPU resource utilization. Our method uses a deep reinforcement learning (DRL) approach called deep Q-learning to learn from experience, such that we find a near-optimal scheduling order for a set of applications. We compare our approach to multiple scheduling algorithms, including a dynamic programming-based approach [47] which similarly seeks to solve a multi-objective optimization problem which we frame below. In addition

to achieving a more optimized scheduling order, our approach is also more readily adaptable to changing dynamics in an HPEEC system because of its DRL design.

5.1 Task Scheduling

Traditionally, task scheduling algorithms (i.e., for CPU) have been designed with simplicity, fairness, and/or throughput in mind [48]. The First-Come-First-Served (FCFS) algorithm is an example of one which is simple to implement and understand, but has certain shortcomings depending on the types of tasks in the scheduling queue. Preemptive scheduling algorithms, such as Round-Robin (RR), exhibit fairness by ensuring that each task gets a predetermined time slice to execute before giving up control to another task in the queue. Throughput is a primary consideration for algorithms like Shortest Job First (SJF), which prevents short tasks from waiting in the queue for disproportionately long periods of time. However, even this approach has unintended negative consequences, for the case of arrival of new short-duration tasks to the queue in which a single long-duration task is waiting.

On the other hand, very little attention has been given to the multiple objectives of throughput and utilization for GPU task scheduling. The architectures studied here, i.e., NVIDIA GPUs, are known to adopt the simpler scheduling approach of FCFS, while providing some mechanisms for user-specified priority scheduling, e.g., CUDA stream priorities [11]. However, prior work has motivated the need for a more nuanced approach, in order to both achieve optimal throughput and to fully utilize the GPU.

Keeping in mind certain understood constraints about the manner in which tasks are selected for execution [15], we define a scheduling algorithm which utilizes deep Q-learning to converge on a near-optimal solution. The Q-learning is a model-free approach to finding an

optimal policy by finding the expected reward for a given state-action pair [38]. Using experience observations, one builds out a matrix of expected future reward values when taking action *a* from state *s*. This value defines the quality of that action (thus, *Q*). Deep Q-learning combines deep neural network approaches with the traditional Q-learning method in order to learn the expected value, *Q(s,a)*, from a subset of prior experience observations.

By utilizing a deep Q-learning approach, our algorithm is expected to avoid certain performance limitations which come with prescribed approaches as discussed above. Furthermore, this algorithm is amenable to new hardware configurations and new types of tasks because it learns from experience.

5.1.1 Problem Formulation

Let *D* be a GPU device with N_{sm} symmetric multiprocessors (SMX). Each SMX *j* has an identical amount of compute resources, r_j. In particular, we focus on four specific resources: the number of resident thread blocks (r^b), the number of resident threads (r^t), the number of registers (r^r), and the number of bytes of shared memory (r^s). Therefore, for each SMX *j* we represent the multiprocessor resources as a tuple, $r_j = (r_j^b, r_j^t, r_j^r, r_j^s)$. Then the total amount of compute resources for *D* can be given as $R_D = N_{sm} * r_j = (R_D^b, R_D^t, R_D^r, R_D^s)$.

Device *D* also has a ready-task queue, $K = \{k_0, k_1, \ldots, k_{n-1}\}$, where $|K| = n$. Each task k_i can be represented by a tuple which describes the task's compute resource requirements as defined above, as well as the estimated execution time under unlimited resources. Therefore,

$$k_i = \left(r_{k_i}^b, r_{k_i}^t, r_{k_i}^r, r_{k_i}^s, t_{k_i}^e\right) \tag{5-1}$$

Compute resource requirements are easily determined from the task specification in code. Estimated execution time can be determined through a variety of methods, including empirically-derived methods. For this research we used the profiling results from execution of the selected tasks when running with exclusive device access to verify the task characteristics.

In general, we assume that the order of tasks in *K* represents the order-of-arrival, i.e. given *arrival time* t_{a_i},

$$t_{a_0} < t_{a_1} < \cdots < t_{a_{n-1}} \tag{5-2}$$

Typically, arrival time would be the primary determining factor in selecting the next task for execution, as in the FCFS algorithm. On the other hand, the SJF algorithm only considers the estimated execution time for making task selections. Our algorithm relaxes the above assumptions, and allows for any ready-task to be selected for execution.

Each task also has a dispatch time t_{d_i}, i.e., the time which the task's first thread block is dispatched to the device, and a completion time t_{c_i}, i.e., the time which the task's latest thread block completes execution on the device. These values are necessary for computing waiting time and execution time, and are used in calculating performance metrics described below.

We use the metrics introduced in Chapter 2 to measure the effectiveness of the evaluated scheduling algorithms, namely system throughput (STP) and average normalized turnaround time (ANTT) for the application performance metrics, and our own metric to measure device utilization using the average function value [49]. Given our definition for *Util*, to calculate the average device utilization over the scheduling window we find the average function value for utilization during

the period between time T_0 and time T_1 using (5-3), where T_0 is the dispatch time of the first task and T_1 is the completion time of the latest task.

$$Avg.Util = \frac{1}{T_1 - T_0}\int_{T_0}^{T_1} Util_D(t)dt \tag{5-3}$$

We compare each scheduling algorithm across all three metrics in experimental results given below.

5.2 Deep Reinforcement Learning

When attempting to apply reinforcement learning techniques to problems with high-dimensional state spaces or complex feature representations, many traditional techniques fail to scale. The introduction of the combination of deep learning and reinforcement learning by Mnih, et al., [50] has spawned a new subdomain of reinforcement learning called deep reinforcement learning (DRL). In DRL, the intelligent agent uses deep learning methods to learn the optimal action-value function, Q(s,a). Such models are called deep Q-networks, and the process in general is referred to deep Q-learning.

In deep Q-learning, the similar setup to a traditional Q-learning problem exists, as described in Section 4.4. The difference resides in the method by which Q is approximated. Whereas in traditional Q-learning, we find the expectation over observed returns for an action taken from a given state, in deep Q-learning the neural network model finds an approximation that maps input states to output actions based on training from prior experience transitions. In particular, we use the concept of experience replay, which stores a history of prior transitions of the form (s_t, a_t, r_t, s_{t+1}). Each training iteration selects a random subset of the prior experiences to

update the model approximation. Action selection by the agent is informed by the policy network, which is an instance of the learned deep Q-network.

5.2.1 Deep Q-Learning Model

In structuring the problem within a reinforcement learning context, we define the *agent* as the task scheduling engine and the *environment* to include the GPU resources and ready queue. Below we describe how we define the state, actions, and reward in our implementation.

State: The state of the environment consists of the current resource utilization on the GPU and the resource requirements of waiting tasks in the queue.

The device state is given by aggregating the utilized resources from each SMX at the device level. Then, the state values represent the percentage of maximum resident resources which are in use at a given system time. Thus, in our implementation the device state values lie within [0, 1].

$$S_{device} = \left[\frac{\sum_j r_j^b}{R_D^b}, \frac{\sum_j r_j^t}{R_D^t}, \frac{\sum_j r_j^r}{R_D^r}, \frac{\sum_j r_j^s}{R_D^s}\right] = [u_b, u_t, u_r, u_s] \tag{5-4}$$

The task state values represent the percentage of maximum resident resources being requested. In many cases, the resource requirements may exceed the amount of maximum resident resources, i.e., a condition known as oversubscription. Such a situation does not mean that a task cannot execute on the chosen device. Rather, it requires that the task execute in waves, i.e., a series of thread block dispatches which fully exhaust one or more resources. Values greater than 1.0 indicate an oversubscription condition.

$$S_{k_i} = \left[\frac{r_{k_i}^b}{R_D^b}, \frac{r_{k_i}^t}{R_D^t}, \frac{r_{k_i}^r}{R_D^r}, \frac{r_{k_i}^s}{R_D^s}\right] \tag{5-5}$$

The environment state, *S*, is represented as a concatenation of the device state vector and the task state vectors for each waiting task in the ready queue.

$$S_{env} = [S_{device}, S_{k_0}, \dots, S_{k_{n-1}}] \tag{5-6}$$

Note that each task can be in one of three conditions: waiting, dispatched, or completed. The *waiting task* is one which is ready to execute but has not be selected for execution. The *dispatched task* is one which has had some number of thread blocks dispatched for execution, where the number of blocks dispatched is less than the total number of blocks. The *completed task* is one which has had all thread blocks dispatched and all blocks completed execution. Dispatched tasks remain on the ready queue, while completed tasks are removed (and the subsequent state for that slot is a zero-tuple). A task which is fully dispatched but not fully completed is still treated as completed for the purposes of scheduling.

Actions: Action selection identifies which of the waiting tasks should be dispatched to the device next. Therefore, the set of valid actions can be viewed as $A = \{0, 1, \dots, n-1\}$, where $n = |K|$. The action value represents the slot in the ready queue from which the next task should be drawn.

We include one unique condition for action selection, in abiding by the GPU scheduling rules as enumerated in [15]. If a task has begun execution by having a subset of thread blocks dispatched to the device, that task moves to the head of the ready-queue and becomes the only valid action selection until all blocks have been dispatched (i.e., rules G3 and X1).

Reward: Finally, we define reward through a combination of the device utilization and turnaround time of waiting tasks. Both of these values are calculated with special attention to the effect of certain scheduling decisions on the composition of an execution wave. In particular, for any kernel

in which one or more resource requirements exceeds the maximum resident capacity, execution is divided up into multiple waves. For such cases, we consider the cumulative effects on utilization and turnaround time in our reward, rather than only instantaneous effects from the next wave period.

Therefore, device utilization $Util_D$ is measured as the average utilization of total resources over the number of execution waves, and $Util_D \in [0, 1]$. We update the definition of turnaround time to include the cumulative effect of waiting and call this *effective turnaround time* (ETT). First, we measure the estimated turnaround time of each waiting task k_w by adding the run time of the selected action to the waiting time of k_w. Then we recalculate an estimated normalized turnaround time (NTT_{est}) for k_w. Finally, we calculate the $ANTT_{est}$ value for the set of waiting tasks by averaging the per-task NTT_{est} values. This value is then normalized to restrict it to a [0, 1] range similar as utilization. We use the results from FCFS experimentation to determine the minimum and maximum limits for ANTT normalization.

Since device utilization is a higher-is-better metric and turnaround time is a lower-is-better metric we put them on an equivalent scale by defining $ETT = 1 - ANTT_{est}$. Our implementation includes weighting factors for each term to allow for solutions which value the metrics disproportionately. Finally, we multiply the weighted sum of reward terms by -1, since negative rewards encourage early solution completion rather than reward accumulation [38].

Given any moderate size ready queue, the potential number of states is exponential and therefore trying to use traditional reinforcement learning methods becomes untenable. However, our objective remains to be developing task scheduling solutions which can be adaptive and flexible.

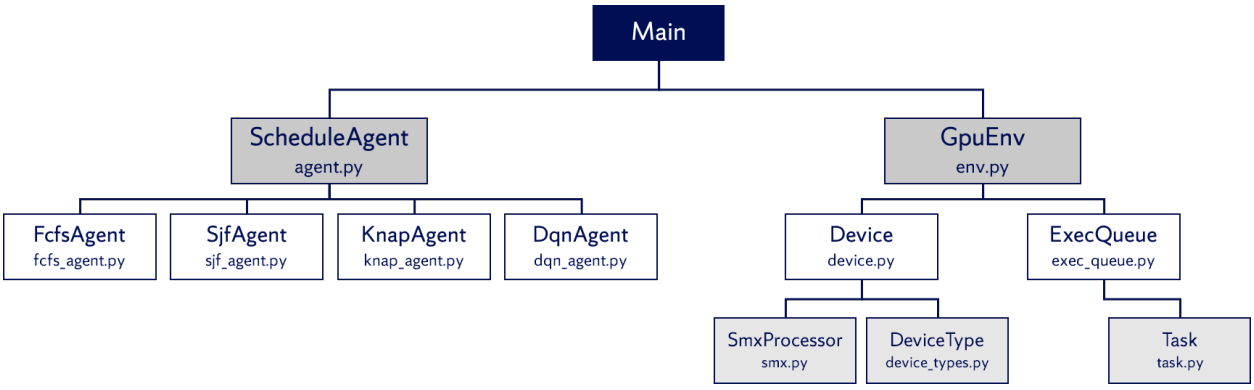

Figure 29. GPU system deep Q-learning simulation architecture design.

Our DQN model is a simple, three-layer neural network. The size of the input layer is determined by the size, n, of the ready queue, but in general can be written as $4 + (n * 4)$, which is also the size of the environment state vector, S_{env}. The intermediate layers are fully-connected with ReLU activation. The output layer size is equal to n, corresponding to $|A|$. We use masking to eliminate certain action selections, i.e., if the corresponding slot in the ready queue is empty.

The DQN model is trained using experience replay. The replay memory logs the initial state, selected action, reward, and next state transitions for each decision point in our algorithm. We exclude the deterministic transitions given by constraints discussed above, namely the adherence to rules G3 and X1. Therefore, each episode consists of a finite set of n actions.

5.2.2 Implementation Details

We developed a custom discrete event simulation to evaluate the selected scheduling algorithms and simulate the GPU device and task queue dynamics. The simulation is developed in Python 3.5 and uses PyTorch 1.4 for the DQN learning. An architecture diagram of the simulation is shown in Figure 29.

Main. The primary functionality of the Main module is to execute the basic implementation of reinforcement learning process, shown in pseudocode in Listing 2. This execution flow is

consistent with many similar implementations of deep Q-learning, including the DeepRM proposed by [51].

GpuEnv. Utilizing the reinforcement learning concept presented in Chapter 4 (Figure 22), we model the environment as the combination of the GPU device, i.e., compute resources, and the execution queue. The Device object allows us to manage the state of compute resources by assigning blocks of a task to the first available SMX processor, and releasing completing blocks from the respective processors. The parameters used to construct the Device object and its component SmxProcessors are drawn from the DeviceType object. At runtime, we can specify a certain GPU type, providing the resource parameters shown in Table II, making the experimentation flexible to evaluate several different GPU architectures. We also implement a `state` method, which returns the instantaneous utilization of the device as defined in Section 2.5.4. The ExecQueue object manages the list of Tasks waiting to execute on the GPU. Because we allow for selection from any slot, the queue is actually implemented as an array. We include a `state` method to return a vector representation of the states of tasks waiting on the queue. The Task object is a simple data structure that contains data members representing resource requirements and execution time, as well as performance metrics (arrival, dispatch, and completion times) for experimental analysis. The Task `state` method returns the vector representation shown above in (5-5).

ScheduleAgent. The ScheduleAgent object provides an abstract base class from which we can implement multiple scheduling policy agents. The base class provides interface methods which are not implemented by most of the derived class agents, but allow for runtime selection of scheduling policy during experimentation. Every agent does implement the `act` method, which differentiates the policy decision for selecting the next task to be executed.

Listing 2. GPU system simulation Main module for deep Q-learning evaluation.

```
# set up the environment
env = env.GpuEnv(device_type, queue_length, weight_slowdown, weight_util)
state_size = env.state_space
action_size = env.action_space

# initialize the agent
agent = initializeAgent(policy, state_size, action_size, batch_size, gamma,
learning_rate)

done = False

# run many episodes to evaluate different behaviors
for e in range(num_episodes):

    # reset the environment for the new episode
    state = env.reset()

    # for a certain amount of time steps, continue choosing actions
    for wave in count():
       # select and perform an action
       action = agent.act(state)
       next_state, reward, done = env.step(action)

       # store the transition in memory
       agent.remember(state, action, reward[2], next_state, done)

       # move to the next state
       state = next_state

       # perform one step of optimization (on the target network)
       agent.optimize()

       if done:
              # progress the environment until all running tasks are completed
              while tasks_running:
                     t0, t1, next_state = env.progress()
                      state = next_state

              if agent.epsilon > agent.epsilon_min:
                     agent.epsilon *= agent.epsilon_decay
              break

        # Update the target network, copying all weights and biases in DQN
        if e % target_update == 0:
            agent.update()
```

The FcfsAgent implementation of `act` method simply returns the first task in the execution queue. The SjfAgent performs similarly, i.e., returns the first task in the queue, but with the additional step that this agent is initialized such that the execution queue is reordered from initial arrival sequence to one arranged by increasing estimated task execution time. The KnapAgent implements the algorithm proposed by [47]. Similar to SjfAgent, upon initialization of the KnapAgent the execution queue is reordered to the resulting solution to the knapsack 0-1 approach. Finally, the DqnAgent implements the deep Q-learning based approach described in this chapter. The agent initializes an initial neural network architecture (Net) which we use to map environment states to agent actions. We also keep a ReplayMemory that stores the experience replay, i.e., transitions, described above. The `act` method implements a ε-greedy approach for selecting the next action.

By utilizing the developed GPU model we have a fully-customizable module that can be used to represent a variety of NVIDIA GPU architectures, and is controllable to ensure repeatable performance. With the smallest units being SMX processors, we can construct a detailed representation of the GPU architecture, including a mapping of thread blocks to processors. The scheduling module abides by well understood GPU scheduling rules, though the underlying algorithms have not been published. In particular, we use a round-robin approach to assigning thread blocks to processors. Furthermore, we restrict scheduling to ensure that only tasks which can fit within the processors resource footprint will get dispatched. Finally, we observe the scheduling rules as defined in [15] to ensure validity and consistency.

For the DQN module, we begin with randomly initialized policy and target networks [50]. Action decisions are selected using the policy network, while the target network allows for a stable

estimation of values for *Q(s,a)* by holding weights fixed for a period of time. Every 10 episodes, we update the target network with a copy of the current policy network.

The replay memory can store 40,000 transitions, which allows us to retain as many observations from our experiments as possible [52]. This memory prioritizes recent observations, therefore if we exhaust the space older observations will be dropped off in favor of more recent transitions. We use a batch size of 128 for selecting transitions from the replay memory for training.

Each episode of our experiment uses ε-greedy method for choosing the next action. The action selected is determined either randomly from among eligible waiting tasks, or by using the policy network. At the end of each episode we multiply ε by a decay rate of 0.999. The effect of this is to allow an adequate amount of exploration early in the learning process, but then shift the focus to exploitation – by following the policy network – in the later stages.

5.3 Experiment Methodology

The experiments in this research focus on scheduling a ready queue of size, $n = 8$. Each task in the queue is independent and unique, and characteristics of each are described in Table VIII. Tasks are drawn from the Rodinia benchmark suite [22] [23]. We model a NVIDIA TITAN X GPU, with compute resources given in Table IX.

In addition to the DQN algorithm proposed here, we include three other scheduling algorithms for comparison. The first algorithm is the FCFS approach which is the baseline method used by current GPU architectures. Solving for an optimal solution using exhaustive search is intractable in practice, even for small n. Consider at $n = 8$, there exist 40,320 possible orderings of tasks in the queue. We evaluate all possible orderings in this experiment to illustrate the upper

and lower bounds of performance, since all other solutions will come from this set, but note that this is not a feasible solution in a real-time system.

Table VIII. Rodinia kernel characterisitics.

Task ID	Kernel name	Number of thread blocks	Number of blocks per thread	Number of registers per thread	Size of shared memory per block	Execution Time (ms)
0	*lud*	1	16	32	1024	26
1	*hs3*	1024	56	36	0	110
2	*srad*	16384	256	21	5120	560
3	*pf*	463	256	13	2048	55
4	*bfs*	1954	512	19	0	20
5	*hs2*	1849	256	38	3072	156
6	*knn*	3840	256	8	0	41
7	*hw*	51	256	38	11872	12055

Table IX. GPU Resource Specifications.

	TITAN X	GRA113Q
Number of Symmetric Multiprocessors (SMX)	24	5
Maximum number of resident blocks per SMX	32	16
Maximum number of resident threads per SMX	2048	2048
Maximum number of registers per thread block	64K	64K
Maximum amount of shared memory per SMX	96KB	48KB

The second algorithm we evaluate is the SJF approach. SJF is a well understood, non-preemptive algorithm that favors high throughput. However, there is no accommodation for resource utilization within the algorithm, hence it cannot be considered an equivalent solution to the problem we pose.

The final algorithm we compare to is the Knapsack 0-1 approach as described by [47]. This approach is most similar to ours because it simultaneously considers resource utilization and turnaround time as optimization objectives. In this algorithm, the knapsack capacity is represented

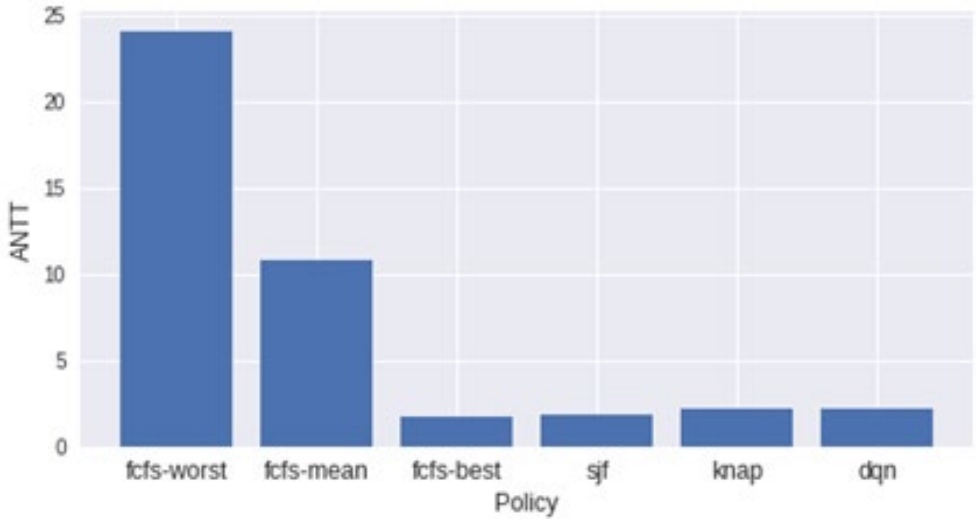

Figure 30. Average normalized turnaround time performance by scheduling algorithm policy.

by the available GPU resources, while the item weights are given by task resource requirements and item values are given by the ratio of resource requirements to execution time.

In the cases of the SJF and Knapsack algorithms for a static schedule, we can determine in advance the order of execution, regardless of the original order of arrival. Therefore, we preload the solution order into our experimental task queue, and do not run the algorithm during execution.

We evaluate our DQN algorithm over 5000 episodes. Given the initial and threshold values for epsilon and the decay rate per episode, we determined that 5000 episodes ensures that we reach the threshold and then conduct many episodes which purely follow the DQN algorithm.

5.4 Experimental Results

By running all 40,320 possible orders in the FCFS algorithm, we are able to evaluate all solutions to the scheduling problem and identify the optimal ordering. Each of the other three algorithms, including our proposed approach, derives an ordering which is in the set of all orders evaluated under FCFS. In Figure 30, Figure 31, and Figure 32, we show where each solution falls

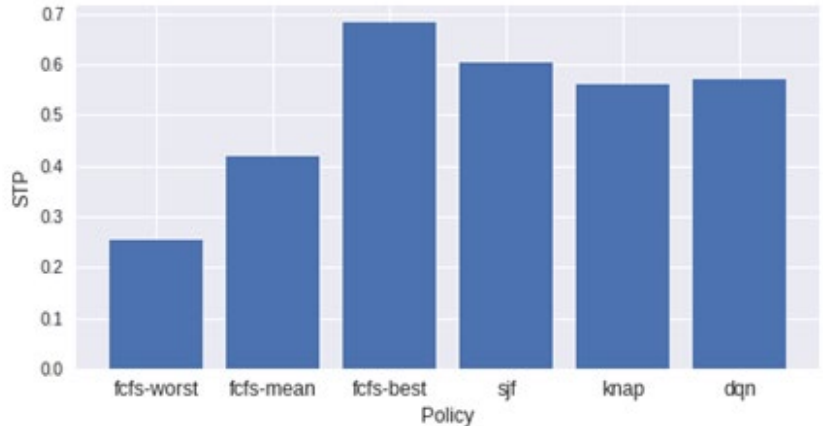

Figure 31. System throughput performance by scheduling algorithm policy.

in the range of FCFS outcomes. Of particular note is the average performance in both ANTT (Figure 30) and utilization (Figure 32). Because exhaustive search is not a feasible solution to task scheduling, we should expect the average performance results to be most representative of the FCFS performance.

We also show the best- and worst-case results for each metric under FCFS method. Note that these results do not correlate across metrics – we show the best-case ANTT, but this solution ordering does not coincide with the best-case utilization. Rather, we show these results to show what is theoretically possible.

The algorithms evaluated all exhibit approximately 80% improvement over the average FCFS performance in ANTT. In addition, all are very close to the optimal ANTT found at 1.887, with the SJF algorithm achieving the lowest ANTT. When compared in device utilization, we again observe that all three algorithms present an improvement over the average FCFS performance, with the DQN algorithm providing the largest increase. Results are summarized in Table X.

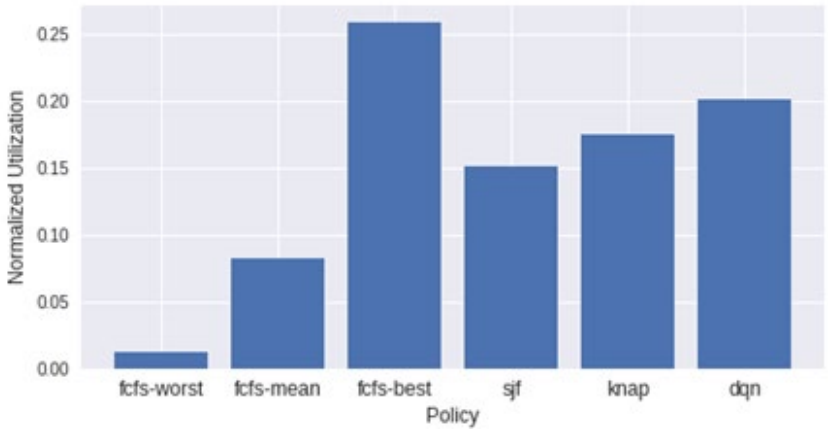

Figure 32. Average device utilization performance per scheduling policy.

Figure 33 shows the multi-objective performance from a different perspective. In this figure, Normalized ANTT is defined as the reciprocal of the measured ANTT. The light gray dots show the results of every possible ordering evaluated under FCFS algorithm. We highlight the specific solution orders for each of the special cases identified above: the three other algorithms under test, plus the optimal solution from FCFS. Values at the upper right of this plot are considered optimal, for they represent solutions which equally balance utilization and ANTT performance. The proposed DQN solution falls within this quadrant, and represents an improvement over the alternative algorithms evaluated.

Table X. Scheduling Algorithm Performance Results.

Algorithm	**ANTT**	**STP**	**Util**	**Q Score**	**Makespan (µs)**
FCFS avg	10.818	0.418	0.082	0.008	12528
FCFS optimal	1.887	0.612	0.249	0.132	12406
SJF	1.818	0.604	0.151	0.083	13025
Knapsack	2.216	0.562	0.175	0.079	13012
DQN	2.166	0.570	0.202	0.093	12391

The *Q Score* column from Table X corresponds to the product of the normalized ANTT and normalized Utilization values, giving a single value with which to measure the schedule

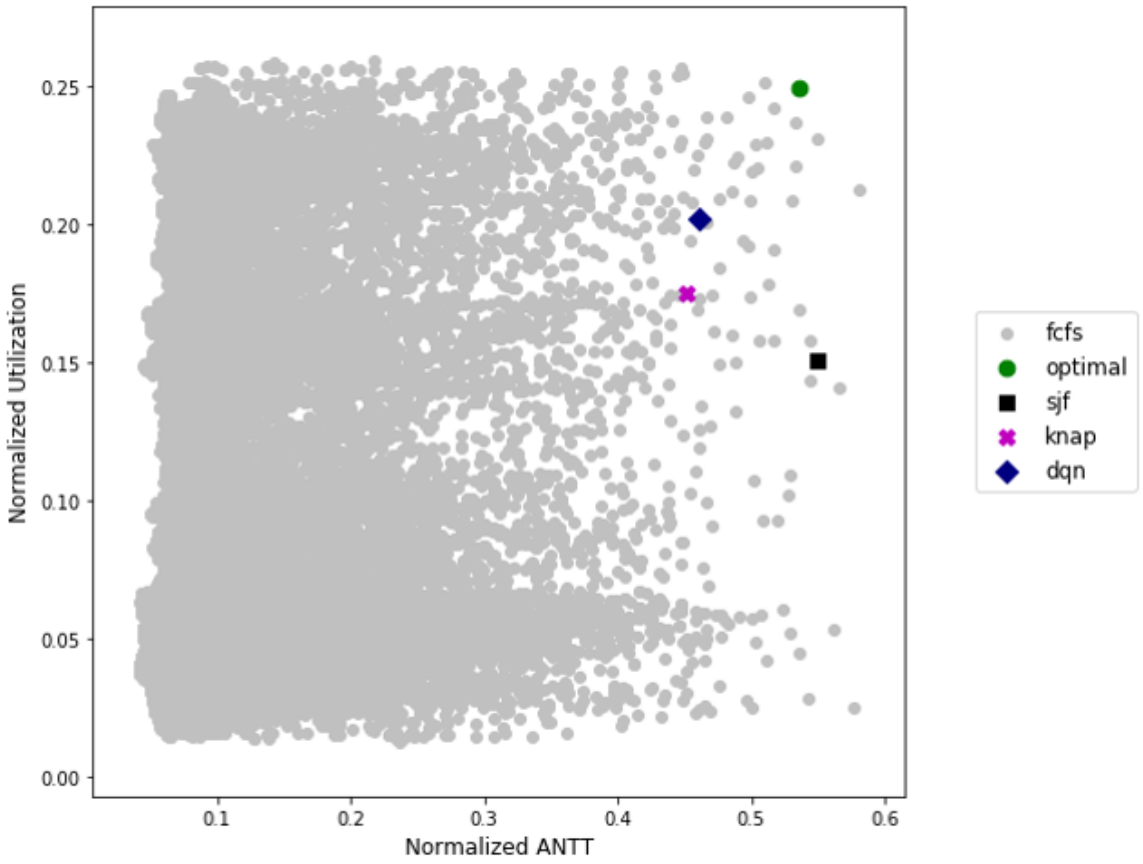

Figure 33. Average normalized turnaround time versus average utilization performance.

performance, i.e., higher is better. Of the three algorithms evaluated, the DQN solution is nearest to the optimal ordering found under exhaustive evaluation of FCFS.

A last perspective on the relative performance of each algorithm comes in the form of per-task performance, in particular the observed waiting time per task as shown in Figure 34. In this figure we show the worst- and average-case waiting time for FCFS (best-case for each task would be 0 when the task is first in order, and is therefore trivial), along with SJF, Knapsack, and DQN. We observe that in general, the three algorithms under test outperform the average case, with the exception of SJF and Knapsack on task 7. This task happens to be the one with the longest execution time (see Table VIII), and these results suggest a bias against longer running tasks. This

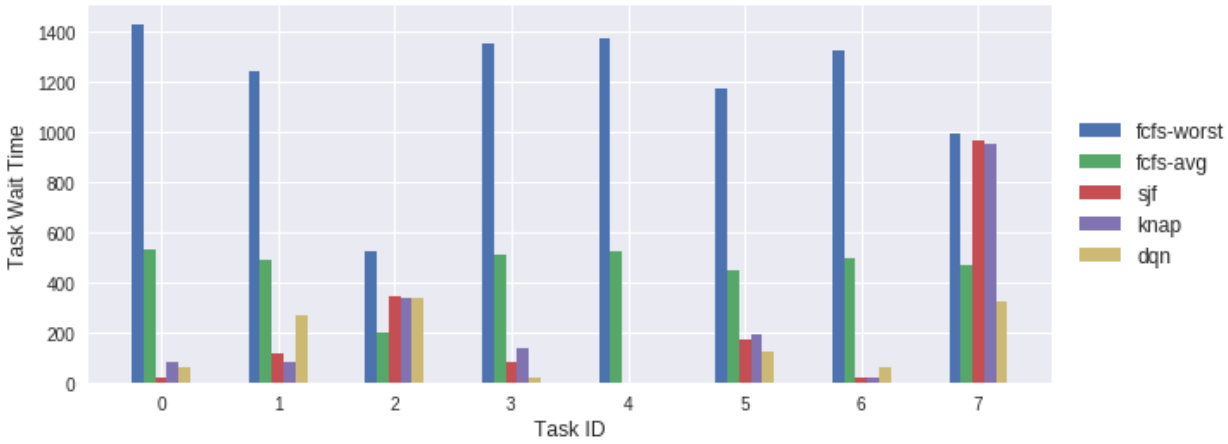

Figure 34. Per-task average waiting time under different scheduling algorithms.

is expected for SJF, but somewhat surprising for the Knapsack algorithm. Furthermore, we observe that nearly all tasks under the DQN approach are bounded in wait time below the overall average wait time. Such performance is indicative of a fair algorithm and could provide quality-of-service guarantees as well.

5.5 Conclusions

The results in the previous section demonstrate that our proposed DQN algorithm for GPU task scheduling outperforms other scheduling algorithms for effectively balancing both GPU resource utilization and application turnaround time. Our algorithm achieves a higher degree of resource utilization and lower average normalized turnaround time than the Knapsack 0-1 algorithm proposed by [47]. Furthermore, our approach is more adaptable and flexible to changing dynamics in a system. For example, should a new task be inserted in to the system, our approach will make an action selection based on the current DQN model whereas other methods require a completely new iteration of the algorithm.

We also expect advantages to our approach when applied to both the variety of GPU architectures that currently exist, as well as the continually evolving options expected in the future.

In particular, through transfer learning we believe that we could quickly adapt an existing model to a new architecture and achieve near optimal results. The knapsack approach from [47] can similarly be applied to different architectures, but the weights and values must all be recalculated, rendering the knapsack matrix nontransferable. In future work, we would explore this transferability by simulating multiple different architectures. We would also explore evaluating the proposed algorithm on real GPU hardware to validate the performance results reported here.

A key direction of future research would be to combine compute resource allocation and task scheduling with the optimization and scheduling of data transfer tasks. As demonstrated by the rules specified by [15], the order of execution in most cases relies on underlying dependencies between data transfer and compute tasks. Our prior work, and that of Belviranli, et al., [35] has demonstrated the effectiveness of different techniques for reordering data transfer tasks in order to achieve optimal transfer bandwidth and reduce system waiting time.

Finally, optimization for a single GPU system presents only a first step to solving a larger problem, as these computational resources are primarily utilized in large-scale systems. Therefore, the ability to optimize the allocation and scheduling of tasks to a multi-GPU system could be constructed as a hierarchical problem. Research on such problems using DRL [53] has shown performance benefits, and could be combined with the algorithm we propose here to formulate a multilevel DRL solution.

Chapter 6 Related Work

This chapter discusses previous works on GPU tasks scheduling under concurrent kernel execution scenarios. In section 6.1, we compare these techniques with contributions presented in Chapters 3 and 5. In section 6.2, we present related work in data transfer approaches and compare to the contributions of Chapter 4. Finally in section 6.3, we discuss the application of reinforcement learning and deep reinforcement learning to resource allocation and task scheduling problems to provide context for the solutions we develop in Chapters 4 and 5.

6.1 Concurrent Kernel Execution and Compute Resource Scheduling

Many of earliest works to address GPU system performance focused on methods for maximizing the potential of concurrent kernel execution. Some of these approaches predate the introduction of CUDA Hyper-Q technology; others propose solutions which are in opposition to one of the core tenets of our approach, in particular to not require modifications to either source code or hardware drivers.

Pai, et. al., [26] defines the concept of an “elastic kernel” which consists of a wrapper around original kernels to enable logical grid and block sizes that are separate from physical grid and block configurations that actually execute on the GPU. Similarly, Jiao, et al., [54] describes a kernel pairing strategy that simultaneously optimizes performance and power consumption for energy efficiency optimization. The resource sharing and concurrent kernel execution is enabled through a kernel slicing technique [55], which requires an intermediate step to generate the CUDA code for the smaller kernel slices. A main limitation to these types of approaches is that they

require code transformation to enable the resource sharing technique, implying that it must be enabled at development time, whereas our approach is transparent to the developer and therefore backwards-compatible with existing applications.

Li, et al., [24] develops a performance model to evaluate resource-sharing among different types of single-program multiple-data (SPMD) applications. In this work, the authors characterize applications into four categories based on the resource utilization of the kernels relative to the per SMX constraints. These categories define how unutilized or underutilized resources can be shared with other programs. The authors then define optimization strategies to maximize performance based on an application's category, to include overlapping kernel execution with input/output (I/O), changing grid dimensions, and splitting the same work among multiple smaller kernels. The effectiveness of the proposed techniques are limited in scope to only homogeneous kernels. By contrast, our analysis considers applications with different kernel behaviors and GPU resource requirements.

In a following work, Li, et al., [25] defines a greedy algorithm to schedule concurrent kernels by examining relative GPU resource utilization and calculating a "symbiosis score" between pairs of kernels. This symbiosis score is meant to rate how efficient a pair of kernels could utilize the GPU if selected to execute concurrently. Some parameters required to calculate the symbiosis score require offline analysis of each kernel, which could presumably be stored in a lookup table for the runtime scheduler. When compared to the naïve (first-come, first-served) scheduler, their approach computes a near-optimal kernel launch sequence for performance and energy efficiency. A limitation of this approach is that it penalizes kernels which oversubscribe GPU resources. Oversubscribing is defined as any scheduling of concurrent kernels which requires greater than the maximum hardware resources available, typically in terms of thread blocks, threads, registers, or

shared memory. In other words, for two kernels to be scheduled concurrently, the sum total of their resource requests must be less than or equal to the total resources available on the GPU. For realistic application kernels with sufficiently large resource requirements, this approach will almost always result in serialized execution.

Liang, et al., [27] develops a dynamic programming approach to pairing concurrent kernels based on maximum execution latency improvement. Using the results of the temporal scheduling, each kernel's thread blocks are scheduled to execute using a leaky bucket algorithm, which artificially interleaves the kernels by merging them into a single kernel. The resultant schedule is evaluated against Hyper-Q performance and demonstrated an average 17% performance improvement. However, the authors note that the practical implementation requires hardware support, and in lieu of such support the solution relies on a software emulation framework which increases the GPU resource requirements by introducing extra variables and global memory for storing the interleaved schedule.

Wang, et al., [34] define Simultaneous Multikernel (SMK) which exploits heterogeneity of independent kernels to fully utilize compute resources. In particular, through the use of "dominant resource fairness" allocation decisions are made in order to equalize resource usage of kernels and make thread block to symmetric multiprocessor (SM) assignments. Lee, et. al., [56] proposes two forms of thread block scheduling – Lazy CTA scheduling and Block CTA scheduling – which vary how thread blocks are assigned to SMX resources. Both approaches assume fine-grained control over the assignment of thread blocks to SMX processors, an assumption that we do not make in our solution.

Reordering mechanisms achieve improved performance in the most pragmatic way by substituting for the specific scheduling algorithm used by the execution queue, rather than the

thread block scheduler. Wende, et al., [57] develops a kernel reordering technique which aims to overcome the limitations of Fermi GPUs (i.e. false serialization) by having each thread insert its kernel executions into separate CPU queues associated with a GPU stream. The reordering algorithm launches kernels in a round-robin fashion across these queues, until all kernels have been scheduled. Thus, an advantage to this technique is that it is backward compatible to GPUs without Hyper-Q support. The authors compare this approach with Hyper-Q performance and note that, when both approaches are combined it achieves close to optimal efficiency [33]. The authors conclude that Hyper-Q alone was not sufficient, since the kernel execution queues were not all equally favored for GPU execution. This work is similar to ours because it describes a host-side reordering technique that affects the order in which executions are launched to the GPU, but does not attempt to explicitly control execution order on the device. However, the differences between this work and ours are two-fold. First, their work only examines a round-robin reordering technique, while we examine several orderings and demonstrate that different orderings are more optimal for different application pairings. Second, their technique associates each CPU thread with a specific GPU stream, which may result in host-side serialization within thread queues, whereas we allow for an independent thread for each application, and dynamically assign GPU streams to these threads as they are needed.

Cruz, et al. [47] proposed a dynamic programming approach using a knapsack 0-1 algorithm. We have compared our method to this work by implementing a version of the algorithm as described in Chapter 5. One subtle difference between our approach and [47] is the required estimate of kernel execution time. For the knapsack algorithm, this is critical information to determine the value, v_i, to be used, whereas we only use estimated execution time for our simulation and it is not a feature in our deep Q-learning model. While learning an estimate over

time is possible, the reliance on this information makes the approach susceptible to inaccuracies, whereas all of the features we utilize are fixed values.

6.2 Data Transfer

Despite early research that showed that data transfer performance is a critical factor for consideration in the evaluation of application performance [58], not a great deal of research on GPU performance optimization has focused this area. Efforts have been made to model the PCIe transfer time in order to describe optimal communication and computation overlap strategies [59], or to develop more accurate performance models [36], but have not focused on maximizing available bandwidth as we have done.

As discussed in Chapter 3, Butler, et al. [17] studied the effects of implicit and explicit synchronizations on performance of concurrently executing kernels on GPU. In particular, as part of the proposed Sync-Free GPU (SF-GPU) runtime mechanism that aims to bypass both types of synchronizations, the authors describe a method of double-buffer memory pools. This approach is most similar to our bandwidth-optimized transfer algorithm, using a fixed buffer size where we had a fixed threshold value, τ. Our reinforcement learning algorithm overcomes this limitation to both approaches.

Belviranli, et al. [8] defines the CUDA Multi-Application Sharing (CuMAS) framework which intercepts certain CUDA API function calls and re-orders the calls in order to improve the resource utilization. Notably, this work appears to be the first to explicitly treat the host-to-device and device-to-host transfer channels as separate resources to be efficiently optimized. However, the authors consider each CUDA API call in the user program as an individual operation. Due to the overhead of the PCIe bus, this may lead to suboptimal performance. Furthermore, dynamic

programming is used to solve the scheduling problem, although it is much faster than brute force approach, the complexity is still too high to be used for online scheduling.

Message aggregation as a means of improving performance and mitigating resource contention has been studied in the past [60] [61]. These approaches often make underlying assumptions about the message characteristics and are generally too application-specific to be applied in the context presented here.

The key distinction between these previous works and the research presented in Chapter 4 is the specific focus on the optimization and efficient utilization of the data transfer engines between the host and device. Compared to some of the recent works, our approach considers more detailed hardware limitations and online scheduling has much higher simplicity.

6.3 Reinforcement Learning for Scheduling Problems

Finally, reinforcement learning has gained increasing attention recently following the trends of machine learning in general [62], and techniques to apply RL to resource management. Mao, et al., [51] presents an approach to use deep reinforcement learning to the job scheduling problem. This work seems to be complementary to our work, as the approach considers a more abstract assignment of jobs to resources to improve performance (i.e. decrease average job slowdown) on clusters, but does not explicitly consider resource utilization a key metric. Furthermore, the system specification is abstract and not specific to the GPU, making our model a more concrete representation.

Liu, et al., [53] examines a hierarchical approach to managing resource allocation and power management in a cloud computing system. Again we consider this to be complementary to our approach since we consider a much finer level of granularity in the job-to-resource allocation

strategy. Huang, et al., [63] describes a deep reinforcement learning approach to manage task offloading and resource allocation in edge computing. The primary focus of this algorithm is on the cost of offloading, whereas as our approach is focused on resource utilization and performance.

Chapter 7 Conclusion and Future Directions

In this dissertation, we studied the optimization and scheduling of GPU resources to maximize resource utilization and increase system throughput. By enabling gains in these critical metrics, we allow for improvements in system efficiency, both in terms of energy efficiency and a reduction in the number of compute devices required.

In Chapter 2, we introduced the key concepts relative to GPU programming and scheduling which are necessary to develop a feasible scheduling algorithm. We introduce the challenges of multi-kernel, multi-stream scheduling, particularly with respect to resource contentions, and provided motivational examples which highlighted the opportunities for performance improvements and the benefits of introducing a new scheduling approach.

In Chapter 3, we explored the ways in which one can make effective use of the CUDA Hyper-Q technology to increase throughput via concurrent kernel execution. In particular, we identified a key limitation to performance due to memory transfer interleaving and proposed a novel approach for mitigating this restraint. We also showed that application launch ordering can significantly affect task overlap potential, requiring a more fine-grained approach to application task scheduling.

Chapter 4 introduced a reinforcement learning-based approach to managing memory transfers to maximize the transfer bandwidth utilization. This approach introduced methods for aggregating or disassembling memory transfers into optimally-sized packets, while balancing the effect of aggregation wait times on the turnaround time of independent transfer tasks. Our approach

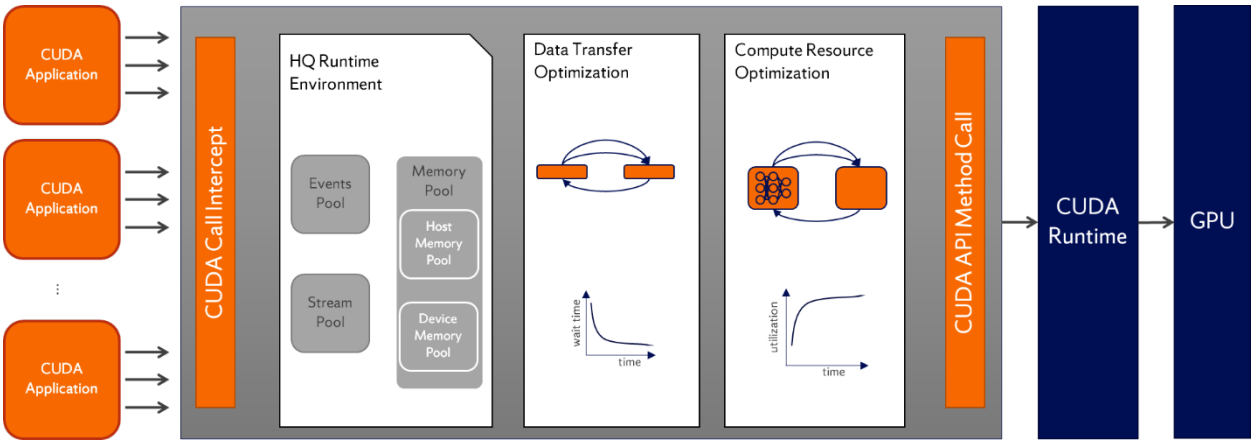

Figure 35. Proposed hierarchical framework for managing data transfer and compute task scheduling on GPU devices.

outperformed other techniques in mean waiting time and provided significant increase in bandwidth utilization over the baseline FCFS algorithm.

Finally, in Chapter 5 we developed a deep Q-learning algorithm for managing task scheduling to maximize the utilization of GPU compute resources while finding a launch ordering with near-optimal ANTT. Our approach is competitive with the best-case performance of the baseline FCFS algorithm, which could only be identified via the infeasible solution of exhaustive search, and is adaptable to changing GPU and task characteristics, unlike other approaches.

7.1 Future Directions

With the successful demonstration of reinforcement learning approaches to managing both the data transfer and compute resource scheduling problems, the natural next step is to combine these approaches into a hierarchical scheduler which can effectively and efficiently schedule task requests to the GPU. The design of such a framework is given in Figure 35.

This framework brings together all of the key concepts discussed in this dissertation into a logical hierarchy that can be seamlessly integrated on a GPU system without requiring either application source code or hardware device driver modifications.

7.1.1 Implementation Details

The implementation of this framework would be accomplished using the method call intercept approach. An example of this type of an approach using CUDA API methods is shown by [9]. With this approach, we can provide a layer between the developer's application code and the CUDA Runtime which is transparent to the system. In this layer, we can extract all of the necessary information for the proposed scheduling algorithms, such as data size for the transfer scheduler or the resource requirements for the compute task scheduler, directly from CUDA API method calls.

Appendix B provides source code implementation details of the primary CUDA API methods that would need to be intercepted to enable this framework. For the HQ Runtime Environment discussed in Chapter 3, this includes the memory, stream, and events creation and destruction methods. For data transfer tasks, we intercept all `cudaMemcpy*` methods. Finally, for the kernel execution tasks, we must intercept two methods – `cudaConfigureCall`, which specifies the grid and block dimensions and signals that the kernel is ready to execute, and `cudaLaunch`, which actually launches the specified kernel function for execution on the GPU.

The insertion of the described methods in this dissertation into this type of hierarchical framework is certainly nontrivial, but also does not require any modifications to either the hardware device drivers or user application code in order to reap the performance benefits. In fact, once developed, the framework would be provided as a shared library object and inclusion of its features is simply enabled through a LD_PRELOAD command such as the following:

```
LD_PRELOAD=/path/to/lib/libmyframework.so ./your-program
```

Thus, we restate our earlier claim that the approach described herein is the most pragmatic and comprehensive approach proposed in this domain.

7.1.2 Multi-GPU Systems

While all of the research in this dissertation, as well as the proposed framework above, focuses on a single-GPU system, we also expect significant benefits to come from applying these methods to a multi-GPU system. Providing another level in the hierarchy would enable more optimal scheduling across multiple devices while introducing additional complexity. The highest level scheduler is envisioned to serve more like traditional load-balancing techniques, handling coarse-grained assignment of applications to devices, whereas our scheduling framework manages fine-grained task scheduling.

In addition, because one of our primary objectives is to maximize device utilization, we believe this multi-GPU, hierarchical approach could drive system designs that require fewer number of GPUs than similarly tasked systems with a naïve multi-device scheduler. This type of design brings considerations about size, weight, and power (SWaP) to a macro (system) level, while simultaneously managing efficiency at the micro (device) level.

Appendix A GPU Scheduling Rules

The following rules are consolidated from [15] and provided here for quick reference.

Identifier	Rule
G1	A copy operation or kernel is enqueued on the stream queue for its stream when the associated CUDA API function (memory transfer or kernel launch) is invoked.
G2	A kernel is enqueued on the EE queue when it reaches the head of its stream queue.
G3	A kernel at the head of the EE queue is dequeued from that queue once it becomes fully dispatched.
G4	A kernel is dequeued from its stream queue once all of its blocks complete execution.
X1	Only blocks of the kernel at the head of the EE queue are eligible to be assigned.
R1	A block of the kernel at the head of the EE queue is eligible to be assigned only if its resource constraints are met.
R2	A block of the kernel at the head of the EE queue is eligible to be assigned only if there are sufficient thread resources available on some SM.
R3	A block of the kernel at the head of the EE queue is eligible to be assigned only if there are sufficient shared-memory resources available on some SM.
C1	A copy operation is enqueued on the CE queue when it reaches the head of its stream queue.
C2	A copy operation at the head of the CE queue is eligible to be assigned to the CE.
C3	A copy operation at the head of the CE queue is dequeued from the CE queue once the copy is assigned to the CE on the GPU.
C4	A copy operation is dequeued from its stream queue once the CE has completed the copy.
N1	A kernel Kk at the head of the NULL stream queue is enqueued on the EE queue when, for each other stream queue, either that queue is empty or the kernel at its head was launched after Kk .
N2	A kernel Kk at the head of a non-NULL stream queue cannot be enqueued on the EE queue unless the NULL stream queue is either empty or the kernel at its head was launched after Kk
A1	A kernel can only be enqueued on the EE queue matching the priority of its stream.
A2	A block of a kernel at the head of any EE queue is eligible to be assigned only if all higher-priority EE queues (priority-high over priority-low) are empty.

Appendix B CUDA Method Intercept Examples

Example B-1. Intercepting cudaSetDevice and cudaDeviceReset for initialization and shutdown of proposed framework.

```
//----< environment setup/cleanup intercepts >----------------------------------
typedef cudaError_t (*cudaSetDevice_t)(int device);
typedef cudaError_t (*cudaDeviceReset_t)(void);

//----< environment setup/cleanup intercepts >----------------------------------
static cudaSetDevice_t realCudaSetDevice = NULL;
static cudaDeviceReset_t realCudaDeviceReset = NULL;

extern "C" cudaError_t cudaSetDevice(int device)
{
    if (realCudaSetDevice == NULL)
        realCudaSetDevice = (cudaSetDevice_t)dlsym(RTLD_NEXT, "cudaSetDevice");

    assert(realCudaSetDevice != NULL && "cudaSetDevice is null");

    // implement framework initialization procedures

    return realCudaSetDevice(device);
}

extern "C" cudaError_t cudaDeviceReset(void)
{
    if (realCudaDeviceReset == NULL)
        realCudaDeviceReset = (cudaDeviceReset_t)dlsym(RTLD_NEXT, "cudaDeviceReset");

    assert(realCudaDeviceReset != NULL && "cudaDeviceReset is null");

    // implement framework shutdown procedures

    return realCudaDeviceReset();
}
```

Example B-2. Intercepting cudaMemcpy, cudaMemcpyAsync to pass to data transfer optimization algorithm.

```
//----< memory transfer intercepts >--------------------------------------------
typedef cudaError_t (*cudaMemcpy_t)(void* dst, const void* src,
                                    size_t count, enum cudaMemcpyKind kind);
typedef cudaError_t (*cudaMemcpyAsync_t)(void* dst, const void* src,
                                         size_t count, enum cudaMemcpyKind kind,
                                         cudaStream_t stream);

//----< native CUDA API function pointers >-------------------------------------
static cudaMemcpy_t realCudaMemcpy = NULL;
static cudaMemcpyAsync_t realCudaMemcpyAsync = NULL;

//----< Memory transfer intercepts >--------------------------------------------
```

```
extern "C" cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum
cudaMemcpyKind kind)
{
    if (realCudaMemcpy == NULL)
        realCudaMemcpy = (cudaMemcpy_t)dlsym(RTLD_NEXT, "cudaMemcpy");

    assert(realCudaMemcpy != NULL && "cudaMemcpy is null");

    // implement data transfer algorithm handling

    return realCudaMemcpy(dst, src, count, kind);
}

extern "C" cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count, enum
cudaMemcpyKind kind, cudaStream_t stream)
{
    if (realCudaMemcpyAsync == NULL)
        realCudaMemcpyAsync = (cudaMemcpyAsync_t)dlsym(RTLD_NEXT, "cudaMemcpyAsync");

    assert(realCudaMemcpyAsync != NULL && "cudaMemcpyAsync is null");

    // implement data transfer algorithm handling

    return realCudaMemcpyAsync(dst, src, count, kind, stream);
}
```

Example B-3. Intercepting cudaConfigureCall and cudaLaunch for compute resource optimization and task scheduling algorithm.

```
// cudaConfigureCall
typedef cudaError_t (*cudaConfigureCall_t)(dim3, dim3, size_t, cudaStream_t);
static cudaConfigureCall_t realCudaConfigureCall = NULL;

extern "C"
cudaError_t  cudaConfigureCall(dim3  gridDim,  dim3  blockDim,  size_t  sharedMem=0,
cudaStream_t stream=0)
{

    if (realCudaConfigureCall == NULL)
        realCudaConfigureCall                                                    =
(cudaConfigureCall_t)dlsym(RTLD_NEXT,"cudaConfigureCall");

    assert(realCudaConfigureCall != NULL && "cudaConfigureCall is null");

    // extract kernel resource requirements for queue state values

    return realCudaConfigureCall(gridDim, blockDim, sharedMem, stream);
}

// cudaLaunch
typedef cudaError_t (*cudaLaunch_t)(const void* func);
static cudaLaunch_t realCudaLaunch = NULL;

extern "C"
```

```
cudaError_t cudaLaunch(const void* func)
{
    if (realCudaLaunch == NULL)
        realCudaLaunch = (cudaLaunch_t)dlsym(RTLD_NEXT,"cudaLaunch");

    assert(realCudaLaunch != NULL && "cudaLaunch is null");

    // implement task scheduling and resource optimization algorithm handling

    return realCudaLaunch(func);
}
```

References

[1] M. Macedonia, "The GPU enters computing's mainstream," *Computer,* vol. 36, no. 10, pp. 106-108, 10 2003.

[2] Z. Fan, F. Qiu, A. Kaufman and S. Yoakum-Stover, "GPU Cluster for High Performance Computing," in *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, Pittsburgh, PA, USA, 2004.

[3] NVIDIA Corp., "NVIDIA CUDA Compute Unified Device Architecture, Programming Guide," 2007.

[4] "The OpenCL Specification," Khronos Group, 2009.

[5] S. S. Vazhkudai, B. R. De Supinski, A. S. Bland, A. Geist, J. Sexton, J. Kahle, C. J. Zimmer, S. Atchley, S. Oral, D. E. Maxwell, V. G. Larrea, A. Bertsch, R. Goldstone, W. Joubert, C. Chambreau, D. Appelhans, R. Blackmore, B. Casses, G. Chochia, G. Davison, M. A. Ezell, T. Gooding, E. Gonsiorowski, L. Grinberg, B. Hanson, B. Hartner, I. Karlin, M. L. Leininger, D. Leverman, C. Marroquin, A. Moody, M. Ohmacht, R. Pankajakshan, F. Pizzano, J. H. Rogers, B. Rosenburg, D. Schmidt, M. Shankar, F. Wang, P. Watson, B. Walkup, L. D. Weems and J. Yin, "The design, deployment, and evaluation of the CORAL pre-exascale systems," in *Proceedings - International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018*, 2019.

[6] W. Cunningham, "Cars drive autonomously with Nvidia X1-based computer," 2015. [Online]. Available: https://www.cnet.com/roadshow/news/cars-drive-autonomously-with-nvidia-x1-based-computer/.

[7] J. H. Laros III, K. Pedretti, S. M. Kelly, W. Shu, K. Ferreira, J. Van Dyke and C. Vaughan, Energy-Efficient High Performance Computing, 1st ed., London: Springer-Verlag, 2013.

[8] M. Philips, "Stock Trading Is About to Get 5.2 Milliseconds Faster; A new ultrafast Atlantic cable will cater to electronic traders," Bloomberg, 29 March 2012. [Online]. Available: https://www.bloomberg.com/news/articles/2012-03-29/stock-trading-is-about-to-get-5-dot-2-milliseconds-faster.

[9] N. Chong, "cudahook," 2014. [Online]. Available: https://github.com/nchong/cudahook.

[10] NVIDIA Corp., "NVIDIA Tesla V100 GPU Architecture," 2017.

[11] NVIDIA Corp., "CUDA C Programming Guide, Design Guide," 2018.

[12] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," *SIGARCH Comput. Archit. News,* vol. 41, no. 3, pp. 332-343, 6 2013.

[13] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer and C. R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," *SIGPLAN Not.,* vol. 48, no. 4, pp. 395-406, 3 2013.

[14] B. Wu and X. Shen, "Software-level task scheduling on GPUs," in *Advances in GPU Research and Practice*, 1st ed., San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2016.

[15] T. Amert, N. Otterness, M. Yang, J. H. Anderson and F. D. Smith, "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed," in *2017 IEEE Real-Time Systems Symposium (RTSS)*, 2017.

[16] J. Bakita, N. Otterness, J. H. Anderson and F. D. Smith, "Scaling Up: The Validation of Empirically Derived Scheduling Riles on NVIDIA GPUs," in *OSPERT'18*, 2018.

[17] S. Hong and H. Kim, "An integrated GPU power and performance model," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010.

[18] J. T. Adriaens, K. Compton, N. S. Kim and M. J. Schulte, "The case for GPGPU spatial multitasking," in *IEEE International Symposium on High-Performance Comp Architecture*, 2012.

[19] L. Eeckhout, "Computer architecture performance evaluation methods," *Synthesis Lectures on Computer Architecture,* vol. 5, no. 1, pp. 1-145, 2010.

[20] NVIDIA Corp., "nvidia-smi - NVIDIA System Management Interface program," [Online]. Available: https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf.

[21] NVIDIA Corp., "CUDA Profiler Users Guide v10.2," 2019.

[22] 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 *2009 IEEE international symposium on workload characterization (IISWC)*, 2009.

[23] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *IEEE International Symposium on Workload Characterization (IISWC'10)*, 2010.

[24] T. Li, V. K. Narayana and T. El-Ghazawi, "Exploring graphics processing unit (GPU) resource sharing efficiency for high performance computing," *Computers,* vol. 2, no. 4, pp. 176-214, 2013.

[25] T. Li, V. K. Narayana and T. El-Ghazawi, "Symbiotic scheduling of concurrent GPU kernels for performance and energy optimizations," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, 2014.

[26] S. Pai, M. J. Thazhuthaveetil and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," *ACM SIGARCH Computer Architecture News,* vol. 41, no. 1, pp. 407-418, 2013.

[27] 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,* vol. 26, no. 3, pp. 748-760, 2014.

[28] M. Butler, K. Sajjapongse and M. Becchi, "Improving application concurrency on GPUs by managing implicit and explicit synchronizations," in *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, 2016.

[29] M. Harris, "How to Optimize Data Transfers in CUDA C/C++," 2012. [Online]. Available: http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/.

[30] M. Boyer, "Memory transfer overhead," [Online]. Available: https://www.cs.virginia.edu/~mwb7w/cuda_support/memory_transfer_overhead.html.

[31] NVIDIA Corp., "NVIDIA's Next Generation CUDA Computer Architecture: Kepler GK110/210," 2014.

[32] R. Di Pietro, F. Lombardi and A. Villani, "CUDA leaks: Information leakage in GPU architectures," *arXiv preprint arXiv:1305.7383,* 2013.

[33] F. Wende, T. Steinke and F. Cordes, "Multi-threaded Kernel Offloading to GPGPU Using Hyper-Q on Kepler Architecture," 2014.

[34] 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 *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[35] M. E. Belviranli, F. Khorasani, L. N. Bhuyan and R. Gupta, "CuMAS: Data transfer aware multi-application scheduling for shared GPUs," in *Proceedings of the International Conference on Supercomputing*, 2016.

[36] M. Boyer, J. Meng and K. Kumaran, "Improving GPU Performance Prediction with Data Transfer Modeling," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 2013.

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

[38] R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction, MIT press, 2018.

[39] C. Watkins, "Learning from delayed rewards," University of Cambridge, England, 1989.

[40] A. Munir, S. Ranka and A. Gordon-Ross, "High-performance energy-efficient multicore embedded computing," *IEEE Transactions on Parallel and Distributed Systems,* vol. 23, no. 4, pp. 684-700, 2012.

[41] C. L. Usmail, M. O. Little and R. E. Zuber, "Evolution of embedded processing for wide area surveillance," *IEEE Aerospace and Electronic Systems Magazine,* vol. 29, no. 1, pp. 6-13, 2014.

[42] M. Barnell, C. Raymond, C. Capraro and D. Isereau, "Agile Condor: A Scalable High Performance Embedded Computing Architecture," in *2015 IEEE Conference on High Performance Extreme Computing (HPEC)*, Waltham, MA, USA, 2015.

[43] A. C. Pineda, J. K. Mee, P. M. Cunio and R. A. Weber, "Benchmarking image processing for space: Introducing the SPACER architecture laboratory," in *2016 IEEE Aerospace Conference*, 2016.

[44] E. Kain, D. Wildenstein and A. C. Pineda, "Embedded GPU Cluster Computing Framework for Inference of Convolutional Neural Networks," in *2019 IEEE High Performance Extreme Computing Conference (HPEC '19)*, Waltham, MA, USA, 2019.

[45] S. Rennich, "CUDA C/C++ Streams and Concurrency," 2011. [Online]. Available: http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf.

[46] M. Harris, "GPU Pro Tip: CUDA 7 Streams Simplify Concurrency," 2015. [Online]. Available: https://devblogs.nvidia.com/gpu-pro-tip-cuda-7-streams-simplify-concurrency/.

[47] R. A. Cruz, C. Bentes, B. Breder, E. Vasconcellos, E. Clua, P. M. de Carvalho and L. M. Drummond, "Maximizing the GPU resource usage by reordering concurrent kernels submission," in *Concurrency Computation* , 2019.

[48] P. B. Hansen, Operating System Principles, USA: Prentice-Hall, Inc., 1973.

[49] G. Strang, Calculus, Wellesley, MA: Wellesley-Cambridge Press, 1991.

[50] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland and G. Ostrovski, "Human-level control through deep reinforcement learning," *Nature,* vol. 518, no. 7540, pp. 529-533, 2015.

[51] H. Mao, M. Alizadeh, I. Menache and S. Kandula, "Resource management with deep reinforcement learning," in *HotNets 2016 - Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 2016.

[52] L.-J. Lin, "Self-improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching," *Machine Learning,* vol. 8, no. 3-4, pp. 293-321, 1992.

[53] N. Liu, Z. Li, J. Xu, Z. Xu, S. Lin, Q. Qiu, J. Tang and Y. Wang, "A Hierarchical Framework of Cloud Resource Allocation and Power Management Using Deep Reinforcement Learning," in *Proceedings - International Conference on Distributed Computing Systems*, 2017.

[54] Q. Jiao, M. Lu, H. P. Hyunh and T. Mitra, "Improving GPGPU Energy-Efficiency through Concurrent Kernel Execution and DVFS," in *Proceedings of the 2015 Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO) : CGO 2015*, San Francisco, California, USA, 2015.

[55] J. Zhong and B. He, "Kernelet: High-Throughput GPU Kernel Executions with Dynamic Slicing and Scheduling," *IEEE Transactions on Parallel and Distributed Systems,* vol. 25, no. 6, pp. 1522-1532, 2014.

[56] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho and S. Ryu, "Improving GPGPU resource utilization through alternative thread block scheduling," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[57] F. Wende, F. Cordes and T. Steinke, "On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering," in *Symposium on Application Accelerators in High-Performance Computing*, 2012.

[58] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, 2011.

[59] B. V. Werkhoven, J. Maassen, F. J. Seinstra and H. E. Bal, "Performance models for CPU-GPU data transfers," in *Proceedings - 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2014*, 2014.

[60] M. Martinasso, G. Kwasniewski, S. R. Alam, T. C. Schulthess and T. Hoefler, "A PCIe congestion-aware performance model for densely populated accelerator servers," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[61] M. Chetlur, N. Abu-Ghazaleh, R. Radhakrishnan and P. A. Wilsey, "Optimizing communication in Time-Warp simulators," in *Proceedings. Twelfth Workshop on Parallel and Distributed Simulation PADS'98 (Cat. No. 98TB100233)*, 1998.

[62] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam and M. Lanctot, "Mastering the game of Go with deep neural networks and tree search," *nature,* vol. 529, no. 7587, p. 484, 2016.

[63] L. Huang, X. Feng, L. Qian and Y. Wu, "Deep reinforcement learning-based task offloading and resource allocation for mobile edge computing," in *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, 2018.

Vita

Author's Name: Ryan Seamus Luley
Place of Birth: New Hartford, NY, USA
Date of Birth: April 21, 1979

Degrees Awarded:

- Bachelor of Science, Mathematics and Computer Science, St. Lawrence University, 2001
- Master of Science, Computer Engineering, Syracuse University, 2008

Professional Experience:

- Senior Mathematician, Air Force Research Laboratory Information Directorate, January 2019 – present
- Mathematician, Air Force Research Laboratory Information Directorate, May 2008 – January 2019
- Mathematician, Air Force Research Laboratory Sensors Directorate, Jan 2007 – May 2008
- Associate Mathematician, Air Force Research Laboratory Sensors Directorate, Mar 2003 – Jan 2007
- Software Engineer Associate, Lockheed Martin Management & Data Systems, Aug 2001 – Mar 2003