Concurrent Deep Learning Workloads on NVIDIA GPUs

by

Guin Gilman

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2021

APPROVED:

Professor Robert J. Walls, Major Thesis Advisor

Professor Charles Davis Roberts, Thesis Reader

Professor Craig E. Wills, Head of Department

Abstract

Deep learning GPU servers that execute latency-sensitive inference requests from clients often seek to run training tasks alongside inference when there are idle resources in order to improve overall system utilization. We empirically derive the thread block scheduler's behavior under such concurrent workloads for NVIDIA's Pascal, Volta, and Turing microarchitectures. In contrast to past studies that suggest the scheduler uses a round-robin policy to assign thread blocks to streaming multiprocessors (SMs), we instead find that the scheduler chooses the next SM based on the SM's local resource availability. We show how this scheduling policy can lead to significant, and seemingly counter-intuitive, performance degradation; for example, a decrease of one thread per block resulted in a 3.58X increase in execution time for one kernel in our experiments. We then investigate the performance of current concurrency mechanisms on NVIDIA's new Ampere microarchitecture under deep learning workloads and demonstrate that fluctuating resource requirements and kernel runtimes make executing such workloads while maintaining consistently high utilization and low, predictable turnaround times difficult on current NVIDIA hardware. Moreover, we conclude that the lack of sufficiently flexible preemption policies, robust task prioritization mechanisms, and contention-aware thread block scheduling techniques limits the effectiveness of NVIDIA's concurrency mechanisms. We estimate that through the use of block-level, contention-aware preemption, it is possible to achieve 1.5X speedups in turnaround time with comparable utilization and improved predictability, as long as preemption overhead remains under 1-2ms.

Acknowledgements

I would like to thank my advisor for his guidance and insight during this process, which has been an immeasurable contribution to making this work one which I am proud of. I also thank Tian Guo and Samuel Ogden for their feedback and constant support, and for their contributions to a great portion of the earlier parts of this work. I would additionally like to thank Professor Charles Robert as the second reader of this thesis for providing valuable input that improved it greatly. The members of the CAKE Lab at WPI have also given me a tremendous amount of support, feedback, and opportunities to share my work that I deeply appreciate. I thank both the faculty and students for their support. Finally, I would also like to thank my family for providing me with encouragement throughout this endeavor, especially my sister Meredith. This accomplishment would have been impossible without them. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

GPU-based servers are attractive for deep learning applications because such systems (e.g., TensorRT [28]) offer high-throughput and low latency for inference requests. However, servicing such requests may not consistently utilize the entire GPU. To make use of such idle resources, recent works have proposed simultaneously running multiple model training tasks on a single GPU [36], which could be extended to inference servers. This requires GPUs to have a scheduling policy that guarantees the latency-sensitive inference requests are completed in time to assure quality of service for the user, while also consistently making use of the unoccupied resources by applying them to resource-hungry training tasks. However, NVIDIA GPUs have a limited set of tools for running and managing multiple independent applications concurrently on a single GPU. Further, it is unclear how these existing mechanisms perform with the aforementioned concurrent training and inference deep learning workloads—workloads which, as we show in this work, have fluctuating resource requirements, variable kernel runtimes, and frequent sequential kernel launches that make existing mechanisms difficult to utilize efficiently in an inference server context.

We examine concurrency on NVIDIA GPUs at two levels: the application level and the kernel level. Despite being capable of running more than one application simultaneously, there is only a limited set of circumstances in which kernels from separate applications are able to share the resources of the GPU at the same time, and in the case of time-slicing, it is not possible at all. Despite this, concurrent kernel execution—i.e., running kernels from separate streams at the same time on the same device—has often been proposed as a means to improve the utilization of general purpose GPUs [37, 3, 31, 34, 8, 30, 16, 6, 7]. In order to take full advantage of kernel concurrency, the scheduler must make intelligent decisions to efficiently divide the GPU's limited resources among the kernels. Suboptimal decisions by the scheduler can lead to inefficiencies that impact kernel performance. However, characterizing the performance implications of such concurrency is challenging due, in large part, to the black-box nature of NVIDIA's proprietary thread block scheduler.

We therefore use empirical observations of real hardware to infer the policies of the thread block scheduler on the Pascal, Volta, and Turing GPU microarchitectures in Chapter 4. We find, for example, that the scheduler chooses where to assign a thread block based on the local resource availability of the streaming multiprocessors (SMs)—we call this the *most-room policy*. In contrast, most literature assumes that the scheduler uses a simple round-robin policy [24, 4, 22]. Our observations lead to the following conclusion: *the performance of a kernel in a concurrent workload is challenging to predict because the performance depends on factors that are external to the kernel itself.* Such factors include *(i)* the scheduling policies of the thread block scheduler; *(ii)* the potential for resource contention across myriad hardware resources; and *(iii)* the impact of possibly unpredictable effects such as kernel launch timing.

At the application level, we examine the three concurrency mechanisms currently

available on NVIDIA hardware—priority streams, time-slicing, and MPS—and evaluate their performance on deep learning workloads on the new Ampere and Turing microarchitectures in Chapter 5. (We omit Volta due to the high degree of similarity to Turing it possesses.) We find them unable to meet the demands of an inference server in terms of turnaround time, utilization, and predictability due to a lack of flexibility and insufficient preemption and prioritization policies. In summary, priority streams are not meant to be utilized by applications with fluctuating or high resource requirements, leading to the kernels of the higher-priority inference task experiencing compounded delay as they are forced to wait behind blocks of training task kernels for GPU resources. Time-slicing disallows separate applications from utilizing GPU resources simultaneously altogether, making it difficult to improve utilization from a serial execution case. MPS is designed for use when the concurrent tasks all have consistently low resource requirements, which we show is not the case for deep learning workloads. None of the mechanisms provide sufficient task prioritization to accomplish the goal of maintaining QoS when servicing inference requests.

With these limitations in mind, we propose a more flexible set of preemption strategies for concurrent deep learning workloads in Chapter 6. We argue the need for changes to NVIDIA hardware to enable per-block preemption that can prioritize latency-sensitive tasks and present a series of strategies to be applied to different types of kernels that appear in deep learning workloads. We suggest an SM-filling policy for large kernels that are contentious and compete directly for per-SM resources like computational units, a round-robin policy for large kernels with low contention whose blocks can share SMs with less significant degradation, and an SM-division policy for smaller kernels that are not able to utilize all available GPU resources. We further discuss the theoretical performance gains that are possible by

making such changes to the behavior of the NVIDIA hardware scheduling hierarchy, demonstrating that it is possible to achieve up to 1.5X speedups in turnaround time with comparable or improved utilization and predictability.

Our efforts differ from much prior work in that the preemption strategies presented in this work are specifically tailored to the use case of deep learning inference server, where one task is latency-sensitive and the other is best-effort. In contrast to many existing spatial sharing policies [3, 37, 16], we introduce task prioritization and individual block-level preemption to achieve efficient performance on inference server workloads with unique traits, such as differences in latency sensitivity and task arrival times. We differ from previously proposed task preemption strategies [34, 31, 35] that are unable to account for block placements that lead to degradation in performance due to resource contention.

The remainder of this work is structured as follows. Chapter 2 provides a description of the CUDA programming model, as well as explanations of the three high-level concurrency mechanisms examined in this work, and present related work in Chapter 3. Our derivation of the most-room policy and analysis of its performance implications for kernel concurrency are in Chapter 4. In Chapter 5, we detail the characteristics of deep learning workloads that influence their performance with different application-level scheduling techniques and analyze the three concurrency techniques available on NVIDIA devices. We then argue the need for fine-grained preemption and examine a set of preemption policies designed to increase the efficiency of concurrent deep learning workloads, and provide estimated performance gains. We then present our conclusions in Chapter 7.

Chapter 2

Background

The following section provides a brief overview of the CUDA programming model for GPU computing, primarily focusing on NVIDIA devices of the Turing [2] and Ampere [1] microarchitectures. It also describes the three techniques currently available for executing multiple tasks concurrently on one GPU: priority streams, time-slicing, and MPS.

2.1 CUDA Programming Model

CUDA is a programming model for GPU computing on NVIDIA devices. We provide a brief overview of the terminology and workflow of the CUDA programming model and NVIDIA GPUs, focusing our discussion on GPUs from the recent Pascal, Volta, Turing, and Ampere microarchitectures; the first devices based on these architectures were released in 2016, 2017, 2018, and 2020, respectively. We provide only the details that are necessary to understand the behavior of the thread block scheduler and concurrency mechanisms examined in this work.

Kernels, Thread Blocks, and Warps. A *kernel* in CUDA programming is the term for the code which is executed on the GPU. A kernel is comprised of a

logical array of independent *thread blocks*, known as a *grid*, that each execute the same block of code in parallel on different subsets of data. A *warp* is a group of 32 threads within a block, and instructions are issued per warp, meaning that a warp is a group of threads which execute in parallel on the GPU.

Streaming Multiprocessors. The GPU executes a kernel by scheduling the thread blocks to hardware units of computation known as *streaming multiprocessors (SMs)*. Each SM in a GPU from the Ampere architecture has four warp-scheduler units, which can each issue instructions to a warp every two cycles [1]. An SM additionally has a fixed set of resources and resource limits, such as threads, shared memory, and registers. During execution, blocks are scheduled under the constraint that the total resource requirements of the *resident* blocks on the SM cannot exceed any one of the SM's resources. For example, consider a GPU that supports 2048 threads and 32 blocks per SM and a kernel with 64 blocks of 32 threads. In this scenario, a single SM can handle at most 32 blocks from the kernel (given the 32 blocks per SM limit) which means that during execution only half of the SMs threads are being utilized (i.e., 32 blocks x 32 threads). The SMs are grouped into Texture Processing Units (TPCs), and these TPCs are further grouped into Graphics Processing Units (GPCs). The number of SMs per TPC and GPC varies per card. An SM is considered to be *saturated* if it can schedule no further blocks due to a lack of the required resources. We consider two blocks to be *colocated* if they are being executed on the same SM simultaneously.

Streams. A *stream* is a sequence of commands which must be executed in issue-order on the GPU; more than one stream can exist at a time, and operations across streams are asynchronous and independent. In a system with a *discrete* GPU, where the GPU is a separate device from the CPU and connected by a link such as PCIe, the kernel and any data it operates on must be transferred to the GPU using

streams. Two kinds of commands can be issued to a stream by a CUDA program: a *data transfer* command, which causes data to be migrated between the GPU and CPU over the PCIe link; and a *kernel dispatch* command, which causes a kernel to be transferred to and executed on the GPU.

Thread Block Scheduler. The *thread block scheduler* is responsible for assigning thread blocks to SMs to be executed. A new block is assigned as soon as the resources become available on some SM [22, 4]. Thus, the thread block scheduler must be aware of the remaining resources of each SM. Once a thread block has been assigned to an SM, groups of 32 threads called *warps* are scheduled to the SM's execution cores by the SM's own *warp scheduler*.

Execution and Copy Queues. There are two separate queues for data transfer commands and kernel dispatches. The former is called the *copy queue* and the latter is called the *execution queue*, and all streams launch kernels and data copy commands to these queues. These operations are independent from each other and performed in the order of their arrival to the respective queues [4]. The only time when the kernel's launch time is not the deteriminant factor for their placement in the execution queue if stream priorities are introduced, in which case the high-priority stream will be treated as if it were launched first [22]. The two queues allow data transfer and kernel execution to take place independently of each other, but the order of arrival still affects the scheduling outcome and therefore the performance of the kernels.

Concurrent Kernel Execution. Broadly, *concurrent kernel execution* is the act of running kernels from separate streams at the same time on the same GPU. Note that kernel concurrency is only possible for kernels from the same *CUDA context*, which is analogous to a CPU process, and contains all resources and actions performed within the CUDA driver API.

This definition of concurrent kernels does not include the case of a single kernel overlapping data transfer and execution; though, a concurrent kernel may employ these techniques and we explore the impact of such behavior in Section 4.3.5. Further, this definition does not include kernels that are executed serially on the same stream, i.e., one kernel's input depends on the output of a previous kernel. For example, a single application may use one kernel to calculate a large matrix multiplication and a separate kernel to further process the result, and launch one after the other on a single stream.

We highlight the distinction between the concept of *data parallelism* from parallel programming—i.e., processing large datasets wherein individual data points can be processed largely independently—and kernel concurrency [19]. In particular, GPUs and the kernels that run on them are designed to exploit data parallelism. Again, only kernels which are separate and independent from each other and occupying the GPU at the same time qualify as concurrent.

2.2 NVIDIA Concurrency Mechanisms

We refer to executing two independent applications simultaneously on one GPU as *concurrent application execution*. An application is defined as a set of one or more kernels to be executed serially with possible dependencies between them. Concurrent application execution can involve sharing the GPU in terms of space, time, or both.

When two kernels from separate applications are executed at the same time on a single GPU, this is referred to as *concurrent kernel execution*. Note that this is only possible through the use of an MPS server, or for kernels from within the same CUDA context using streams. Thread blocks of kernels from separate processes run on a GPU with no MPS server cannot occupy the GPU at the same time. Thus,

concurrent application execution can include concurrent kernel execution but does not necessarily.

Priority Streams. In this approach, the kernels of the two applications are launched from within the same process on different streams. Streams can be defined with one of three priorities; higher-priority streams will preempt the execution of lower-priority streams at the boundary of a thread block. This means that when a high-priority stream arrives at the GPU while a low-priority stream is already occupying the computational resources, the high-priority stream's thread blocks will be scheduled after the blocks of the low-priority stream finish, even if the low-priority stream has unexecuted thread blocks remaining. The high priority stream cannot interrupt the execution of the lower priority stream's thread blocks which are already being executed as it arrives. It is important to note that with this mechanism, the applications must be launched from within the same process; despite this, the use of separate streams makes their execution independent from each other to the extent that they are not reliant on the progress of the other to make progress in their own execution. However, they are still sensitive to competition for GPU resources.

Time-Slicing. When two applications are run as separate processes (with separate CUDA contexts), they cannot share the GPU's resources as their thread blocks cannot be located on the GPU at the same time. However, they can share the GPU through time-slicing. The CUDA application-level scheduler will alternate between the processes over time, yielding the GPU's resources completely to one process for the fixed length of a time-slice [32, 9]. The total resources required by the set of processes needs to be less than the resource limits of the GPU; if a process is launched that needs more resources combined with all other currently executing processes than a GPU has to offer, it will receive an error instead of being scheduled. This is because the resources such as shared memory and registers that are used by a

process are not transferred on and off the GPU between time-slices.

Multi-Process Service. MPS allows applications run as separate processes to share the GPU; an MPS server is run on the target GPU, and each process that starts on that GPU is scheduled by that MPS server. This differs from time-slicing in that the thread blocks of kernels from separate processes can now occupy the GPU at the same time, possibly even sharing an SM; in contrast to priority streams, the kernels can be from separate CUDA contexts. The MPS server can be configured to limit the number of threads that can be used by any one client; for example, it can be set so that each client can use no more than 50% of the total amount of threads offered by the GPU. However, this limit cannot be set for each process launched by a client individually. NVIDIA recommends that this limit be set to $100\%/0.5n$, where n is the number of clients, to allow the load balancer to overlap execution between clients whenever there are idle resources. MPS is recommended by NVIDIA to be used in cases where the kernels utilize less than the total available resources of the GPU in order to achieve resource saturation.

Chapter 3

Related Work

There are two broad categories of works which are related to this one: work which focuses on using empircal analysis to reverse-engineer components of the scheduling hierarchy on actual NVIDIA hardware, and work which proposes or characterizes different multiplexing techniques or scheduling policies to more efficiently execute concurrent workloads.

3.1 Reverse-Engineering NVIDIA Hardware

A significant amount of work has been done to understand the hardware scheduler when executing a single CUDA application. When assigning thread blocks to SMs, the only kernel whose blocks can be scheduled at any given time is the one at the head of the execution queue [4, 22], and a kernel is not removed from the execution queue until all of its blocks have been scheduled [4]. Blocks are typically assumed to be assigned to SMs in a round-robin manner [24, 4, 22], which results in the fewest number of blocks assigned per SM for a given kernel. This continues until there is not enough of any given resource (e.g. threads, shared memory, registers, etc.) on any of the SMs to schedule another block [4]. Additional blocks are scheduled as

soon as enough resoures become available to do so due to another block completing its execution [4, 22].

It has been widely observed that the scheduler uses a *leftover policy* when scheduling blocks from kernels launched on different streams [24, 37, 4, 22]. This means that, because only blocks from the kernel at the front of the execution queue can be scheduled, the blocks of other kernels in the queue will not be scheduled until all of the blocks from the current kernel have been, even if there is space for them but not for any more blocks of the current kernel. There is no form of preemption [4]; the queue cannot be skipped, and blocks cannot be paused or stopped partway through their execution. Making use of priorities, where a stream can either be high or low prioritiy, works as though the high priority stream is added to the front of the execution queue [4, 22]. In other words, its blocks are the next to be scheduled, and no other kernels' blocks can be scheduled before the high priority kernel's blocks have all been scheduled.

Olmedo et al. [32], in contrast, describe in detail all levels of the NVIDIA scheduling hierarchy. It contains three levels: the warp-scheduler, the thread block scheduler, and the application scheduler. This is also the only work that provides evidence that the thread block scheduler does not use a purely round-robin policy to place blocks on SMs. Additionally, it provides evidence that the copy and execution queues are scheduled independently from each other, and that this can cause interference in the execution of applications which rely on both in the concurrent context.

The time-slicing application scheduler is detailed by Capodieci et al. [9]. Each application maps to a set of channels, and each channel is associated with a particular time slice length and interleaving level. The runlist is a list of the channels to check for work in order. To emulate the behavior of a higher priority application being worked on more, its channels can have a higher interleaving level, meaning

that it shows up in the runlist more often. So for every lower priority application that appears in the runlist, every higher priority application has to appear before it as many times as its interleaving level dictates. The time slice length is how long work from the channel is computed until the channel is preempted. Additionally, when preempting, context is saved, including the constant memory, shared/L1 data, registers, and L2 data.

3.2 Concurrent Scheduling Policies

Techniques for remedying the lack of efficient mechanisms for concurrent application execution on NVIDIA GPUs can be divided into two broad categories: time-based multiplexing and space-based multiplexing. Space-multiplexing focuses on efficiently sharing GPU resources between kernels with techniques for intra-SM scheduling, thus improving the utilization of the GPU over time, while time-multiplexing methods are based on improving the turnaround time of various general-purpose task sets.

3.2.1 Temporal Multiplexing

Time-multiplexing methods are a technique used for improving turnaround time as opposed to utilization, and there are a number of approaches which have been taken to enable preemption on GPUs. The first of these is context-switching, in which a currently-executing block's execution context is saved in some reserved section of global memory so that it can be abandoned and another block can be scheduled in its place [34]. The cost of this is the memory access throughput which is necessary for saving the state (the execution contexts of every thread, the shared memory, the register file, etc. will all use up a not unreasonable amount of the

available bandwidth for global memory accesses). While the transfer is occurring, not only can that bandwidth not be used, the resources can't, so computation time is sacrificed. This cost is typically thought of as so prohibitive as to render the benefits of preemption on GPUs worthless [30, 3].

The second technique is SM-draining, where once a kernel is designated for preemption, its currently-executing blocks finish executing, but no new ones are scheduled, and the new kernels' blocks get scheduled instead [34]. This avoids the overhead of saving the execution state, since a kernel's blocks are all independent and execution can be resumed by continuing to schedule the rest of the interrupted kernel's blocks, but it comes at the cost of the remaining execution time left for all of the currently-executing blocks.

Lastly, there is SM-flushing, which involves abandoning the execution of the blocks on the preempted kernel entirely [31, 30], meaning that the cost is no longer the remaining execution times of the blocks, but the recomputation that will have to occur when the blocks are re-scheduled later. Additionally, this only works for blocks that are idempotent at the point in time in which they are preempted [31, 30]. Due to the different costs and benefits of these approaches, using different techniques for different scenarios in order to achieve the lowest cost for preemption at any given moment is more efficient than choosing one over the others at all times [31]. However, time multiplexing on its own has been shown to provide less speedup than space multiplexing techniques overall [16], and these techniques on their own do not guarantee the optimal assignment of resources at any given point in time, only an efficient way to preempt kernel execution if necessary.

Another technique for enabling preemption involves the use of a technique called persistent threads programming (PTP), which was developed as a way of bypassing the proprietary hardware thread scheduler on NVIDIA devices and allowing the

programmer to influence the scheduling of work to the GPU through software [35]. This is done by scheduling exactly as many blocks as can fit on the SMs of the GPU in one round (termed as 'thread groups' instead of thread blocks to differentiate them from the traditional sense of CUDA thread blocks, as these are persistent blocks that occupy the GPU until the entire kernel's runtime is complete). These thread groups pick tasks from a work queue, where tasks are a subset of the work of the kernel that would originally have been performed by one standard thread block. The thread groups keep performing tasks from this work queue until it is empty, at which point they terminate. [35]. This allows the scheduler to place all of the kernel's blocks on the GPU at once, and makes interrupting and resuming them simpler, but requires rewriting the kernels themselves, which is not always possible given the use of proprietary libraries such as CuDNN in many real deep learning workloads.

Other approaches involve reordering the kernels in the execution and copy queues to avoid serialization due to data transfer dependency bottlenecks. For example, one approach is to separate the copy operations and kernel execution from one application into two different CUDA streams [30]; another is to group dependent operations into tasks, which can then be reordered in the queue to achieve a greater degree of overlap [8]; and a third is to reorder GPU operations and launch them to different CUDA streams to better overlap computation [21]. While these approaches do achieve better PCIe bandwidth utilization and overall execution time, they do not directly address the problem of SM resource underutilization.

Some deadline-based approaches exist [9], although for specific architectures with more limited resources than those available on a server GPU. None of these approaches are tailored to the specific type of deep learning workloads we are interested in scheduling efficiently. The closest to this area would be Xiao et al.'s scheduler

for GPU server clusters, which dynamically scales the resource requirements of deep learning jobs at runtime and then schedules high-priority jobs and best-effort jobs cooperatively in the cluster through over-provisioning [36]. However, this approach does not consider any lower aspects of the NVIDIA scheduling hierarchy such as the thread block scheduler which are necessary to maintain contention-aware block placement.

3.2.2 Spatial Multiplexing

For space multiplexing, the GPU shares its resources between the blocks of multiple kernels concurrently, thus improving resource utilization of the GPU over time. One way to accomplish this is inter-SM slicing, which means dividing the SMs among the kernels and scheduling blocks to run on the SMs which were assigned to that block's kernel. Assigning kernels a number of SMs based on profiles of their performance relative to resource assignment in isolation achieves a higher turnaround time than cooperative multitasking by itself, which is a method of preemption in which applications voluntarily yield to others at times as when they are idle. However, even simpler methods such as assigning all the kernels an equal number of SMs also achieve a lesser but not insignificant speedup in turnaround time [3].

However, this does not solve the problem of wasted resources per-SM, as most kernels do not use up every available resource that an SM has. Intra-SM slicing addresses this by assigning blocks from different kernels to the same SM to execute concurrently instead of assigning kernels to SMs. The division of blocks between kernels on an SM is not optimal when using first-come first-served, leftover, or an even-partitioning strategy (where each kernel gets a fixed and equal amount of the SM's resources) [37]. Running portions of the kernels to profile their performance relative to resource availability, and then assigning kernels' blocks to SMs using a

water-filling approach to minimize the performance loss of any given kernel achieves higher resource utilization and improved turnaround time [37].

Separate from these two approaches are those which are based on modifying or profiling the kernels themselves in order to reduce contention and interference when they are run simultaneously. Elastic kernels are kernels which are able to be, at a fine granularity, transformed into kernels which use a less contentious amount of an SM's resources. This is accomplished by mapping 2D logical grids and 3D logical thread blocks into 1D versions [30]. Approaches which analyze colocated kernels as they run and adjust the scheduling to meet QoS requirements tend to be generalized enough that they assume no knowledge about the behavior of the concurrent application's kernels which could be used to prevent contention [38], which is a tighter restriction than necessary in the case of a deep learning inference server.

There has also been some preliminary work to suggest that a combination of time and space multiplexing can lead to even greater turnaround time and resource utilization than either in isolation [16]. However, all of these techniques typically assume a task set that is fixed, with uniform start times, making them difficult to apply to an inference server without also developing a sufficient preemption and eviction policy, given the stochastic nature of the inference requests.

Chapter 4

Kernel Concurrency

Improving utilization requires that the kernels of two separate applications be scheduled together, with thread blocks arranged onto SMs to occupy as many resources as possible. However, NVIDIA GPUs severely limit the ability to share SMs between blocks of separate kernels, and due to being proprietary in nature, it is difficult to know how thread block scheduling choices are made.

Thus, in this chapter, we examine the performance of kernel-level concurrency on NVIDIA devices. We characterize the behavior of the hardware thread block scheduler on GPUs under concurrent kernel workloads in Section 4.2. We introduce the *most-room policy*, a previously unknown scheduling policy used to determine the placement of thread blocks on SMs. We define the most-room policy as follows:

The most-room policy dictates that a kernel block will be scheduled to the streaming multiprocessor that, at the time of scheduling, can support the most blocks from the current kernel, with only one block scheduled to that SM at a time. This calculation takes into account each SM's current resource availability, but it does not account for potential resource contention with blocks already on the SM. This policy breaks ties between SMs using a pre-defined device-specific ordering.

We then examine the performance implications of the most-room policy under concurrent kernel workloads in Section 4.3. We demonstrate that the policy can result in counter-intuitive performance drops with only small changes made to the structure of the concurrent kernels. For example, a decrease of one thread per block resulted in a 3.58X increase in execution time for one kernel in our experiments. Finally, we highlight the scheduler's impact on concurrent kernel workloads with purpose-built kernels that emulate common classes of general purpose GPU kernels: L1-cache-dependent, compute-intensive, memory-intensive, and PCIe-bandwidth-dependent. We found performance differences due to resource contention between kernels and a lack of kernel-level fairness.

4.1 Methodology

We selected three GPUs which are representative of NVIDIA's three recent microarchitectures: Pascal, Volta, and Turing. These three GPUs represent a range of use cases; the Pascal GPU is found in laptops, the Volta GPU is used in cloud computing servers, and the Turing GPU is a high-end desktop GPU. All three are discrete GPUs. We ran a similar set of experiments on all three devices, with adjustments to tailor the workload to the specific hardware capabilities and resource limits of the GPU under observation. See Appendix A.2 for a summary of each device's architectural details.

We identified individual streaming multiprocessors (SMs) using the `smid` register, which returns a unique value for each SM. We differentiated thread blocks by their `blockIdx` values, a predefined tuple of identifiers for each thread block. We use SM0 to denote the SM with id 0, and B0 to denote the block of a kernel B with a `blockIdx.x` value of 0. Our experiments in this work fall into two categories:

deriving the scheduler's policy and characterizing the performance impact of the derived policy.

4.1.1 Deriving the Most-Room Policy

The results presented in Section 4.2 are based on the following empirical methodology.

To derive the most-room scheduling policy of the thread block scheduler, we used a basic workload structure consisting of two kernels, X and Y, for our experiments, where each kernel was launched on a separate CUDA stream. In all cases, Kernel X was launched first and followed later by Kernel Y.

The kernels consisted of code that used the `globaltimer` register to spin each block for a number of seconds proportional to the id of the assigned SM. In particular, this difference in block execution time guaranteed that the blocks for Kernel X would finish executing in the order in which they were assigned. Further, the timing guaranteed that Kernel Y's blocks were scheduled after the first block of Kernel X finished executing, but before any of Kernel X's other blocks had finished. In other words, at the moment Kernel Y was launched, SM0 was empty while SMs 1–n each contained exactly one of Kernel X's blocks. Kernel X consisted of a set of n blocks (where n was the number of SMs on the GPU), while Kernel Y had three thread blocks, so all of the SMs contained one block of Kernel X except the empty one. The number of threads per block and the execution times of these kernels were configured such that the placement of the blocks from Kernel Y allowed us to derive the scheduler's policy.

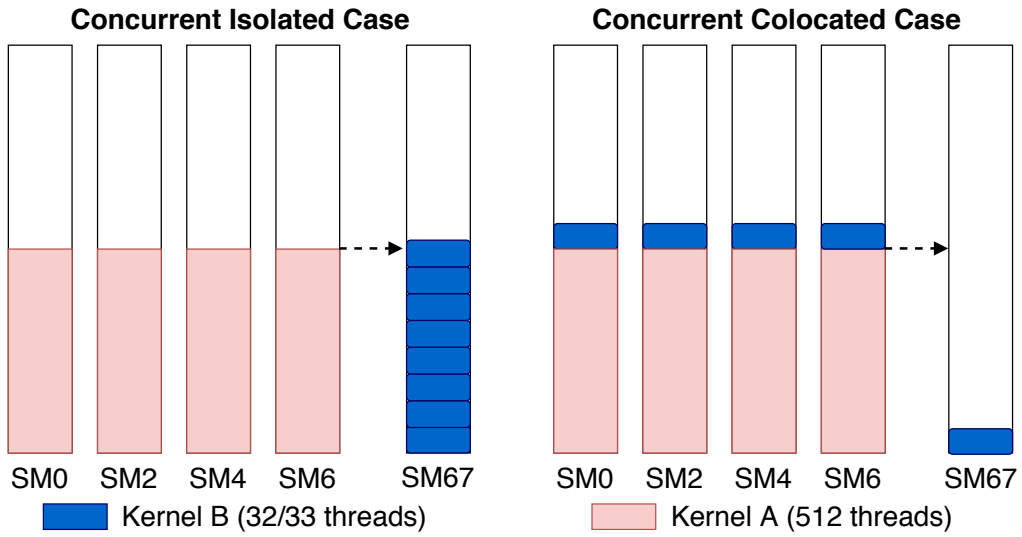


Figure 4.1: Concurrent Workloads Experiment. The experimental setup for the concurrent workloads on the Turing architecture. This example uses threads as the limiting resource.

4.1.2 Measuring Workload Performance

The results presented in Section 4.3 are based on the following empirical methodology.

To investigate the performance implications of the most-room policy, we designed a set of concurrent workloads with kernels whose block dimensions made their block placement sensitive to the most-room policy. We wrote these workloads instead of running kernels from an existing benchmark suite (e.g., Rodinia [10]) in order to have more control over the scheduling outcome and the particular resource under contention. We used the execution time of the individual kernels as the performance metric, measured with NVIDIA's kernel profiling tool `nvprof` [25]. We used four different classes of purpose-built kernels: L1-cache-dependent, compute-inten-sive, memory-intensive, and PCIe-transfer-dependent.

All of the performance experiments followed the same basic structure, an example of which is illustrated in Figure 4.1. First, each experiment consisted of two kernels

from separate applications, termed Kernel A and Kernel B. Note that these kernels were distinct from the Kernels X and Y described in Section 4.1.1. Kernel A was launched first, with $n - 1$ blocks, where n was the number of SMs on the GPU. This guaranteed that all $n - 1$ blocks were scheduled to a separate SM, leaving one empty SM remaining.

We varied the number and specific resource requirements of Kernel B's blocks, such that the scheduler assigned all of B's blocks to the empty SM in some experimental runs, and in other runs colocated B's blocks with Kernel A's blocks. We refer to these scenarios as the *concurrent-isolated case* and the *concurrent-colocated case*, respectively, and illustrate both in Figure 4.1. As a baseline, we also ran each kernel serially (i.e., without concurrency); we refer to this as the *serial case*. Note that in the serial case experiments, the blocks of Kernel B were scheduled to separate SMs.

4.2 The Most-Room Policy

Understanding the thread block scheduler requires answering the following questions. First, *when* does the scheduler choose to schedule another block? Second, *which* block does the scheduler choose? And third, *where* will that block be placed? It has been shown in previous work that the scheduler chooses *when* and *which* block using a leftover policy (see Section 3). However, in contrast to previous studies, we find that the scheduler chooses *where* to place a block based on the SMs' local resource availability; we call this behavior the *most-room policy*. Due to the black-box nature of the NVIDIA hardware, we draw our conclusions from empirical observations of the scheduler.

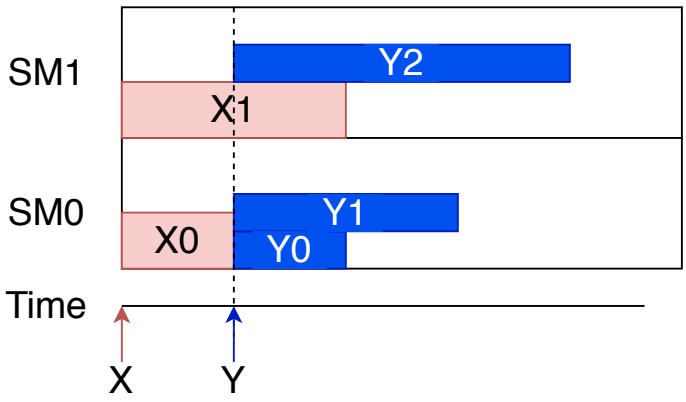


Figure 4.2: Most-Room Policy Experiment. An illustration of the experiment demonstrating the scheduler's most-room policy on the Pascal GPU. Here, SMs 2-4 were omitted for space, as they each contained only blocks of Kernel X.

4.2.1 A Demonstrative Experiment

We illustrate the most-room policy with the following experiment run on the Pascal GPU and depicted in Figure 4.2. For this experiment, we used a workload consisting of two kernels, A and B, which were launched in that order. Kernel X was composed of five blocks, as the Pascal GPU had five SMs, with 256 threads in each block. Block X0 (assigned to SM0 by the scheduler) always finished executing first, while block X4 (assigned to SM4) finished executing last. Kernel Y was composed of three blocks, each of 160 threads.

If the scheduler followed a pure round-robin policy, as is widely believed to be the case [24, 4, 22], then we would expect that blocks Y0, Y1, and Y2 would be placed on SM0, SM1, and SM2 respectively. Instead, the scheduler placed two blocks on SM0 and one block on SM1—a decision which, as we argue below, was based on each SMs' local resource availability.

Let us first consider why Y0 was scheduled to SM0. At the time of the decision, SM0 was empty and thus could support the maximum of 2048 threads, meaning that it had room for up to 12 blocks of Kernel Y. The other four SMs, having one

block of Kernel X already resident, had only 1792 threads available and thus only had room for 11 blocks of Kernel Y.

The second block of Kernel Y was also scheduled to SM0, resulting in Y0 and Y1 executing on the same SM. With Y0 already executing, SM0 had 1888 threads available and could then fit only 11 blocks of Kernel Y. As all of the SMs could fit 11 blocks of Kernel Y, the first SM was chosen (SM0) per the tie-breaking ordering.

Finally, the third block of Kernel Y was scheduled to SM1. At the time of the decision, SM0 was executing two blocks of Kernel Y, and thus had 1728 threads available. As SM0 could fit only 10 blocks of Kernel Y, the scheduler chose the first SM out of the remaining four that could fit 11 blocks each (SM1).

This behavior, where the scheduler places the next block onto the SM which can host the largest number of blocks of the current kernel, is what we term the *most-room policy*. We discuss the finer details of this policy below.

4.2.2 SM Resource Limits

Determining which SM has the most room is dependent on a number of factors, and we find that it is specific to the moment in time when the block is being scheduled and is therefore re-evaluated for each block. As discussed previously, blocks require a number of computational resources which the SM provides, including shared memory, threads, and registers. If a block requires more of any one of these resources than an SM has available, it cannot be assigned to that SM. Thus, the first resource to run out when assigning blocks of that kernel to SMs becomes the limiting resource.

For the previous experiment, threads were the limiting factor, but we have also identified shared memory, the hardware limits on blocks per SM, and warps per SM as limiting factors. However, we cannot be certain that we have identified all

limiting factors given the black-box nature of the scheduler.

We ran a modified version of the experiment discussed above, where Kernel X consisted of five blocks with 1024 threads per block and Kernel Y consisted of three blocks with 32 threads per block. This meant that the limiting factor was the number of blocks allowed per SM, since 1024 free threads is enough room for up to 32 blocks of Kernel Y. On the Pascal GPU, this limit of blocks per SM was 32. In this experiment, the first two blocks of Kernel Y were assigned to SM0, and the last was assigned to SM1. This result is consistent with the most-room policy: at the time block Y0 was scheduled, SMs 1–4 had room for 31 blocks of Kernel Y (already having one block of Kernel X each), so SM0 was chosen because it was empty and had room for up to 32 blocks of Kernel Y. Then, as all the SMs were tied with space for 31 blocks, Y1 and Y2 were placed on the first two SMs respectively.

When we increased the number of threads per block in Kernel Y to 33, but left Kernel X the same, all three blocks of Kernel Y were placed on SM0. The limiting factor had become the number of warps per SM instead of the maximum number of blocks per SM. On the Pascal GPU, each SM can have up to 64 warps scheduled, and the blocks of Kernel Y at the size of 33 threads now required two warps instead of one. The SMs running one block of Kernel X already had 32 active warps, and could thus fit only 16 blocks of Kernel Y.

4.2.3 Tie-Breaking

The scheduler appears to use a per-device fixed ordering to break the ties between SMs, always picking the first SM that appears in that ordering. In our experiments with the Pascal GPU, for instance, we observed that when SM0 was empty, the scheduler always chose to place the next block on SM0, no matter which other SMs were also empty.

However, this tie-breaking ordering is not as simple as choosing the SM in ascending order of id number, as the previous observation might suggest. For instance, on the Pascal GPU, the ordering was a simple ascending order: 0, 1, 2, 3, 4. On the Turing GPU, however, the order can be best summarized as an evens-then-odds ordering: 0, 2, 4, 6, ..., 66, 1, 3, 5, 7, ..., 67. While the orderings were different among the different GPUs, none of the GPUs' thread block schedulers ever deviated from their respective orderings when breaking ties between blocks in our experiments.

We suspect the ordering depends in part on the grouping of SMs into Texture Processing Clusters (TPCs) and TPCs into Graphics Processing Clusters (GPCs). On the Turing GPU, for example, there was a total of six GPCs, with two SMs per TPC and 5-6 TPCs per GPC. We used the methodology of Pai [29] to determine which SMs belonged to which GPC, and found that the even-then-odds ordering caused blocks to be spread across GPCs and TPCs. This behavior may be intended to be a form of load balancing.

4.2.4 Further Details

The most-room policy is often indistinguishable from round-robin in the case of a single kernel. This is because blocks of the same kernel are of equal dimensions and typically there is little divergence in the executed instructions of the blocks, so the resources available on each SM will remain mostly identical when a single kernel is executing. We posit this as a reason for the frequent use of round-robin as a description of the scheduler's placement policy.

Additionally, we found no evidence that the shape of the block or grid influences the scheduler's decision. For example, a $32x2$ thread block was indistinguishable from a $64x1$ thread block from the perspective of the scheduler.

Finally, the most-room policy has performance implications for concurrent ker-

nels, including performance drops that are difficult to understand without knowledge of the scheduler's most-room policy. The results from concurrent kernel execution can seem counter-intuitive at first glance without knowledge of the most-room policy. We explore these issues in the next section.

4.3 Performance Implications of the Most-Room Policy

In this section, we highlight the impact of the scheduler and its most-room policy on concurrent kernels. We empirically show how minute variations in the structure of the kernel's blocks cause the scheduler to make different placement decisions that result in large variations of kernel performance. While the observations given below apply to all three GPUs used in this study, this section only presents the empirical results for the Turing GPU; the results for the Pascal and Volta GPUs can be found in Appendix A.2. See Appendix A.1 for the kernel implementation details.

4.3.1 A Demonstrative Experiment

Consider the basic experimental structure that we used for the Turing GPU. The two kernels A and B were each launched on two different CUDA streams. Kernel A had 67 blocks of 512 threads and it was launched first, guaranteeing that all 67 blocks would be scheduled to a separate SM (SMs 0-66), with SM67 left empty. Kernel B had two versions: one with 8 blocks of 32 threads, and one with 8 blocks of 33 threads. The version with 33 threads was the concurrent-isolated case. The limiting agent for Kernel B was the number of threads, so all of the 8 blocks of Kernel B were scheduled to the empty SM67. The version with 32 threads was the concurrent-colocated case; the limiting agent for Kernel B was the hardware limit

Table 4.1: Turing Execution Times. Kernel execution times on the Turing GPU, with their increase from the serial case noted in parentheses. Times were averaged over 30 runs; coefficient of variation was less than 3% for all cases.

	Serial (ms)			Concurrent-Isolated (ms)		Concurrent-Colocated (ms)	
	Kernel A	Kernel B	Total	Kernel A	Kernel B	Kernel A	Kernel B
L1 Cache-Dependent	85	79	164	85	79	105 (1.24X)	105 (1.33X)
Compute-Intensive	523	365	888	527	529 (1.45X)	530	676 (1.85X)
Memory-Intensive	949	10	959	951	224 (22.4X)	955	961 (96.1X)
Transfer-Bandwidth-Dep	369	130	499	385 (1.04X)	355 (2.73X)	388 (1.05X)	466 (3.58X)

on the number of blocks per SM, so the first of Kernel B's blocks was placed on the empty SM67. The rest were placed according to the tie-breaking ordering (see Section 4.2.3), with one per SM. This resulted in one block of Kernel A and one block of Kernel B on SMs 0, 2, 4, 6, 8, 10, and 12.

As described above, the only difference between the con-current-isolated and concurrent-colocated cases is that the Kernel B uses 33 threads per block in the concurrent-isolated case and 32 threads per block in the concurrent-colocated case. This minor difference in threads per block had a negligible impact on runtime (in the serial case), but triggered different scheduling decisions.

4.3.2 L1-Cache-Dependent Kernels

As all blocks on an SM share the same L1 cache, the performance of *L1-cache-dependent kernels* depends primarily on the amount of cache contention [37] (i.e., L1-cache-dependent kernels perform better when the cache is available for their exclusive use).

As summarized in Table 4.1, the execution time of both kernels in the concurrent-isolated case mirrored that of the baseline (i.e., the serial case). In other words, when the scheduler placed all of Kernel B's blocks on a separate SM from Kernel

A's blocks, both kernels executed with the same performance as if they each had a dedicated GPU. However, when blocks from both kernels were scheduled to the same SM (i.e., the concurrent-colocated case), there was a 1.24X increase in execution time for Kernel A and a 1.33X increase for B.

We attribute this loss of performance to increased cache contention caused by the scheduler's decision to co-locate blocks from Kernels A and B. In particular, the most-room policy does not account for interactions between separate kernels. In the concurrent-isolated case, there was no increase in execution time, as the kernels were executing on separate SMs and each SM had a separate L1 cache—there-fore there was no increase in cache contention.

Further, we observed performance degradation even when a single block of Kernel B was placed on an SM with Kernel A. In particular, we also ran the 33-thread version of Kernel B with 9 blocks instead of 8, which led to one block of Kernel B being assigned to an SM other than 67—one where Kernel A had a block running, thus causing interference. This experiment resulted in an execution time of 105ms for both kernels, the same as the concurrent-colocated case.

Finally, while concurrent kernel execution was faster than serial execution in both cases, the end-to-end execution time of the workload was also impacted by the scheduling decisions; for the Turing GPU we observed 164ms for the serial case, 85ms (0.52X) for the concurrent-isolated case, and 105ms (0.64X) for the concurrent-colocated case. Note that the total execution time of the concurrent cases is the max of the execution times of A and B.

4.3.3 Compute-Intensive Kernels

Compute-intensive kernels perform a high number of computational operations, and their performance is bounded by the number of these operations that can be per-

formed on an SM per unit of time. In the concurrent-isolated case, Kernel A saw no change in performance, while Kernel B experienced a 1.45X increase in execution time. Again, we attribute this decrease in performance to increased contention for resources (i.e., the functional units which perform these operations). However, in this case, the contention is between blocks of Kernel B only rather than contention between blocks of Kernels A and B. Recall that in the baseline serial execution, all of the blocks of Kernel B were scheduled to separate SMs, whereas in the concurrent-isolated case, all eight blocks of Kernel B were scheduled to the same SM. In other words, there was more contention for computational resources than when Kernel B was run by itself and each of the eight blocks were executing on a different SM.

In the concurrent-colocated scenario, Kernel A remained unaffected, but Kernel B experienced a 1.85X increase in execution time from the serial case. However, when we swapped the launch order of A and B, we observed only a small degradation for both kernels. Both concurrent cases exhibited only a slight improvement in total execution time compared to the serial case.

These results demonstrate that when two compute-inten-sive kernels are run concurrently, the scheduler's decisions can have a disproportionate impact on the performance of the kernels. This observation has important implications for kernel-level fairness, as the second kernel gets starved for resources in the concurrent-colocated scenario. Thus, even if Kernel B gets scheduled to an SM, the scheduler's implicit preference for Kernel A (due only to the fact that it was launched first) seemingly resulted in the majority of the functional units being assigned exclusively to Kernel A, preventing Kernel B from using the resources it needed to finish executing.

4.3.4 Memory-Intensive Kernels

Memory-intensive kernels are dependent on global memory throughput for their performance due to the high volume of global memory accesses that they incur. In the serial baseline, Kernels A and B exhibit very different execution times due to the difference in the number of threads per block. When run concurrently (i.e., isolated and colocated executions), the execution time of Kernel A was mostly unaffected; however, the execution time of Kernel B was impacted significantly. In the concurrent-isolated case, the execution time of Kernel B increased 22.4X, and in the concurrent-colocated case, the execution time increased by 96.1X. Both concurrent cases had total execution times comparable to the serial case (i.e., concurrency offered little improvement).

The increase in execution time for Kernel B during the concurrent-isolated case can be explained by the increase in contention for global memory throughput when all eight blocks reside on the same SM. The performance of Kernel B worsened drastically, whereas Kernel A saw almost no change in execution time. This is most likely due to the difference in size of the two kernels. Kernel A, having a much higher number of threads, made a much larger number of global memory accesses. Global memory is SRAM, physically present on the GPU and accessible by all SMs. Therefore, Kernel A used a larger portion of the global memory transfer bandwidth. Thus, when sharing the bandwidth with Kernel B, it was less affected by the contention.

4.3.5 Transfer-Bandwidth-Dependent Kernels

Transfer-bandwidth-dependent kernels depend on the speed at which page faults can be handled by the GPU. For a system with a discrete GPU, the PCIe link

connects the CPU and GPU. All input data and code must be transferred over this link. The classic model for handling this transfer is to send all of the input data over the link prior to the start of kernel execution. However, as the PCIe link becomes a performance bottleneck in this load-then-execute model, NVIDIA has been progressively adding features that allow for the overlap of data transfer and kernel execution. One such feature is Unified Virtual Memory (UVM).

For NVIDIA GPUs, UVM allows the programmer to treat memory as if it is shared between the CPU and GPU, even though in actuality, data must still be transferred between them over the PCIe link. With UVM, data can be transferred completely asynchronously as the kernels are being executed, with data being fetched on-demand as it is accessed by the kernels using paging. Thus, PCIe transfer bandwidth becomes another resource that is shared between concurrently executing kernels.

When run concurrently, we observed a minor performance degradation for Kernel A but a substantial degradation for Kernel B. In the concurrent-isolated case, the runtime for Kernel A increased by 1.04X and the runtime for Kernel B increased by 2.73X. In the concurrent-colocated case, Kernel A saw a 1.05X increase, while Kernel B experienced a larger 3.58X increase from the baseline. One possible explanation for the larger increase in the concurrent-colocated case is that there was more contention for transfer-related, SM-specific resources, like the translation lookaside buffer (TLB). Despite the increase in the individual execution times, the total execution time of the workload was slightly less than the serial case. Note that even in the serial case, the performance of PCIe transfer bandwidth-sensitive kernels depends on the rate at which paged memory transfers can be performed during kernel execution, which can depend on qualities such as TLB sensitivity and prefetching policies [39, 6, 11].

4.3.6 Performance Summary

The impact of the most-room policy on performance depends on the type of kernel being executed. For example, the scheduler's decisions disproportionately affect individual kernel performance for compute- and memory-intensive kernels, resulting in poor kernel-level fairness. Transfer-bandwidth-dependent and L1-cache-dependent kernels are impacted by contention for resources such as PCIe bandwidth, the TLB, and the L1 cache, which is worsened when the two concurrent kernels are colocated.

Finally, when the blocks of Kernels A and B were executed on separate SMs, concurrency offered an improvement in total execution time versus serial execution. However, when blocks from different kernels were placed on the same SM, that improvement lessened and, in some cases, dissipated entirely.

4.4 Summary

In summary, we have presented evidence that the thread block scheduler on NVIDIA devices uses a most-room policy to assign thread blocks to SMs, as opposed to the round-robin scheduling assumed by prior work. We have also demonstrated how scheduling decisions made under this policy can impact the performance of concurrent workloads.

Our results evince three factors that influence the performance of a kernel in a concurrent workload: *(i)* the scheduling policies of the thread block scheduler; *(ii)* the potential for resource contention across myriad hardware resources; and *(iii)* the impact of possibly unpredictable effects such as kernel launch timing. The implication is that predicting the performance of concurrent kernel execution is challenging because the kernel's performance depends on factors that are external to the kernel itself.

Chapter 5

Application Concurrency

While NVIDIA GPUs possess certain limitations that restrict the sharing of SMs between kernels, they do have the ability to run multiple indepedent applications simultaneously on a single device. However, the mechanisms which provide this functionality are limited by certain factors, including lack of preemption, insufficient prioritization, and being usable only with applications that share a CUDA context.

In this chapter, we analyze the performance of currently-existing mechanisms for concurrently executing multiple applications on a single NVIDIA GPU, in the context of a deep learning inference server: priority streams, time-slicing, and MPS. We explain the deep learning models that we use in addition to our experimental design to represent a deep learning inference server in Section 5.1. We then analyze the attributes of deep learning workloads that influence the effectiveness of priority streams, time-slicing, and MPS in Section 5.2. A significant portion of the runtime of deep learning training and inference tasks are small kernels with short runtimes that underutilize the GPU's resources. However, is also the case for some models that up to half of the runtime is spent on long-running or large kernels that occupy GPU resources for a significant amount of time. This poses a problem for achiev-

ing high utilization, as the resource demands of deep learning jobs are fluctuating. We additionally evaluate the performance of the three mechanisms in terms of the metrics relevant to a deep learning inference server, and find that all three mechanisms are lacking in features that they would need to possess to maintain both high turnaround time and high utilization consistently.

5.1 Measurement Methodology

We examine the performance of priority streams, time-slicing, and MPS for concurrently executing multiple applications on one GPU by analyzing deep learning training and inference tasks designed to resemble the scenario of an inference server responding to user requests and training models with spare resources. We utilized a set of five Pytorch deep learning models to measure the actual impact of the three scheduling mechanisms on performance when running inference and training concurrently. All tests were performed on the recently released NVIDIA Geforce RTX 3090 GPU of the Ampere microarchitecture.

We trained and tested these five models on a subset of images from the ImageNet database [14]. We ran the training task for each model for a length of five epochs, and the batch sizes we used were the maximum possible before encountering an out-of-memory error, detailed in Table 5.1. This was to ensure that the GPU's resources could reach as close to saturation as possible. We ran the inference task for each model as a series of 5000 consecutive image classifications at a batch size of one, to represent a series of incoming inference requests. The data reported in Table 5.1 were collected using NVIDIA's kernel profiling tool Nsight Systems [27].

We first ran both inference and training without any other concurrent tasks as a baseline for comparison. The only modification necessary was for the priority

streams method, which required some small changes to the models so that the training and inference tasks were launched from the same process on different streams. All of the kernels from the training and inference tasks were launched to separate streams, such that the inference task's kernels were always on a stream that was of a higher priority than that of the training task.

5.2 Characterizing Concurrency Mechanisms

In this section, we empirically examine and characterize the performance of priority streams, time-slicing, and MPS for running concurrent deep learning workloads on NVIDIA GPUs, presenting both their advantages and limitations. As we discuss below, we find that priority streams and MPS result in comparatively poor turnaround times and predictability for the inference task due to lack of fine-grained preemption, while time-slicing results in low utilization resulting from an inability to colocate kernels from different applications.

5.2.1 Performance Metrics

	Training Batch Size (imgs)	Long-Running Training Kernels (% of runtime)	Large Inference Kernels (% of kernels)
ResNet-50 [13]	128	56.63	15.85
ResNet-152 [13]	64	6.72	7.75
AlexNet [20]	256	3.28	2.28
VGG-19 [33]	64	41.60	42.19
DenseNet-201 [15]	64	6.76	34.39

Table 5.1: Deep Learning Workload Characteristics. The deep learning models analyzed, along with their relevant attributes to concurrent performance. Note that the long-running column shows the proportion of execution time spent on executing long-running kernels, while the large kernels columns show the proportion of large kernels to total kernels. Long-running inference kernels were omitted because they involved a negligible number of such kernels.

In evaluating the performance of a concurrency mechanism for the use case of a deep learning inference server, it is necessary to analyze both the degree to which users will be able to expect their requests are executed in a reasonable amount of time and how well-utilized the server's resources are. Therefore, the three main factors that we consider are as follows.

Turnaround time. The inference requests sent to the server represent latency-critical jobs that should be prioritized in order to complete them as fast as necessary to meet QoS requirements. Acceptable turnaround times depend on the application, but the best case turnaround time is when the inference request is executed alone, with no interference from other concurrent work being performed on the GPU. We define the inference turnaround time as the time it takes to service an inference request once received. We then use the inference time when running alone on the GPU as the baseline and compare that to the inference time achieved in the concurrent cases.

Utilization. Ideally, between the training and inference tasks, GPU resources would be as close to peak utilization at any given point in time as possible. With more requests to compute, this becomes easier to accomplish, but the balancing of those resources between tasks that need to make progress becomes more complex. For the purposes of our evaluation, we measure utilization as the end-to-end time of the training task. Since the training task is ideally computed with spare resources not needed for meeting inference requests' deadlines, any time that the training task is being executed on the GPU represents those idle resources being utilized. High utilization means that the training task finishes sooner while still maintaining low turnaround times for the inference requests.

Predictability. Maintaining a consistent response time is a key factor in providing a high quality of experience for the end-user. An ideal system would show little

variation in the time it takes to service inference requests. We therefore measure predictability as the variance in turnaround time.

5.2.2 Workload Characteristics

A deep learning model, whether performing training or inference, consists of a sequence of kernels that are launched onto the GPU serially to perform computations on subsets of the data. Two kernel properties play an important role in the performance of the examined techniques: length and size. A long-running kernel is one that takes longer than 1ms to run when executed on the GPU in isolation. A large kernel is one with a big enough grid size that its kernels cannot all fit onto the GPU to be executed simultaneously. This occurs when a kernel requires more of a particular resource than is available on the SMs. Once one resource on an SM is used up completely, no more blocks can be scheduled to that SM, even while other resources remain unused. The first resource to run out is known as the *limiting resource* for a kernel [12].

Both long-running and large kernels pose a problem for the task of efficiently servicing inference requests as they are submitted to the server. Long-running kernels occupy GPU resources for a significant amount of time, and so mechanisms that lack the ability to interrupt thread blocks mid-execution must instead wait for them to finish before reassigning those resources. Large kernels often inefficiently occupy GPU resources by preventing further thread blocks from being scheduled and making use of the non-limiting resources.

Overall, Table 5.1 shows that a significant portion of the training task execution time will be spent on executing large kernels from either the training or inference tasks, sometimes reaching up to half of the runtime for training. For some models, such as VGG-19 and ResNet-201, it is also the case that approximately half of

the training task's runtime will be comprised of executing long-running kernels, in addition. Therefore, we can say that for these deep learning workloads, there are potentially large gains to be achieved by using preemption-based scheduling techniques, as explored in Chapter 6. Additionally, preemption will be a key factor in reducing delay and ensuring predictability/avoiding QoS violations. However, because there is also a significant amount of kernels that are small and/or short-running, there is also a large potential gain in being able to spatially share the GPU during execution, as no task is occupying all of the GPU's resources constantly during their execution.

5.2.3 Priority Streams

For the priority streams case, both the inference and training workloads were launched from within the same process on separate CUDA streams, with the inference kernels being on higher-priority streams than those of the training tasks. Both workloads were comprised of a sequence of consecutive kernels that get launched serially to the GPU to be executed. When one of the high-priority inference kernels arrives, its blocks become the next in line to be scheduled, so it replaces the blocks of any executing lower-priority kernels as they finish.

Priority streams result in a low turnaround time in cases where the training task contains many short-running kernels. AlexNet demonstrates this in Figure 5.1a, where turnaround time was only 3.54ms, the shortest of the three, and almost 97% of the training execution time was spent on short-running kernels. When a training task kernel uses up enough of one resource (e.g., threads or shared memory) so that no more blocks can be scheduled, this disallows kernels on other streams from progressing effectively and prevents the other underutilized resources from being used. For AlexNet, the majority of the training task's large kernels were limited by

the hardware limit on the number of blocks per SM, meaning that no other blocks could be scheduled when these kernels were occupying the GPU. Even for the rest of the kernels, which may have left room for blocks of the higher-priority inference task to be colocated with the training task blocks, the SM's resources were not being utilized efficiently, as the inference task was only able to use resources leftover from the training task's grid.

Priority streams did, however, see consistent results in terms of utilization, as seen in Figure 5.1b, usually increasing the training execution time by 20-30 seconds. This tended to be on the lower side compared to the other two mechanisms. Priority streams can achieve good utilization primarily because when kernels are run on separate streams from within the same process, it is possible to *colocate* blocks from different kernels by scheduling them to the same SM and thus utilize more resources within SMs between them. A large reason for the consistency in utilization is the fact that almost all of the inference kernels for each of the five models were short-running. While the training kernels were interrupted by the higher-priority kernels of the inference task, they were able to return their thread blocks to the GPU quickly, as well.

Priority streams additionally perform better when potential contention due to colocation is low. Colocation of blocks from different applications allows for finer-grained resource assignment, but it also presents the problem of contention for resources when the blocks that are sharing an SM require conflicting amounts of the same resource. This, in turn, can lead to significant performance degradation [12, 37]. However, if this contention is low enough due to the blocks sharing an SM requiring complementary resources, priority streams will see increased utilization with a less significant degradation in turnaround times.

With high contention, priority streams' performance can become unpredictable.

In fact, unless it is clear what effects contention will have on the runtimes of the kernels, it is challenging to predict the performance of colocated kernels[37]. The increases in turnaround times compared to time-slicing observed in Figure 5.1a is partially explained by the fact that colocation is occurring in the priority streams case and not in the time-slicing case, introducing some contention.

The biggest reason that the priority streams approach performs the worst of the three in terms of turnaround time is due to a phenomenon we term *compounded delay*, which is an instance of the convoy effect [5]. The inference workload is comprised of a sequence of consecutive kernels that frequently depend on the output of the previous kernel, as is the training workload, and this fact is what causes the compounded delay. When a high priority kernel is finished executing, there is a window of time before the next kernel is launched and reaches the GPU. In this time, the lower-priority kernel resumes executing and fills the GPU with its thread blocks. As the GPU cannot preempt executing blocks, the next high priority kernel must wait for those blocks to finish executing. This adds the execution times of those blocks to the high priority kernel's runtime every time that one is submitted to the GPU, resulting in longer turnaround times.

We can see the effects of this delay in the results from the models besides AlexNet in Figure 5.1a, where the turnaround times are frequently over twice as long compared to the baseline. Models such as ResNet-50 and VGG-19 saw the worst turnaround time, and these models spent about half of their execution time on long-running kernels; this resulted in a larger slowdown incurred by compounded delay as the inference kernels waited longer behind the blocks of these longer-running kernels. In fact, we can see that despite being able to consider the inference task as higher-priority, the effects of compounded delay are enough for the priority streams turnaround times to be worse than that of MPS in almost all cases, despite MPS

having no notion of priorities.

Compounded delay is also the reason for the large degree of variance in turnaround time seen in Figure 5.2a. Spikes in delay were experienced during the time the training epochs were executing on the GPU, as those kernels, typically being longer-running, forced the inference tasks' kernels to wait for longer. In contrast, the turnaround times were lower during validiation, when the training task was launching shorter-running kernels to the GPU.

A final limitation to the priority streams approach is that it requires that all tasks are launched from within the same process. Given the difficultly in designing a system where both the training framework and the inference applications are a part of the same process, this is a considerable disadvantage.

5.2.4 Time-Slicing

The training and inference tasks in this case were launched as separate processes to the same GPU. As demonstrated in Figure 5.2b, time-slicing offers the most predictable performance of the three because blocks from the training kernels and blocks from the inference kernels never execute at the same time. This eliminates contention for SM resources during block execution, and the inference kernel does not need to wait for any blocks of the training task to finish executing before being scheduled to the GPU. Thus the primary factor that influences turnaround time is the number of other jobs that are being executed concurrently, as this changes the amount of time that any one job must wait for access to the GPU's resources. The reason for this is that time slots are a fixed size and are assigned round-robin to each process. As far as we could determine, the time slot size and interleaving level assigned to each process cannot be configured.[1]

[1]The Jetson devices do allow such configuration [9].

The major trade-off inherent in using time-slicing is predictability at the cost of utilization. The inference task effectively has the entire GPU during its timeslice; while great for predictability, this also means that extra GPU resources sit idle. None of the models' inference tasks spend a majority of the time on large kernels, and some, such as AlexNet and ResNet-152, spend virtually none (2-7% of the total execution time). This means that when using time-slicing to execute inference requests, a large portion of time is spent with vastly underutilized GPU resources. Further, time-slicing offers limited ability to configure based on application needs. As the time-slice length is fixed, the inference requests can never be completed any faster than the current schedule for the set of concurrently executing tasks will allow. Average turnaround time increased the most for the VGG-19 and DenseNet-201 models at around 10ms, but because the inference requests are always run in isolation when using time-slicing, the increase in turnaround time tends to be consistently small.

Low turnaround time came at the expense of utilization, which was frequently the worst of the three surveyed mechanisms. This is the major detriment of time-slicing; due to the lack of spatial sharing capabilities, it does not truly solve the GPU resource utilization problem being addressed by concurrently executing training and inference. For the ResNet models and particularly for DenseNet-201, the lack of ability to colocate tasks makes utilization suffer dramatically, increasing the training time to over 100 seconds more in Figure 5.1b. The reason VGG-19 and AlexNet don't see such an increase is due to the shorter lengths of their inference tasks; they completed earlier and ceased interfering with the training task partway through, allowing it to then utilize the GPU resources in isolation. Turnaround time stayed consistently low using time-slicing, and this is because the tasks launched by the inference workload were guaranteed consistent access to the SMs whenever their

time-slice window would appear.

Another limitation of time-slicing is that the resource requirements of any tasks being run simultaneously as separate processes cannot together exceed the resource limitations of the GPU, or an error will be thrown. For instance, if two applications are launched that use an amount of shared memory within the GPU's limitations, but together utilize more shared memory than the upper bound, this will cause the second process to reach the GPU for scheduling to crash with an out-of-memory error. Thus, the inference task is not actually getting full access to the GPU's resources; it can use the warp scheduler and compute units without interference, but still has to share the global memory, shared memory, and registers. The reason these resources are shared is, presumably, because swapping them out would introduce prohibitively high context switching overheads.

This also places a further limit on the training task, which had to be scaled down from its maximum batch size in order to allow space for the inference task without running into this error. Thus, despite the fact that the two tasks never occupy the GPU at the same time, they are drastically underutilizing the resources even when only blocks from their kernels are on the GPU. Given that the point of executing the training task alongside the inference requests was to improve utilization, this is a rather debilitating limitation.

This problem is compounded by the fact that it cannot be known ahead of time precisely how many resources the inference task, in particular, will require. Given that we don't know the rate at which inference requests will be received by the server, we can either perform inference for a single image at a time, choose a fixed batch size to use for performing inference, or perform inference using variable-sized batches. Single image inference has predictable resource usage, so an out-of-memory error can be avoided; however, this will add queueing delay (i.e., one image now has to wait

for the previous request to be serviced first). Fixed batch sizes also have predictable resource usage as they are just the generalized case of single-image inference, so we can tune the training task to accommodate that while minimizing queueing delay. However, if we don't fill up the batch for a particular run, then we will have even lower utilization. Dynamic batch sizing is probably impossible given the possibility of incurring an out-of-memory error.

5.2.5 Multi-Process Service

The main strength of MPS is that it guarantees that a set of tasks with small enough kernels can all have some GPU resources simultaneously, similar to the priority streams approach. However, MPS is not able to prioritize the execution of one task over another; instead, it uses load-balancing to provide more equitable progress whenever there are idle resources [26]. Thus, in those cases, both the training and inference tasks are guaranteed to make progress that is more balanced between the two applications. This is the main reason that the MPS turnaround times and utilization in Figure 5.1 are typically better than priority streams; given that Table 5.1 shows that at least half of all of the models' inference and training kernels are small, MPS can employ this load-balancing during a significant portion of the tasks' execution.

One important limitation of MPS is that the resource limit is the same for all processes, so it is not possible to prioritize one application over another. Further, resources are assigned on essentially a first-come, first-served basis (up to the per-process limit). More specifically, the GPU schedules kernel blocks from separate processes as if they originated from different streams within the same CUDA context. This means that the blocks are scheduled according to the *leftover policy*, which dictates that all of the blocks from the most recently-arrived kernel must first

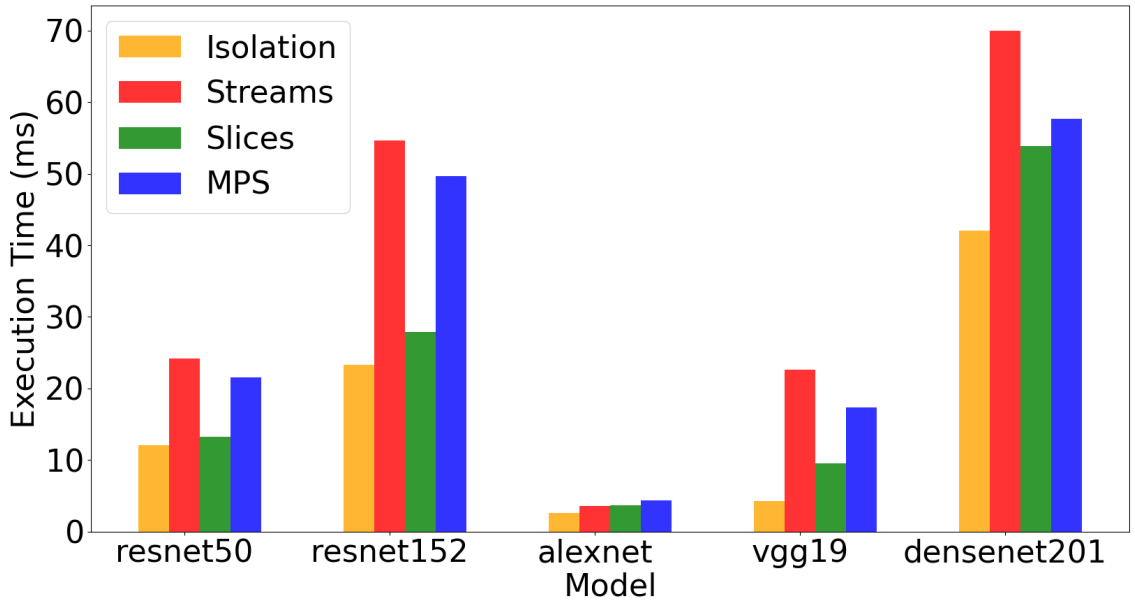
be dispatched and executed on the GPU before any other kernels' blocks can be scheduled [4]. This presents a problem for the task that arrives at the GPU later, especially if that task is of a higher priority than the currently-executing one, as the running time of the second task is needlessly throttled. A more proportionate division of resources would result in a more acceptable amount of degradation to the later-arriving and higher-priority task's turnaround time.

Thus, MPS causes a much greater degree of degradation for the inference tasks than the training tasks in Figure 5.1 due to its first-come, first-served scheduling policies. For instance, ResNet-152 saw the turnaround time double, but the training task execution time only increased by a few seconds, which was the best performance of the three mechanisms. The training task has, on average, longer-running kernels with larger grids for all models except DenseNet-201. This was the model where MPS performed the best in terms of both turnaround time, at an increase of 15.7ms, and utilization, increased by only 11 seconds. For the other four models that averaged longer-running training kernels, the inference task was more often starved for resources, forced to make progress with what was leftover.

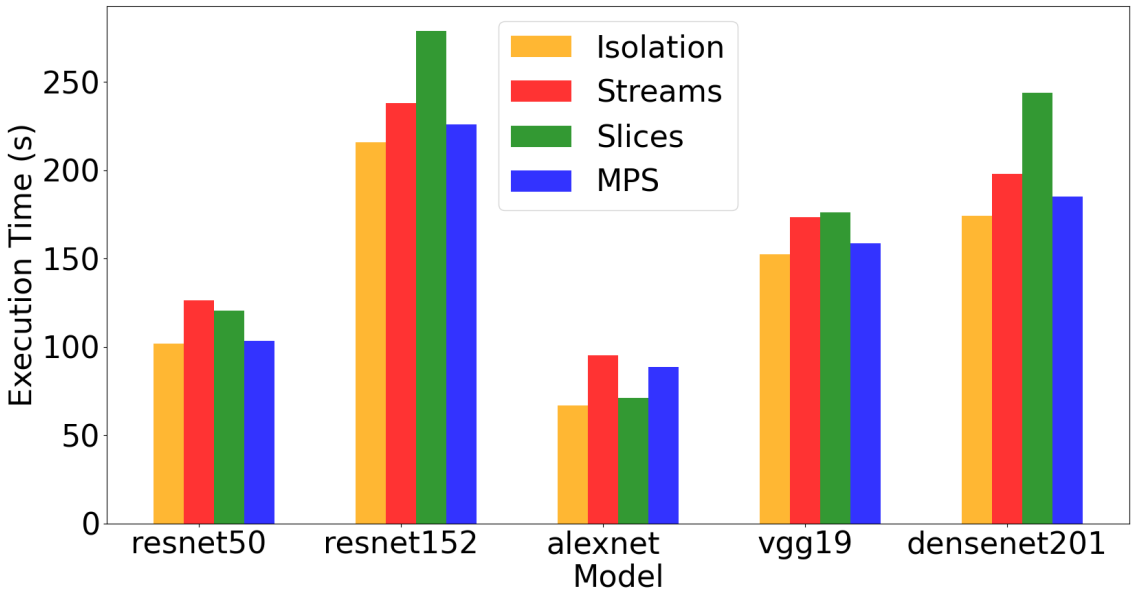
This also caused some degree of variance in turnaround time, as seen in Figure 5.2c. This variance was not as large as that observed in the priority streams case, as inference request satisfaction is partially dependent on the degree to which the training task is utilizing the GPU's resources. The advantage of MPS over time-slicing, however, is that it does improve utilization by being able to host both tasks on the GPU simultaneously. Thus, resources are far less underutilized for almost all of the models analyzed, despite most of the degradation in execution time being for the latency-sensitive inference tasks.

5.3 Summary

In summary, we have shown the three mechanisms for executing concurrent workloads currently available on NVIDIA GPUs—priority streams, time-slicing, and MPS—have limitations that reduce their ability to handle concurrent deep learning workloads efficiently. In particular, the features of these deep learning workloads, such as kernel size, length, and their nature as a sequence of frequntly-launched kernels introduce severe inefficiencies when executed concurrently by these mechanisms. Their inflexible preemption policies and limited notions of task prioritization are poorly-suited to execute this type of mixed, latency-sensitive workload while also maintaining high utilization.

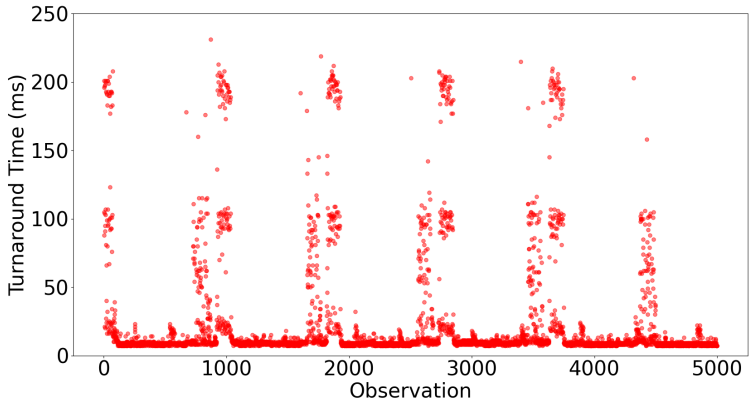


(a) Average Turnaround Times

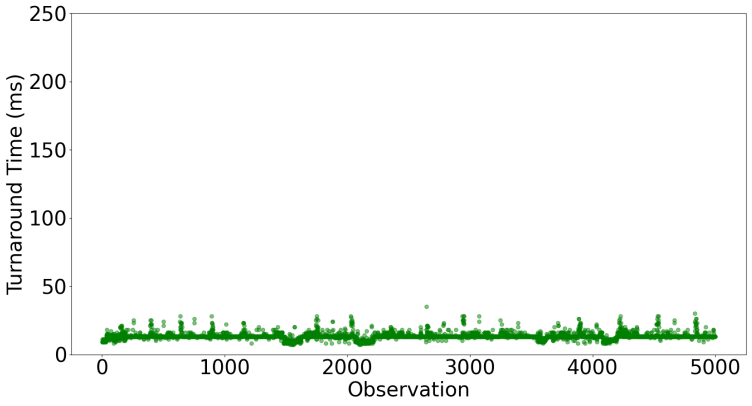


(b) Average Utilization

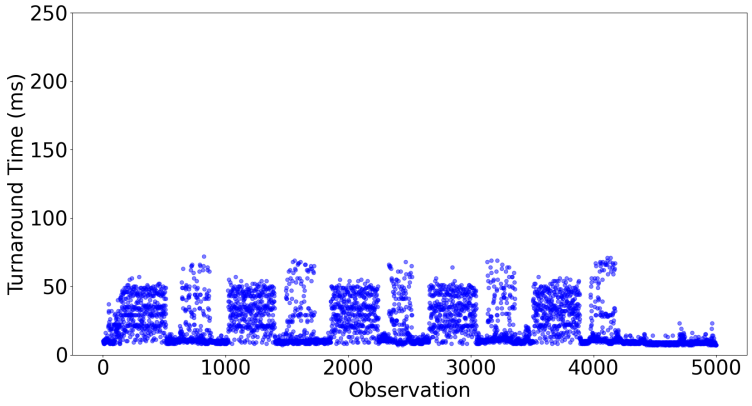
Figure 5.1: Ampere Turnaround Time and Utilization. The average turnaround times and utilization for each of the three mechanisms on five different models. Note that the turnaround times are the averages of 5000 inference requests, and the measurement of training execution time is the average of 10 runs.



(a) Priority Streams



(b) Time-Slicing



(c) MPS

Figure 5.2: Ampere Variance. The variance of the turnaround times for the ResNet-50 model. Other models' variance results were omitted for space, but resemble these.

Chapter 6

Proposed Scheduling Policies

While using priority streams does allow for the preemption of a kernel being executed on the GPU if the arriving kernel is of a higher priority, it does not actually interrupt any of the blocks currently on the GPU, instead waiting for them to finish execution before scheduling any blocks of the new kernel. MPS similarly has no mechanism for interrupting the execution of a block; this lack of block-level preemption is what causes the performance degradation seen in Section 5.2. The compounded delay incurred as a result causes the priority stream turnaround times to be comparable to that of MPS, which has no notion of priorities at all. Time-slicing, while able to preempt blocks in the middle of their execution, is only able to clear the entire GPU of all currently-executing thread blocks, with no ability to partially preempt the GPU. Additionally, the preempted task is still occupying resources such as shared memory and registers, limiting the amount of computation that can be done by the preempting task.

To address these issues, we consider the need for *fine-grained thread block preemption*, which allows for any arbitrary set of thread blocks to be removed from the GPU during execution and relaunched at a later time. This will allow tasks to

spatially and temporally share the GPU's resources for improved utilization without coming at the expense of turnaround time. This will vastly improve predictability over the priority streams and MPS approaches, as the effects of compounded delay and the leftover policy are eliminated. The preempted kernel will take longer, but the overhead of preemption will be offset by more efficient resource utilization and the improvements to predictability and turnaround time for the inference task.

We frame the problem of the concurrent execution of inference and training tasks as follows. The training task will be executing on the GPU when the inference request arrives; the GPU must preempt precisely as many thread blocks of the training task as necessary to meet the desired turnaround time for the inference request so that utilization can be maximized and the overhead for preempting the blocks of the training task is limited. Additionally, the thread block scheduler will have to know to replace completed inference blocks with waiting inference blocks and completed training blocks with the queued training blocks.

The question that is posed by such a scenario is which blocks of the training task should be preempted to make space for the blocks of the inference task. The choice of blocks must be proportional, meaning that enough of the resources end up assigned to the inference task that it can make acceptable progress and meet its deadline; and it must be contention-aware, so that progress on the inference task is not impeded unnecessarily by competition for hardware resources.

Kernels require different sets of resources, and can be classified as such into categories like memory-intensive, compute-intensive, or L1-cache-dependent; if two blocks whose performance depends on the same resource are colocated on an SM together, competition for those resources can cause significant performance degradation and drastically increase the kernels' runtimes [12]. Block placement should therefore seek to minimize the amount of contention that will be incurred when

colocating blocks on a given SM, so as not to impede the progress of the latency-sensitive inference task. In particular, colocating blocks from different kernels can result in performance gains for both kernels if they fall into different categories. If the kernels fall into the same category, we want to avoid colocating their blocks; we would instead want to put each onto their own SMs. There will still be contention, but it will be more predictable.

It is known which resources are required by different layers in convolutional neural networks, so it is reasonable to assume we would know this information about the kernels of a given deep learning workload. For example, fully-connected layers of deep learning models are memory-intensive, while convolutional layers are compute-intensive [23]. Further kernel profiling can also be conducted for information specific to the models being hosted by the server before beginning training.

Below, we consider three potential block preemption strategies that may be employed to reduce contention: *round-robin*, *SM-filling*, and *SM-division*. The first two policies are useful when preempting large training kernels, and the last is for sets of small kernels. We can consider kernels to belong to different categories based on their reliance upon certain resources for performance, as described above, and the implementation of the chosen preemption strategy can take that into account when choosing which blocks to preempt. We estimate the performance gains that are possible to achieve through the use of these strategies on a set of ablative workloads, described below, which emulate the relevant properties of deep learning workloads but allow for fine-grained control over their attributes and behaviors.

6.1 Methodology

Given that it is not possible to modify proprietary NVIDIA hardware to directly test other preemption strategies, we designed these ablative kernels to be representative of actual training and inference workloads based on our characterization of them in Section 5.2. We analyze the three proposed strategies regarding three different categories of ablative workloads on a Geforce RTX 3090 GPU of the new Ampere microarchitecture. The following analysis could be applied to the previous Turing microarchitecture, as well, because we achieved similar results on the previous Turing microarchitecture; those results are elided for space.

Each workload was comprised of a sequence of ten kernels, and their runtimes in isolation are shown in the first column of Table 6.1. The total execution time is the time it takes for all ten kernels to finish executing serially, mimicking the training and inference structure of a sequence of kernels. We examined workloads comprised of both small and large kernels, as both types make up a significant portion of actual deep learning workloads. For the small kernels, the 82 training kernel blocks fit one per SM and left available 1024 threads of the total 1536 threads per SM. The inference kernel blocks were small enough (128 threads) to fit alongside the blocks of the training kernel. The large kernels had thread blocks of the same size as the small workload for both training and inference, but the grid size was 5248, which was large enough that only a portion of the blocks could fit on the GPU at once.

Contentious kernels, which include the small kernel workload, performed a series of computational instructions repeatedly, to spin for the desired runtime. These kernels were therefore compute-intensive, as they depended on access to computational units for their performance. The low-contention kernels performed a series of local memory operations to reduce the impact of resource contention between the

blocks of the kernels. Thus, both types of kernels were able to spin for enough time to mimic the inference execution time and individual kernel runtimes of real deep learning workloads.

6.1.1 Round-Robin Policy

	Baseline Priority Streams (ms)	Time-Slicing (ms)	MPS 100% (ms)	Proposed Estimate (ms)
Small Kernels				
Low Priority	12.6	49.2	30.9	31.8	23.8
High Priority	17.2	39.1	29.5	21.3	19.9
Low-Contention Kernels				
Low Priority	19.0	41.5	29.5	27.6	34
High Priority	13.1	21.8	26.6	27.9	19
Contentious Kernels				
Low Priority	14.2	43.5	29.7	26.3	29
High Priority	14.7	28.4	29.8	25.8	20

Table 6.1: The turnaround times and utilization of the ablative workloads. *Note that these are the averages of ten runs, where the kernels were ensured to have been launched to the GPU within 1-2ms of each other.*

The round-robin policy preempts one block of the training task per SM in round-robin order, replacing it with as many blocks as possible from the inference task until the desired number of blocks are scheduled. To account for any slowdown due to contention that may arise from colocating blocks from two different kernels, the round-robin policy will slightly overestimate the number of high-priority blocks needed to be scheduled, by one block per SM. This policy has the advantage of spreading out the blocks of the preempting task between SMs, which as described above will reduce contention between blocks of the inference kernel. If the preempting and preempted kernels are of different types, the round-robin policy also improves effective utilization by placing more low-contention blocks together on the same SMs.

To analyze the effectiveness of the round-robin policy on low-contention, large kernel tasks, we look at the ablative workload termed low-contention in Table 6.1. We set the target turnaround time to 19.5ms (approximately 1.5X the 13ms time in isolation), and so we would need to place 75% of the maximum number of inference task blocks on the GPU to make the required amount of progress. This would provide a 9ms improvement over the turnaround time observed using MPS, or a roughly 1.45X speedup.

The original low-contention kernels, shown in row two of Table 6.1, could fit 984 thread blocks on the GPU at once in isolation, so the minimum number required to meet the deadline would 738 blocks, or nine per SM. The round-robin policy will schedule the equivalent of ten blocks per SM to offset any degradation due to contention; this would mean 820 total blocks. The round-robin policy would result in a block placement where for half of the total SMs, all blocks of the training task would be preempted and replaced with inference task blocks, while for the other half of the SMs, all but one block of the training task would be preempted and replaced with blocks of the inference task. Thus, half of the SMs will have 12 blocks of the inference task, and the other will have eight blocks of the inference task and one block of the training task. There will be 820 blocks of the inference kernel on the GPU, along with 41 blocks of the training kernel.

The training task makes approximately 1/6th of the progress it would in isolation for the 19ms the inference task is still executing, and then full progress once the inference task completes. Thus, the training task will get through about 1/6th of its work during the inference task execution, and then the remaining 5/6ths will be completed in isolation. This will result in a turnaround time of about 19ms, and a utilization level of approximately 34ms.

This turnaround time is an improvement of 2-9ms compared to all three concur-

rency mechanisms; the utilization is a 7ms reduction in the priority streams case, as it guarantees some progress by the training task. Thus, as long as the overhead of the preemption remains at or around 1-2ms, round-robin preemption will significantly reduce turnaround time while achieving comparable utilization levels to the MPS and time-slicing cases.

6.1.2 SM-Filling Policy

The SM-filling strategy isolates the blocks of the two tasks as much as possible, for when their expected contention is high. To do so, SM-filling places the blocks of the preempting inference kernel by filling up entire SMs, only evicting blocks on a different SM when the current SM has no more room. If both the training and inference tasks are bound by the same resource requirements, using this technique limits their interference with each other by keeping them as separate as possible. Thus, this policy is most useful in the case where the kernels come from the same contention category. It is also good for predictability because we can profile performance a priori. Note that SM-filling preempts an entire SM before scheduling any blocks; in the case where some blocks of the inference task could have fit before preemption, the blocks currently on the GPU are preempted first and relaunched to a different SM to minimize contention.

We next look at the performance of the contentious, large-kernel workload, labeled as contentious in Table 6.1. We set the target turnaround time to be approximately 1.5X the baseline inference time, which would be about 20ms. This means that again 3/4ths of the inference kernels' blocks should be scheduled to the GPU at any point in time, and using the SM-filling policy, this would mean that 61 SMs will have 12 blocks of the inference task, and one will have six leftover blocks, leaving space for one block of the training task on that final SM. The remaining 20 SMs will

each have 3 blocks of the training task each, for a total of 61 training task blocks on the GPU.

To estimate the training task time, we can use a similar method to the analysis performed on the round-robin policy. The training task makes roughly 1/4th of the progress it would in isolation for the 20ms that the inference task is still executing, and then full progress once the inference task completes. Thus, the training task will get through about 1/3rd of its work during the inference task execution, and then the remaining 2/3rds will be completed in isolation. This will result in a turnaround time of about 20ms, and a utilization level of about 29ms. The SM-filling preemption policy can thus achieve considerable improvements in turnaround time and utilization, as long as the preemption overhead remains less than 1-2ms.

It is important to note that the preemption overhead for both the SM-filling and the round-robin techniques will only occur once, when the inference task first arrives to be scheduled. This is in contrast to time-slicing, which incurs an overhead every timeslice. While SM-filling will have a more expensive overhead, as the execution contexts of the interrupted blocks will have to be flushed from the GPU to allow the inference task to utilize those resources, the fact that it does not occur repeatedly will limit its impact.

It is also possible to consider a slight modification of SM-filling, where instead of flushing everything from the SM, we take a similar approach to the time-slicing strategy and only flush the scheduling information for a subset of the training blocks. This will still have the problem of needing to tune the resource utilization to be lower, but the context-switching overhead will be much smaller. This concept could be applied to reduce the overhead of the other preemption strategies discussed, as well.

6.1.3 SM-Division Policy

The final policy we propose is SM-division, which functions similarly to MPS, but allows for dynamic resource scaling. For small kernels, the round-robin policy suffices when contention is low. However, using the round-robin policy on contentious kernels could result in severe performance degradation. Additionally, using the SM-filling policy will result in most of the inference blocks placed on a small subset of SMs while the others are left empty, causing unnecessary contention.

Thus, SM-division instead divides the SMs between the tasks proportionally based on their priority. This is in contrast to the SM-filling policy, which simply preempts entire SMs and fills them with kernels of the higher-priority task. The SM-division policy instead sets aside a portion of the SMs for the higher-priority task that minimizes the number of blocks per SM as much as possible to reduce contention, while also allowing the training task to fit on the remaining SMs.

For small kernels, the runtime is essentially as long as the longest-running block, so contention between any two blocks increases the runtime. Thus, we divide SMs between the tasks by first giving the inference task as many SMs as it needs to place the minimum number of blocks m per SM possible (in other words, the baseline). Then, the remaining SMs go to the training task. If there are not enough remaining SMs to fit all of the blocks of the training task, the division is shifted to give the inference task enough SMs to fit $m + 1$ blocks per SM, and the remaining SMs are given to the training kernel. This continues until either a division is found where all blocks can fit on the GPU simultaneously without sharing SMs, or the inference kernel blocks cannot take up any fewer SMs.

Using SM-division, the inference task would get 42 SMs with two blocks per SM, leaving the training task the other half of the SMs, also with two blocks per SM. To estimate these runtimes, we ran actual workloads that would amount to these

placements (two blocks of each per SM) and report their average runtimes.

Thus, SM-division can achieve comparable results to MPS and improve turnaround time compared to time-slicing, as long as preemption overhead is less than 5-6ms. This technique has the advantage of being able to assign more resources to higher-priority and latency-sensitive tasks. Like MPS, both tasks can make progress, but with the SM-division policy, the higher-priority task is as uninhibited as possible.

6.2 Summary

We have argued the need for fine-grained block preemption and a more robust notion of task prioritization to more efficiently execute concurrent deep learning workloads. We then proposed a set of three policies which use those: mechanisms: round-robin, SM-filling, and SM-division, which allow for kernels to share SMs but take into account the potential performance degradation investigated in Chapter 5 due to contention when similar kernels have to share resources. We achieved estimated potential performance improvements of up to 1.5X turnaround times when using these policies which utilize fine-grained preemption, demonstrating the need for such a mechanism to exist on NVIDIA GPUs.

Chapter 7

Conclusions

Current NVIDIA GPUs are not capable of efficiently executing the kinds of concurrent deep learning workloads which would be useful to an inference server provider. Such providers would be concerned with maintaining high utilization while also continuously meeting QoS requirements for servicing latency-sensitive requests, and NVIDIA devices are currently unable to meet that demand. We have enumerated the *most-room policy* as the block placement policy for concurrent kernels, which leads to counter-intuitive scheduling outcomes and often significant performance degradation as it does not take into account contention for per-SM resources. Additionally, NVIDIA's current mechanisms for application-level concurrency—priority streams, time-slicing, and MPS—each have their own set of limitations, and each lack a fine-grained preemption policy which we have shown could drastically improve the turnaround times of concurrent applications.

While the proposed preemption strategies show promise, testing fine-grained preemption on actual hardware will require modification to proprietary NVIDIA components and, as such, cooperation from the NVIDIA corporation. In future work, we intend to build on these findings through analysis of a wider set of deep

learning models, such as recurrent neural networks. We will also use such results to suggest further modifications to the NVIDIA scheduling hierarchy at both the kernel and application level which are capable of addressing the effects of other myriad factors not considered in this work, such as memory transfer bandwidth contention.

It is important to reiterate that we are limited to empirical observations of the scheduler, and thus, the policies we describe are not guaranteed to be precisely what the hardware implements—though, the most-room policy description is consistent with all of our empirical observations. Thus, we hope that our work will be useful in improving the accuracy of existing GPU simulators and, consequently, assist in the development of concurrency-aware scheduling policies. We intend to use our understanding of NVIDIA's concurrency mechanisms and thread block scheduler to implement more accurate simulators and present the theoretical advantages of fine-grained preemption and contention-aware spatial multiplexing policies through the use of such simulators as future work.

We intend for this work to catalyze the creation of more robust and efficient techniques for concurrent deep learning workloads in the future. We emphasize that such mechanisms should involve both efficient preemption mechanisms and contention-aware block placement policies to achieve greater concurrent workload performance. We also expect this work to serve as a baseline for comparison for work on concurrency mechanisms on NVIDIA devices. Additionally, we hope to see the proposed fine-grained preemption mechanisms implemented in future NVIDIA devices, as we have demonstrated their utility for real GPU workloads.

Appendix A

Appendix

Table A.1: Architectural details of the GPUs used in our experiments.

	Arch.	Compute Capability	SMs	Threads per SM	Threads per Block	Blocks per SM	Warps per SM
GeForce GTX 1080	Pascal	6.0	5	2048	1024	32	64
Tesla V100	Volta	7.0	80	2048	1024	32	64
GeForce RTX 2080 Ti	Turing	7.5	68	1024	1024	16	32
GeForce RTX 3090	Ampere	8.0	82	1536	1024	32	64

A.1 Kernel Implementations

To emulate an L1-cache-dependent kernel, we used an approach based on the one taken by Naghibijouybari et al [24]. We implemented a kernel which uses each thread in a block to repeatedly access texture memory. We used knowledge of the specific structure of the L1 caches on each GPU, such as their size and set-associativity [18, 17], to make these accesses highly cacheable but vulnerable to replacement by repeatedly accessing data from different sets of the cache. To confirm these kernels' dependence on the cache, we measured the L1 cache hit rate in the serial case, and found that they experience a 90% hit rate on average, ranging from 75%-95%.

In order to emulate a compute-intensive kernel, we designed the kernels to occupy the functional units by performing repeated floating point operations—these *functional units* are the hardware components that the SMs use to perform computations. Further, we avoided memory accesses in our kernel design to prevent global memory access contention from impacting performance.

To emulate a memory-intensive kernel, we designed a kernel which repeatedly accesses indices of a large array stored in global memory. When threads write to memory addresses nearby each other in global memory, their individual accesses can be coalesced to save memory transfer bandwidth. Using writes as opposed to reads prevents the data from being cached in the L1/texture cache. To avoid memory coalescing impacting these results, the threads' memory accesses were spaced apart. As these addresses were far away from each other for all threads within a warp, they cannot be coalesced efficiently, causing more data to be transferred per global memory access.

To emulate dependence on PCIe transfer bandwidth, we designed a kernel which triggers a large number of *page far-faults*, meaning that the page is not in the GPU's global memory but instead must be transferred over the PCIe link. To do this, the kernel accesses memory such that the threads within a block target addresses nearby each other, but threads from different blocks target addresses that are distant. Further, accesses can be coalesced within blocks, which limits the effect of global memory transfer bandwidth contention on the performance of these kernels.

A.2 Other GPU Kernel Concurrency Results

The relevant architectural details for each of the three GPUs looked at in this work can be found in Table A.1. The only differences in the kernels run on the Pascal and

Table A.2: Pascal Execution Times. Average execution times for kernels in differing scenarios on the Pascal GPU with 5 SMs.

	Serial (ms)			Concurrent-Isolated (ms)		Concurrent-Colocated (ms)	
	Kernel A	Kernel B	Total	Kernel A	Kernel B	Kernel A	Kernel B
L1 Cache-Dependent	63	45	108	63	45	94 (1.49X)	94 (2.09X)
Compute-Intensive	780	415	1195	780	415	895 (1.15X)	915 (2.20X)
Memory-Intensive	1233	49	1282	1233	274 (5.59X)	1270	1270 (25.9X)
Transfer-Bandwidth-Dep	1588	239	1827	1680 (1.06X)	523 (2.19X)	1689 (1.06X)	1686 (7.05X)

Table A.3: Volta Execution Times. Average execution times for kernels in differing scenarios on the Volta GPU with 80 SMs.

	Serial (ms)			Concurrent-Isolated (ms)		Concurrent-Colocated (ms)	
	Kernel A	Kernel B	Total	Kernel A	Kernel B	Kernel A	Kernel B
L1 Cache-Dependent	85	51	136	84	55	104 (1.22X)	104 (2.04X)
Compute-Intensive	869	333	1202	870	480 (1.44X)	871	986 (2.96X)
Memory-Intensive	2458	34	2492	2459	622 (18.29X)	2492	710 (20.88X)
Transfer-Bandwidth-Dep	3156	121	3277	3194 (1.01X)	1113 (9.2X)	3295 (1.04X)	1325 (10.95X)

Volta GPUs were related to the hardware differences between them and the Turing GPU, such as the number of threads per block and the total number of blocks. The only other change was that for the L1-cache-dependent kernel, adjustments were made for the differences in the size and architecture of the caches.

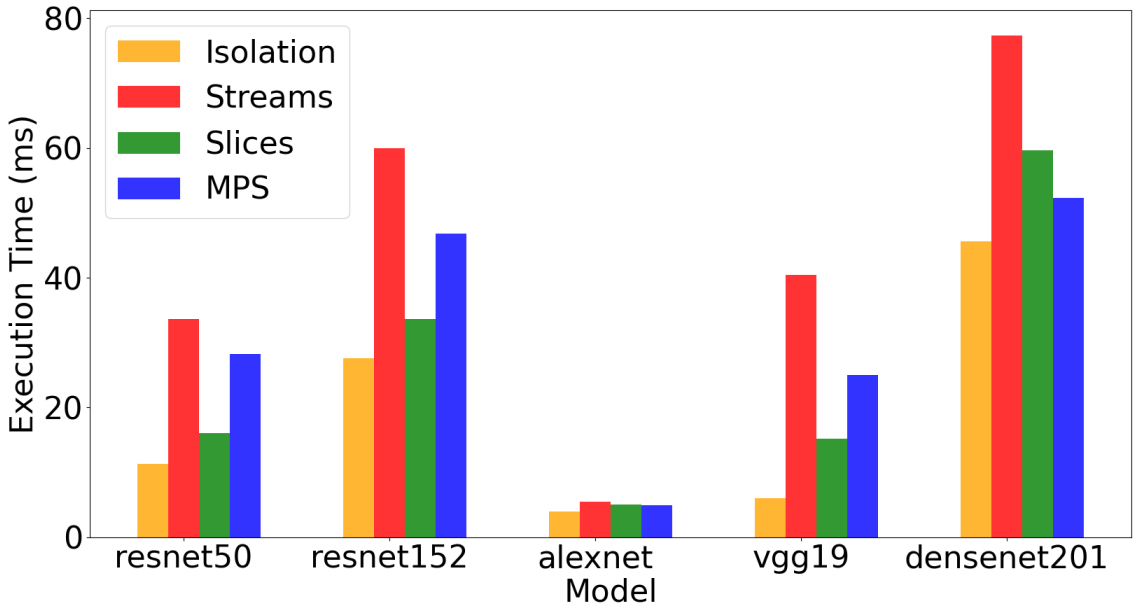
The results for the Pascal GPU can be seen in Table A.2, and those for the Volta GPU can be found in Table A.3. The only major difference in these results from the Turing GPU is that for the memory-intensive kernel on the Pascal GPU, Kernel B did not see any performance degradation in the concurrent-isolated case. This is because of the fact that with only four blocks, there was no contention for computational resources when they were all scheduled to the same SM; the Turing GPU kernel had eight blocks, and scheduling all eight to one GPU was enough to cause some contention. However, we stress that the ultimate meaning behind these results remains the same; the same impacts on execution time were found in the Pascal and Volta GPU results as in the Turing results, indicating the same behavior

from the scheduler during the execution of these concurrent workloads.

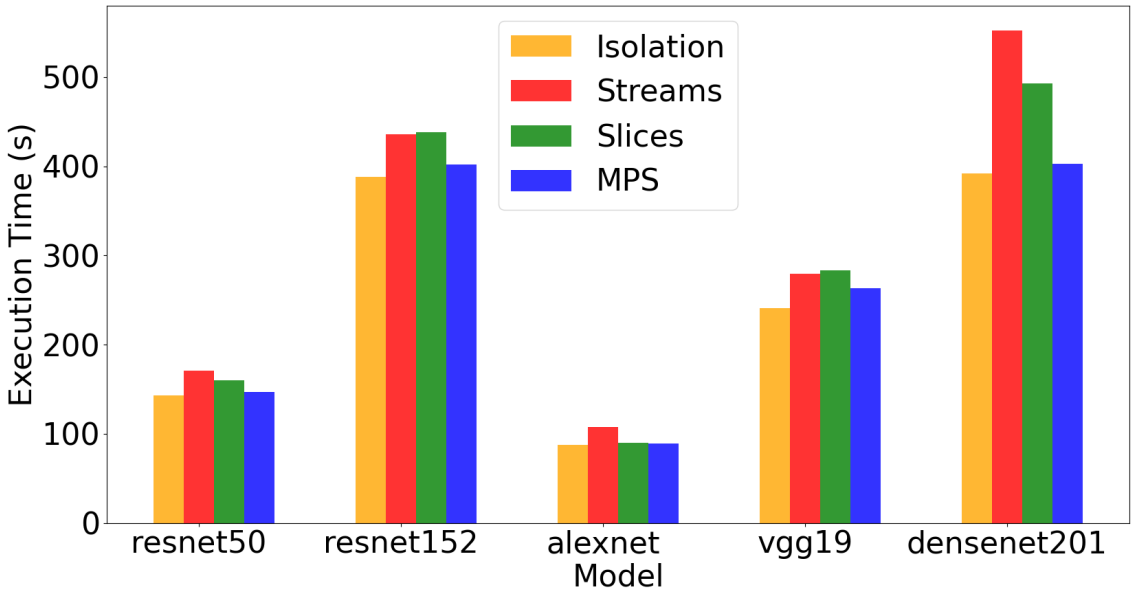
A.3 Turing GPU Deep Learning Workload Results

Figures A.1 and A.2 show the results for the same set of five deep learning models examined in Section 5.2 run on an NVIDIA 2080 RTX GPU of the Turing microarchitecture, the predecessor of Ampere. Note that batch sizes for the models had to be reduced in most cases from the Ampere cases, reported in Table 5.1, with a batch size of 64 for ResNet-50, 16 for ResNet-152, 256 for AlexNet, 32 for VGG-19, and 16 for DenseNet-201.

These results resemble those of the Ampere microarchitecture, with the exception that priority streams perform much worse in terms of utilization in general. The reason for this is the reduced computational power and resources of the Turing GPU; it had fewer SMs and could fit fewer blocks of the kernels at a time than the Ampere GPU, and coupled with the lower batch sizes, the kernel execution times and overall runtimes of the tasks increased. This caused fewer opportunities for colocation of the kernels of the two tasks, resulting in worse utilization.

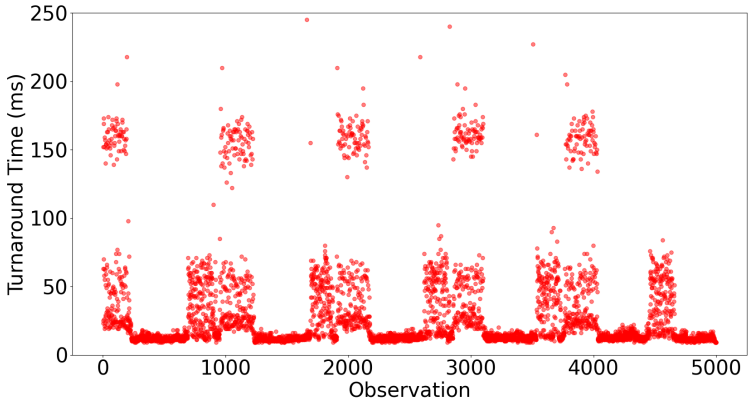


(a) Average Turnaround Times

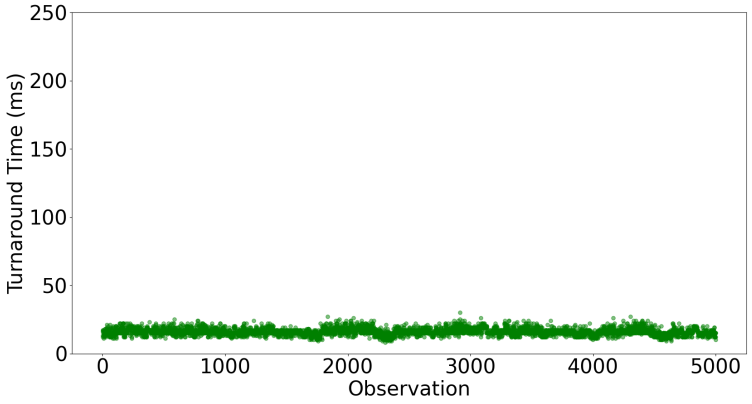


(b) Average Utilization

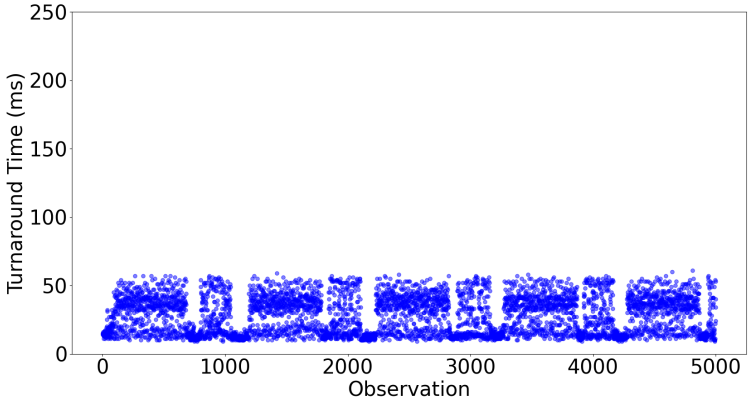
Figure A.1: Turing Turnaround Times and Utilization. The average turnaround times and utilization for each of the three mechanisms on five different models, on the Turing GPU. Note that the turnaround times are the averages of 5000 inference requests, and the measurement of training execution time is the average of 10 runs.



(a) Priority Streams



(b) Time-Slicing



(c) MPS

Figure A.2: Turing Variance. The variance of the turnaround times for the ResNet-50 model on the Turing GPU. Other models' variance results were omitted for space, but resemble these.

Bibliography

[1] Nvidia ampere ga102 gpu architecture: The ultimate play. Technical report, NVIDIA, September 2018.

[2] Nvidia turing gpu architecture: Graphics reinvented. Technical report, NVIDIA, September 2020.

[3] 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.

[4] 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.

[5] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2018.

[6] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu. Mosaic: A gpu memory manager with application-transparent support for multiple page sizes. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[7] M. Awatramani, J. Zambreno, and D. Rover. Increasing gpu throughput using kernel interleaved thread block scheduling. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013.

[8] Mehmet E. Belviranli, Farzad Khorasani, Laxmi N. Bhuyan, and Rajiv Gupta. Cumas: Data transfer aware multi-application scheduling for shared gpus. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, 2016.

[9] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru. Deadline-based scheduling for gpu with preemption support. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 119–130, 2018.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[11] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, 2019.

[12] Guin R. Gilman, Samuel S. Ogden, Tian Guo, and Robert J. Walls. Demystifying the placement policies of the gpu thread block scheduler for concurrent kernels. In *38th International Symposium on Computer Performance, Modeling, Measurements and Evaluation 2020*, 2020.

[13] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[14] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018.

[15] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018.

[16] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic space-time scheduling for gpu inference, 2018.

[17] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the nvidia turing t4 gpu via microbenchmarking, 2019.

[18] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking, 2018.

[19] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2012.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, June 2017.

[21] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel gpu task scheduling for deep learning, 2020.

[22] Hao Li, Di Yu, Anand Kumar, and Yi-Cheng Tu. Performance modeling in cuda streams - a means for high-throughput data processing. *IEEE International Conference on Big Data*, 2014.

[23] Sparsh Mittal and Shraiysh Vaishay. A survey of techniques for optimizing deep learning on gpus. *Journal of Systems Architecture*, 99:101635, 2019.

[24] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael Abu-Ghazaleh. Constructing and characterizing covert channels on gpgpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 354–366, 2017.

[25] NVIDIA. Profiler user's guide, 2007.

[26] NVIDIA. *Multi-Process Service*, June 2020.

[27] NVIDIA. Nsight systems user's guide, 2020.

[28] NVIDIA. Nvidia tensorrt, 2020.

[29] Sreepathi Pai. How the fermi thread block scheduler works (illustrated), 2014.

[30] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 2013.

[31] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. *SIGPLAN Not.*, 50(4), April 2015.

[32] Ignacio Sañudo, Nicola Capodieci, Jorge Martinez, Andrea Marongiu, and Marko Bertogna. Dissecting the cuda scheduling hierarchy: a performance and predictability perspective. 04 2020.

[33] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[34] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on gpus. *SIGARCH Comput. Archit. News*, 42(3), June 2014.

[35] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. Flep: Enabling flexible and efficient preemption on gpus. *ACM SIGARCH Computer Architecture News*, 45:483–496, 04 2017.

[36] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, November 2020.

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

[38] Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Mawhirter, Bo Wu, Chao Li, and Minyi Guo. Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. pages 58–68, 06 2019.

[39] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. Towards high performance paged memory for gpus. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.