DEVELOPING REAL-TIME GPU-SHARING PLATFORMS FOR ARTIFICIAL-INTELLIGENCE APPLICATIONS

Nathan M. Otterness

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2022

Approved by:

James H. Anderson

Don Smith

Don Porter

Lars Nyland

Richard (Dick) Sites

ABSTRACT

Nathan Otterness: Developing Real-Time GPU-Sharing Platforms for Artificial-Intelligence Applications
(Under the direction of James H. Anderson)

In modern autonomous systems such as self-driving cars, sustained safe operation requires running complex software at rates possible only with the help of specialized computational accelerators. Graphics processing units (GPUs) remain a foremost example of such accelerators, due to their relative ease of use and the proficiency with which they can accelerate neural-network computations underlying modern computer-vision and artificial-intelligence algorithms. This means that ensuring GPU processing completes in a timely manner is essential—but doing so is not necessarily simple, especially when a single GPU is concurrently shared by many applications.

Existing real-time research includes several techniques for improving timing characteristics of shared-GPU workloads, each with varying tradeoffs and practical limitations. In the world of timing correctness, however, one problem stands above all others: the lack of detailed information about how GPU hardware and software behaves. GPU manufacturers are usually willing to publish documentation sufficient for producing logically correct software, or guidance on tuning software to achieve "real-fast," high-throughput performance, but the same manufacturers neglect to provide details used when establishing temporal predictability.

Techniques for improving the reliability of GPU software's temporal performance are only as good as the information upon which they are based, incentivising researchers to spend inordinate amounts of time learning foundational facts about existing hardware—facts that chip manufacturers must know, but are not willing to publish. This is both a continual inconvenience in established GPU research, and a high barrier to entry for newcomers.

This dissertation addresses the "information problem" hindering real-time GPU research in several ways. First, it seeks to fight back against the monoculture that has arisen with respect to platform choice. Virtually all prior real-time GPU research is developed for and evaluated using GPUs manufactured by NVIDIA, but this dissertation provides details about an alternate platform: AMD GPUs. Second, this dissertation works towards establishing a model with which GPU performance can be predicted or controlled. To this end, it uses

a series of experiments to discern the policy that governs the queuing behavior of concurrent GPU-sharing processes, on both NVIDIA and AMD GPUs.

Finally, this dissertation addresses the novel problems for safety-critical systems caused by the changing landscape of the applications that run on GPUs. In particular, the advent of neural-network-based artificial-intelligence has catapulted GPU usage into safety-critical domains that are not prepared for the complexity of the new software or the fact that it cannot guarantee logical correctness. The lack of logical guarantees is unlikely to be "solved" in the near future, motivating a focus on increased throughput. Higher throughput increases the probability of producing a correct result within a fixed amount of time, but GPU-management efforts typically focus on worst-case performance, often at the expense of throughput. This dissertation's final chapter therefore evaluates existing GPU-management techniques' efficacy at managing neural-network applications, both from a throughput and worst-case perspective.

ACKNOWLEDGEMENTS

This dissertation and my years in academia would not have been possible without the help and support of many people. First, I thank my advisor, Jim Anderson, for starting me on the fascinating path of GPU research, providing significant material support, welcoming me into an excellent group, and patiently helping me through numerous setbacks over the past six years. Additionally, I am thankful to all of the members of my committee, Lars Nyland, Don Porter, Dick Sites, and Don Smith, who have all instructed or assisted me in various ways during my time as a student, in addition to agreeing to serve on my committee.

Next, I thank the many former and current students with whom I have collaborated on papers: Micaiah Chisholm, Namhoon Kim, Stephen Tang, Thanh Vu, Bryan Ward, and especially the "GPU group," including at various times: Tanya Amert, Joshua Bakita, Vance Miller, Sarah Rust, and Ming Yang. Of these, Joshua Bakita deserves an additional mention for having me as an officemate for several years, meaning that he humored my numerous rants, debates, and off-the-wall ideas, as well as occasional bits of research-related conversation.

I am extremely grateful to the department staff who handled the many odd requests that enabled my research to proceed smoothly. Among many others, I would like to thank Murray Anderegg, Robin Brennan, Jodie Gregoritsch, Denise Kenney, Jim Mahaney, Mike Stone, Missy Wood, and the late Bil Hays for always being willing to hear my requests regarding computer parts, office assignments, travel paperwork, course registration, and a myriad of additional issues.

I thank Professor Fabian Monrose for being willing to take the risk of hiring me in a full-time research position while I only had an undergraduate degree. His decision allowed me to entirely change the trajectory of my career, giving me firsthand experience in computer security, operating systems, low-level programming, and post-undergraduate academia. Without my formative years working under Professor Monrose, I may never have pursued a graduate degree in the first place.

Next, I am grateful to my friends Ivan Pogrebnyak, Patrick Domico, and Perry Harabin for the continued visits, games, and antics, which we have managed to continue even after I became the only one remaining in Chapel Hill. Finally, I am grateful to my family, including my brother and my parents, who always

encouraged my academic pursuits, and especially my wife Nina, and my sons Felix and Jethro, for their endurance of my time in school. As a student, is incredibly difficult to separate my work from my life at home. It was with their encouragement, most of all, that I made it to the point of writing this dissertation.

The research in this dissertation was supported by funding from NSF grants CNS 1409175, CNS 1563845, CNS 1717589, CPS 1239135, CPS 1446631, CPS 1837337, CPS 2038855, and CPS 2038960, AFOSR grant FA9550-14-1-0161, ARO grants W911NF-20-1-0237, W911NF-17-1-0294, and W911NF-14-1-0499, ONR grant N00014-20-1-2698, and funding from General Motors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
AQL	Architected Queueing Language
CDF	Cumulative Distribution Function
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CU	Compute Unit
DRAM	Dynamic Random Access Memory
CUDA	Compute Unified Device Architecture (See Chapter 2, Footnote 1)
DNN	Deep Neural Network
EDF	Earliest Deadline First
FIFO	First In First Out
FLOPs	Floating-point Operations
GCN	Graphics Core Next
GEDF	Global Earliest Deadline First
GPC	Graphics Processing Cluster
GPU	Graphics Processing Unit
GPGPU	General-Purpose GPU (computing)
HIP	Heterogeneous-compute Interface for Portability (See Chapter 2, Footnote 1)
HSA	Heterogeneous System Architecture
IPC	Inter-Process Communication
JSON	JavaScript Object Notation
MIG	Multi-Instance GPU
MMIO	Memory-mapped I/O
MPS	Multi-Process Service
OS	Operating System
ROCclr	ROCm Common Language Runtime
ROCm	Radeon Open Compute (See Chapter 2, Footnote 1)

RT	Real Time
SIMD	Single Instruction Multiple Data
SWaP	Size, Weight, and Power
TPC	Texture Processing Cluster
WCET	Worst-case Execution Time

CHAPTER 1: INTRODUCTION

Nothing is more destructive to the traditional concept of a *safety-critical real-time system* than attempting to build such a system using hardware with unknown timing behavior, and software that exhibits frequent, unavoidable logical failures. Yet, the advent of machine learning, made practical by graphics processing units (GPUs), means that such systems are becoming commonplace—with or without correctness guarantees.

The application of GPU acceleration to machine-learning algorithms, and neural networks in particular,[1] has led to enormous improvements in the field of computer vision over the past decade. With this improvement came an explosion in the number of potential applications, such as handwriting recognition (Memon, Sami, Khan and Uddin 2020), 3D object reconstruction (Han, Laga and Bennamoun 2019), and medical image analysis (Litjens, Kooi, Bejnordi, Setio, Ciompi, Ghafoorian, Van Der Laak, Van Ginneken and Sánchez 2017), to name a few. However, none of these aforementioned applications are as significant to real-time research as *autonomous vehicles*.

Autonomous vehicles are unlike any of the other examples due to their immediate impact on human safety. Outside of contrived examples, incorrect handwriting recognition is unlikely to immediately result in death or injury. Incorrect medical image analysis can certainly result in injury, but, when necessary, medical applications can use high-end computers and results can be reviewed by human experts—reducing the chance that a slow or incorrect algorithm is singly responsible for any mistakes. The potential risks of autonomous-vehicle mishaps are far greater. In this case, a mistake not only harms the vehicle's passengers, but also the vehicle itself, other humans, and other vehicles and property. Put another way, autonomous vehicles are a classic example of a safety-critical real-time system.

A brief overview of real-time systems. Generally, a *real-time system* is a set of hardware or software that must react to events or regularly perform actions within a specific time frame. A computer-controlled machine performing actions on a factory assembly line can certainly be classified as a real-time system. Arguably, even a lower-stakes goal of rendering a video game at 60 frames per second is a real-time system; like the

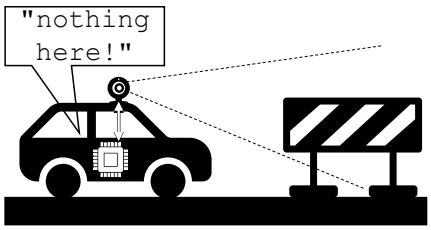
[1] I define neural networks and other computer-vision and machine-learning terminology more thoroughly in Section 2.3.

assembly line, video games must operate within timing constraints to be considered "correct." However, the distinction between real-time and non-real-time systems is most important in cases that are *safety-critical*: where failure to perform on time can risk lives, injury, or large monetary costs. In terms of software, a real-time system generally consists of a set of *tasks*. Tasks are most easily conceived of as computer programs that run multiple, repeating *jobs*. For example, a video-processing *task* may launch a *job* to analyze a new image from a camera every $1/30^{th}$ of a second. Each job of a real-time task has a *deadline*—a point in time by which the job must complete. Note that this simplistic overview is only intended to make it easier to understand the remaining introductory material. The academic community has developed more extensive models for discussing real-time systems, complete with formal notation, terminology, and increased nuance. We save most of this material for a detailed discussion in Section 2.5.

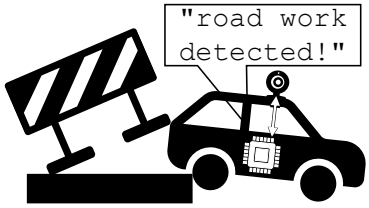
1.1 GPU-Augmented Real-Time Systems

The academic discipline of real-time systems is primarily concerned with guaranteeing the *temporal correctness* of safety-critical systems: ensuring that jobs complete by their deadlines. Focusing primarily on timing is justified by treating *logical correctness* (producing correct results) as an orthogonal concern. If one assumes that safety-critical computations produce correct results, the remaining challenge is to ensure that results arrive in time for them to be useful.

This is easier said than achieved, especially for GPU-augmented systems. Ensuring temporal correctness requires accurate *models*: sets of assumptions, formulas, or rules used to derive or justify predictions about computations' timing requirements. Well-established real-time models of CPU execution often assume that a (relatively small) number of processors operate independently, with some limited contention for shared resources. This does not apply to the highly parallel architecture of a GPU, where a very large number of computations occur, but cannot operate independently. Additionally, ensuring that real-world behavior matches the models' assumptions requires careful *hardware management*: controlling how applications access CPUs, memory, *etc.* Once again, CPU-focused real-time literature contains a large number of well-understood strategies for scheduling computations on one or more CPUs, many of which are difficult to implement or apply on GPU hardware. Novel approaches for both modeling and hardware management must be developed in order to provide a traditional sense of timing correctness for GPU-using computations. All of these problems are compounded when considering *GPU sharing*: allowing multiple, separate, applications to use a



(a) Timeliness is meaningless without logical correctness.



(b) Logical correctness becomes meaningless when delivered too slowly.

Figure 1.1: In modern applications, both timing and logical failures must be prevented.

single GPU in a system. In addition to modeling the behavior of a single GPU-using application, one must also understand how to arbitrate GPU access when several applications attempt to use it at the same time. This is particularly important for autonomous vehicles, where size, weight, and power constraints motivate a desire to install a smaller number of less-powerful computers.

1.2 New Challenges from Modern Applications

Unfortunately, as hinted by this paper's opening sentence, the challenge GPUs pose to this traditional view of real-time systems comes not only from the GPUs themselves, but also *from the applications they are expected to run*. As recently as 2015, entire dissertations about real-time GPU management (*e.g.*, Elliott (2015)) do not even mention the flagship application of modern GPUs: neural networks.

Consider the example in Figure 1.1, which depicts a realistic application (Abodo, Rittmuller, Sumner and Berthaume 2018) of neural-network-based computer vision: detecting road construction. Clearly, timeliness is required for an autonomous-driving application. As shown in Figure 1.1b, correctly detecting road work only helps if the detection is fast enough. At the same time, almost any nontrivial neural network is incapable of guaranteeing correct results for all possible inputs. As we will discuss in Section 2.3, a neural networks' behavior is more akin to *estimating probabilities*. Consider again the actual research that inspired Figure 1.1: in it, Abodo *et al.* (2018) design a neural network that only outputs a relative "confidence" value that a video frame is a road-construction scene. *Using neural networks in a safety-critical system invalidates the assumption that computations always produce logically correct results*. Figure 1.1a shows that a system is not necessarily safe, even if computations complete on time. One may argue that, given slightly more *time*, the situation in Figure 1.1a can be rectified; *e.g.*, if the construction is detected on a subsequent video frame. In fact, this type of application exhibits a fundamental tradeoff between time and logical correctness. Even if

we abandon rigorous requirements for perfect logical correctness, this tradeoff exposes a different problem for the traditional view of real-time systems: *it contradicts the assumption that timing and logical correctness are orthogonal*.

Even leaving aside the difficulties with logical correctness, neural-network applications pose a problem for existing real-time GPU-management approaches simply due to their *complexity*. Even small neural-network applications can easily invoke GPU code several hundred times in the course of analyzing a single image. Applications may also require arbitrary points of synchronization, where the CPU must wait for the GPU (or vice versa) throughout this process. This behavior may not even be the direct responsibility of the computer-vision researchers developing the neural networks—complexity also arises due to implementation details of the high-level software frameworks frequently used to facilitate the neural networks' development. In real-time systems, guaranteeing temporal correctness hinges on the ability to *understand* the interplay of various hardware and software components. As a system's complexity increases, developing a sufficient understanding of the interactions between tasks and hardware becomes increasingly difficult.

While we reserve further discussion of these software frameworks and the changing landscape of GPU applications for Chapters 2 and 5, they are worth mentioning here to provide another point of motivation for our research: simplistic models of GPU behavior and restrictive requirements placed on GPU software are unable to handle modern GPU-accelerated AI. In order to support GPU acceleration for the flagship application of GPU-augmented real-time systems, we require both detailed models and GPU-management techniques capable of handling arbitrarily complex software.

1.3 The Monoculture in Real-Time GPU Research

While the battle among chip manufacturers for dominance in the GPU market has raged for years, the most prominent contenders for at least the past decade have been two companies: NVIDIA and AMD. Of these, NVIDIA is currently the clear leader, boasting over 80% of the market share for discrete desktop GPUs as of mid-2021 (Mujtaba 2021). It is difficult to find reliable information about NVIDIA's share of the embedded-GPU market, but we are forced to assume that it is even more dominant than their share of the desktop market—due to the simple fact that NVIDIA produces relatively popular embedded platforms, such as the "Jetson" line of single-board computers (Otterness, Yang, Rust, Park, Anderson, Smith, Berg and Wang 2017), whereas AMD apparently offers no off-the-shelf embedded-GPU development platforms at

all. NVIDIA's market dominance extends into the research domain, reflected in the near-monopoly its GPUs have in prior work from the real-time community (Section 2.6).

Much of NVIDIA's dominance in the area of general-purpose GPU computing can be attributed to the popularity of CUDA, NVIDIA's proprietary GPU programming framework (NVIDIA 2022). On top of CUDA itself, NVIDIA has heavily supported and encouraged the use of its GPUs in the area of artificial intelligence (AI), as exemplified by its popular CUDA library supporting deep neural networks, cuDNN (NVIDIA 2021). cuDNN was first released in 2007, and has since been adopted into many prominent deep-learning and image-processing frameworks such as Caffe (Jia, Shelhamer, Donahue, Karayev, Long, Girshick, Guadarrama and Darrell 2014), PyTorch (Paszke, Gross, Massa, Lerer, Bradbury, Chanan, Killeen, Lin, Gimelshein, Antiga *et al.* 2019), and TensorFlow (Abadi, Barham, Chen, Chen, Davis, Dean, Devin, Ghemawat, Irving, Isard *et al.* 2016).

In contrast, these popular deep-learning frameworks lacked AMD support for years, causing AMD to fall behind in research adoption. However, AMD has recently been improving its own response to CUDA with the ROCm software stack (AMD Corporation 2022), which includes HIP, a GPU programming interface sporting near-identical features to CUDA (AMD Corporation 2021). The introduction of HIP has allowed AMD to quickly develop versions of the aforementioned deep-learning and AI frameworks compatible with its own GPUs. For example, as of May 2018, the official PyTorch repository has shipped with scripts that allow it to be compiled for AMD GPUs, using ROCm.[2] Similarly, AMD maintains a fork of TensorFlow compatible with its own GPUs.[3] Despite AMD's efforts, however, the dearth of research directed at AMD GPUs could lead one to question whether a serious competitor to NVIDIA even exists. Our work seeks to address this deficiency: when designing systems where failure risks human lives, developers should seek the safest option irrespective of brand loyalty or sales numbers. In short, the real-time GPU research community should not ignore either side in the NVIDIA-vs.-AMD battle.

1.4 Thesis Statement

The ultimate goal of real-time research is to make systems safer. Improving both the *depth* and *breadth* of knowledge for GPU-augmented platforms serves the same goal: improving safety in GPU-using real-time

[2] `https://github.com/pytorch/pytorch/commit/cd86d4c5548c15e0bc9773565fa4fad73569f948`

[3] `https://github.com/ROCmSoftwarePlatform/tensorflow-upstream`

systems. A fair amount of prior real-time GPU research seeks to apply existing high-level concepts to GPU management, but is held back due to lacking an *in-depth* view of GPU details. Other prior research seeks to expose and apply details about GPU behavior, but a larger number of details, often by necessity, end up underrepresenting a greater *breadth* of potential platforms such as AMD GPUs.

Additionally, we must work to ensure that existing research remains relevant in the face of increasingly complex software ecosystems and application designs. Much existing research not only relies on simplifying assumptions about hardware, but also (sometimes implicit) assumptions about the structure of the entire software stack. Modern neural-network applications may communicate with a GPU hundreds of times in a single real-time job and rely on new, under-studied, hardware or software features to maintain sufficient throughput. In addition, neural-network-based applications break out of the mold required by virtually the entire history of real-time study due to providing no logical-correctness guarantees.

This dissertation attempts to address these issues hindering the safety of GPU-augmented real-time systems. In it, we provide details about internal behaviors for both NVIDIA and AMD GPUs, expanding both the breadth and depth of information available for developers producing GPU-augmented real-time systems. We also document the many ways that GPU software has grown in complexity in a short amount of time, and how existing approaches to GPU management may or may not continue to apply. This leads to our thesis statement:

> *Guaranteeing safe execution of modern real-time GPU-accelerated applications is only possible with accurate, sufficiently detailed models of GPU behavior, covering a wide range of hardware and applications. These models can be applied to produce techniques for managing behavior of GPU hardware, while remaining aware of the tradeoffs between timing and logical correctness.*

1.5 Contributions

In support of this thesis, our work makes the following contributions:

1.5.1 Queueing Behavior for NVIDIA GPUs

Our thesis statement implies the need for more sophisticated models for GPUs used in real-time systems. Despite their popularity in real-time research (and real-world embedded systems), prior research contains relatively little information about issues as basic as the order in which jobs will execute when multiple tasks

share an NVIDIA GPU running unmodified, default software. Naturally, unknown internal behavior implies unpredictable timing. While it is certainly possible to produce some real-time management techniques without making any assumptions about such GPU-internal behavior (we discuss some such approaches in Section 2.6.1), if one's goal is simply *predicting* timing behavior, then applying an accurate model of unmanaged NVIDIA hardware and software is likely both simpler and more efficient. Unfortunately, this is only possible if such a model is available in the first place.

In order to discern an adequate model of NVIDIA GPU behavior, we constructed a *microbenchmarking framework* capable of answering *fundamental questions* about how pending GPU work is prioritized—posing insightful questions and developing tools with which to answer them are both key components of a successful reverse-engineering attempt. In Chapter 3, we discuss our questions, the experiments we ran, and the answers we obtained.

1.5.2 Scheduling and Hardware Partitioning Behavior for AMD GPUs

While AMD GPUs remain far less popular than their NVIDIA counterparts in real-time or computer-vision research, they still offer a distinct advantage over their competitors: a fully open-source software stack. Additionally, even lower-tier AMD GPUs provide an oft-desired feature for real-time management: hardware-level support for *partitioning* computational resources—allowing fine-grained control of how much of the GPU's processing capacity gets dedicated to each concurrent task. Unfortunately, even with these apparent benefits, serious barriers still stand in the way of AMD GPUs as a test platform for research. Any researcher hoping to use AMD GPUs is likely to quickly learn a difficult lesson: "open source" does not imply "open information."

Apart from source code, AMD generally offers *less* public documentation about their GPUs than what NVIDIA makes available. Source-code availability is certainly not without benefit, though: it changes the tools available for acquiring information, and gives a first-party, official, way to confirm findings. To use existing software-engineering terminology, we can use *white-box* experiments to study AMD GPUs, whereas the lack of internal information makes only *black-box* experiments possible on NVIDIA. In the prime example from this dissertation, we used knowledge of AMD GPU internals, gleaned from source code and other scant documentation, to design a proof-of-concept experiment that specifically targets weaknesses in AMD's hardware-partitioning implementation. In this experiment, we are able to increase an application's execution

time by more than tenfold, by *increasing* its computational-resource allocation. Chapter 4 both describes this experiment in detail, and more importantly, uses it as a gateway into explaining the full stack of how computational operations are requested and scheduled throughout AMD's software and hardware.

1.5.3 Updated Assumptions About Real-time GPU Software

Our thesis statement claims that it is *insufficient* for proposed real-time GPU-management systems to make overly simplistic assumptions about the behavior of either GPUs' behavior or the software they are running. This claim is largely due to the fact that GPU-accelerated workloads have changed, in a short span of time, to become dominated by neural networks. The individual algorithms underlying neural networks may not be particularly complex, but their utility stems less from novel mathematics and more from the manner in which many operations are composed.

Neural-network tasks are often developed using high-level scripting languages, and may interact with the GPU hundreds of times per job, amplifying the impact of overhead in any proposed management techniques. Also, as mentioned multiple times throughout this introduction, neural-network applications are not even capable of guaranteeing logical correctness, meaning that any claims about "safety" must account for uncertain logical correctness in tandem with uncertain temporal correctness. The primary contributions of Chapters 3 and 4 focus on producing or updating models of GPU hardware, but Chapter 5 addresses these new application-level needs by focusing on updating models of GPU software.

We structure Chapter 5 around our creation of a new benchmark for evaluating real-time GPU management: an image-processing neural network initially designed for performance-sensitive contexts (Yu and Huang 2019*b*). Unlike the microbenchmark-centric evaluation from Chapters 3 and 4, the high-level frameworks used in neural-network development come with with their own set of performance pitfalls. In adapting an existing neural network into a suitable benchmark, we encounter, explain, and fix several such pitfalls. In investigating these pitfalls, we conduct a case study using *KUtrace* (Sites 2021), a tool for tracing system-wide CPU activity including the operating system (OS),[4] to fully analyze the activity of a single job of the new benchmark.

[4]For unfamiliar readers, we discuss operating-system basics in Section 2.4.

1.5.4 Implement and Reevaluate Prior GPU-Management Approaches

At least as stated, our contributions so far primarily serve to enrich models of GPUs and applications, but our thesis statement also includes *realistic management of GPU hardware*—which must involve testing real-world code. If prior management approaches are wholly insufficient, then we should be able to demonstrate this is the case, or, more constructively, demonstrate specifically where they fall short and how these shortcomings may be addressed.

In light of this, we draw on the GPU- and software-related information gained earlier in the dissertation to form our final contribution: we take multiple GPU-management approaches from existing real-time literature and implement these techniques on a common platform. With a common baseline, we can use our implementation to conduct a study: how do these existing management proposals affect both timing and logical-correctness tradeoffs in a modern neural network? Chapter 6 describes the implementation, experimental setup, and results of this evaluation.

1.6 Organization

Following this introduction, Chapter 2 provides necessary background material, including information about GPU programming, computer-vision applications, and real-time systems. Chapter 2, Section 2.6, contains our discussion of prior related work. Next, Chapters 3 and 4 cover our investigation of NVIDIA and AMD GPU behavior, respectively. We explore new complexities in modern GPU-using software and develop a representative benchmark application in Chapter 5. In Chapter 6, we build on the material from earlier chapters to evaluate GPU-management approaches from prior real-time literature. Finally, we make our concluding remarks in Chapter 7.

CHAPTER 2: BACKGROUND AND MOTIVATION

The research covered in this dissertation naturally requires understanding general-purpose GPU programming and a general overview of GPU hardware. Additionally, due to the software applications in our case studies, we cover some basic computer-vision and neural-network fundamentals in Section 2.3. We assume familiarity with some basic real-time concepts and notation, which we define in Section 2.5. When discussing practical applications, however, we rely on several concepts related to operating systems, covered in Section 2.4. Finally, Section 2.6 of this chapter considers prior related work.

2.1 GPU Programming

It is usually easiest to introduce GPU programming from the top down, by following the high-level life cycle of a GPU-using application. For practical purposes, we assume that any code being executed by the GPU is running at the request of a CPU program. CPU programs almost always issue requests to the GPU using higher-level APIs (application programming interfaces). For example, OpenGL (Khronos Group 2021) and Vulkan (Khronos Vulkan Working Group 2022) are APIs designed to leverage GPUs' original purpose: rendering graphics. It may come as a surprise, then, that we almost never refer to these APIs in a dissertation about GPUs—in fact, we focus very little on graphics at all. Instead, we primarily concern ourselves with the frameworks designed for non-graphical *general-purpose GPU* (GPGPU) computations: *CUDA* for NVIDIA GPUs, and *HIP* for AMD GPUs.[1]

2.1.1 An Example GPU Program

In their basic form, both CUDA and HIP programs follow a similar pattern when offloading work to the GPU: copy data to GPU memory, invoke some GPU code to process data, and copy results back to CPU memory. Figure 2.1 shows an example of such a program, using the HIP API to offload the addition of two

[1] While both "CUDA" and "HIP" began as acronyms, the meaning of both abbreviations is rarely intended, even in official documentation. We, likewise, use these terms only as names of the respective GPU-programming frameworks, and do not intend to imply further meaning behind the acronyms.

vectors of floating-point numbers to the GPU. The following steps give a more detailed explanation of the activity carried out by Figure 2.1:

1. **Allocate buffers in GPU memory, both to hold input data and to receive results.** This is carried out by calls to the `hipMalloc` function shortly after the start of `main` in Figure 2.1. Note that Figure 2.1 does not explicitly define the `vector_size` variable—in real code, `vector_size` would need to be set to the size of vectors `a`, `b`, and `c`, in bytes. Additionally, Figure 2.1 does not include code for allocating or initializing the contents of the vectors in CPU memory, but this can be accomplished using standard C library functions such as `malloc`, `mmap`, `fread`, *etc.*

2. **Create a "stream:" a queue to hold GPU operations.** This is accomplished in Figure 2.1 using the `hipStreamCreate` function. For this example, we only need to think of streams as FIFO (first-in-first-out) queues that hold operations to be carried out on the GPU. Put another way, each successive operation enqueued in a stream only *begins* executing on the GPU after all previous operations in the same stream have completed. Streams can be useful for purposes beyond ordering requests, but we save these additional details for Section 2.1.2.

3. **Enqueue several operations in the stream.** The stream created in the previous step ensures that the following operations execute in FIFO order:

   (a) **Copy input data from CPU memory to GPU memory.** The code in Figure 2.1 issues two `hipMemcpyAsync` calls, specifying the `stream` into which they should be enqueued. Each of these two operations is responsible for copying the content of one input vector in CPU memory to the corresponding buffer in GPU memory.

   (b) **Invoke a piece of GPU code, called a *kernel*.** As is common throughout GPU literature, we use the term "kernel" to refer to a section of code that runs on the GPU. When it is necessary to refer instead to the operating-system notion of "kernel code," we use alternate terminology such as "driver code" or "Linux kernel code" (see Section 2.4). Programs using the HIP API call the `hipLaunchKernelGGL` function to launch kernels. This function takes several arguments:

      - The kernel to launch. This has the appearance of a standard C-style function pointer, specifying the `VectorAdd` kernel. In source code, GPU kernels such as `VectorAdd` are

nearly identical to standard C or C++ functions, but are annotated with the `__global__` keyword. This indicates to the compiler that the function's code is to be run on the GPU.

- The `block_count` and `thread_count` arguments, which control the number of parallel GPU threads with which to run the kernel. We discuss these arguments further in Section 2.1.2.
- The size of a shared buffer allocated from GPU scratchpad memory. The simple code in Figure 2.1 sets this to `0` because it does not need any such memory. More sophisticated kernels may request shared memory for communication between concurrent GPU threads.
- The stream into which the kernel-launch request is enqueued.
- The "normal" arguments to the kernel. `VectorAdd` requires pointers to the three buffers in GPU memory holding the input and result vectors, and we provide these as the final three arguments to `hipLaunchKernelGGL`.

(c) **Copy the resulting data from GPU memory to CPU memory.** This is accomplished by the final call to `hipMemcpyAsync` in Figure 2.1. Recall that enqueueing this operation after the kernel launch ensures that it will execute only after the kernel has completed.

4. **Wait for all operations in the stream to complete.** Even though the operations enqueued in the stream execute sequentially with respect to one another, they execute asynchronously with respect to the CPU. In order to wait for all of the enqueued GPU operations to complete, the program calls `hipStreamSynchronize`, which blocks the calling CPU thread until the last operation in the given stream has finished executing.

5. **Further processing and cleanup.** While we omit any further code from Figure 2.1, most GPU-using applications continue to process the data obtained from the GPU (displaying or storing results, *etc.*). Applications may also free any GPU memory for other uses, using other functions from the GPU-programming API.

Remarks on HIP and the CUDA APIs. Our choice to provide a detailed explanation of the HIP example in Figure 2.1 was arbitrary; our explanation could have just as easily been based on the CUDA API. In fact, the code in Figure 2.1 barely changes at all when translated into a program using CUDA; we give equivalent CUDA code in Figure 2.2. When comparing these two figures, we see that the code in the

```
  __global__ void VectorAdd(int *a, int *b, int *c) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}

int main() {
  float *a, *b, *c;
  // Allocate vectors a, b, and c in GPU memory.
  hipMalloc(&a, vector_size);
  hipMalloc(&b, vector_size);
  hipMalloc(&c, vector_size);

  float *a_cpu, *b_cpu, *c_cpu;
  // [Omitted for space: allocating and initializing vectors
  // a, b, and c in CPU memory.]

  // Create a stream
  hipStream_t stream;
  hipStreamCreate(&stream);

  // Copy input data to the GPU. [Omitted: initializing the
  // content of the vectors a_cpu and b_cpu in CPU memory.]
  hipMemcpyAsync(a, a_cpu, vector_size, hipMemcpyHostToDevice,
    stream);
  hipMemcpyAsync(b, b_cpu, vector_size, hipMemcpyHostToDevice,
    stream);

  // Launch the kernel. [Omitted: setting block_count and
  // thread_count to match the number of vector_elements.]
  hipLaunchKernelGGL(VectorAdd, block_count, thread_count,
    0, stream, a, b, c);

  // Copy results from the GPU back to CPU memory.
  hipMemcpyAsync(c_cpu, c, vector_size, hipMemcpyDeviceToHost,
    stream);

  // Wait for operations enqueued in the stream to complete.
  hipStreamSynchronize(stream);

  // [Omitted: Cleanup, etc.]
}
```

Figure 2.1: Code using AMD's HIP API to define and launch a GPU kernel.

`VectorAdd` kernel does not need to change at all, and most API function names only require replacing `hip` with `cuda`. The most noticable difference between Figures 2.1 and 2.2 is the syntax for launching a kernel. In CUDA, kernels are launched using a special syntax involving the `<<<...>>>` characters, whereas HIP opts for standard C syntax: calling the `hipLaunchKernelGGL` function.

HIP and CUDA programs are unfortunately not entirely interchangeable—for example, some GPU-specific inline assembly instructions may exist for NVIDIA GPUs but not AMD, or vice versa. In general,

```
__global__ void VectorAdd(int *a, int *b, int *c) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c[i] = a[i] + b[i];
}

int main() {
  float *a, *b, *c;
  // Allocate vectors a, b, and c in GPU memory.
  cudaMalloc(&a, vector size);
  cudaMalloc(&b, vector size);
  cudaMalloc(&c, vector_size);

  // ...

  // Create a stream
  cudaStream t stream;
  cudaStreamCreate(&stream);

  // Copy input data to the GPU.
  cudaMemcpyAsync(a, a_cpu, vector_size, cudaMemcpyHostToDevice,
    stream);
  cudaMemcpyAsync(b, b_cpu, vector_size, cudaMemcpyHostToDevice,
    stream);

  // Launch the kernel.
  VectorAdd<<<block_count, thread_count, 0, stream>>>(a, b, c);

  // Copy results from the GPU back to CPU memory.
  cudaMemcpyAsync(c_cpu, c, vector_size, cudaMemcpyDeviceToHost,
    stream);

  // Wait for operations enqueued in the stream to complete.
  cudaStreamSynchronize(stream);
  // ...
}
```

Figure 2.2: Code using NVIDIA's CUDA API to define and launch a GPU kernel.

though, the difference between CUDA and HIP code is minimal enough that AMD provides an open-source tool, `hipify`,[2] capable of automatically transforming most CUDA code into HIP code compatible with AMD GPUs. Porting code in the opposite direction is arguably even easier, as HIP code typically supports NVIDIA GPUs without modification. This is due to AMD providing a set of C header files[3] that use thin wrapper functions and C macros to implement the HIP API on top of the CUDA API.

[2] `https://github.com/ROCm-Developer-Tools/HIPIFY`

[3] `https://github.com/ROCm-Developer-Tools/hipamd/tree/develop/include/hip/nvidia_detail`

2.1.2 Controlling Parallel Execution of GPU Code

Threads and blocks. In GPU programming, the term *thread* refers to a single logical thread of computation on the GPU. As mentioned already, when invoking a kernel, the (CPU) program specifies the number of parallel GPU threads with which to execute the kernel code. Each individual thread executes identical kernel code, but is able to maintain local variables and operate on separate pieces of data. In truth, GPU threads are a convenient abstraction designed to simplify writing software that takes advantage of the GPU's parallel-processing hardware in the same way one may write a multi-threaded CPU application. The underlying hardware responsible for thread execution, however, is quite different from what one would see in a typical CPU core. In AMD GPU hardware, for example, each thread will be executed using a single lane in one of the GPU's SIMD (single instruction, multiple data) vector-processing units (AMD Corporation 2011).

Both Figures 2.1 and 2.2 illustrate the concepts of threads, *blocks*,[4] and streams as used in real applications. The beginning of the code in both figures defines a kernel for setting vector `c` to the sum of input vectors `a` and `b`. The `main` function later launches the kernel, using either `hipLaunchKernelGGL` or CUDA's special kernel-launch syntax to enqueue a kernel-launch request in a stream. Of particular interest are the `block_count` and `thread_count` arguments when issuing this kernel-launch request: these two arguments control the number of parallel threads created to execute the kernel code. A *thread block*, or simply *block* consists of up to 1,024 parallel threads (specified by `thread_count`), all of which execute the same kernel code. The number of threads in a block is limited to 1,024 by both NVIDIA and AMD hardware, but billions of blocks may be requested per kernel—effectively an unlimited quantity for practical applications. The total number of GPU threads is given by multiplying the number of blocks by the number of threads per block.

As shown in Figures 2.1 and 2.2, threads in the `VectorAdd` kernel are able to use the special `blockIdx` and `threadIdx` variables to access their per-thread block and thread indices, meaning that threads and blocks serve the essential purpose of distinguishing between GPU threads within a running kernel. In the example kernel, each thread uses this information to compute a unique index into the vectors. However, thread blocks also play another role: they serve as schedulable entities when dispatching work to the GPU's

[4] After shifting to a heavier focus on the HIP API, AMD seems to have adopted CUDA terminology, likely to reduce mismatched terms in cross-platform code. In material pertaining to APIs other than HIP, such as OpenCL or the lower-level HSA runtime (HSA Foundation 2018*b*), AMD typically uses the term *work item* to refer to a single GPU thread and *workgroup* to refer to a thread block.

computation hardware. We cover this role in greater detail in the relevant chapters: Chapter 3 for NVIDIA, and Chapter 4 for AMD GPUs.

Additional uses for streams. As discussed, the `stream` argument is also present when launching a kernel. We already covered one important aspect of using HIP or CUDA streams: ensuring that operations, such as memory transfers or kernel launches, occur in FIFO order. Explicitly creating and specifying a stream in source code is actually optional; both kernel launches and memory-transfer operations default to being enqueued in a special "null stream" if the user chooses not to create and provide a `stream` argument to the corresponding function calls. Since Figures 2.1 and 2.2 only require simplistic in-order execution within a single thread, omitting the `stream` variable from them entirely would still result in valid code, and the code would even produce identical results; each operation would execute sequentially in the null stream.

Nonetheless, our usage of a user-created stream in Figures 2.1 and 2.2 serves an intentional illustrative purpose. Using the null stream ends up being a poor choice in many sufficiently complex GPU-using applications, namely any application issuing GPU work from multiple concurrent CPU threads. This is due to a guarantee made by both the CUDA and HIP APIs: using the null stream intentionally prevents parallel execution with other kernels, potentially preventing full use of available hardware.[5] By contrast, kernels submitted to separate user-defined streams can at least be *potentially*[6] scheduled concurrently on GPU hardware. Unintentional loss of parallelism due to careless use of the CUDA or HIP API is an easy pitfall for real-time applications to fall into, and we cover a couple examples of this in Section 3.5. In the meantime, manually creating and specifying a stream is simply a good practice, which we adopt in Figures 2.1 and 2.2.

2.2 GPU Hardware

Broadly speaking, GPUs manufactured by both NVIDIA and AMD share some basic design similarities. Refer to Figure 2.3 as an introductory example, which contains a high-level block diagram of the compute hardware in AMD's Radeon VII GPU. The bulk of the Radeon VII's compute power comes from its collection of *compute units*, abbreviated as CUs. Each CU contains an independent L1 cache, a *scalar unit* used for logical and control-flow operations, and several SIMD vector processing units responsible for carrying out

[5] This behavior is documented at `https://rocmdocs.amd.com/en/latest/ROCm_API_References/HIP_API/Stream-Management.html` for HIP and `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#implicit-synchronization` for CUDA.

[6] See Chapters 3 and 4 for details.

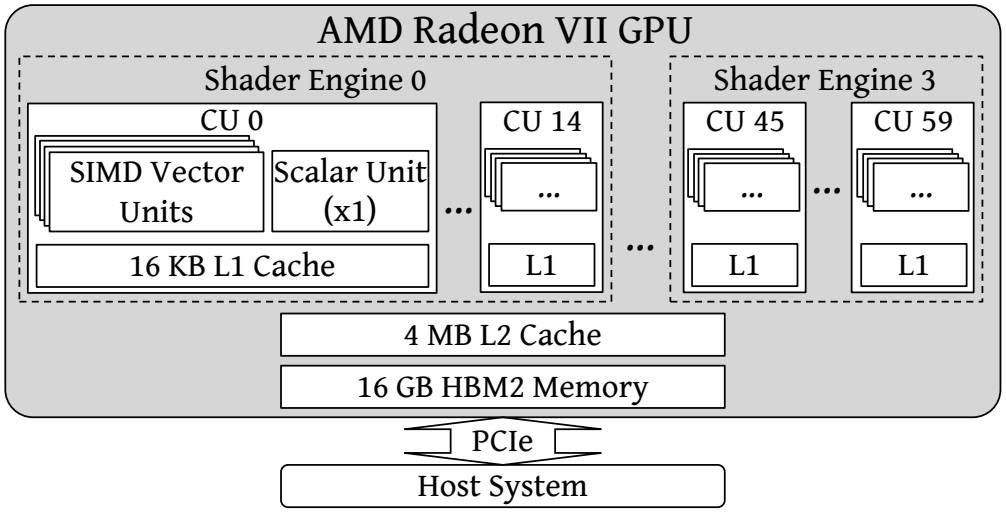


Figure 2.3: Overview of hardware in an AMD Radeon VII GPU.

the parallel computations. CUs are themselves organized within higher-level hardware units known as *shader engines*. As shown in Figure 2.3, the Radeon VII GPU contains four shader engines and 60 CUs in total. The 60 CUs are divided evenly among the four shader engines, meaning that each shader engine contains 15 CUs. All CUs in the entire GPU share four megabytes of L2 cache, and sixteen gigabytes of HBM2 DRAM.

While internal details certainly differ, a block diagram for an NVIDIA GPU has some visually similar structure to that of an AMD GPU. Figure 2.4 shows the high-level architecture for NVIDIA's H100 GPU, which was released in March 2022 and is NVIDIA's most powerful off-the-shelf GPU at the time of writing. In NVIDIA GPUs, *streaming multiprocessors* (SMs) are most directly analogous to CUs in AMD GPUs. In a similar fashion to how the Radeon VII's CUs are grouped into shader engines, the SMs in NVIDIA's H100 are grouped into *texture processing clusters* (TPCs), which are in turn grouped into *graphics processing clusters* (GPCs). Lower-end NVIDIA GPUs may contain fewer GPCs, or a reduced number of TPCs per GPC.

We include Figure 2.4 for two purposes. First, it allows us to draw the comparison between NVIDIA's *SM* terminology vs. AMD's *CU*s, but it also serves as an illustration of the current state of top-end GPU hardware. Despite the H100's superior performance, the "modest" GPUs used in this dissertation's experiments retain one major advantage over the H100: *cost*. With a price speculated to exceed 36,000 dollars,[7] the H100 is simply cost-prohibitive as a platform for timing-related research.

[7] `https://www.tomshardware.com/news/nvidia-hopper-h100-80gb-price-revealed`

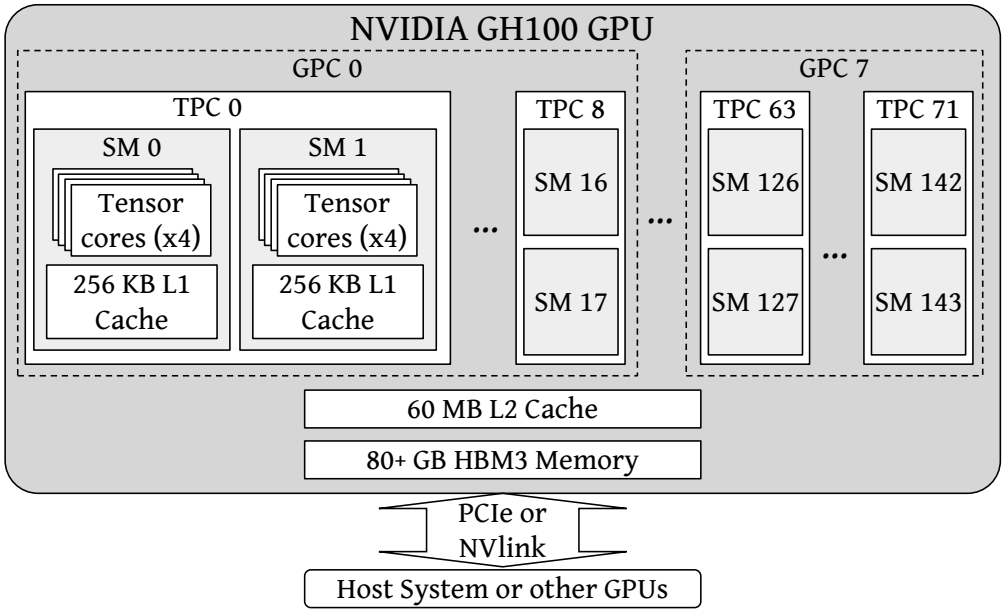


Figure 2.4: Overview of hardware in an NVIDIA H100 GPU.

Compute hardware and kernel execution. CUs and SMs occupy more than similar positions in our block diagrams: they serve analogous roles when executing kernels. Recall from Section 2.1.2 that a program requests a number of *thread blocks* when launching a kernel. GPU hardware assigns these thread blocks to CUs or SMs. In this respect, both NVIDIA and AMD hardware treats blocks similarly: threads from a single block are never divided across multiple CUs or SMs, and never migrate between CUs or SMs. Both CUs and SMs are capable of hosting a total of 2,048 in-flight threads at a time (Otterness and Anderson 2020). Note that the limit of 2,048 threads per CU or SM will always be sufficient for at least two concurrent blocks: as we stated in Section 2.1.2, each block contains at most 1,024 threads. In Chapters 3 and 4 we revisit this topic in detail, by asking fundamental questions about *how* thread blocks are mapped to SMs and CUs.

While up to 2,048 threads may be assigned to a CU or SM, only a smaller subset of these will actually be "executing" at once. Each SM on NVIDIA GPUs contains four *warp schedulers*, each of which is responsible for selecting a *warp* of threads to execute. Warps are simply groups of 32 threads from within a block, and all threads in a warp execute in lockstep. At first, it may sound inefficient to only run four warps of 32 threads when up to 2,048 threads are assigned to an SM, but doing so enables *latency hiding*: if any warp stalls, *e.g.*, in order to wait for memory or a shared hardware unit, the warp scheduler can immediately switch to a different warp from among the SM's threads. Latency hiding attempts to maintain consistent throughput by

ensuring that at least *some* threads are always making progress, even if a large number of other threads have stalled.

The story is very similar on AMD, albeit with different terminology. AMD's documentation uses the term *wavefront* anaologously to NVIDIA's "warp." Similarly to NVIDIA, the CUs in AMD GPUs also can schedule up to four wavefronts at a time, but unlike warps, AMD's wavefronts may not always contain 32 threads. AMD's published specifications technically allow for any wavefront size that is a power of 2 (HSA Foundation 2018*b*), and the Radeon VII, for example, uses 64-thread wavefronts (AMD Corporation 2011).

Discrete and integrated GPUs. Both example GPUs shown so far, AMD's Radeon VII and NVIDIA's H100, are *discrete* GPUs, meaning that they are on separate chips from the system's CPU. Discrete GPUs must interact with other hardware over a communication bus, such as PCIe or proprietary alternatives like *NVlink* (mentioned in Figure 2.4). Alternate buses like NVlink may offer higher bandwidth, but are usually only available for higher-end GPUs using manufacturer-specific hardware. For these reasons, we limited ourselves to the PCIe bus in our experiments involving discrete GPUs. This does not mean that we ignore different communication methods between the CPU and GPU altogether—our work also covers some *integrated* GPUs.

Unlike discrete GPUs, integrated GPUs typically *share hardware*, such as DRAM, with the CPU. Integrated GPUs are actually quite prevalent in day-to-day life, mostly in systems like smartphones or lightweight laptops—where powerful GPUs are either unneeded or unwanted. When producing CPUs for these systems, hardware manufacturers package GPU functionality into their chips to provide basic acceleration for graphical displays. While laptops and smartphones are more common in real life (or at least, more visible), we do not explicitly consider such systems in this dissertation, due to our focus on safety-critical AI and computer vision. Our focus allows us to narrow the scope of the integrated-GPU systems we consider: if safety-critical computations are being carried out using an integrated GPU, then the GPU is likely to be part of an *embedded system*. This assumption is justified by a simple fact: non-embedded safety-critical contexts, with fewer restrictions on size and electricity, would be better served by discrete GPUs, as discrete GPUs can offer vastly more computational power and greater cost effectiveness.

Further narrowing the scope of hardware platforms, there are actually very few integrated GPUs designed for embedded applications. In fact, we are only aware of one off-the-shelf set of platforms meeting this

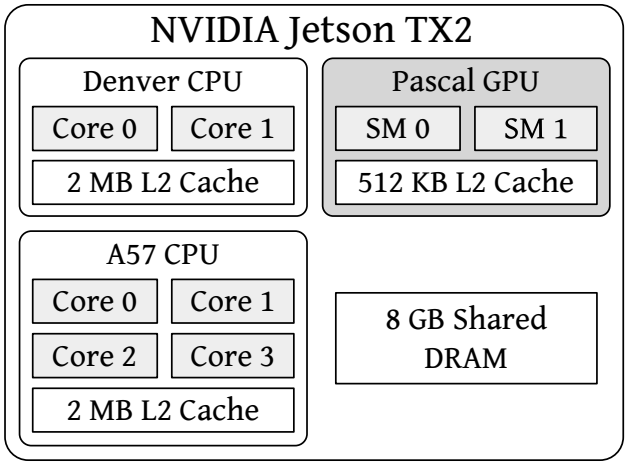


Figure 2.5: Overview of hardware in an NVIDIA Jetson TX2 system-on-chip.

requirement: NVIDIA's "Jetson" line of single-board computers.[8] Figure 2.5 shows the hardware available in one of these systems used in our work, the *Jetson TX2*.

Figure 2.5 illustrates the main property we expect from an integrated-GPU system-on-chip: its Pascal-architecture GPU shares eight gigabytes of DRAM with all of the CPU cores. Leaving aside the integrated GPU for now, the Jetson TX2 is also notable for its heterogeneity. Its six CPU cores are divided between two 64-bit ARM CPUs: a four-core ARM Cortex A57 CPU and a two-core NVIDIA "Denver" architecture CPU with higher single-threaded performance (Franklin 2017). While the CPUs and GPUs share main memory, Figure 2.5 illustrates the fact that the TX2's CPUs and GPU each have separate L2 caches.

As for the GPU, the TX2 is similar to many integrated GPUs in that it is incredibly impoverished compared to its discrete counterparts. The most striking comparison is in the number of SMs available: the discrete H100 from Figure 2.4 contains 144 SMs, while the integrated TX2 offers only two. Granted, this is not an entirely direct comparison: the internal SM architecture differs, and the TX2 is about five years older than the H100. We can offer some closer comparisons, however. NVIDIA's *Titan X Pascal* discrete GPU was released shortly after the Jetson TX2, used the same basic GPU architecture, and contained 30 SMs. Alternatively, we can consider a modern integrated GPU rather than an older discrete GPU. NVIDIA's *Jetson AGX Orin* platform, released in 2022, contains 16 SMs (Karumbunathan 2022)—vastly beyond the TX2's two SMs, but still a far cry from the power available on a modern discrete GPU.

[8] `https://developer.nvidia.com/embedded-computing`. NVIDIA offers other embedded-GPU systems such as its DRIVE platform, but these tend not to be "off-the-shelf" and harder to obtain.

The changing landscape of GPU hardware. This brief introduction of GPU hardware has already mentioned several GPUs released over the course of about six years:

- The NVIDIA Jetson TX2 (Figure 2.5), an integrated GPU containing two SMs and released in late 2016. Approximate cost at launch: 600 dollars.
- The NVIDIA Titan X Pascal, a discrete GPU containing 30 SMs and released in 2017. Approximate cost at launch: 1,200 dollars.
- The AMD Radeon VII (Figure 2.3), a discrete GPU containing 60 CUs and released in 2019. Approximate cost at launch: 700 dollars.
- The NVIDIA Jetson AGX Orin, an integrated GPU containing 16 SMs and released in 2022. Approximate cost at launch: 2,000 dollars.
- The NVIDIA H100 (Figure 2.4), a discrete GPU containing 144 SMs and released in 2022. Approximate cost at launch: 36,000 dollars.

One pattern is already clear from this short list: the computational power available in both integrated and discrete GPUs is growing rapidly. Like all processors, GPUs' performance has continually increased since they first became available. However, computational power is not the only interesting story contained in this small list of GPUs: the implication behind the noticable jumps in monetary cost is equally relevant to this dissertation.

To this day, a large portion of GPUs sold by both AMD and NVIDIA still go towards applications where fast graphics are paramount; NVIDIA reports that slightly over half of their revenue in 2021 came from video-game and visualization-related markets (Reiff 2022). Nonetheless, NVIDIA's customers are certainly not spending 36,000 dollars on an H100 GPU in order to play video games. NVIDIA's whitepaper (NVIDIA Corporation 2022*d*) about the H100 GPU states:

> Note that the H100 GPUs are primarily built for executing datacenter and edge compute workloads for AI, HPC, and data analytics, but not graphics processing. Only two TPCs in both the SXM5 and PCIe H100 GPUs are graphics-capable...

Similar changes are taking place within the embedded-GPU domain. A review written shortly after the TX2's release (Benchoff 2017) calls the Jetson TX2 "high end" and concludes with some lines that, in light of the 2,000 dollar Jetson AGX Orin, already seem comically out of date:

> This [the Jetson TX2] is not a toy. This is an engineering tool. This is a module that will power a self-driving car, or a selfie-capturing quadcopter. These are hard engineering problems that demand fast processing with a low power budget.
>
> There's a reason the TX2 Developer Kit is expensive. The market for a device like this is tiny...

There are three relevant points to draw from these small examples. First, they offer clear support for our assertions in Chapter 1: the evolution of GPUs is characterized not just by increasing computational power, but also by tailoring new hardware towards general-purpose applications. Second, the rapid pace of change in GPU architecture keeps real-time GPU research in a perpetual state of immaturity. This immaturity is not just due to numbers of SMs or CUs: each successive generation of GPUs also reflects internal architectural overhauls. We have only mentioned one AMD GPU so far, but as we discuss in Chapter 4, AMD is equally involved with reorienting their GPU and software architectures to meet new demands. The third and final point from this list is that *GPUs are valued higher than ever before*. Contrary to the TX2 review quoted above, it is unlikely that NVIDIA would increase the costs of their embedded GPUs by a factor of four if they were having trouble selling Jetson TX2s for 600 dollars. If anything, the Jetson AGX Orin indicates that the niche occupied by embedded GPUs is *expanding*.

2.3 Deep Learning and Computer Vision

A full account of computer-vision research is not required for understanding this dissertation, but some knowledge of current applications provides helpful context about the evaluation methods and the role of GPUs as an enabling technology.

The application of GPUs to machine learning has led to advancements in the past decade that can scarcely be overstated. The introduction of the *AlexNet* image-classification system in 2012 (Krizhevsky, Sutskever and Hinton 2012) is frequently cited as the start of this paradigm shift. *Image classification* is a fundamental problem in computer vision: as depicted in Figure 2.6, an image-classification system attempts to assign the correct "class" to an input image, where the class is one of a finite set of possible labels representing the

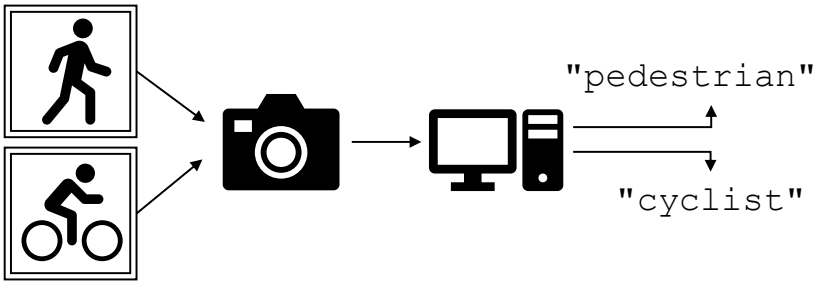


Figure 2.6: Computer vision's "classification" task.

image's subject (*e.g.*, "dog", "car", *etc.*). *AlexNet* was remarkable not only due to the fact that it exhibited a 10% improvement in accuracy over prior image-classification algorithms, but also due to *how* it achieved such an improvement. At the core of *AlexNet*'s approach was, in the words of the authors, "a very efficient GPU implementation," making use of two GPUs to accelerate the training process by which a *convolutional neural network* (CNN) "learns" to classify images.

A *neural network* is a sequence of functions or transformations that map some input data (*e.g.*, an image) to a desired output (*e.g.*, a label). A neural network is composed of many small processing units, known as *neurons*, which individually carry out small, fast, mathematical operations. Neurons are typically organized into sequential *layers*, in which neurons are arranged into specific structures to carry out different operations. A CNN is simply a type of neural network that includes *convolution layers*. In convolution layers, neurons are configured to perform *convolution* operations on small regions of neighboring pixels, making them particularly powerful at identifying recurring small features (*i.e.*, small shapes such as corners or edges) at different locations in an input image. Even though neural networks were introduced in data-analysis contexts as early as the 1960s, the recent success of CNNs and GPUs in systems like *AlexNet* has led to a major surge in their popularity (Schmidhuber 2015).

2.3.1 Adaptable Neural Networks

Given our desire to evaluate some of the tradeoffs between logical and temporal correctness in real-time GPU usage, it is worth noting that computer-vision research often explores similar tradeoffs between accuracy and computational cost. This is especially true for recent *adaptable neural network* architectures, such as Kim, Ahn and Oh (2018), Vu, Eder, Price and Frahm (2020), Yu and Huang (2019*b*), and Yu, Yang, Xu, Yang and Huang (2019). However, computer-vision publications typically choose to view tradeoffs in terms of

accuracy and required compute power, *i.e.*, floating-point operations (FLOPs), which is subtly different from our evaluation in Chapter 5. While the approach favored in computer-vision domains allows researchers to present results in a manner agnostic to underlying platforms, it fails to capture the notion that *timeliness* is required in real systems, which depends on more than raw power.

2.3.2 Deep-Learning Programming Frameworks

Most modern computer-vision research (or any other neural-network-based research) relies on software frameworks to simplify the process of designing, implementing, training, and evaluating increasingly complex neural-network architectures. While a handful of such frameworks exist, *PyTorch* (Paszke *et al.* 2019) and *tensorflow* (Abadi *et al.* 2016) are the most popular by far. Of these two, PyTorch currently dominates academic computer-vision research (He 2019), and serves as a key component of our studies in Chapter 5. Unlike many traditional real-time applications, PyTorch applications are typically written in the high-level Python programming language and, to simplify development, involve many layers of software abstraction before requesting any GPU computations. We devote much of Chapter 5 to these application-level implementation details.

2.4 Operating Systems

Though different definitions exist, for our purposes an *operating system* (OS) is the piece of software responsible for a variety of essential services across an entire computer. Among the operating system's foremost responsibilities is *managing shared hardware*: determining which applications have access to which CPU cores, managing DRAM allocation, arbitrating access to peripheral devices, *etc.* This is possible due to hardware support: the OS is able to execute *privileged code*, which is allowed to access special CPU instructions and registers with which the rest of the system can be managed.

"Kernel" and "user" in an OS context. When discussing operating systems, one typically uses the term *kernel* to refer to the aforementioned portion of privileged code ultimately responsible for managing system hardware. Other, unprivileged code, including the applications running on the system, is referred to as *user code*. (Depending on definitions, many consider a full operating system to not just include the kernel, but also some essential utilities consisting of unprivileged user code.) One frequently encounters terms like *kernelspace* in reference to code or data that is used only within privileged contexts, or, respectively the term

userspace in reference to unprivileged code or data. In this dissertation, we attempt to avoid using the typical "kernel" terminology (without additional clarification) in the context of operating systems, in order to avoid confusion with its other meaning in the context of GPU programming.

Mediating between applications and devices. This introductory material on operating systems becomes especially important in Section 4.2, where we discuss some of the code in the Linux operating system: Linux's *device driver* for AMD GPUs. While certainly important, we are unable to investigate NVIDIA's driver code in as much detail, as significant portions of it are closed source.[9] A device driver (or simply a *driver*) is a portion of operating-system code responsible for interacting with a particular hardware device. Privileged code is almost always required for at least some stages of hardware configuration, but ultimately the goal of a device driver is to provide an interface with which userspace applications can access the device. Therefore, the entire code path with which applications control GPU hardware involves both unprivileged "userspace" code and privileged "kernelspace" code running within the operating system's GPU driver.

2.5 Real-Time Systems

In this dissertation, we sometimes use notation and definitions conventionally used throughout the study of real-time systems. A real-time system is characterized by a *task set*, usually denoted τ, comprised of n tasks: $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task executes a series of *jobs*. In formal notation, jobs are conventionally denoted with a J, with subscripts indicating the task that released the job, as well as the job's number. For example, job $J_{2,4}$ would indicate the fourth job released by task τ_2.

Tasks are characterized by several parameters. First, is the *period* of a task, which is usually denoted with a T; T_i is the period of task τ_i. The period determines the rate at which the task releases jobs. Under the *periodic* task model, T_i denotes the exact amount of time between subsequent job releases of τ_i (Liu and Layland 1973). Under the *sporadic* task model, T_i instead denotes the *minimum amount of time* separating the release of two subsequent jobs of τ_i (Mok 1983).

[9]While not particularly well advertised, NVIDIA does provide source code for its embedded systems such as the Jetson TX2. NVIDIA also eventually released source code for a Linux driver supporting its most recent discrete GPUs, but this unfortunately arrived while this dissertation was being written. This timing meant we did not use this open-source driver in our research, though it will certainly be a good resource for future work.

Second, each task requires a *deadline*, usually denoted D. A task's deadline falls into one of the following categories, ranging from least to most restrictive: *arbitrary*, *constrained*, or *implicit*. As the name implies, an *arbitrary* deadline can be anything, regardless of a task's period. If task τ_i has a *constrained* deadline, then $D_i \leq T_i$. Finally, if a task has an *implicit* deadline then $D_i = T_i$. The parameter D_i itself is a *relative deadline*, meaning that every job J_i of τ_i must complete D_i time units from its release. Sometimes, however, we also use the term *absolute deadline* to refer to a concrete point in time by which a specific given job must complete. In order for a task system to be *temporally correct*, all jobs must complete by their absolute deadlines.

The task parameter that is likely the most difficult to obtain is its *cost*, usually denoted C. Task τ_i's cost, C_i, indicates the maximum amount of time that a job of τ_i requires in order to complete execution. The term *worst-case execution time* (WCET) is often used to refer to a task's cost. It is often useful to compute a task's *utilization*, u, using the task's cost and period: $u_i = C_i/T_i$. One can think of task τ_i's utilization as the maximum fraction of a processor's time that is required in order to complete all jobs of τ_i by their deadlines. This can be combined into the *total utilization*, U, of an entire task system: $U = \sum_{i=1}^{n} u_i$.

Once we have established each task's deadline, cost, and period, we can define a real-time task system: $\tau = \{\tau_1, ..., \tau_n\}$, where $\tau_i = \{\phi_i, T_i, C_i, D_i\}$. The one parameter we have yet to describe is each task's *phase*, denoted ϕ_i, indicating the absolute time at which the first job of τ_i is released. We may simplify task-system definitions when certain parameters are not important. For example, we can omit ϕ if all tasks start releasing jobs at the same time, or we can omit D if a task uses implicit deadlines.

Scheduling and schedulability. When executing a real-time task system in practice, a *scheduling algorithm* is responsible for determining when each job (or set of jobs) is allowed to execute on available hardware. Scheduling algorithms can be broadly classified into two groups: *fixed priority* and *dynamic priority*. A fixed-priority scheduler makes decisions based on task parameters: period, deadline, arbitrary user-specified priorities, *etc.* For example, the well-known *rate-monotonic* scheduler always prioritizes jobs from tasks with shorter periods over jobs from tasks with longer periods (Liu and Layland 1973). Dynamic-priority scheduling, by contrast, prioritizes jobs based on factors that may change at runtime. The *earliest-deadline-first* (EDF) algorithm is perhaps the most prominent example of a dynamic-priority scheduler: as its name implies, it prioritizes whichever job that has the nearest deadline, regardless of periods, costs, or other factors.

The goal when defining task systems is to be able to make formal guarantees about timing constraints. Part of this process is known as *schedulability analysis*: given a task system, a platform, and a scheduling algorithm, a *schedulability test* determines whether all of the jobs in the task system are guaranteed to be able to meet their deadlines. A particularly famous example within real-time literature is the schedulability test for the EDF scheduler on a platform with a single processor. In this case, the task system with implicit deadlines is schedulable if and only if $U \leq 1$. While this dissertation pertains more to tangential hardware- and software-related issues than it does to formal schedulability analysis, the terminology is still essential in order to understand the existing body of traditional real-time work and subsequent attempts to integrate GPU usage into this legacy.

2.6 Prior Work

For convenience, we break our list of relevant GPU-management research into different categories, based on how each work prevents or reduces the harmful side effects of GPU sharing.

2.6.1 Enforcement of Exclusive GPU Access

Early GPU-management approaches sought to prevent the issue of unpredictable interference by preventing concurrent GPU access in the first place. Perhaps the first example in real-time literature comes from Kato, Lakshmanan, Rajkumar and Ishikawa (2011), who modified Linux's *Nouveau* GPU driver code[10] to provide additional prioritization and scheduling mechanisms for graphical GPU-using applications. Shortly thereafter, the same group published a follow-up work providing similar control over general-purpose GPU applications, albeit requiring a now-defunct third-party CUDA implementation (Kato, Lakshmanan, Kumar, Kelkar, Ishikawa and Rajkumar 2011).

Other groups extended these exclusive-access models to apply to multiple GPUs, including Verner, Schuster, Silberstein and Mendelson (2012), Verner, Mendelson and Schuster (2014*b*) and Verner, Mendelson and Schuster (2014*a*). However, like the preceding work by Kato *et al.*, these subsequent publications from Verner *et al.* still treated GPU access as a *scheduling* problem. Elliott, Ward and Anderson (2013) took a different approach. While still supporting multi-GPU systems, Elliott *et al.* managed GPUs using *locking*,

[10]The nouveau driver is an open-source driver for NVIDIA GPUs, and is based on extensive research and reverse-engineering efforts. While reasonably stable for graphical applications running on older GPUs, nouveau is often unstable for newer GPUs and notably lacks CUDA support.

treating GPUs as resources that individual tasks must request (exclusive) access to. In the work of Elliott *et al.*, the key question is not about when to run GPU tasks, but instead a question of how to prioritize tasks' GPU-access requests.

While these approaches are all effective at preventing unpredictable interference, they do not adequately address the needs of embedded systems like autonomous vehicles. Notwithstanding big-budget research prototypes, size, weight, power, and monetary cost constraints all serve to discourage the use of multi-GPU supercomputers in vehicles intended for widespread public use. Instead, based on current devices aimed at autonomous vehicles, such systems are likely to share a one or two, possibly less-powerful, GPUs among all applications.[11] In this resource-constrained case, requiring exclusive access to the GPU can result in unacceptable *capacity loss*, when an application that does not require full use of the GPU nonetheless prevents other applications from running. Granted, this limitation is known, even within research that still proposes solutions mandating exclusive GPU access. For example, Kim, Patel, Wang and Rajkumar (2017) developed a *server-based* system that grants tasks GPU access while loosening some of the strict lock-based requirements in prior work such as Elliott *et al.* This, however, serves to reduce symptoms without addressing the underlying cause of capacity loss: limiting GPU access to a single task in the first place.

2.6.2 Spatially Partitioning GPU Hardware

Perhaps one of the most obvious solutions to the capacity-loss problem on shared GPUs is *spatial partitioning*, where a single GPU's hardware components are divided into two or more partitions, such that accessing resources in one partition causes little or no unpredictable interference with the other partitions. Recall that the GPU's SMs (or CUs, on AMD GPUs) contain the bulk of the computational resources, as well as independent L1 caches, so forcing independent applications to execute on non-overlapping sets of CUs or SMs has been a popular method for partitioning GPUs. Several prior publications instrument GPU kernel code in order to implement SM partitioning on NVIDIA GPUs despite a lack of dedicated hardware support. To our knowledge, the first example of such an approach comes from Janzén, Black-Schaffer and Hugo (2016), who proposed a simple but apparently effective implementation. Janzén *et al.*'s approach modifies the start of GPU kernel code to make each thread block check its SM assignment at runtime. If

[11] For example, NVIDIA's Drive AGX Pegasus system, which NVIDIA claims is "built for Level 4 and Level 5 autonomous systems," contains a single discrete GPU and a single integrated GPU: `https://developer.nvidia.com/drive/drive-hyperion#section-2`.

any thread block detects that it is executing on a SM that is *not* part of the kernel's partition, then the block exits immediately. This approach is not without drawbacks: it depends on the ability to modify kernel source code, it requires launching a larger number of blocks than otherwise necessary to account for the number of badly-assigned blocks that exit immediately, and a tiny amount of code may unavoidably execute outside of a kernel's intended partition (for each block to test its SM assignment).

Despite the shortcomings, Janzén *et al.*'s approach has remained popular in real-time literature due to its conceptual simplicity and the fact that it runs on off-the-shelf NVIDIA hardware. Saha (2018) applied the technique in combination with other resource-allocation algorithms in order to assign SMs to tasks on both discrete and embedded GPUs. Jain, Baek, Wang and Rajkumar (2019) not only adapted the Janzén *et al.* approach to partition SMs in certain discrete NVIDIA GPUs, but also successfully partitioned the L2 cache and DRAM banks in order to reduce possible hardware interference even further.

Not all real-time GPU hardware-partitioning research uses Janzén *et al.*'s technique. In Chapter 4, we document our experiences with partitioning CUs using the hardware support available in AMD GPUs. Others have investigated GPU cache partitioning using simulation to avoid the implementation difficulties using real closed-source hardware and software (Wang, Li and Yang 2016, Liang, Li and Xie 2017).

2.6.3 Temporally Partitioning GPU Access

GPU access can also be managed using *temporal partitioning*: dividing GPU access into blocks of time. In a sense, the exclusive locking or scheduling techniques from Section 2.6.1 are a coarse-grained form of temporal partitioning. The issue that continues to hamper temporal partitioning attempts is the fact that GPUs have traditionally been non-preemptive: after starting to execute, a GPU kernel does not relinquish resources until it completes. The assumption that GPUs are non-preemptive is slightly outdated: In 2016, NVIDIA first added hardware support for preempting kernels at an instruction-level granularity to its then-new "Pascal" GPU architecture. However, to this day there is no documented user-facing API for controlling this behavior. This has not prevented Capodieci, Cavicchioli, Bertogna and Paramakuru (2018) from making use of this functionality, albeit in collaboration with NVIDIA. In their 2018 work, Capodieci *et al.* implemented preemptive scheduling on NVIDIA's embedded-oriented DRIVE platform, including an EDF scheduler for GPU operations. While impressive, it is difficult for other research groups to expand on

this effort without assistance from NVIDIA insiders, and a fully open-source real-time GPU-management system making use of fine-grained hardware preemption has yet to be produced.

Absent hardware support, several papers from the real-time and high-performance computing communities propose subdividing large, long-running GPU kernels into multiple smaller chunks in order to reduce the amount of non-preemptive blocking. Basaran and Kang (2012) were among the first to attempt this for real-time GPU usage, subdividing larger kernels into what they termed "subkernels." In practice, Basaran and Kang implemented their "subkernels" by dividing a single kernel-launch command into multiple commands, each of which launch a smaller number of thread blocks. The overhead due to the larger number of underlying kernel launches quickly increased, becoming as high as 340% for certain kernels. For most of the kernels investigated by Basaran and Kang, overheads only remained in negligible ranges when subdivided into four or fewer subkernels. While any amount of kernel subdivision can certainly reduce worst-case nonpreemptive blocking times, Basaran and Kang's description of their 2012 approach as "fine-grained preemption" seems outdated by modern standards.

Fine-grained or not, subdividing large GPU kernels into multiple, "smaller" kernel launches has remained popular in real-time literature. While focused more on throughput than traditional real-time requirements, Zhong and He (2013) extended the tactics used by Basaran and Kang, allowing subkernels originating from entirely different kernels to execute on the GPU concurrently. A few years later, Zhou, Tong and Liu (2015) implemented another system using an approach highly similar to Basaran and Kang (2012), though Zhou *et al.* provided a more streamlined, lower-overhead implementation, wrapping parts of the CUDA API and involving support from the operating system. In another application of the same idea targeted towards embedded GPUs, Lee and Al Faruque (2016) proposed a system for dynamically adjusting subkernel sizes at runtime.

Despite the popularity of subdividing kernels based on thread blocks, it is not the only approach towards greater control over GPU kernels in time. Chen, Zhao, Shen and Zhou (2017) instead proposed a *voluntary preemption* system, in which kernel source code transformations enable exiting a kernel early if a preemption is needed, and resuming it in a subsequent launch. In a rare example of a non-NVIDIA-based work, Lee, Roh and Seo (2018) implemented a system for transactional kernels using the OpenCL API (Khronos Group 2020), allowing terminating the execution of low-priority non-real-time kernels without undesired side effects.

2.6.4 Mitigating Memory Interference

In addition to contending for compute resources, GPUs have limited bandwidth for transferring data or accessing memory (to differing degrees, depending on whether the GPU is discrete or integrated). Particular attention has been paid to managing memory contention on integrated GPUs, where GPU activity can potentially interfere with the CPU's ability to access memory. Some of this work expects GPU workloads to adhere to the *PREM* (*PRedictable Execution Model*) structure, which requires programs to be divided into memory-intensive and compute-intensive phases for the sake of scheduling (Pellizzoni, Betti, Bak, Yao, Criswell, Caccamo and Kegley 2011). Forsberg, Marongiu and Benini (2017) proposed a system to extend PREM-like protections to include GPU tasks. They did so by combining the PREM model with concepts from an older CPU-focused work called *MemGuard* (Yun, Yao, Pellizzoni, Caccamo and Sha 2013), using hardware performance counters to detect and throttle memory-heavy tasks.

Capodieci, Cavicchioli, Valente and Bertogna (2017) also applied the PREM model to tasks running on an embedded, integrated GPU. In this work, Capodieci *et al.* designed a server that arbitrates when memory-intensive portions of GPU tasks are allowed to execute, in order to reduce interfere with CPU tasks' memory accesses.

Other works favor a model more akin to locking, where applications must notify the operating system about ongoing GPU work, either to acquire a lock for a discrete GPU's data-transfer hardware (Elliott, Ward and Anderson 2013) or to enable a bandwidth-throttling mechanism when needed (Ali and Yun 2018). As with kernel execution, prior work typically considers copying data to discrete-GPU memory to be *non-preemptive*, meaning that a long data-transfer operation may block other work. Some prior works (Basaran and Kang 2012) (Kato, Lakshmanan, Kumar, Kelkar, Ishikawa and Rajkumar 2011) (Kato, McThrow, Maltzahn and Brandt 2012) (Zhou, Tong and Liu 2015) attempted to alleviate this issue by subdividing long memory transfers into multiple, shorter chunks—a similar approach to the one discussed in Section 2.6.3 for dealing with long-running kernels.

2.6.5 Exposing Black-Box GPU Behavior

Some publications seek to determine what may happen if one forgoes additional GPU management, sharing a GPU between multiple tasks under the "default" hardware and software behavior. Put another way,

this type of research seeks to establish an accurate model of GPU behavior, as discussed in Section 1. Much of our own work falls into this category, discussed in Chapters 3 and 4 of this dissertation.

Several papers from outside of the real-time community provide useful information about GPU-internal behavior. In 2013, Peres used reverse engineering to infer power-management controls for NVIDIA GPUs. Later that year, Fujii, Azumi, Nishio and Kato (2013) reverse engineered and modified the microcontroller firmware for the NVIDIA GTX 480 to improve response times for some workloads. Mei and Chu (2016) used microbenchmark experiments to infer the cache and memory layout for several generations of NVIDIA GPUs. Later, Jia *et al.* published successive papers using microbenchmarking to infer instruction-set details about both NVIDIA's Volta (Jia, Maggioni, Staiger and Scarpazza 2018) and Turing (Jia, Maggioni, Smith and Scarpazza 2019) GPU architectures.[12]

While potentially applicable in a real-time setting, the papers mentioned in the previous paragraph all focuses on aspects of performance other than *predictable timing*. Apart from our own contributions, Capodieci *et al.* (2018), mentioned already in Section 2.6.3, serves as a rich source of information about scheduling behavior of some embedded NVIDIA GPUs due to the authors' collaboration with NVIDIA. In Section 2.6.2, we mentioned Jain *et al.* (2019) in the context of GPU partitioning, though the work deserves another mention here. Using thorough black-box reverse engineering efforts, Jain *et al.* discovered and published the formulas used by certain NVIDIA GPUs to map memory addresses to DRAM banks and L2 cache lines.

To our knowledge, the most recent real-time paper focused on GPU scheduling internals comes from Olmedo, Capodieci, Martínez, Marongiu and Bertogna (2020). In their paper, Olmedo *et al.* provide new details regarding the assignment of thread blocks to SMs in NVIDIA GPUs. Olmedo *et al.* specifically refute some simplistic round-robin scheduling models assumed in a handful of prior papers, demonstrating the importance of accurate information when developing GPU scheduling models.

2.6.6 Orienting Our Contributions Within the Wider Field

In summary, prior real-time research has explored a variety of tactics for GPU management: mediating GPU access, temporal and spatial partitioning, and revealing details of GPU behavior. The study of GPUs in real-time systems is still quite active, and our own research is necessarily interwoven with several of these

[12] We list and discuss NVIDIA GPU architectures in Section 3.1.1.

topics, as other GPU-oriented research groups conducted their studies at the same time as our own. To the point, our work was among the first to focus on producing information sufficient for building a model of GPU behavior, being conducted prior to the other real-time-oriented papers listed in Section 2.6.5. Our second major contribution lies in our efforts to expand the available research platforms to include a more open option: AMD GPUs. This is particularly unique in our field, as all but one paper we mention in Section 2.6 uses NVIDIA GPUs. Finally, while several of the papers in this section contain case studies of varying size and complexity, few attempt to address the particular difficulties that arise with modern neural-network applications, despite such applications easily being the largest motivation for inclusion of GPUs in safety-critical systems. Our own work, discussed in Chapter 5, attempts to illuminate the challenges posed by these applications.

2.7 Summary of this Chapter

We began this chapter with an overview of general-purpose programming for both NVIDIA and AMD GPUs, and their respective CUDA and HIP programming frameworks. Next, in Section 2.2, we discussed the high-level architecture of several different GPUs, including discrete NVIDIA and AMD GPUs along with an integrated NVIDIA GPU. We concluded the section by discussing how the demand for GPUs continues to increase apace with continual increases to GPUs' computational capabilities—providing a moving target for the still-immature field of real-time GPU management.

We next switched topics to the main applications that we care about accelerating using GPUs: AI, and especially neural networks, in the service of computer vision. In Sections 2.4 and 2.5, we provided some key background information used throughout the rest of this dissertation: an overview of operating systems and some basic definitions and notation used in real-time literature. Finally, we discussed a variety of prior real-time GPU-management publications, roughly divided into five categories based on the how each approach addresses the challenge associated with GPU sharing.

CHAPTER 3: NVIDIA GPUS[1]

Our goal, as mentioned, is modeling GPU behavior: discovering sets of rules with which we can predict GPU behavior. Models of GPU behavior can include as much or as little detail as desired, with differing results: the predictive power of insufficiently detailed models is likely unreliable or lacking, but a model with too much detail may be impractical or impossible to apply to complex workloads.

The spectrum of possible models is directly reflected in the various approaches taken in prior real-time GPU-management proposals. In an example of an overly sophisticated model, one prior paper claimed to produce upper bounds on the response time for an approximation of GPU code by exhaustively exploring all possible interactions between every concurrent GPU thread (Berezovskyi, Bletsas and Petters 2013). Not only was this approach computationally intensive due to relying on integer linear programming, its timing predictions were incredibly pessimistic, and it could not be applied to anything remotely resembling a real workload: the paper only reports results for sequences of at most five *instructions* running on fewer than ten warps on a single SM. Rather than attempting to improve the performance or applicability of such exhaustive models, many approaches follow the opposite path: modeling the GPU as an exclusively accessed black box, handling one request at a time. Notable examples include locking (Elliott, Ward and Anderson 2013) or server-based GPU management techniques (Kim, Patel, Wang and Rajkumar 2017), which, as discussed, are limited by their inability to avoid losing some of the GPU's computational capacity.

[1]Contents of this chapter previously appeared in the following papers:

Otterness, Miller, Yang, Anderson and Smith (2016). *GPU Sharing for Image Processing in Embedded Real-Time Systems*. Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT).

Otterness, Yang, Rust, Park, Anderson, Smith, Berg and Wang (2017). *An Evaluation of the NVIDIA TX1 for Supporting Real-Time Computer-Vision Workloads*. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).

Otterness, Yang, Amert, Anderson and Smith (2017). *Inferring the Scheduling Policies of an Embedded CUDA GPU*. Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT).

Amert, Otterness, Anderson and Smith (2017). *GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed*. IEEE Real-Time Systems Symposium (RTSS).

Yang, Otterness, Amert, Bakita, Anderson and Smith (2018). *Avoiding Pitfalls when using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems*. Euromicro Conference on Real-Time Systems (ECRTS).

Our own work on NVIDIA GPUs attempts to develop, or at least enable, a model that falls into neither modeling extreme. We do not wish to lose computational capacity or risk safety by ignoring GPU internals, but we also understand that real GPU software, with millions of instructions and concurrent threads, is far too complex to properly analyze by exhaustively simulating clock cycles on a thread-by-thread basis. Instead our goal is pragmatism: autonomous-vehicle manufacturers have *already started* putting NVIDIA GPUs into safety-critical systems[2]—so what can we do to produce models capable of making them safer and more predictable without sacrificing efficiency?

Among other topics, the primary sections of this chapter give three different answers to this question. In Section 3.2, we show evidence that basic GPU sharing, termed *co-scheduling*, is capable of improving throughput without disproportionate impacts on predictability, for computer-vision microbenchmarks running on an embedded GPU. Using tools described in Section 3.3, Section 3.4 outlines underlying rules governing NVIDIA's GPU sharing, along with the techniques we used to discover them. These rules allow us to produce predictive models about GPU behavior, without requiring additional modifications to NVIDIA's default software. Finally, Section 3.5 lists several pitfalls that can lead to particularly devastating timing behavior for co-scheduled kernels on NVIDIA GPUs, and how they may be avoided.

3.1 Overview of NVIDIA GPUs

The details we provided in Sections 2.1, 2.1.2 and 2.2 are foundational for understanding GPU programming in general, but there are some remaining details that apply specifically to NVIDIA GPUs.

3.1.1 NVIDIA GPU Architectures

There are many important architectural distinctions even between GPUs from a single manufacturer. While GPU architecture changes frequently impact raw performance, power usage, memory capacity, *etc.*, we care mostly about architectural differences that affect timing and co-scheduling. We provide the scheduling-related details we consider important in the following list of NVIDIA GPU architectures, starting with the first architecture in which co-scheduling is even relevant:

[2]Tesla's "Model S" electric car is one such example: `https://electrek.co/2017/05/22/tesla-nvidia-supercomputer-self-driving-autopilot/`

- **The Kepler GPU architecture.** NVIDIA's *Kepler* architecture was first released in 2012. The Kepler architecture is a notable milestone for GPU-sharing research such as ours. With Kepler, NVIDIA introduced what it terms *Hyper-Q*: the ability to launch kernels from multiple streams concurrently. Prior to Kepler GPUs, even separate streams within a single process ultimately had their contents multiplexed onto a single hardware queue (NVIDIA Corporation 2014). Kepler-architecture discrete GPUs include models such as the GTX 770 and GTX 780, though a particular integrated GPU is a greater landmark in our own research: the Kepler-architecture Tegra K1 GPU is used in the *Jetson TK1*, the first of NVIDIA's embedded-oriented Jetson systems.

- **The Maxwell GPU architecture.** NVIDIA's *Maxwell* architecture was first released in 2014. The Maxwell architecture introduced fewer notable changes to high-level behavior, instead focusing on power efficiency in a bid to make further inroads in mobile and embedded domains (Smith and T. S. 2014). Maxwell GPUs include discrete models such as the GTX 970 and GTX 980, as well as integrated models such as the Tegra X1, used in the Jetson TX1 embedded platform.

- **The Pascal GPU architecture.** NVIDIA's *Pascal* GPU architecture was first released in 2016, and introduced several features that are significant to our work. Most importantly, Pascal-architecture GPUs were the first to support instruction-level *preemption*. As we shall see in Section 3.2, prior to Pascal, ongoing computations on NVIDIA GPUs could only be time-sliced at thread-block boundaries: long-running blocks could delay the GPU from switching to different tasks. Additionally, Pascal introduced support for *unified memory* in discrete GPUs: allowing automatic data transfers between CPU and GPU memory during kernel execution. Notable Pascal-architecture GPUs include the GTX 1070 and 1080, as well as the integrated Tegra X2, used in the embedded Jetson TX2 system.

- **The Volta GPU architecture.** NVIDIA's *Volta* architecture was released in 2017, and was ultimately only used in a small number of GPU models. Nonetheless, Volta is relevant to us due to its introduction of enhanced quality-of-service capabilities for NVIDIA's *multi-process service* (MPS), a tool facilitating GPU sharing that we discuss further in Section 3.1.2. The Volta architecture only appears in a small number of discrete GPUs, such as the Titan V, but it still makes an appearance in an embedded platform: the Jetson AGX Xavier.

- **The Turing GPU architecture.** NVIDIA's *Turing* architecture was first released in 2018. Unlike Volta, Turing GPUs tended to be more consumer-oriented, and the architecture introduced several features relevant to graphics, though nothing with an obvious application for real-time GPU sharing. To our knowledge, NVIDIA's Turing architecture does not appear in any embedded-oriented integrated GPU, though it is used in several discrete GPUs, such as the RTX 2070 or RTX 2080.

- **The Ampere GPU architecture.** NVIDIA's *Ampere* architecture was first released in 2020. One of the most interesting features introduced with Ampere hardware was NVIDIA's *multi-instance GPU* (MIG) system. MIG offers enhanced hardware partitioning, allowing a single Ampere GPU to be subdivided into several sub-GPUs (called *instances* in documentation), each of which uses separate compute units, L2 cache, and DRAM regions (NVIDIA Corporation 2020*c*). The introduction of MIG indicates that NVIDIA places increasingly high importance on GPU sharing, and a MIG-capable GPU would be a desirable platform for researching shared-hardware interference. Unfortunately, MIG is only available in top-end, expensive GPUs. To our knowledge, the only Ampere GPUs to support MIG are the A100 and A30, costing around 10,000 and 5,000 US dollars respectively. The Ampere architecture is also used, without MIG support, in a wide number of less expensive consumer GPUs, such as the RTX 3070 or RTX 3080. Additionally, the most recent Jetson platform, the Jetson AGX Orin, features an Ampere-architecture GPU.

- **The Hopper GPU architecture.** NVIDIA's *Hopper* architecture was first announced in 2022, and, at the time of writing, it still remains unclear whether any Hopper-architecture GPUs are even available for purchase by the general public (especially with the price speculation discussed in Chapter 2). Official publications indicate a continued emphasis on GPU sharing in Hopper GPUs, which support additional enhancements to MIG such as encrypted memory on a per-instance basis (NVIDIA Corporation 2022*d*). So far, the only Hopper-architecture GPU seems to be the high-end H100 model; NVIDIA's documentation indicates that other variants of the H100 exist (NVIDIA Corporation 2022*d*), but it is unclear at the time of writing if these other models are available for purchase, or whether they even go by the same name.

In summary, like our list in Section 2.2, the points we chose to highlight about each GPU architecture hint at a story. Certainly, each GPU generation continues to become more powerful than its predecessor, but

different themes emerge as GPUs become more established in new, non-graphical domains. For example, the improvements made with the Volta and Hopper architectures make it clear that these entire architectural revisions are targeted at producing GPUs for artificial-intelligence or data-processing workloads rather than graphics. At the same time, NVIDIA remains aware of the strong desire for greater amounts of GPU sharing, in progression mirroring research goals in the real-time field: the whitepaper describing the Kepler architecture (NVIDIA Corporation 2014) uses capacity loss as a direct motivation for its Hyper-Q hardware, while the whitepaper for the Volta architecture explicitly states that its enhanced MPS feature "reduces average latency and jitter" in a shared-GPU system (NVIDIA Corporation 2017, page 31). Eventually, this develops into the introduction of full partitioning support in some Ampere GPUs, specifically to address the hardware interference that remained possible when sharing a Volta GPU (NVIDIA Corporation 2020*b*, pages 44-45).

3.1.2 CUDA Details

A large portion of our work with NVIDIA GPUs, specifically that described in Section 3.5, deals with the CUDA API used for programming NVIDIA GPUs. CUDA as a whole is more accurately described as a *framework*, of which the API is only one component. In truth, *CUDA* entails a collection of software, encompassing the API with which programmers request GPU work, the compiler capable of converting C or C++ kernel code into GPU-compatible bytecode, a collection of userspace libraries required to support the API, and even the kernelspace driver responsible for communicating with the hardware. (We note that NVIDIA's official "CUDA" installer will install all of these components.)

CUDA's "driver" and "runtime" APIs. CUDA's API is actually divided into two layers: the higher-level *runtime* API with which most applications are developed, and the lower-level *driver* API, which is available for advanced users desiring more explicit control.

Despite the name, CUDA's "driver API" is not part of the operating system, and is accessible to ordinary userspace programs. It consists of a C-language interface for managing GPU code and memory, and in several cases exposes control over operations that cannot be directly managed using the runtime API alone, such as loading kernel code onto the GPU. Most application developers do not use the driver API directly, as it leads to CUDA programs that are more verbose and inconvenient to develop. Nonetheless, the runtime API is actually implemented on top of the driver API. The fact that the driver API exists below the runtime API and

exposes some additional controls has led a handful of prior GPU-management works to implement additional real-time controls by intercepting driver-API calls. For example, Zhou, Tong and Liu (2015), mentioned in Section 2.6, intercepted the CUDA driver API to implement their kernel subdivision in a manner transparent to higher-level code.

Most CUDA developers are likely to be far more familiar with the runtime API, which transparently handles many "boilerplate" aspects of GPU programming, such as managing connections to the GPU and loading kernel code. Our example from Chapter 2, Figure 2.2, uses the runtime API.

Our work focuses exclusively on CUDA's runtime API. While the driver API's lower-level access may sound better for real-time GPU management, for our purposes, the additional degree of control is largely superficial, while being significantly harder to use. For example, we have no need to create multiple CUDA contexts (discussed in the next paragraph) within a single Linux process, as this feature is mostly intended for applications using multiple GPUs. We also had no need to intercept or modify the behavior of the CUDA API, though our work does hint at cases where doing so could be useful. Finally, we sought to use benchmarks based on real-world applications as much as possible, where appearances of the driver API are quite rare.

The role of CUDA contexts. The CUDA userspace runtime libraries store state opaquely, in a structure known as the *CUDA context*. The CUDA driver API allows users to manually create new CUDA contexts and to change the current "primary" context that is used by runtime-API functionality. However, without manual intervention using the driver API, the CUDA runtime API will transparently create a single context when users first attempt to interact with the GPU. A CUDA context is associated with exactly one GPU, so in real-world applications manual context management is typically only used in order to access multiple GPUs from within a single process.

CUDA contexts, however, do play an important role in GPU scheduling. As we shall demonstrate with experiments in Section 3.2, CUDA only allows a single context to access the GPU at once, and switches between active contexts using *time slicing* (this is quite different from the behavior of AMD GPUs, discussed in Section 4.2). In order for "true" GPU sharing, in which separate kernels actually run on the GPU at the same time, the kernels must be launched from the same context. Ordinarily, it would impossible to share a single context between separate system processes, but NVIDIA provides a tool for working around this limitation.

Multi-process service. NVIDIA's *multi-process service* (MPS) is middleware produced by NVIDIA intended to reduce the capacity loss associated with GPU sharing (NVIDIA Corporation 2020*a*). MPS only supports the Linux operating system and discrete GPUs, and takes the form of a Linux service: a long-running, persistent process intended to run non-interactively. When launching a CUDA application, the CUDA runtime will automatically detect the presence of an active MPS instance on the system, and interact with MPS using standard inter-process communication (IPC) mechanisms.

On GPUs using Pascal and earlier architectures, MPS works by creating a single CUDA context and requiring client processes to request all GPU operations via IPC. This has the downside of increased overheads and lack of isolation between client virtual address spaces, *i.e.*, one MPS-connected client process can corrupt memory buffers used by another MPS client process (NVIDIA Corporation 2020*a*). In Volta and later architectures, additional hardware support allows MPS clients to maintain separate virtual address spaces, and to directly interact with GPU hardware, though it is not immediately clear from available documentation how NVIDIA allows this newer version of MPS to avoid the time-slicing behavior.

Unfortunately, MPS remains exclusively available for Linux, and additionally only supports discrete GPUs. We find the lack of availability on embedded platforms to be more of a limitation than being restricted to Linux, at least in real-time research. Unfortunately, the lack of MPS on embedded platforms, such as the "Jetson" systems, forces us to structure GPU-sharing workloads as a single Linux process in order to enable true concurrent GPU usage. We explore some of our efforts to conduct meaningful experiments despite this limitation in Section 3.3.2.

3.2 GPU Co-Scheduling

In this section, we revisit an old assumption: namely, that GPU co-scheduling must be avoided due to unpredictable interference effects.

3.2.1 What is GPU Co-Scheduling?

We use the term *GPU co-scehduling* to make a careful distinction: in our work, *co-scheduling* implies nothing more than the phrase's literal meaning: more than one application is scheduled on (*i.e.*, allowed to access) the GPU at a time. We employ this phrase in our work to merely indicate what we *allow*. This notion establishes two useful points of contrast to which we must draw attention. First, it sets our work apart from

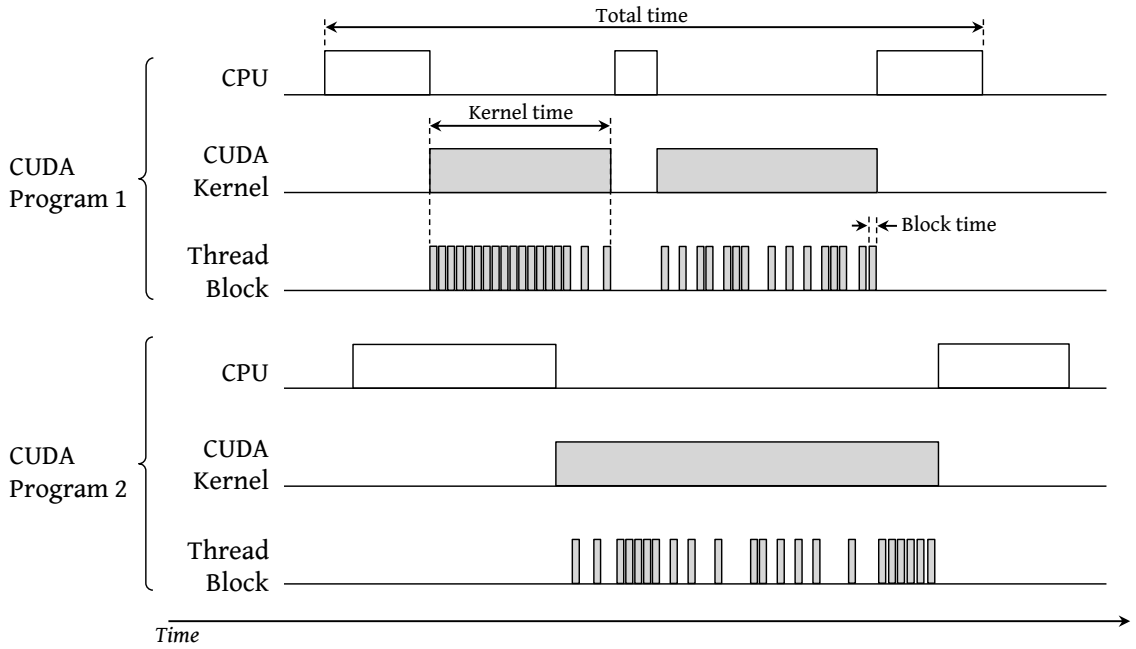


Figure 3.1: Simplified depiction of the relation between CPU, kernel, and block times in co-scheduled CUDA applications in the Kepler and Maxwell architectures.

prior approaches that intentionally *forbid* concurrent GPU access. Second, it allows us to make a careful distinction: *co-scheduling does not necessarily imply "true" concurrency*.

Consider Figure 3.1, which represents the activity that may be carried out by two hypothetical tasks that have been co-scheduled on a GPU. Even though this figure is based on a now-dated GPU architecture, it still provides a useful illustration of the hierarchy between a GPU-using application, the GPU kernels used by the application, and the thread blocks executed on behalf of the kernels. It also provides a visual definition for three different spans of time, to which we frequently refer throughout Section 3.2. Finally, Figure 3.1 presents a useful illustration of what we mean by distinguishing co-scheduling from "true concurrency."

Clearly, the two applications depicted in Figure 3.1 are running concurrently on separate cores in a multicore-CPU system. Additionally, we can see that they are co-scheduled on the GPU: for example, the second kernel launched by "CUDA Program 1" overlaps entirely with the execution of the single kernel launched by "CUDA Program 2." However, when we consider the actual thread blocks responsible for carrying out the GPU computations, we see that they do *not* execute concurrently; the thread blocks for the first program's kernels never overlap with the blocks for the second kernel (this behavior arises because two programs use separate CUDA contexts). We only consider GPU operations to be truly concurrent when the

GPU is actively carrying out computations on behalf of more than one independent task. Even so, a lack of true concurrency does not necessarily mean that co-scheduling is useless—as we shall see in the following experiments.

3.2.2 Co-Scheduling Experiments on Embedded Platforms

Our foray into embedded GPUs must begin by revisiting a key motivation behind all of our research, discussed in Chapter 1: autonomous vehicles. The size, weight, and power (*SWaP*) constraints present in autonomous vehicles, along with manufacturers' desires to reduce monetary cost, serve as downward pressure on the computational power available to carry out such vehicles' safety-critical processing. Under these constraints, autonomous vehicles clearly must rely on the presence of computational accelerators such as GPUs, but manufacturers (as well as eventual customers) are likely to balk at the size, power, and monetary costs of large server-grade systems with multiple GPUs (*e.g.*, those used by Elliott, Ward and Anderson (2013)).

GPU manufacturers such as NVIDIA certainly took notice of this need, as evidenced by the emergence of embedded platforms such as the Jetson TK1 (launched in late 2014) and Jetson TX1 (launched in early 2015). In fact, NVIDIA's marketing material at the time even referred to the Jetson TX1 as *the embedded platform for autonomous everything*.[3] While solving SWaP concerns, the shift to low-power embedded hardware platforms gives rise to new dilemmas for prior GPU-management approaches: when using a single, less-capable GPU, any waste of the GPU's capacity becomes untenable.

Experimental objectives. Despite the "embedded everything" slogan, there were few or no published studies (at least at the time of the TX1's release) that expressly evaluated the effectiveness of the TX1, or any other comparable energy-efficient embedded-GPU platform, in hosting safety-critical real-time workloads. This provides us with two opportunities: we can examine the impact of GPU co-scheduling while simulataneously conducting a high-level study of the embedded platforms' capabilities to host the computer-vision workloads essential to autonomous vehicles. Specifically, we seek to gauge the behavior of co-scheduled GPU workloads, and the extent to which the workloads experience shared-hardware interference due to uncontrolled co-scheduling.

[3]This no longer appears in NVIDIA's marketing, even for the far-more-capable Jetson AGX Orin, but remains visible in archived versions of NVIDIA's embedded-systems website: `https://web.archive.org/web/20170103045909/http://www.nvidia.com/object/embedded-systems.html`.

3.2.2.1 Observed Co-Scheduling Behavior on the Jetson TX1

We chose to study the Jetson TX1's behavior under unmanaged co-scheduling by executing one or more instances of certain microbenchmark programs on the system. Here, we consider one such set of experiments, and the effects of co-scheduling on the total times, kernel times, and block times depicted in Figure 3.1.

The stereo disparity microbenchmark. For this set of experiments, we used the *stereo disparity* (SD) microbenchmark as an exemplar of a computer-vision application. We adapted SD from NVIDIA's official CUDA samples, and the source code for our experiment is available online.[4] SD continually runs iterations in which it extracts 3D depth information from 2D images taken with a stereo camera. The inputs to SD are 640×533 color images corrseponding to the left and right frames from the camera, and the output is a 640×533 grayscale image, where the brightness of each pixel corresponds to the estimated depth in the 3D scene.

Experimental structure. Our experiment consisted of launching up to four concurrent instances of SD, each of which was allowed unrestricted access to the TX1's GPU. In our experiments, we launched each instance of SD as separate Linux processes, which we pinned to separate CPU cores (the Jetson TX1's ARM Cortex-A57 CPU has four CPU cores). We allowed each instance of the microbenchmark to run for ten minutes, during which we recorded timing measurements. We used the same pair of images for each instance of SD, and pre-loaded the input image data into memory before beginning measurements. We also disabled graphics on the system, and used an NVIDIA-provided script[5] to disable the GPU's dynamic frequency scaling. These experiments used CUDA version 8.0.

Impact of co-scheduling on SD's total time. As stated, this experiment was devised to evaluate how much of a benefit GPU co-scheduling *can* produce on the Jetson TX1. Each instance of SD executed as many iterations as possible for 10 minutes. To begin, we measured each iteration's total time (as defined in Figure 3.1), in order to capture all possible benefits due to co-scheduling.

Observation 3.1 *GPU co-scheduling can lead to reduced total time, compared to sequentially executing the co-scheduled tasks.*

[4] `https://github.com/yalue/PeriodicTaskReleaser`

[5] `https://github.com/yalue/PeriodicTaskReleaser/blob/master/Benchmark/TX-max_perf.sh`

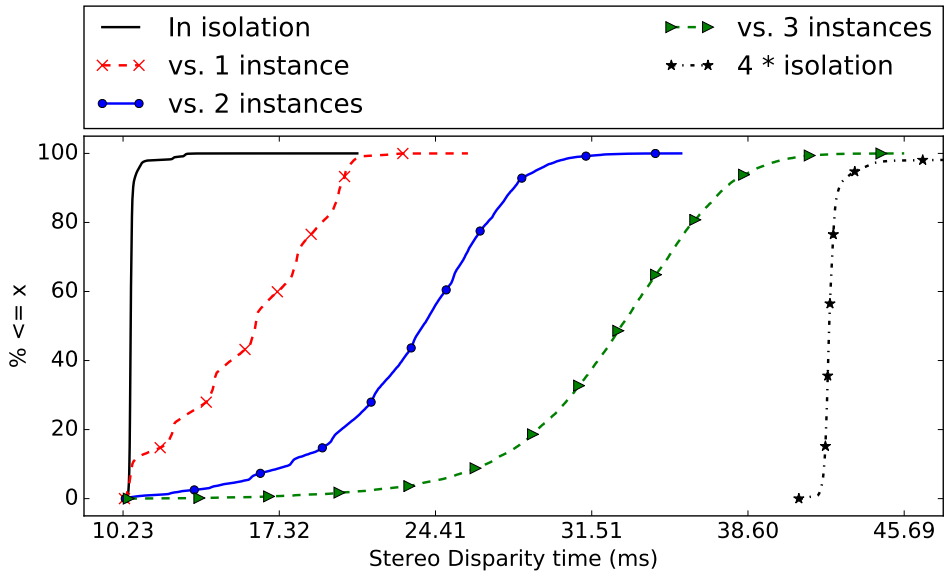


Figure 3.2: CDFs of total times for SD with up to four co-scheduled SD instances. The maximum value for the "4 * isolation" curve is 110 ms.

Observation 3.2 *Total times with GPU co-scheduling are still significantly longer than total times in isolation.*

These observations are supported by Figure 3.2, which plots the *cumulative distribution functions* (CDFs) of the recorded total times for a single instance of SD executing alongside a varying number of identical SD competitors.

The "In isolation" curve of Figure 3.2 corresponds to the case in which no competitors exist, and the subsequent curves correspond to cases where one, two, or three competitors exist. The "4 * isolation" curve is not a real measurement, but was obtained by scaling up the isolation curve by a factor of four. We include it to provide a rough estimate of the time necessary to complete four instances if all four benchmarks were forced to run sequentially.

In the co-scheduling case, the distribution of total times necessary for four instances to complete is represented by the "vs. 3 instances" curve, which includes measurements of the time a single instance takes to complete when running concurrently with three competitors. The difference in median times between these two curves leads to Observation 3.1: we can save approximately 10 milliseconds on average by co-scheduling. The measured worst-case execution time (WCET) shows an even greater benefit due to co-scheduling: approximately 70 milliseconds can be saved.

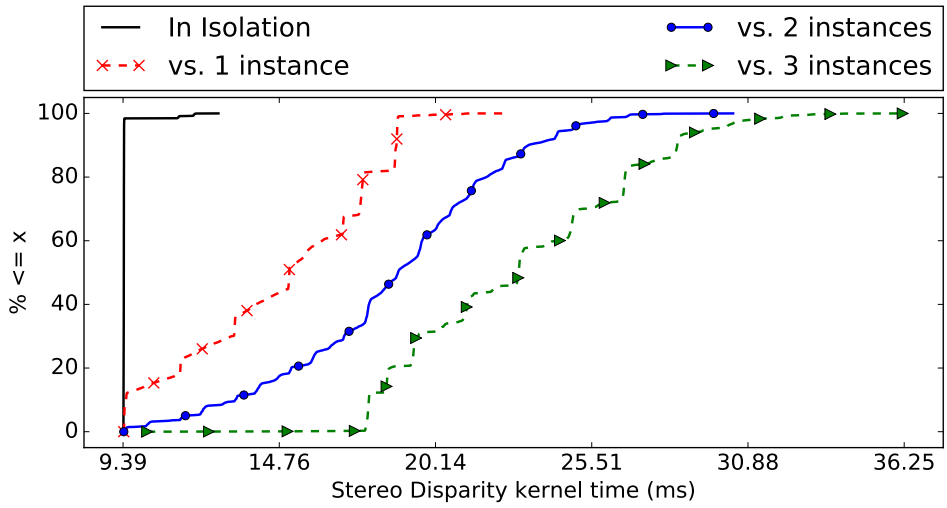


Figure 3.3: CDFs of kernel times with up to four co-scheduled SD instances.

On the other hand, as stated in Observation 3.2, total times appear to scale almost linearly on average, with the median total time increasing by about 7 milliseconds with each additional competing instance. This implies that there is still a significant amount of resource contention between co-scheduled workloads, but further experimentation is needed to determine exactly which resources are a source of contention.

Impact of co-scheduling on SD's kernel time. The results in the prior paragraphs establish that co-scheduling can be beneficial in some cases, but give little indication of the root cause for the increasing total times during co-scheduling. The next step in our examination involved measuring kernel times in the presence of co-scheduling. We recorded kernel times in our experiments using the `nvprof` utility, which NVIDIA provides as part of the default CUDA installation.

Observation 3.3 *Co-scheduling affects kernel times similarly to total times.*

Observation 3.4 *A majority of SD's total time is spent executing kernels.*

Observation 3.3 is supported by the kernel-time CDFs shown in Figure 3.3. Like total times, kernel times still expanded in the presence of co-scheduling, but not to the point where any benefit of co-scheduling no longer exists. The kernel-time CDFs are less distinct and more noisy than the total-time CDFs, which, as we shall see in subsequent paragraphs, is likely a result of block-scheduling behavior.

Comparing Figure 3.2 with Figure 3.3 allows us to infer Observation 3.4. This comparison is clearest when viewing SD in isolation: the "In isolation" curve in Figure 3.2 indicates that SD has a median total time of slightly over 10 ms, while the "In isolation" curve from Figure 3.3 shows that slightly over 9 ms of this

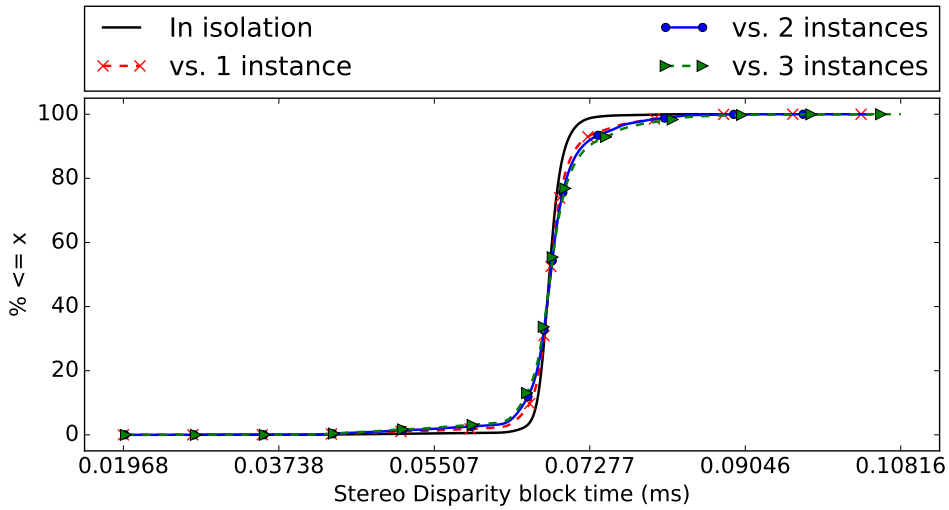


Figure 3.4: CDFs of block times with up to four co-scheduled SD instances.

time were spent executing kernels. Recall from Section 2.1.1 that this does *not* necessarily mean that SD only spends one millisecond executing on the CPU: kernels are launched asynchronously, enabling SD to carry out CPU computations while kernels are executing on the GPU.

Impact of co-scheduling on SD's block times. In the final step of our experiment, we measured the time of every thread block launched on behalf of SD's kernel (SD launches only a single kernel per iteration). Unfortunately, tools such as `nvprof` do not directly enable block-time measurements, meaning that we had to modify SD's kernel code to carry out this part of our experiments. Our modifications ended up being small: we instrumented one thread from each block to use inline assembly to read the GPU's `globaltimer` register, which is shared between all SMs and maintains a count of nanoseconds. The designated GPU thread records a timestamp both at the start and end of the kernel code, writing the times into a buffer in GPU memory. After the kernel completes, we copy the buffer of timestamps back to CPU memory, where we can compute block times and produce CDFs plots like we did for kernel and total time. Figure 3.4 shows these results.

Observation 3.5 *Block times are minimally affected by co-scheduling.*

Observation 3.5 is supported by the block-time CDFs given in Figure 3.4. This figure shows that block times are virtually unaffected by co-scheduling. Especially striking is the fact that the median block time, approximately 70 microseconds, was indistinguishable regardless of whether one or four instances of SD were running. The complete lack of interference between block times serves as a striking testament to the

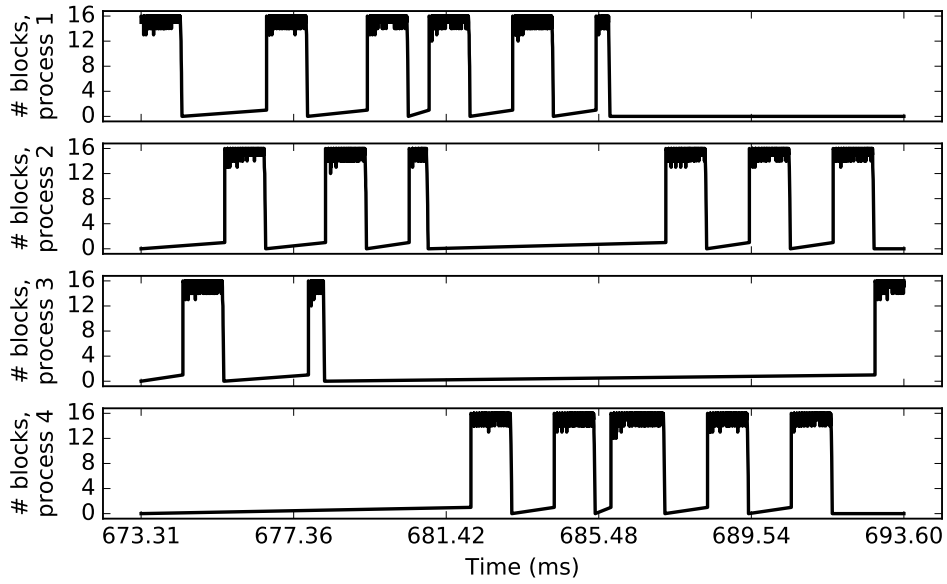


Figure 3.5: Timeline of when blocks from four co-scheduled SD instances ran on the Jetson TX1's GPU.

efficacy of NVIDIA's time-slicing approach between different CUDA contexts. Since block times do not change under co-scheduling, we can only conclude that the majority of the slowdown due to co-scheduling is due to inter-block scheduling behavior. We further verify this conclusion by visualizing our block-time data in a different way.

Using the block start and end times we already recorded in order to produce Figure 3.4, we are also able to generate Figure 3.5. Figure 3.5 shows a snippet of a timeline of the number of thread blocks being executed by each of four concurrent SD processes. We used `nvprof` to confirm that the timespan depicted in Figure 3.5 only includes a single kernel per process, rather than multiple kernels beginning and ending. From this figure, it is clear that the Maxwell-architecture GPU on the Jetson TX1 executes blocks on behalf of one context for a certain time span before switching to executing blocks from a different context.

Remarks on the TX1's performance. Our full suite of experiments included more than just the SD microbenchmark. For example, we also attempted co-scheduling various applications alongside the CaffeNet neural network, which implements the *AlexNet* (Krizhevsky, Sutskever and Hinton 2012) image-classification network using the *Caffe* (Jia *et al.* 2014) programming framework. While we do not include full results from our CaffeNet experiment here (its complexity and slower performance yielded few interesting insights about co-scheduling), we found that CaffeNet on the TX1 could only perform approximately 30 classifications per second, even when running in isolation. While this is likely adequate for hobbyist drones or self-driving R/C

cars, it would undoubtedly be inadequate in a truly safety-critical system such as an autonomous car, where image classification would only be one of many stages in multiple computer-vision pipelines.

3.2.2.2 Objections to Co-Scheduling: Well-Founded?

Co-scheduling in the manner of Section 3.2.2.1 should *not* be particularly susceptible to unpredictable interference, as NVIDIA GPUs already perform some variant of time-slicing to prevent true concurrency between separate CUDA contexts (*i.e.*, separate Linux processes, as MPS is unavailable on the TX1). The block-level time-slicing may certainly be a cause for concern in a system where one block may take a very long time, but, in later experiments, we verified that even this potential co-scheduling pitfall goes away with the Pascal GPU architecture, which switches from time slicing at the block granularity to finer-grained preemption.

On the other hand, our initial goal with co-scheduling was *not* to prove it had low interference—the entire point was to *increase hardware utilization*. While the results in Figure 3.2 show that co-scheduling is beneficial, Figures 3.4 and 3.5 indicate that the performance improvement is due to overlapping *CPU* activity (in addition to pontentially using internal GPU queueing hardware more efficiently); GPU computations are not being carried out at the same time. Can we do better by enabling "true" concurrency by switching to a single CUDA context? If so, will we still be able to establish some clear boundaries on when hardware contention may occur?

3.3 Developing a Microbenchmarking Framework for Investigating GPU-Sharing Behavior

Our experiments from Section 3.2 focused on the co-scheduling of multiple independent CUDA contexts, and concluded that this type of co-scheduling is unlikely to cause problems with respect to unpredictable interference, at the expense of making it impossible to fully claim the GPU's computational capacity. We next push our investigation farther by using a single CUDA context to enable true concurrency. Unfortunately, doing so may re-open the door to destructive shared-hardware interference, in addition to posing some software-engineering challenges. While MPS is available on discrete GPUs, we care the most about recouping lost capacity on embedded platforms where the only way to share a CUDA context is to also share a single system process.

It is at this point that we must reemphasize our goal of developing a model for NVIDIA GPU behavior. Assuming we can overcome the software-engineering hurdle of restructuring independent GPU applications into a single process, we still need a way to predict how they can interact. Our next set of experiments attempts to address this need. As is often the case with NVIDIA GPUs, we lack detailed information about internal details regarding GPU behavior, or even details about CUDA's runtime-library software, so we are forced to rely on black-box testing.

Black-box testing. *Black-box testing* refers to the practice of developing tests or experiments to evaluate a system for which internal details are unknown—contained in a "black box." We begin our experiments with only a limited set of presuppositions about NVIDIA GPU behavior. We can already safely assume that NVIDIA GPUs manage shared-GPU access via some hierarchy of queues, as the CUDA documentation guarantees that streams are FIFO queues. The question remains of how NVIDIA GPUs arbitrate between kernel-launch requests from multiple, independent, streams. CUDA documentation only notes that kernels from seprate streams *can* execute concurrently (NVIDIA Corporation 2022*a*, Section 10.5), which gives little information about whether kernels *will* execute concurrently, and if not, the order in which they run.

3.3.1 GPU Microbenchmarking Framework Requirements

Effective black-box experiments always seek to answer *specific questions*. Examples of such questions could be "Will the GPU reassign resources from a kernel that is already executing to a newly launched kernel?" or "Can an ongoing memory-transfer operation prevent a kernel from executing?" The more such questions we can answer, the better our understanding of GPU behavior will be.

A *microbenchmarking framework* allows answering questions like these in far less time. In its most fundamental form, a microbenchmarking framework is simply a piece of software that simplifies setting up microbenchmark experiments. To see why this can be useful, consider again the fact that black-box experiments are designed to answer specific questions—in our experience, specific questions almost always imply an experiment. For example, the previous paragraph asks "Can an ongoing memory-transfer operation prevent a kernel from executing?" While not *entirely* specific (there are many missing details, *e.g.*, kernel size, memory-transfer direction, *etc.*), this question certainly implies an experiment:

1. Launch a lengthy memory-transfer operation in one stream.

2. Immediately afterwards, launch a kernel from a separate stream.

3. Observe whether the kernel begins execution immediately, or whether it only begins execution after the memory transfer completes.

In this contrived example, the three experimental steps illustrate different operations that a microbenchmarking framework can carry out. For the first two steps, it could execute two microbenchmarks: one that invokes a lengthy memory-transfer operation of some arbitrary data, and another that launches some arbitrary kernel. The third step requires *making detailed observations*—an error-prone, nuanced task that becomes especially cumbersome when microbenchmarks use separate, ad-hoc code for making and recording measurements. If the framework handles as much of this step as possible, a large portion of measurement code becomes easier to reuse and scrutinize for errors.

We developed a GPU-specific microbenchmarking framework that addresses these needs among several others. The key software-engineering principles upon which we based our framework were:

- **Modularity**: Microbenchmarks are structured as interchangeable plugins, configured through a common interface and producing output in a standard format.
- **Reusability**: Asking most "questions" should usually require reconfiguring existing microbenchmarks rather than writing new ones.
- **Programmability**: It is possible to use additional programs to automatically configure a large number of experiments and process their results.

3.3.2 Framework Operation

Figure 3.6 gives an overview of the basic steps carried out by our microbenchmarking framework.

Step ❶: configuration. When launched, our microbenchmarking framework requires a JSON-format configuration file, marked with a ❶ in Figure 3.6. This configuration file specifies which microbenchmarks to run along with many other parameters. We provide Figure 3.7 as an example used by one of our actual experiments.

While several of the lines in Figure 3.7 only make sense in later steps, we already have sufficient background to explain a large portion of it. The bulk of the configuration consists of the list of microbenchmarks to run, listed in a JSON array starting with the line containing the `"benchmarks"` label. In the entries of this array, the microbenchmark to run is determined by the `"filename"` field, which gives a path

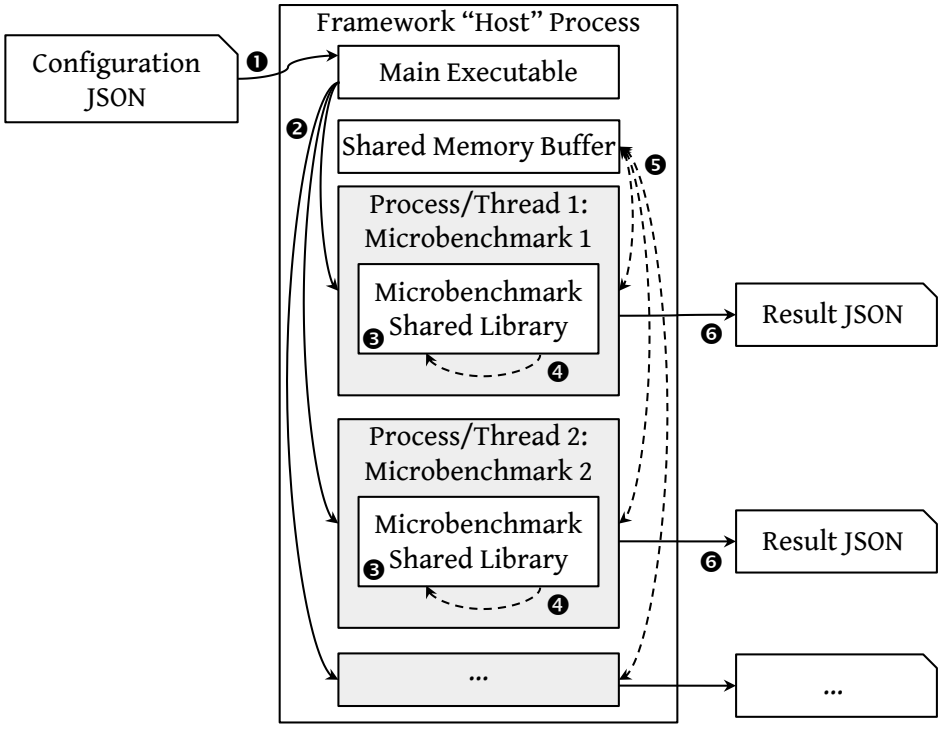


Figure 3.6: Flowchart of actions carried out by our microbenchmarking framework.

to a shared-library plugin containing the microbenchmark's code. In Figure 3.7, all three entries use the `timer_spin.so` microbenchmark: a synthetic microbenchmark that launches a single kernel in which each GPU thread executes a busy loop polling the `globaltimer` register until a certain number of nanoseconds have elapsed.

The configuration requires specifying the number of blocks and the number of threads per block for each microbenchmark. These settings are contained in the `"block_count"` and `"thread_ count"` fields, respectively, though microbenchmarks with less-flexible kernel code may ignore one or both of these settings. Next, the `"additional_info"` setting is an arbitrary JSON object containing microbenchmark-specific parameters; in Figure 3.7 it is simply a number containing the number of nanoseconds for which each iteration of the `timer_spin` instance is supposed to spin. Finally, we specify a `"label"` that will simply be copied to the output file for each microbenchmark. The remaining settings in Figure 3.7 are more easily understood in the context of later steps of the framework's execution.

Step ❷: spawn processes or threads. Our framework requires each microbenchmark to be capable of executing either as independent Linux processes or as separate threads within a single process. This

```
{
  "name": "Cutting ahead test",
  "max_iterations": 1,
  "max_time": 0,
  "use_processes": false,
  "benchmarks": [
    {
      "filename": "./bin/timer_spin.so",
      "log_name": "cutahead_1.json",
      "label": "Released first",
      "thread_count": 512,
      "block_count": 7,
      "additional_info": 1000000000
    },
    {
      "filename": "./bin/timer_spin.so",
      "log_name": "cutahead_2.json",
      "label": "Released second",
      "thread_count": 1024,
      "block_count": 2,
      "additional_info": 500000000,
      "release_time": 0.25
    },
    {
      "filename": "./bin/timer_spin.so",
      "log_name": "cutahead_3.json",
      "label": "Released 3rd, could cut ahead",
      "thread_count": 256,
      "block_count": 1,
      "additional_info": 500000000,
      "release_time": 0.5
    }
  ]
}
```

Figure 3.7: An example JSON configuration file for launching three `timer_spin` microbenchmarks.

behavior is determined by the "use_processes¨ configuration option seen near the top of Figure 3.7, and is reflected in Figure 3.6 by Step ❷.

When first envisioning this framework, one of our key goals was to *contrast the scheduling behavior of multiple CUDA contexts vs. multiple CUDA streams within a single context*. As we also need to support NVIDIA's embedded platforms, we are unable to rely on MPS. Instead, if we wish to examine intra-CUDA-context scheduling behavior, we are forced to run all of our microbenchmarks from a single Linux process. This puts some restrictions on our microbenchmarks: they must not rely on global state, and they must launch all GPU operations from separate CUDA streams.[6]

In fact, it may be more helpful think of separate microbenchmarks as *instances* of a microbenchmark. These instances may execute identical code and even share an address space (when running as threads rather than processes), but each instance can be configured with different settings, and must be able to operate independently.

Step ❸: load and initialize plugins. It is only in the context of a child process or thread that our framework actually attempts to load each microbenchmark plugin's shared library, indicated by step ❸ in Figure 3.6. In a purely thread-based model, it would likely be equally effective to load the shared libraries prior to spawning threads, but in order to support child processes, we must avoid initializing a CUDA context from within our host process. The problem is that CUDA forbids copying contexts between processes. If the parent process issues a `fork` system call to spawn a child process after initializing a CUDA context, then the operating system will copy the parent's CUDA context to the newly created child—violating the "no copying" rule and likely resulting in errors or crashes. This limitation means that we must only load and run our microbenchmark shared-library plugins within child processes, and for simplicity the framework follows the same procedure even when configured to use multiple threads rather than processes.

Step ❹: execute microbenchmark iterations. We use Step ❹ in Figure 3.6 to indicate the repeated iterations executed by each microbenchmark. In practice, the framework divides a single iteration into three "phases" corresponding to the typical execution pattern of GPU-accelerated applications: transfer data to the GPU, process the data, and copy results back to the CPU (see steps 3(a) through 4 listed in Section 2.1.1). The shared-library plugins for each microbenchmark expose these phases as three functions:

[6]For example, if any microbenchmark issues operations to the NULL stream (described in Section 2.1.2) when running in a thread, then those operations will block kernel launches from *all other microbenchmarks* until completion. We discuss the NULL stream's behavior more in Section 3.5.

1. `copy_in`: The framework begins each iteration begins by calling each plugin's `copy_in` function, prompting microbenchmarks to transfer any input data held in CPU memory to the GPU. Microbenchmarks also may use this function to reset any bookkeeping data from prior iterations, in preparation for new measurements.

2. `execute`: Next, the framework invokes the microbenchmark's `execute` function, which is expected to launch any kernels (and CPU computations in some cases) and wait for the kernels to complete. Each microbenchmark is responsible for conducting its own finer-grained timing measurements during this function.

3. `copy_out`: At the end of each iteration, the framework calls the microbenchmark's `copy_ out` function, during which the microbenchmark is expected to copy resulting data from the GPU to the CPU. In addition, it is during the `copy_out` function that the framework requires each microbenchmark to report detailed timing information (*e.g.*, individual block times) gathered during its most recent execution.

Naturally, not all GPU-using applications can divide their execution neatly among these three phases. For example, microbenchmarks requiring both CPU and GPU computations may need to intersperse kernel launches with data-copy operations. This does not pose a problem for our framework; the three phases are primarily used for rough time estimations, which we can choose to use or ignore as we see fit in later analysis. The only hard requirement is that the microbenchmark must complete all execution prior to returning timing information from the `copy_out` function.

Step ❺: optionally synchronize between iterations. After parsing the configuration and spawning the processes or threads responsible for running the microbenchmarks, the host process is typically uninvolved in actual execution. However, occasionally we may wish to *synchronize* all of the microbenchmarks prior to each iteration, *e.g.*, to cause more interference by forcing more overlap in their execution. The point at which this occurs is annotated as Step ❺ in Figure 3.6. In order to support this, the main process of the framework allocates a shared-memory buffer through which the microbenchmark processes or threads may carry out spin-based barrier synchronization, based on the approach developed by Mellor-Crummey and Scott (1991).

Step ❻: record output files. Timing information from each microbenchmark is recorded to a separate JSON-format output file (this is specified by the `"log_name"` lines visible in Figure 3.7). As mentioned in

```
{
  "scenario_name": "Cutting ahead test",
  "benchmark_name": "Timer Spin",
  "label": "Released second",
  "release_time": 0.25,
  "times": [
    {
      "copy_in_times": [0.340182176, 0.340182432],
      "execute_times": [0.340182528, 1.590260352],
      "copy_out_times": [1.590260832, 1.590398464]
    },
    {
      "kernel_name": "GPUSpin",
      "block_count": 2,
      "thread_count": 1024,
      "cuda_launch_times": [
        0.340182816,
        0.340254688,
        1.590259712
      ],
      "block_times": [
        1.090255109, 1.590273405,
        1.090255109, 1.590273405
      ],
      "block_smids": [
        0,
        1
      ]
    }
  ]
}
```


Figure 3.8: An slightly abbreviated JSON result file, produced by the the second of the three microbenchmarks configured in Figure 3.7.

Step ❹, one of the responsibilities of each microbenchmark plugin is to populate a data structure with timing information during its `copy_out` function. Common framework code serializes this data into the JSON format and writes it to the log file.

Figure 3.8 shows an abbreviated example of one of these output files, resulting from running the configuration file shown in Figure 3.7 on a real Jetson TX2 system. To produce Figure 3.8, we only modified the resulting JSON file to have a more human-readable layout, and to remove lines containing irrelevant bookkeeping. For every iteration, the output file contains the CPU times at the start and end of the three stages described in Step ❹, in addition to per-kernel timing measurements made by the microbenchmark plugin. (The microbenchmark instance that produced Figure 3.8 only ran for a single iteration; more iterations would simply append more entries to the `"times"` array.) The output file also includes the block and thread counts

used by the kernels, as well as the list of the start and end timestamps for each thread block and a list of the SM IDs to which each block was assigned. A kernel's `"cuda_launch_times"` entry contains three CPU timestamps: prior to launching the kernel, immediately after the kernel-launch invocation ends (we use this to detect whether the kernel launch was blocked on the CPU), and immediately after the return of the `cudaStreamSynchronize` call indicating the kernel's completion.

Summary of microbenchmarking framework capabilities. Using only different configuration files, our microbenchmark framework is capable of:

- Running an arbitrary number of microbenchmarks
- in either one or multiple CUDA contexts,
- with configurable GPU-resource requirements,
- starting at arbitrary offsets in time,
- for an arbitrary number of iterations or amount of time,
- while recording detailed per-microbenchmark timing information for each iteration.

Note that this does not cover the full list of settings we support, not to mention a large number of varied microbenchmarks with their own specific settings. Furthermore, our choice to use the JSON format for the framework's configuration enables automating sets of experiments with arbitrary complexity: nearly all modern programming and scripting languages have mature support for parsing or producing JSON data, allowing us to write scripts capable of generating configurations and processing results.

3.4 NVIDIA GPUs' Intra-Context GPU-Sharing Behavior

With a suitable microbenchmarking framework, we can return to the topic of how one designs black-box experiments in order to build a model of GPU scheduling. At the start of Section 3.3, we stated our desire to build an understanding of GPU co-scheduling within a single context: official documentation tells us that kernels launched from different streams *can* run concurrently, but we wish to know when they *will* run concurrently. In line with the black-box approach discussed at the start of Section 3.3.2, our goal is to ask *specific questions* about GPU sharing until we can distill a set of consistent rules with which we can predict yet-to-be-observed behavior: a model.

3.4.1 Initial Foray Into Black-Box Experiments: Can CUDA Kernels "Cut Ahead?"

We start our investigation with an important question about whether NVIDIA GPUs can reorder pending work in order to maximize hardware usage. Obviously, the title of this subsection already states the question in brief, but the specific formulation, implying the requisite black-box experiment, requires introducing another concept: GPU *occupancy*.

Understanding GPU occupancy. Occupancy is a term frequently used in GPU literature to refer to the fundamental constraints on the number of blocks that may execute per SM, limited by factors such as the number of threads per block, the number of registers required, or the amount of shared memory used. NVIDIA's official definition of occupancy is "*the ratio of active warps to the maximum number of warps supported on a multiprocessor [SM] of the GPU*" (NVIDIA Corporation 2022*c*).

If this is still unclear, consider a simple example. As discussed in Section 2.2, each SM on a GPU only supports up to 2,048 concurrent GPU threads.[7] If a kernel is launched with an arbitrary number of 768-thread blocks, each SM on the GPU is only able to support a maximum of two concurrent blocks (together totalling 1,536 threads), because the SM's 2,048-thread capacity is insufficient for a third block of 768 threads. Assuming threads are the only limiting factor, this kernel using 768-thread blocks achieves an occupancy of $1536/2048 = 0.75$. If a similar kernel were to use 512-thread blocks instead, it would achieve the highest possible occupancy of 1.0: each SM could fill its entire 2,048-thread capacity with four 512-thread blocks.

In our research, we rarely (if ever) discuss occupancy ratios, and NVIDIA mostly uses the concept when providing guidance on how to optimize GPU-using software. Our focus is not on developing the GPU-using software, but examining the extent to which capacity loss occurs in existing software, and whether it can be avoided. Therefore, even though we do not use occupancy ratios, we still often adopt "occupancy" phraseology, usually to explain *why* capacity loss may occur.

Posing a specific question. With an understanding of occupancy and the limits on per-SM resource usage, we can return to the topic at hand and finally pose a specific question:

[7] For convenience, we choose to focus on the number of threads per SM as the limiting factor on occupancy. We did carry out many experiments focusing on other resources (*e.g.*, shared memory), and observed that the GPU behaves the same (with respect to queueing, cutting ahead, *etc.*), regardless of which occupancy metric was a limiting factor. We choose to only report results using threads simply because the thread limit applies to all GPU architectures, including even to AMD GPUs, whereas limits on per-SM registers and shared memory change from GPU to GPU and can be harder to manipulate during experiments.

Imagine that two kernels, A and B, are launched from different streams, with kernel A being launched slightly before B. Meanwhile, preexisting work is occupying most, but not all, of the resources on the GPU. If there is insufficient capacity for a block from A to start running, but sufficient capacity for a block from B, will the GPU allow B to start executing? In other words, will B be allowed to "cut ahead" of A?

As we would hope, our question is now detailed enough to imply an experiment:

1. Ensure that the GPU resources are sufficiently occupied so that every SM only has sufficient capacity for a "small" thread block.
2. Launch kernel A, using "large" thread blocks that cannot start executing immediately.
3. Using a separate stream, launch kernel B using "small" thread blocks that could start executing immediately if the GPU allows cutting ahead.
4. Observe whether blocks of B are allowed to start executing immediately, or whether they must wait until after A starts executing.

If blocks from B start executing before blocks from A, then we conclusively know that the GPU allowed B to cut ahead. If not, we can conclude that, at least *in this situation*, the GPU does not allow cutting ahead.

Setting up a black-box experiment. We can use our microbenchmarking framework to carry out steps one through four listed above. In fact, we have already provided the exact configuration file required to carry out these steps: Figure 3.7. Returning to Figure 3.7 with a new understanding of the experiment it was designed to conduct allows us to move beyond the definition of each configuration line, and into the *intent* behind them.

The configuration launches three microbenchmark instances, each using the same `timer_spin` plugin. The `timer_spin` microbenchmark is appropriate, since we only care about occupying GPU resources and not specific computations. The configuration in Figure 3.7 is intended to run on the Jetson TX2, which has exactly two SMs, and the first microbenchmark instance it requests (with a "Released first" label) launches seven blocks of 512 threads. Since four blocks of 512 threads occupy an entire SM, we know that these seven blocks will entirely occupy one SM, and only leave 512 threads' worth of available capacity on the TX2's second SM. This satisfies the conditions required by step one of our experiment.

Second, Figure 3.7 launches another `timer_spin` instance, this time requesting two blocks of 1,024 threads. The order in which the framework launches microbenchmark instances is *not* determined by their ordering in the `"benchmarks"` list in the configuration file (even though they appear in the correct order in Figure 3.7), but instead by their `"release_time"` setting: a floating-point number of seconds the child process or thread sleeps before carrying out the first microbenchmark iteration.[8] As established by the previous paragraph, the GPU will only have 512 threads' worth of capacity remaining, so neither of these two 1,024-thread blocks will be able to start executing until after one or two blocks of the first kernel complete. This satisfies the conditions required by step two of our experiment.

Third, Figure 3.7 requests the framework to launch a final `timer_spin` instance, this time requiring only a single block of 256 threads. Clearly, if allowed to cut ahead, this block could occupy part of the remaining 512-thread capacity and begin execution. However, we configured its release time to be a full 0.25 seconds after the second kernel's release. (Note: the first kernel is configured to spin on the GPU for a full second, so it will not have completed execution even by the time the third kernel is launched.) This satisfies the conditions required by step three of our experiment.

Observing the experiment's results. The fourth and final step in our "cutting ahead" black-box experiment is to observe the microbenchmarks' actual behavior. After the framework completes execution, we can observe the results using a script we developed that consumes the output JSON files produced by the framework (such as Figure 3.8), and visualizes the assignment of each thread block to each SM on a timeline. Figure 3.9 contains the plot we obtained from this process.

In Figure 3.9, each thread block is represented by a shaded rectangle. Rectangles are labeled with the block index and name of the kernel they are executing, and are shaded according to the stream to which the kernel was submitted (in Figure 3.9 this is less relevant, as all three kernels were submitted to separate streams). Each rectangle's height is determined by the number of threads in the block, and its width is determined by how long it executed: the left and right edges of a rectangle correspond to the times at which the block began and ended execution. The vertical axis of the plot is subdivided into regions corresponding to each of the GPU's SMs (the TX2 only has two SMs), and the rectangle for each thread block is placed in the

[8] Enforcing order in this way simplifies structuring our microbenchmarks as independent processes or threads. This is less of a limitation than it may seem at first glance. Transient scheduling "jitter" is never the subject of our experiments, and if, for some reason, Linux's scheduler mis-orders the releases despite a full quarter-second difference in sleep times, our visualization scripts would make the mishap easy to detect.

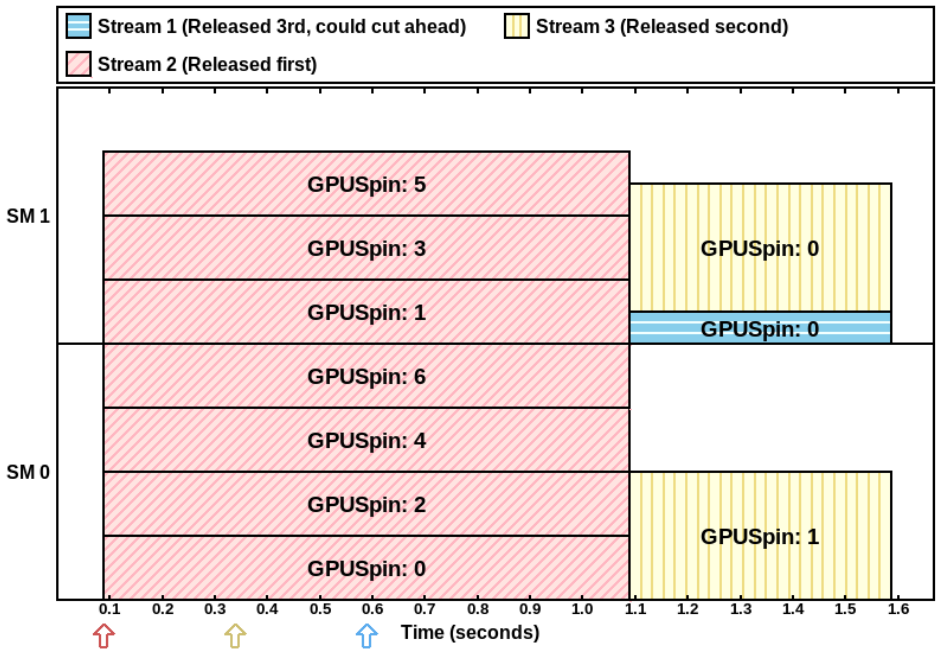


Figure 3.9: Scheduling of blocks on the Jetson TX2 in our "cutting ahead" microbenchmark experiment.

section corresponding to the SM on which it was executed. Finally, the colored arrows below the plot mark the times at which each kernel was launched.

Observation 3.6 *CUDA kernels do* not *cut ahead.*

Observation 3.6 is clearly consistent with the behavior shown in Figure 3.9: the seven blocks of the first kernel leave 512 threads available, but the third kernel (shaded in blue) does not start executing before the second kernel (shown in yellow), even though it was released slightly before the 0.6-second mark (the blue arrow). Naturally, Figure 3.9 is only one situation, so we also conducted many variants of this experiment with different block sizes, kernel orderings, *etc.*, but all of our experiments behaved consistently: CUDA kernels did not cut ahead.[9]

Given the simplicity of the experiment and its consistency across multiple variations, it is reasonable to assume that our conclusion about cutting ahead is correct. At the same time, it is not possible to make the leap from a single rule to a model without many more experiments: we only have a model once we can move from answering questions to making predictions.

[9]Lest any readers happen to think that this behavior is obvious and our experiments are unnecessary: we shall see an interesting contrast when discussing AMD's behavior in Chapter 4.

3.4.2 NVIDIA GPU Scheduling Rules

Recall the driving question behind all of these experiments: resolve the ambiguity in NVIDIA's documentation by figuring out when kernels submitted to separate streams *will* execute concurrently. Answering our initial "cutting-ahead" question is part of the solution—it demonstrated a situation when kernels *will not* execute concurrently despite being launched from separate streams and having sufficient computational resources. In order to progress beyond basic observations, we must discover underlying mechanisms, capable of explaining the lack of cutting ahead as well as being consistent with all other scheduling behavior we happen to observe.

Rather than explaining every single black-box experiment in detail as we did in Section 3.4.1, we choose here instead to start with our final result: the queueing rules for CUDA kernels launched within a single context. After explaining the rules, we give the result of corroborating black-box experiments, which exhibited the behavior on real hardware.

Intra-context kernel scheduling rules. For our set of rules, the GPU scheduler consists of one FIFO *primary queue*[10] per CUDA context, and one FIFO queue per CUDA stream.[11] This layout is depicted in Figure 3.10, which we explain in detail after presenting the rules below.

According to our observations of simple workloads being submitted from a single CUDA context, the following rules dictate the order in which kernels execute on the GPU, and whether two or more kernels will execute on the GPU concurrently:

A. A CUDA kernel is inserted into the primary queue when it arrives at the head of its stream.

B. A CUDA kernel can begin execution on the GPU if both of the following are true:

   B1. The kernel is at the head of the primary queue.

   B2. Sufficient GPU resources are available for at least one block of the kernel.

C. A CUDA kernel is dequeued from the head of the primary queue if all of its remaining blocks have either completed execution *or are currently executing*.

D.

[10] Subsequent work from Olmedo *et al.* (2020), indicates that our notion of a "primary queue" does not necessarily correspond to a separate queue of operations in GPU hardware. Instead, Olmedo *et al.* state that operations from separate streams are arbitrated by a mechanism called a *stream scheduler*. However, the ultimate outcome of this is that operations from separate streams behave exactly as described by our rules, so we continue to use our notion of a primary queue—this should not change a model of queueing behavior.

[11] This structure becomes more complex if more hardware or CUDA features (*e.g.*, copies or stream priorities) are considered.

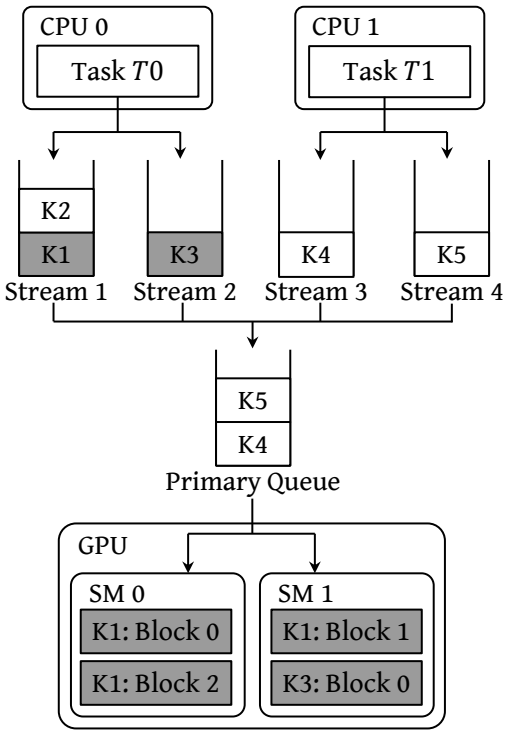


Figure 3.10: Flow of CUDA kernels through streams and the "primary queue."

A CUDA kernel is dequeued from the head of its stream if all of its blocks have completed execution.

In this paper, we only show experimental results supporting these rules that we obtained using a Jetson TX2, but we validated our conclusions on a wide variety of platforms. We found this set of rules to be consistent across all of the integrated and discrete NVIDIA GPUs we tested, including GPUs using the Maxwell, Pascal, and Volta architectures.

Explanation of GPU scheduling rules. Rules A and D restate the property of streams given in Section 2.1: operations submitted to a single stream always complete in FIFO order. Rules A and B1 imply that kernels submitted from multiple streams will run on the GPU in the same order that they arrived at the heads of their streams. (Rule B1 is also what prevented the "cutting ahead" we tested in Section 3.4.1.) Rule C is the rule that allows concurrent execution of multiple kernels on the GPU. In particular, the clause stating that a kernel is removed from the head of the primary queue if it has no remaining incomplete or unassigned blocks means that a second kernel can reach the head of the primary queue while the previous kernel is still executing. Lastly, Rule B2 determines whether a kernel at the head of the primary queue can begin execution. Figure 3.10, which we next describe in detail, serves as a visual example of these rules' applications.

Queueing rules: an example. In Figure 3.10, two concurrent CPU threads, $T0$ and $T1$ share a single CUDA context. (We label these as "tasks" rather than "threads" in the figure to avoid the ambiguity with GPU threads.) $T0$ and $T1$ create two streams each. In total, these two tasks submit five kernels in order, labeled $K1$ through $K5$. Each kernel may have multiple blocks, so kernel $K1$'s i^{th} block is labeled $K1{:}i$, and so on. ($K3$ only has a single block: $K3{:}0$.) In this example, all blocks of K1 and K3 (with shaded boxes) are currently assigned to the GPU. K1 and K3 have therefore been removed from the primary queue (Rule C), but are still present at the heads of their streams. Kernels K4 and K5 are at the heads of their streams, so they have been added to the primary queue (Rule A). Even so, neither is able to begin executing because K5 is not at the head of the primary queue (Rule B1), and insufficient GPU resources exist for a block of K4 (Rule B2). When K1 completes, it will be dequeued from the head of its stream (Rule D), and K2 will reach the head of its stream and be added to the primary queue (Rule A).

3.4.2.1 Corroborating Evidence for Rules B1, B2, and C.

The first of our microbenchmark experiments simply demonstrate that co-scheduling can occur when multiple kernels are submitted from different streams in a single CUDA context, and that kernels become eligible to run as soon as sufficient resources are available. These experiments only required submitting one kernel per stream, so the per-stream processing given by Rules A and D is trivial in these cases. Results of this first set of experiments are represented in Figures 3.11 and 3.12.

Of these first two experiments, Figure 3.11 represents the simplest, optimal co-scheduling situation in which we released Kernels 1 and 2 at time $t = 0s$ and Kernels 3 and 4 at time $t = 0.25s$. Each kernel was launched in a separate stream, configured to run for the same amount of time, and required two blocks of 1,024 threads. The kernels that were released first, 1 and 2, were co-scheduled due to Rule B2—each kernel only required half of the available thread resources. This meant that whichever kernel came first was fully assigned to SMs and dequeued from the primary queue. Kernels 3 and 4 could not commence execution until one of the first two kernels completed, freeing thread resources.

The second experiment, depicted in Figure 3.12, illustrates the greedy behavior implied by Rule C. Kernel 1 was released at time $t = 0s$ and required executing 18 blocks of 512 threads, which exceeded the GPU's capacity. Kernel 2, requiring fewer threads, was released at time $t = 0.25s$, but the scheduler did not allow it to execute until Kernel 1 had no blocks left to assign to the GPU. In accordance with Rule C, Kernel

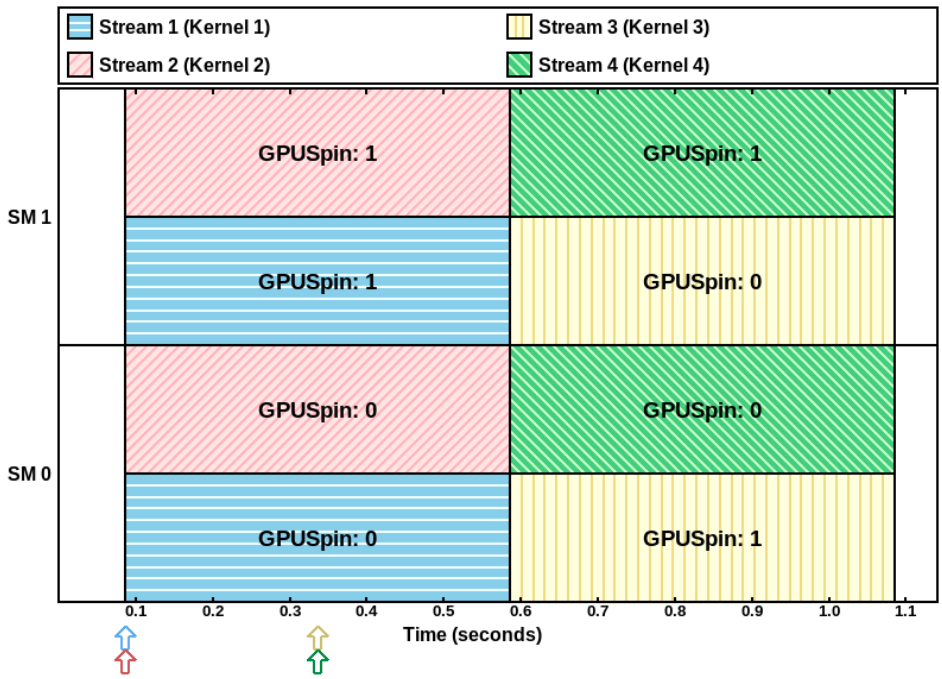


Figure 3.11: Basic concurrent kernel execution using multiple streams.

2 was able to reach the head of the primary queue and begin executing while the final block two blocks of the first kernel were still completing.

3.4.2.2 Corroborating Evidence for Rules A and D

Our set of experiments in Section 3.4.2.1 supported our observations about the ordering of kernels between multiple streams, but did not include situations that can occur when multiple kernels are submitted to a single stream. Our next set of tests illustrates the rules pertaining to intra- and inter-stream ordering of kernels, and therefore focuses on the additional constraints given in Rules A and D. Situations arising due to these rules are illustrated in Figures 3.13 and 3.14.

Figure 3.13 contains an example of how kernels within a single stream are executed in FIFO order. In this figure, Kernels 2 and 3 (shaded pink) were issued to a single stream, and, in accordance with Rules A and D, Kernel 3 did not begin execution until after Kernel 2 completed. Furthermore, Kernel 2 required too many resources to execute concurrently with Kernel 1 (shaded blue), even though Kernel 1 was issued in a different stream. This is in line with earlier observations, but it still serves as an illustration where a kernel with very low resource requirements is blocked not only by a predecessor in its own stream, but also transitively by another kernel from a different stream.

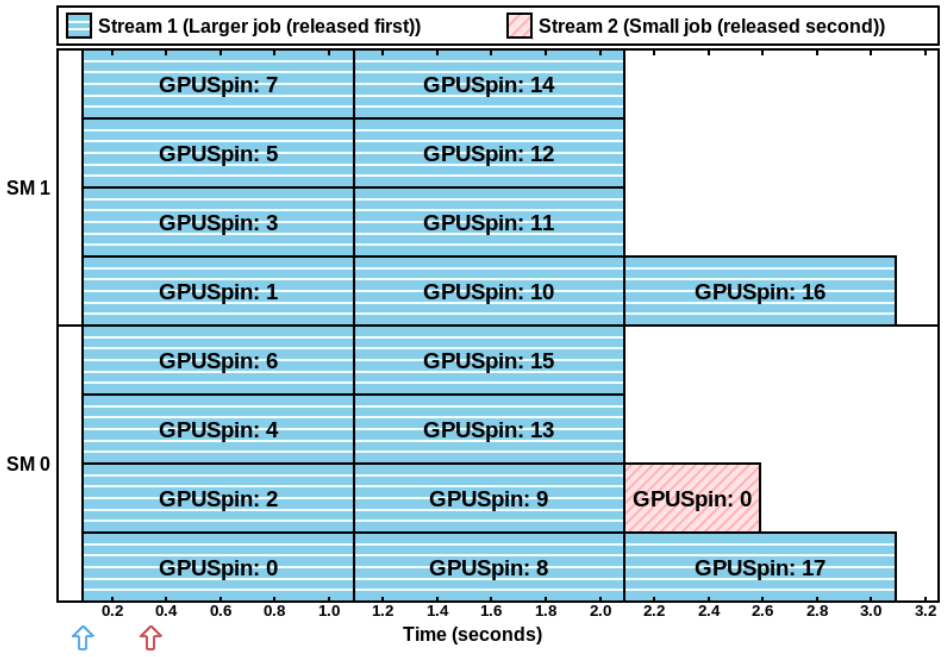


Figure 3.12: "Greedy" scheduling behavior.

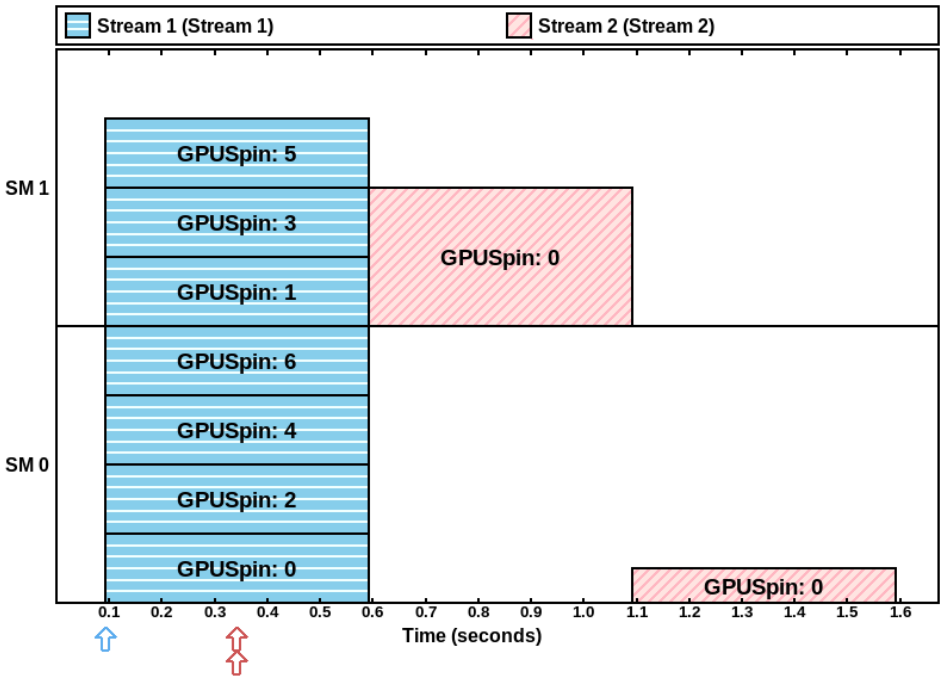


Figure 3.13: FIFO ordering within a single stream.

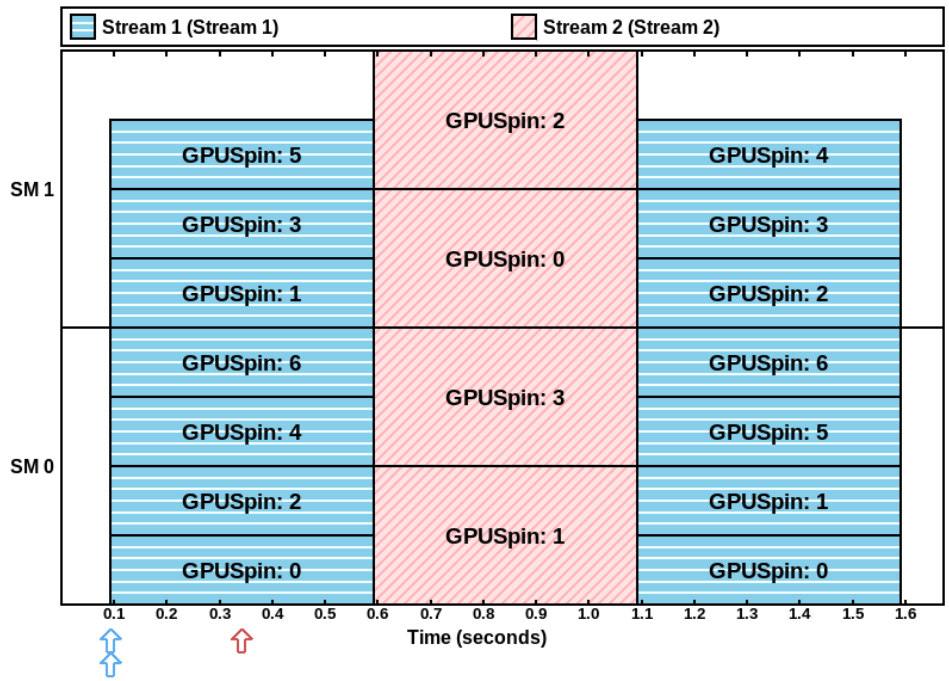


Figure 3.14: FIFO ordering within the primary queue.

We provide Figure 3.14 as a second illustration of Rules A and D. Unlike in Figure 3.13, the kernels in Figure 3.14 were executed in a different order from that in which they were issued. Kernels 1 and 2 (shaded blue) were issued back-to-back at time $t = 0s$ into the same stream, and Kernel 3 (shaded pink) was issued into a separate stream at time $t = 0.25s$. Even though Kernel 2 was issued earlier, Kernel 3 executed before Kernel 2 because Rule D prevented Kernel 2 from reaching the head of its stream until Kernel 1 completed. Kernel 3, on the other hand, reached the head of its stream and entered the primary queue as soon as it was submitted.

3.4.2.3 The Interplay Between Scheduling Rules and Resource Constraints

Having presented (some of) our corroborating evidence for all of the scheduling rules given in Section 3.4.2, we still cannot neglect the amount of complexity packed into Rule B2, which requires "sufficient GPU resources." Only Rules B2 and D are capable of preventing concurrency, but of these two, Rule B2 *appears* to rely on a factor outside the programmer's control: the hardware's computational capacity. If the GPU is fully occupied, there is indeed little reason for a programmer to pay further attention to Rule B2. However, fully occupied hardware, by definition, is not experiencing *capacity loss*, the original motivation for our GPU-sharing investigation.

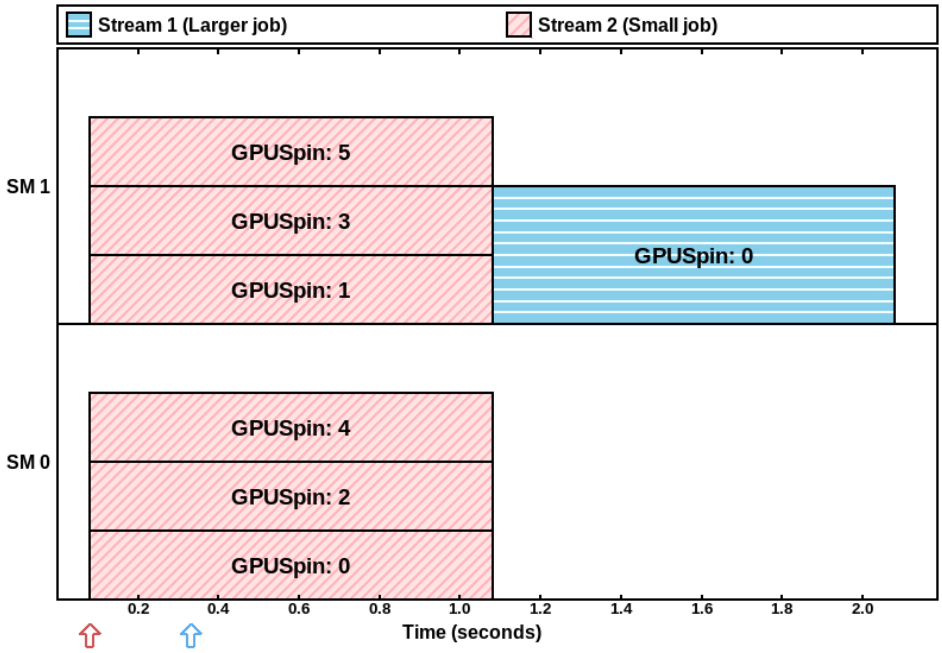


Figure 3.15: A kernel-launch ordering preventing kernel concurrency.

When considering scenarios where not all GPU hardware is occupied, we can demonstrate situations where Rule B2 plays a pivotal role in determining kernels' response times. We illustrate one such example by contrasting two highly similar experiments.

We present the first of the two related experiments in Figure 3.15. In this figure, the first kernel to be launched required 6 blocks of 512 threads. Since nothing else was currently executing, the GPU scheduler evenly distributed the six blocks across the TX2's two SMs,[12] leaving only 512 unassigned threads remaining on each SM. This meant that when the second kernel, requiring a single block of 1,024 threads, was launched at time $t = 0.25s$, it had to wait because neither SM could hold 1,024 threads.

Figure 3.16 contains the same two kernels as Figure 3.15, but with the two kernels released in the opposite order. Here, the single 1,024-thread block was assigned to SM 0, and the GPU scheduler distributed K1's six blocks to fill up all remaining thread resources—nearly halving the overall response time for the two kernels together. While this may not be a surprising result, it illustrates a situation in which reordering kernels could improve GPU utilization and reduce overall execution time. Without perfect clairvoyance, it is likely impossible to automatically reorder kernels to ensure maximal occupancy, but this still serves to

[12]For a detailed discussion about how NVIDIA GPUs assign blocks to SMs, see Olmedo *et al.* (2020).

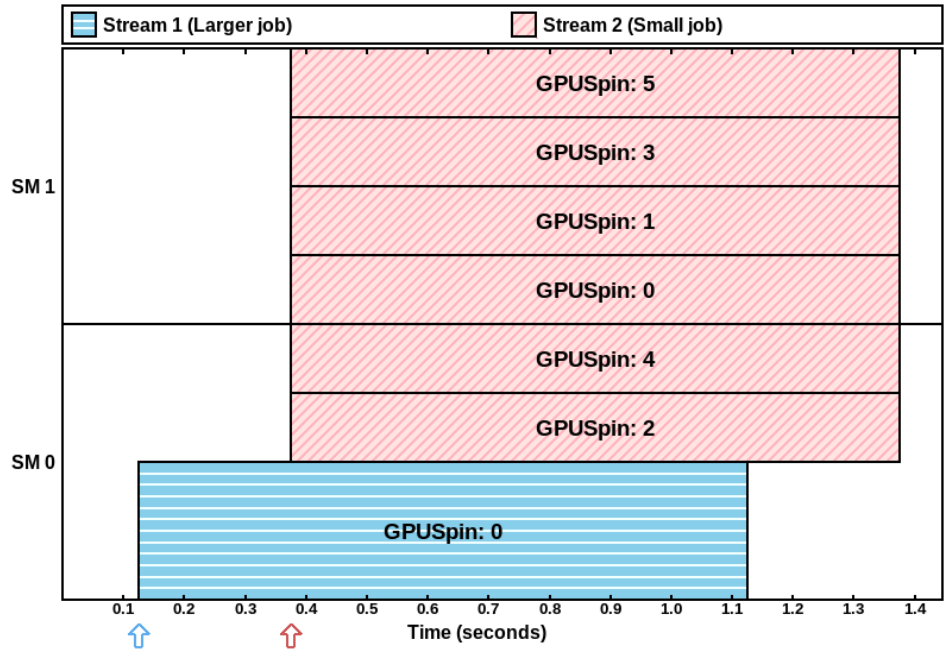


Figure 3.16: A kernel-launch ordering allowing kernel concurrency.

illustrate the deceptive depth of Rule B2—a slight change in the timing between two independent streams can have an outsized impact on the GPU's resource allocations.

3.4.3 Towards a Model of a Shared CUDA Context

The scheduling rules in this section generally only apply in two situations: when a kernel is first launched, and when a kernel's last few blocks begin completing, freeing GPU resources. This is due to Rule C—the kernel remains at the head of the primary queue, preventing other kernels from starting to execute ("cutting ahead"), so long as it still has more blocks to dispatch. In other words, long-running kernels that launch many blocks will almost certainly become the sole running kernel on the GPU at some point in their execution.

This means that the longest-running kernels are unlikely to either cause or experience unpredictable hardware contention during the bulk of their execution times, as even without external intervention they are likely to wind up executing "in isolation." The remaining factors in a kernel's response time are solely determined by its position in its stream, its position in the primary queue, and the blocks from other kernels it may need to contend with near the start and end of its execution. The restrictions that we discovered were successfully applied in subsequent works to produce models capable of predicting response-time bounds for GPU-sharing kernels (Yang, Amert, Yang, Otterness, Anderson, Smith and Wang 2018). However, this

follow-up work was mostly conducted by another member of our group, to whose dissertation we direct interested readers (Yang 2020).

3.5 Timing Pitfalls Pitfalls With NVIDIA GPUs

So far in this chapter, we have discussed two non-intrusive approaches for improving timing predictability on NVIDIA GPUs: co-scheduling (Section 3.2), which improves throughput without undue costs on predictability, and understanding queueing behavior (Section 3.4), which makes GPU concurrency less unpredictable in the first place. Still, we can apply the tools developed in Section 3.3 for another purpose: identifying and investigating aspects of the CUDA API that, when used naïvely, result in extremely degraded worst-case timing behavior.

In this section, we identify several such pitfalls, including both causes and effects. We separate them broadly into two categories: *synchronization-related pitfalls*, discussed in Sections 3.5.1 and 3.5.2, and *CUDA API pitfalls*, discussed in Section 3.5.3.

3.5.1 Synchronization-Related Pitfalls

We have already discussed GPU synchronization in one context, *i.e.*, calling `cudaStreamSynchronize` to wait until a kernel completes. Synchronization inherently requires blocking CPU computations, but, unlike explicit calls to `cudaStreamSynchronize`, not all of the causes of synchronization in a GPU-using application are necessarily obvious. Some forms of synchronization occur due to seemingly non-synchronization-related CUDA API functions, and may cause blocking in unrelated CPU and GPU code. In this section, we construct experiments that clearly illustrate these synchronization effects, along with a list of associated pitfalls timing-cognisant CUDA programmers likely wish to avoid.

In Section 3.4, we focused only on a limited context where kernel launches were the *only* active GPU operations.[13] Clearly, this will not be the case in virtually any sophisticated real-world CUDA software, where programmers will likely encounter (both intentionally and unintentionally) the need for synchronization between CPU and GPU operations.

[13] Our full set of experiments also considered memory copies, see Amert *et al.* (2017).

For this section, we continue to limit our attention to CPU threads that share a single CUDA context and submit kernels or other operations to separate streams, meaning that any sets of kernels described in this section's experiments *could* run concurrently were it not for synchronization-related effects.

3.5.1.1 Overview of GPU Synchronization

Most developers are familiar with the concepts of synchronization in a CPU-only context where two or more processes must communicate or coordinate their actions. Synchronization becomes more complicated when a CPU process must coordinate with code executed on the GPU. Commonly, this occurs when the CPU process must determine when data in GPU memory is safe to access (*e.g.*, kernel code is not accessing the same memory). This is accomplished by waiting for the GPU to complete outstanding work and reach a point in time when the data access can safely occur. For convenience, we refer to such a point in time as a *synchronization point*.

In CUDA, there are multiple ways to achieve GPU synchronization. They fall into two categories: *explicit synchronization*, which is always programmer-requested, and *implicit synchronization*, which can occur as a side effect of CUDA API functions intended for purposes other than synchronization. Our research has uncovered some pitfalls relating to both categories of synchronization, many of which result in unnecessary additional *blocking*. While the pitfalls we discuss should not change *logical* correctness, ignoring them would be perilous in a safety-critical system where all forms of blocking must be anticipated and accounted for.

3.5.1.2 Explicit Synchronization

Explicit synchronization refers to synchronization points that a CUDA program explicitly requests using CUDA API functions such as `cudaStreamSynchronize`. Programs typically invoke explicit-synchronization functions after launching one or more asynchronous CUDA kernels or memory-transfer operations, in order to wait for the operations to complete. In contrast to implicit synchronization, the sole purpose of explicit-synchronization functions is to block the calling CPU code until the GPU reaches a synchronization point.

The CUDA documentation states that explicit synchronization will block the calling process until "all preceding commands" have completed (NVIDIA Corporation 2022*b*, Section 3.2.6.5.3). For example, when invoking `cudaDeviceSynchronize`, "preceding commands" encompasses all commands issued

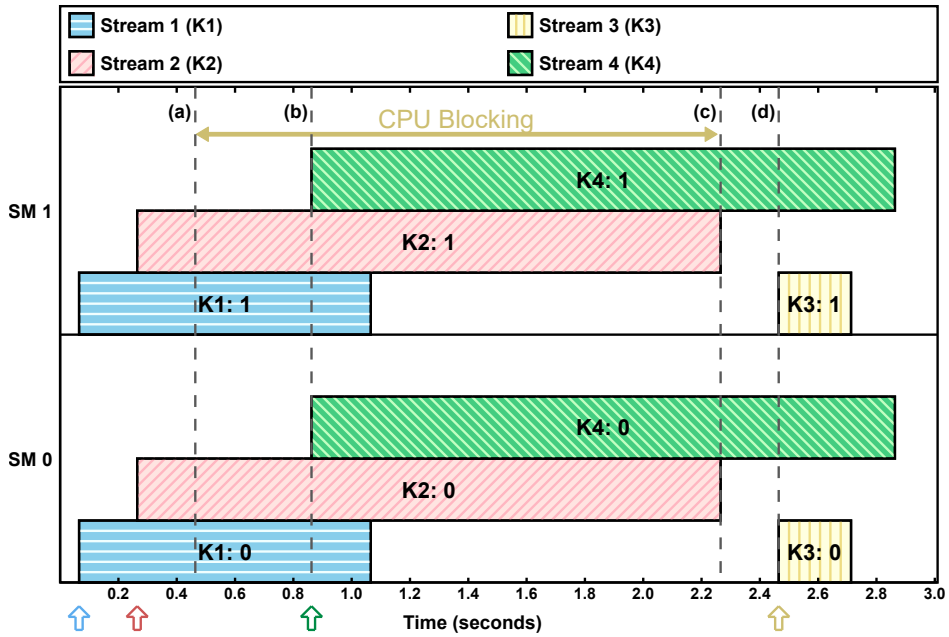


Figure 3.17: Explicit synchronization requested before K3, observed on the Jetson TX2.

to the device from all streams within the CUDA context. Other explicit-synchronization options, including `cudaStreamSynchronize`, will only block until preceding commands from a specified stream have completed.

We investigated the behavior of explicit synchronization using a Jetson TX2, running the same microbenchmarking framework described introduced in Section 3.3. Figure 3.17 shows the behavior of explicit synchronization we observed in one experiment. Figure 3.17 uses the same format as the figures used in Section 3.4, but we annotated it with additional information.

To produce Figure 3.17, we configured our framework to launch four microbenchmark instances that share a single CUDA context. Each microbenchmark instance launches one kernel in a separate stream, separated by a small amount of time. Each kernel requires two blocks of 512 threads, and the figure shows that the GPU hardware assigned one block from each kernel to each SM. All four kernels use the `timer_spin` code described in Section 3.3.2 to perform a busy loop for a set amount of time.

With minor modifications to our microbenchmarking code, we caused the CPU thread responsible for launching Kernel K3 in Figure 3.17 to launch an explicit-synchronization command, `cudaDeviceSynchronize`, at time **(a)**. Calling this function caused the CPU thread to be blocked until the prior GPU commands, Kernels K1 and K2, completed at time **(c)**, after which it carried out its original behavior: sleep for 0.2 seconds and launch Kernel K3. This behavior is exactly what one would expect, given the description

of explicit synchronization from official documentation. However, this experiments also uncovered Pitfall 3.1 for the unwary:

Pitfall 3.1 *Explicit synchronization does not block future commands issued by other CPU threads.*

The fact that the launch of K4 by its CPU task was not blocked at time **(b)** is an example of this pitfall. Logically, this does not violate the semantics of `cudaDeviceSynchronize`, which only pertains to preceding commands. It does, however, illustrate that the CUDA framework takes advantage of this nuance to improve efficiency—one cannot assume that explicit "device" synchronization guarantees the absence of *any* GPU work. Implicit synchronization, which we cover next, presents more serious pitfalls.

3.5.1.3 Implicit Synchronization

Implicit synchronization occurs as a side effect of CUDA API calls that are otherwise unrelated to synchronization. For example, implicit GPU synchronization may occur due to freeing GPU memory or launching a kernel to the default stream. Presumably, this is because some modifications to GPU device state can only occur while no kernels are executing. The CUDA documentation about implicit synchronization (NVIDIA Corporation 2022*b*, Section 3.2.6.5.4) states:

> Two commands from different streams cannot run concurrently if any one of the following operations is issued in-between them by the host thread:
>
> - A page-locked host memory allocation
> - A device memory allocation
> - A device memory set
> - A memory copy between two addresses to the same device memory
> - Any CUDA command to the NULL stream

Unlike the relatively straightforward documentation about explicit synchronization, our experiments revealed that this list includes several operations that do not necessarily cause implicit synchronization, and fails to include some functions that do. We consider this particularly problematic for real-time systems, where the ability to accurately model blocking is critical.

Pitfall 3.2 *Documented sources of implicit synchronization may not occur.*

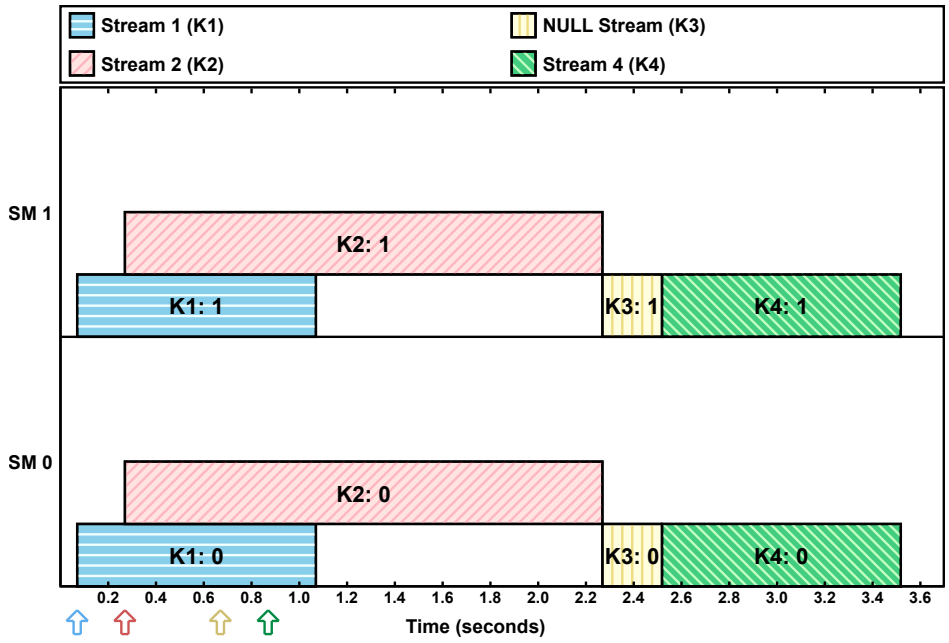


Figure 3.18: Implicit synchronization caused by launching Kernel K3 in the NULL stream.

Pitfall 3.2 became apparent to us when, in all of our experiments, we never observed implicit synchronization as a result of a device-memory operation, regardless of whether the operation was an allocation, set, or copy. Neither did we find that a page-locked[14] host memory allocation (presumably referring to the `cudaMallocHost` API function) caused implicit synchronization. Our experiments covered CUDA versions 8.0 and 9.0, and the Maxwell, Pascal, and Volta GPU architectures. This, of course, does not prove that implicit synchronization can *never* happen under such circumstances (this documentation was likely written to include the behavior of older GPU architectures), but it does indicate that the documentation's statement that "two commands cannot run concurrently" is not a reliable rule. For example, this documentation could mislead a programmer into thinking it adequate to use `cudaMallocHost` in lieu of explicit synchronization, when this is not the case. In fact, the only case from the list on Page 72 in which we *did* observe implicit synchronization was launching GPU operations in the NULL stream.

Figure 3.18 shows a similar scenario to the one in Figure 3.17, with one key difference: the CPU process for K3 does not call `cudaDeviceSynchronize` before launching K3. Instead, it launches K3 in the NULL stream. The implicit synchronization, and resulting loss of concurrency, is clearly visible in the figure. Execution of K3 must wait for the first two kernels to complete, and, in contrast to explicit synchronization,

[14] *Page-locked*, or *pinned*, memory refers to CPU memory buffers that the operating system guarantees will remain present at the same location in physical DRAM.

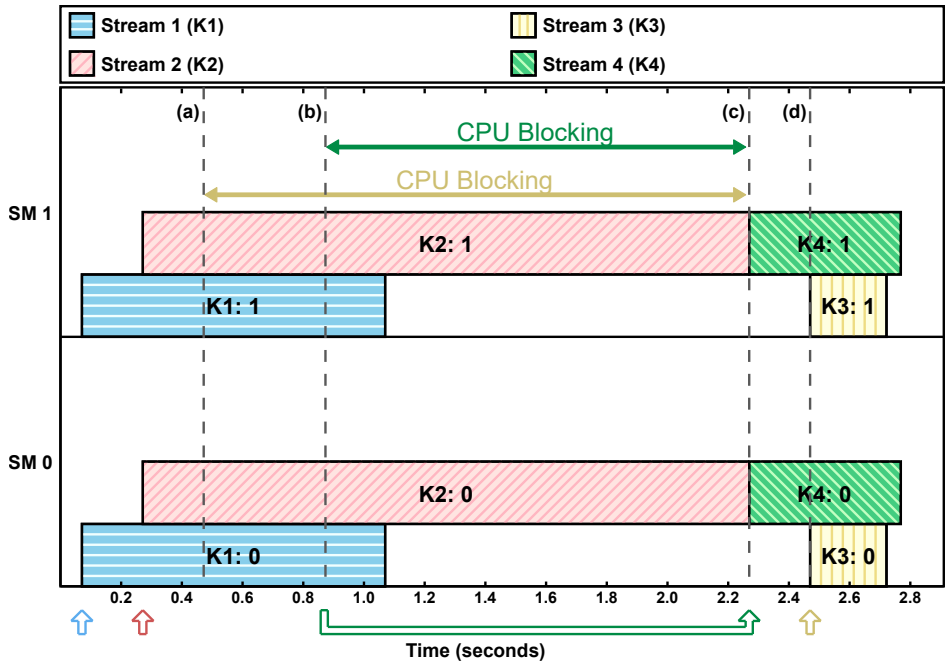


Figure 3.19: Implicit synchronization causing additional CPU blocking due to `cudaFree`.

using the NULL stream even prevents K4 from running concurrently. Even though this loss of concurrency may be striking, it at least is explicitly documented, and can be avoided (or used, if desired) in a carefully designed real-time application.

We found, however, a different source of implicit synchronization that is a far more problematic pitfall, and is not even listed in the documentation on synchronization: *freeing device memory*.

Pitfall 3.3 *The CUDA documentation neglects to list some functions that cause implicit synchronization.*

Pitfall 3.4 *Some CUDA API functions will block future, unrelated, CUDA tasks* on the CPU.

Figure 3.19 shows the results of an experiment identical to the one in Figure 3.17, but this time the call to `cudaDeviceSynchronize` at time **(a)** was replaced with a call to `cudaFree`, which is used to free allocations of GPU memory. In this particular experiment, `cudaFree` freed an arbitrary buffer of GPU memory that we allocated only for the sake of the experiment, and was never accessed by any kernel code. As with the previous synchronization experiments, this API call was performed in the CPU thread responsible for launching kernel K3.

Figure 3.19 includes both Pitfalls 3.3 and 3.4. The fact that this blocked the calling CPU thread until all prior GPU work had completed at time **(c)** indicates that `cudaFree` resulted in implicit synchronization.

Similar to the NULL-stream behavior, implicit synchronization also prevented subsequent kernels from starting to execute until `cudaFree` completed at time **(c)**. We speculate that this behavior by `cudaFree` is necessary because alterations to memory-mapping state require a quiescent execution environment. However, the most surprising effect was not that K4 was blocked, but that K4's task was blocked *on the CPU* until time **(c)**, even though it issued an "asynchronous" kernel launch.[15] Pitfall 3.4, resulting from this behavior, can be particularly harmful for would-be real-time software, as it indicates that CPU threads can experience blocking from GPU operations launched by other, unrelated, tasks.

3.5.2 Overcoming Synchronization-Related Pitfalls

GPU synchronization has two problematic effects—introducing indeterminate amounts of blocking and reducing GPU concurrency. This means that programmers who develop real-time systems must understand the pitfalls inherent in explicit and implicit synchronization. This is especially true if the schedulability of a real-time task system relies on minimizing blocking or high GPU utilization. Avoiding pitfalls can be accomplished through careful construction of CUDA programs to, for example, avoid using the NULL stream or freeing memory outside of certain time intervals.

Our experiments indicate that GPU synchronization does not extend across GPU-using tasks that are using separate CUDA contexts. We also conducted a limited number of experiments using MPS on discrete GPUs, resulting in observations that using MPS also removes these sources of implicit synchronization between separate processes. Even if MPS is unavailable, if synchronization is the dominant limiting factor on real-time schedulability, it may be desirable to require tasks to use separate CUDA contexts. Even though this would prevent GPU concurrency between tasks, such an organization may still be beneficial overall if synchronization-related blocking is a greater limiting factor.

3.5.3 Programming-Related Pitfalls When Using the CUDA API

Our research has necessarily involved writing thousands of lines of code requiring careful use of the CUDA API. In addition to blocking-related effects, we have identified tangential issues such as simple documentation errors, version-specific performance changes, and hard-to-detect programming mistakes. We present a list of these pitfalls along with accompanying experiments in this section.

[15] As an aside, our decision to record timestamps immediately after a kernel-launch call returns (noted in our discussion of Figure 3.8) is what enabled us to notice this oddity.

```
if (!CheckCUDAError(cudaMemsetAsync(
  state->device_block_smids, 0,
  data_size))) {
  return 0;
}
```

```
if (!CheckCUDAError(cudaMemsetAsync(
  state->device_block_smids, 0,
  data_size, state->stream))) {
  return 0;
}
```

Figure 3.20: Contrasting a code snippet that causes implicit synchronization (on the left) with one that does not (on the right).

3.5.3.1 Synchronous Defaults

Even though it may be simple for a programmer to just submit GPU operations to a user-defined stream as opposed to the NULL stream, some simple mistakes in doing so may be easy to miss. This is particularly true when using the `Async` versions of CUDA API functions, such as `cudaMemsetAsync`. For example, consider the code snippets in Figure 3.20, which present a particular example of Pitfall Pitfall 3.5 below.

Pitfall 3.5 *`Async` CUDA functions use the GPU-synchronous NULL stream by default.*

In Figure 3.20 the left-hand snippet's call to `cudaMemsetAsync` is missing a final argument specifying a user-defined stream, which causes the NULL stream to be used by default. Given how the function is defined in the C++ CUDA API, this is not a syntactic error, so it will not be flagged by the compiler. Neither is it a logical error, as the memory-set operation still successfully executes on the GPU as expected. The code remains a timing pitfall, however, and the lack of syntactic or logical errors can make it particularly hard to catch. As pointed out in Section 3.5.1.3, this usage of the NULL stream will prevent concurrency on the GPU. We corrected the mistake in the right-hand snippet of Figure 3.20 by supplying a `stream` argument. On a personal note, this specific mistake led to months of inconsistent results in our own experiments, despite our relatively extensive experience examining the subtleties of CUDA behavior (these snippets are parts of much larger programs). Without additional assistance, would it be reasonable to expect developers of machine-learning or AI applications to catch all such such errors or appreciate their impact?

Even though the examples in Figure 3.20 only use `cudaMemsetAsync`, Pitfall 3.5 applies to other CUDA API functions as well, such as `cudaMemcpyAsync`. The fact that the CUDA documentation indicates that these functions may cause implicit synchronization in some cases, as discussed in Sections 3.5.1.3 and 3.5.3.2, makes potential programmer errors even harder to notice in cases where synchronization is due to NULL-stream usage rather than memory operations.

3.5.3.2 Flawed Documentation

Another substantial danger stems from inaccuracies or omissions in official documentation provided by NVIDIA. While function signatures and data structures receive accurate (but sometimes sparse) official documentation, scheduling and synchronization remain under-discussed.

CUDA documentation sources. The most fundamental CUDA documentation comes from two official sources: the *CUDA C++ Programming Guide* (NVIDIA Corporation 2022*b*), which focuses on high-level concepts related to GPU programming, and the *CUDA Runtime API documentation*, which lists information for the individual functions in CUDA's runtime API (see Section 3.1.2). We refer to both sources in this section.

Pitfall 3.6 *Observed CUDA behavior often diverges from what the documentation states or implies.*

Our observation about `cudaFree` in Pitfall 3.3 is an obvious example of Pitfall 3.6. Both `cudaFree` and `cudaFreeHost` not only cause implicit synchronization, but block other CUDA API calls from other CPU threads. The CUDA documentation for `cudaFree` mentions neither of these side effects,[16] leaving the reader to assume that these functions behave similarly to other CUDA functions and have no side effects.

Pitfall 3.2 is a milder example of Pitfall 3.6. Both the CUDA Programming Guide[17] and the CUDA Runtime API documentation[18] state, entirely unambiguously, that `Async` memory-transfer operations such as `cudaMemcpyAsync` *will* behave synchronously when transferring between the GPU and unpinned regions of CPU memory. Our experiments found that this is simply not the case, regardless of whether the CPU memory buffer was pinned or not.

3.5.3.3 Unknown Future

Underlying these pitfalls is a single overarching problem: the black-box nature of current GPU-enabled platforms. Our experiments and attempts to enumerate timing-related pitfalls focus on alleviating some of

[16] In an update from our original research into the topic (circa 2017), the current description of `cudaFree` in the CUDA Runtime API documentation now includes a note about specific circumstances in which the function will *not* perform implicit synchronization, albeit without explicitly stating that the function *will* cause implicit synchronization in most or all other cases. The documentation still does not differentiate `cudaFree` from other sources of implicit synchronization by including the function's ability to block CUDA API calls from other CPU threads.

[17] Section 3.2.6.1 in version 11.7.0: "`Async` memory copies will also be synchronous if they involve host memory that is not page-locked."

[18] Section 2 of version 11.7.0: "For transfers from device memory to pageable host memory, [`Async` memcpy functions] will return only once the copy has completed."

this problem for the time being, but any work we do is subject to the same drawback faced by virtually all black-box investigations of systems under active development:

Pitfall 3.7 *What we learn about CUDA and GPU behavior may not apply in the future.*

Despite the fact that we validated our experimental results on several CUDA versions and GPU architectures, there is no guarantee that our results will hold after future GPU-architecture or CUDA-version updates. This applies not only to rules about scheduling or blocking (*i.e.*, Pitfall 3.6), but also may even apply to performance characteristics like memory-access times. In particular, we encountered an example of this pitfall in some older experiments, after updating from version CUDA version 7.0 to version 8.0. The following paragraph provides some additional context.

Pitfall 3.7 example: CUDA-version-dependent memory performance. In our earlier explorations of embedded GPUs, we attempted to profile the performance of the Jetson TX1's *zero-copy memory*, a feature specific to embedded GPUs that allows CUDA code to directly share memory buffers with CPU code. We were particularly interested in contrasting zero-copy with *unified memory*, which is similar from a programming standpoint but operates by transparently copying data between the CPU and GPU on demand. We ran two similar microbenchmarks to measure memory-access times using zero-copy, unified, and "traditional" memory (requiring manual copies). Our two microbenchmarks were:

- **Random Memory Walk**: This microbenchmark created 64 CUDA threads, grouped into two blocks of 32 threads each, that accessed 256 MB of GPU memory in a shuffled pattern. Each thread began at a different offset in the random walk and performed 256,000 reads in a loop. We intended this to measure memory performance with reduced cache benefits.

- **In-Order Memory Walk**: This was identical to the random memory walk, but accessed the array of memory in order, with the different GPU threads accessing elements spread evenly across the array. In contrast to the random memory walk, we intended this to measure performance with full caching benefits.

The results of these microbenchmark experiments are shown in Tables 3.1 and 3.2. In order to exclude overheads due to kernel launches, synchronization, *etc.*, the times in the tables are based on *block times* measured on the GPU using the `globaltimer` register (in an identical manner to Section 3.2.2.1).

	CUDA 7.0 (Times are in milliseconds)				CUDA 8.0 (Times are in milliseconds)			
Memory type	Min	Max	Mean	Std. Dev.	Min	Max	Mean	Std. Dev.
Traditional	238.1	411.5	245.1	4.4	671.0	684.7	679.3	3.3
Zero-copy	676.3	677.0	676.5	0.1	671.9	684.7	680.2	3.3
Unified	686.0	702.2	686.2	0.1	671.9	690.6	680.1	3.3

Table 3.1: Best-, worst-, and average-case times for the random memory walk microbenchmark for CUDA 7.0 and 8.0, measured on the Jetson TX1.

	CUDA 7.0 (Times are in milliseconds)				CUDA 8.0 (Times are in milliseconds)			
Memory type	Min	Max	Mean	Std. Dev.	Min	Max	Mean	Std. Dev.
Traditional	4.4	7.8	4.4	0.06	3.1	3.2	3.1	0.005
Zero-copy	48.1	77.5	75.9	4.8	3.1	3.2	3.2	0.006
Unified	4.4	4.6	4.4	0.005	3.1	3.2	3.2	0.006

Table 3.2: Best-, worst-, and average-case times for the in-order memory walk microbenchmark for CUDA 7.0 and 8.0, measured on the Jetson TX1.

Observation 3.7 *Random traditional memory accesses* slowed down *after updating the Jetson TX1 from from CUDA 7.0 to CUDA 8.0.*

Observation 3.7 is supported by Table 3.1. The most surprising change due to transitioning from CUDA 7.0 to 8.0 was that the newer version of CUDA caused traditional memory to become around three times *slower* in the random walk.

Observation 3.8 *Under CUDA 8.0, unified and zero-copy memory perform nearly identically, which was not the case under 7.0.*

This observation is supported by Table 3.2. Under CUDA 7.0, unified memory was slower than zero-copy memory during the random walk, but as fast as traditional memory during the in-order walk. Under CUDA 8.0, in-order zero-copy memory accesses see a significant improvement. Among other factors, we speculate that the changes in memory performance from CUDA 7.0 to CUDA 8.0 are the result of optimizing for sequential memory accesses at the expense of some performance degradation for arbitrary access patterns, but any such speculation is likely impossible to confirm without access to CUDA source code. Regardless, the point of mentioning both Observations 3.7 and 3.8 is to illustrate that Pitfall 3.7 can even apply to timing behaviors that users may otherwise (incorrectly) assume is determined entirely by hardware.

3.5.4 Avoiding CUDA Pitfalls: Closing Words, and an Application

Even though other safety-critical hardware inevitably undergoes changes and updates, future-proof programs can still be developed against a stable specification. Likewise, the only way to truly mitigate some pitfalls, like Pitfall 3.7, is for GPU manufacturers to release stable, accurate documentation about their GPU platforms, along, preferably, with giving developers greater control over GPU scheduling and synchronization.

In the meantime, spreading awareness of pitfalls is a pragmatic measure for improving timing properties in GPU applications today. Our brief catalog of pitfalls has already borne fruit in further research carried out by others in our group: Amert and Anderson (2021) developed a middleware system that intercepts and flags CUDA-API calls that may result in undesired implicit synchronization. Amert and Anderson applied their framework to monitor the execution of the HOG image-analysis algorithm from the popular *OpenCV* open-source computer-vision library.[19] By fixing the problems identified (largely, calls to `cudaFree` or NULL-stream usage), they successfully reduced the frequency and magnitudes of HOG's worst-case response times.

3.6 Chapter Summary

In this chapter, we explored three methods for improving the safety and predictability of shared-GPU systems without sacrificing efficiency. First, we investigated *co-scheduling*: simply allowing separate processes to access GPUs simultaneously. We found that, so long as real-time tasks use separate CUDA contexts, CUDA's time-sliced scheduling of separate contexts keeps unpredictable interference highly unlikely (barring as-yet unknown interference channels). Next, in an effort to overcome any limits imposed by CUDA's time-slicing behavior, we investigated how separate kernels are queued and scheduled on GPU hardware, inferring a set of rules capable of accurately predicting the order in which kernels will run. Even though this sort of knowledge may not directly lead to improvements in raw throughput, revealing previously unknown policies fundamentally makes the behavior of GPU-using real-time tasks more *predictable*. Finally, we documented a list of possible pitfalls that may ensnare unwary developers of real-time CUDA code. Several pitfalls imply "best practices," *e.g.*, avoiding use of CUDA's NULL stream, that should be easy to adopt with no performance sacrifices whatsoever. Furthermore, this work has already been applied to

[19] `https://opencv.org/`

develop an automated system for analyzing and improving the timing performance of sophisticated CUDA programs (Amert and Anderson 2021).

Acknowledgements. The material in this chapter was drawn from a collection of several papers published in the years 2016 through 2018.

The content in Section 3.2 as well as the memory-performance experiment discussed in Section 3.5.3.3 came from two publications: Otterness *et al.* (2016) and Otterness, Yang, Rust, Park, Anderson, Smith, Berg and Wang (2017). In the 2016 paper, Otterness and Miller collaborated on the microbenchmark experiments, which Otterness continued to expand and maintain for the follow-up publication in 2017.[20] The memory-access microbenchmarks presented in Section 3.5.3.3 were developed in collaboration between Otterness and Rust.[21] Yang was responsible for designing and conducting the experiments using CaffeNet, which we briefly mention in Section 3.2.2.1.

The microbenchmarking framework described in Section 3.3 was primarily developed by Otterness, prior to the publication of Otterness, Yang, Amert, Anderson and Smith (2017).[22] Otterness was responsible for initial "cutting ahead" experiment of Section 3.4.1, as well as the experiments and formulation resulting in the scheduling rules of Section 3.4.2. However, Amert and Yang conducted many additional experiments to discern NULL-stream behavior, behavior of CUDA streams with different priorities, occupancy limits due to shared memory, and memory-copy scheduling. These details are included in Amert *et al.* (2017). We are also particularly indebted to Amert's development of the visualization script, used to produce many figures in the aforementioned papers as well as Figures 3.9, 3.11, and many other similarly styled figures throughout this chapter.

Finally, the material in Section 3.5 was primarily from Yang, Otterness, Amert, Bakita, Anderson and Smith (2018). In this collaboration, investigation of CUDA documentation was primarily conducted by Bakita. As before, Amert contributed greatly to plots and visualizations. In the original publication, Yang conducted a variety of experiments evaluating the temporal performance of various GPU benchmarks both with and without using MPS. Otterness collaborated with Yang and Amert on producing code for experiments, and was responsible for formulating the various "pitfalls" based on accumulated observations.

[20] Source code is available online: `https://github.com/yalue/PeriodicTaskReleaser`.

[21] Source code is available online: `https://github.com/Sarahild/CudaMemoryExperiments`.

[22] Source code is available online: `https://github.com/yalue/cuda_scheduling_examiner_mirror`.

CHAPTER 4: AMD GPUS[1]

Given that NVIDIA GPUs are well-established and better-studied, what do AMD GPUs have to offer real-time programmers? Perhaps unsurprisingly, the answer requires returning to the topic of GPU sharing. Some features currently unique to AMD GPUs, especially *an open-source software stack* and *broad support for hardware partitioning*,[2] have the potential to accelerate and aid the long-term viability of real-time GPU research.

Ideally, we hope our research may eventually produce unified models of GPU behavior that apply to NVIDIA, AMD, and GPUs made by any other manufacturer. Whether this is possible depends on how similar GPUs are, or alternatively, determining how much it would cost to ignore the differences. Unfortunately, when it comes to timing behavior in AMD and NVIDIA GPUs, we shall see in this chapter that there are some important differences indeed.

This is not unexpected. Naturally, AMD GPUs also have their own set of drawbacks, some of which may justifiably warrant continued research on NVIDIA GPUs regardless of the availability of an alternative. Whether or not AMD GPUs see further adoption in future work, as we argued in Section 1.3, a greater variety of research platforms can only benefit the field, even if some work is not possible on every alternative. After all, we do not seek to declare victory for one GPU manufacturer or the other, but to further the development of safer GPU-accelerated software.

The structure of this chapter follows a similar form to Chapter 3: we begin with an overview of AMD-specific details in Section 4.1. Next, we cover some of the important aspects of AMD GPU scheduling

[1]Contents of this chapter previously appeared in the following papers:

Otterness and Anderson (2020). *AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads*. Euromicro Conference on Real-Time Systems (ECRTS).

Otterness and Anderson (2021). *Exploring AMD GPU Scheduling Details by Experimenting With "Worst Practices"*. International Conference on Real-Time Networks and Systems (RTNS).

Otterness and Anderson (2022). *Exploring AMD GPU Scheduling Details by Experimenting With "Worst Practices"* (extended version for journal publication). Real-Time Systems, Volume 58, Number 2, Pages 105–133. Springer.

[2]NVIDIA's MIG partitioning support is only currently available in its top-end, expensive GPUs (see Section 3.1.1), with no indication as to whether this support will appear in less-expensive devices.

behavior in Section 4.2. Section 4.3 ends the chapter with a discussion of AMD-specific pitfalls that developers of GPU-using real-time systems may encounter.

4.1 Overview of AMD GPUs

As with NVIDIA GPUs, AMD GPUs are grouped into several architectures and are programmed using a specific software stack.

4.1.1 AMD GPU Architectures

Unlike the lengthy list of relevant NVIDIA architectures in Section 3.1.1, AMD generally opts to produce multiple versions of a single named architecture, only rarely naming new architectures to coincide with major reorganizations of GPU internals. Our AMD-centric work exclusively uses GPUs from AMD's *GCN* (graphics core next) architecture.

The GCN architecture. AMD first announced its GCN GPU architecture in 2011. One of the stated goals of the GCN architecture was an increased focus on general-purpose GPU computing, reflected in hardware by a transition away from the VLIW (very long instruction word) processors used by AMD's previous GPUs to a SIMD model more conducive to general-purpose computations (AMD Corporation 2011). The GCN architecture remained in use for many years, being progressively updated through five versions. The Radeon VII GPU (shown in Figure 2.3) uses the fifth version of the GCN architecture.

The RDNA architecture. While we do not study any AMD GPUs from non-GCN architectures in this dissertation, AMD started producing GPUs using the *RDNA* (Radeon[3] DNA) architecture in 2019. According to a whitepaper published by AMD, the RDNA architecture maintains backwards compatibility with the GCN architecture, with an increased focus on scalability. Perhaps of greater interest to our experiments in Section 4.2 would be the fact that the RDNA architecture reorganizes CUs within shader engines, adding an additional level of hierarchy to that displayed in Figure 2.3. RDNA introduces "adjacent compute unit cooperation," in which pairs of compute units can cooperate to execute a warp ("wave" in AMD terminology) of 64 threads rather than 32 threads (AMD Corporation 2019). Unfortunately, practical limitations on GPU availability and support prevented our research from exploring the impact this restructuring has (if any) on CU masking or partitioning.

[3] *Radeon* is the name of a computer-hardware brand, which was acquired by AMD in 2006.

Why study only a single AMD GPU architecture? From the length of this section alone, it should be clear that our research involved far fewer unique types of AMD GPUs than NVIDIA GPUs. In a clear divergence from NVIDIA's widespread CUDA support, AMD's ROCm (Radeon Open Compute) software only supports a very small number of GPUs. For almost the entire duration of our research, ROCm only supported GPUs using the later revisions of the GCN architecture. We consider this a major drawback of conducting research using the AMD GPU ecosystem, so we save most of our discussion of this issue for Section 4.3, about pitfalls of AMD-targeted GPU research.

Even though we only used AMD's GCN architecture, we still tested more than one AMD GPU. We carried out some of our earlier AMD GPU experiments using the AMD RX 570 GPU, which uses the fourth generation of the GCN architecture. At the time of its release, the RX 570 was intended as a mid-range GPU, featuring 32 CUs and either four or eight GB of DRAM. However, in our experiments we noticed few practical differences between the RX 570 and the Radeon VII, beyond the latter's superior computational power. Therefore, we used the Radeon VII, with 60 CUs and sixteen GB of DRAM, for all of the AMD GPU experiments and results we report in this dissertation.

4.1.2 AMD GPU Software

For most of our AMD GPU experiments, we used version 4.2 of the *ROCm* (Radeon Open Compute) software stack. Aside from the obvious difference of supporting only AMD GPUs, ROCm is highly analogous to CUDA in terms of function. Like CUDA (see Section 3.1.2), ROCm encapsulates the full suite of software required to compile and run GPU applications, including the GPU-programming compiler, API specification, runtime libraries, and driver. On the other hand, unlike CUDA, ROCm's open-source nature means that the internal distinctions between ROCm's components are entirely visible to end users. Figure 4.1 shows the primary stack of components involved in ROCm. The top user-facing component is typically the HIP API, which is nearly identical to CUDA, with the main practical difference only being the names of the API functions. (We refer readers back to our example HIP-code snippet in Figure 2.1, and its CUDA equivalent in Figure 2.2.) GPU kernel code in HIP programs such as the `VectorAdd` function from Figure 2.1, are compiled using the LLVM compiler's AMDGPU backend.[4] Next, HIP is implemented using ROCclr (ROCm

[4]The LLVM documentation for this backend serves as a rich source of information about the instruction-set architecture and binary formats used by AMD GPUs: `https://www.llvm.org/docs/AMDGPUUsage.html`.

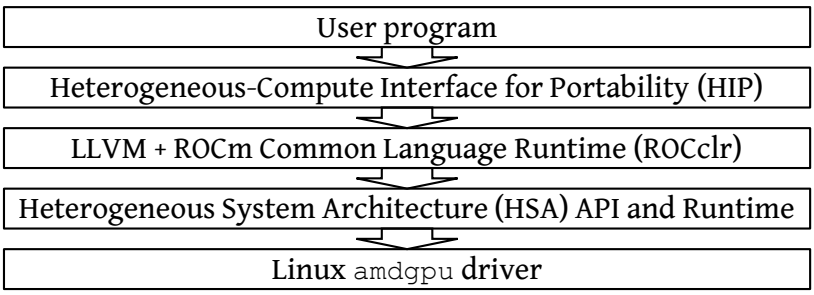


Figure 4.1: Components of the ROCm software stack.

Common Language Runtime), which provides an ostensibly[5] cross-platform API for managing AMD GPUs. At least on Linux, ROCclr is implemented on top of a lower-level userspace library implementing the HSA (Heterogeneous System Architecture) API,[6] which creates and manages the memory-mapped queues and commands that interface with the driver and hardware. Finally, the HSA runtime library interacts with the Linux kernel's `amdgpu` driver.

The importance of openness. The appeal of a software stack like ROCm hinges on the answer to one question: why should developers care that ROCm is open source? Fortunately, the potential benefit of an open-source stack can be easily demonstrated by revisiting the implementation approaches taken by the authors of the prior real-time GPU-management work from Section 2.6.

TimeGraph (Kato, Lakshmanan, Rajkumar and Ishikawa 2011) and *GPES* (Zhou, Tong and Liu 2015) require using the third-party open-source "Nouveau" driver for NVIDIA GPUs, which does not support CUDA. *RGEM* (Kato, Lakshmanan, Kumar, Kelkar, Ishikawa and Rajkumar 2011) uses a third-party CUDA implementation by PathScale, which has by now been discontinued for several years.[7] Papers that rely on reverse engineering, including our own work from Section 3.4 and others such as Jain *et al.* (2019) or Olmedo *et al.* (2020), require re-validation after every hardware or software change. Other works, that intercept driver communication like *GPUSync* (Elliott, Ward and Anderson 2013), are subject to a stable interface between the CUDA runtime libraries and the underlying device driver, but NVIDIA makes no such stability guarantees

[5] See our discussion of Pitfall 4.6 in Section 4.3.

[6] The HSA API is defined by the *HSA Specification*, of which AMD is a member. The HSA foundation seeks to publish sets of standards for interacting with computational accelerators: `http://hsafoundation.com/`.

[7] According to the PathScale website: `https://www.pathscale.com/`.

for these non-public implementation details. The key point here is that, when using NVIDIA GPUs, *working around the lack of source-code access is a prerequisite for meaningful research*.[8]

Other prior work demonstrates what is possible with a greater amount of access to internal GPU details. For example, working in collaboration with NVIDIA enabled Capodieci *et al.* (2018) to implement the popular preemptive EDF real-time scheduler for GPU workloads. In a second example, Jain *et al.* (2019) needed to conduct a major reverse-engineering effort to gather information for their memory and SM-partitioning system. Even though these two papers also needed to work around the lack of public source-code access or hardware details, their strong results illustrate an additional conclusion: *improved access to GPU internals enables fundamentally more powerful management paradigms*.

The ROCm stack means that AMD GPUs are subject to almost none of these software-based limitations. Developers can modify the underlying device drivers and user-level libraries, removing the need to use less-supported libraries or to enter non-disclosure agreements simply to add more functionality to existing code. Additionally, intercepting API calls or transforming application-level code can be rendered unnecessary by the ability to directly modify HIP, the LLVM compiler, or the ROCclr library. All of this means that a GPU-management system targeting AMD GPUs using the ROCm stack should not only be easier to implement, but also easier to use and easier for third parties (*i.e.*, other researchers) to validate.

What's there to study in an open-source system? Given our extensive black-box efforts to study NVIDIA GPUs and the points from the prior paragraph, one may think that an open-source platform, like AMD's ROCm software, would be in an unrivaled position of prominence within real-time GPU research. Unfortunately, mere source-code access is far from an ideal solution to the information difficulties discussed above. By some counts, AMD's open-source code base contains upwards of millions of lines of code—the source code for AMD's GPU driver for Linux exceeds 2.1 million lines (though much of this is due to auto-generated C header files) (Larabel 2020). Driver code aside, the userspace portions of the ROCm stack require hundreds of thousands of additional lines. We performed a basic line count on the source code for the non-driver components of Figure 4.1, for version 4.2 of ROCm:

- **HIP**: Contains about 56,000 lines of source and header files, not including sample or test code.

[8] In a recent development, NVIDIA released an open-source version of their GPU driver for Linux systems. While this is an encouraging stride forward, it unfortunately arrived too late for us to consider, and only supports their Turing architecture and later. Additionally, we remind readers that *CUDA* as a whole consists of far more than the driver, and there is no indication that NVIDIA plans to release source code for their userspace libraries. In summary, NVIDIA's open-source driver is a highly welcome development, but ROCm still maintains a distinct advantage over CUDA in terms of openness.

- **ROCclr**: Contains about 111,000 lines of source and header files.
- **HSA API and Runtime**: Contains about 110,000 lines of source and header files.

This code is accompanied by strikingly little public documentation about the behavior and design of the software itself, likely due to the fact that contribution and maintenance comes almost exclusively from AMD insiders. The sheer line count means that ROCm's source code can certainly be a rich source of information, but without external documentation AMD's public repositories are a set of encyclopedias without an index: mostly useful to those who already know where to look.

The following section therefore fills an information gap for AMD GPUs that is similar to that filled by our work on NVIDIA GPUs in Chapter 3. Unlike in Chapter 3, however, we do not rely on black-box tests or reverse engineering in our study of AMD GPUs. Instead, we collected the details in Section 4.2 from several sources, including public presentations, white papers, and specific source code references.[9]

4.2 Discovering the Behavior of AMD GPUs

We use a different methodology to study AMD GPUs than we did for NVIDIA—the availability of an open-source software stack means that we can study AMD GPU behavior using *white box* experiments: examples designed to exercise specific aspects or corner cases of AMD GPU behavior that we already have reasons to expect. Our investigation is particularly focused on AMD GPUs' *compute-unit masking* feature, which allows partitioning GPU-sharing tasks to specific sets of compute units within the GPU. Doing so is entirely in line with our goal of discovering AMD GPU scheduling behavior in general, as compute-unit masking is inextricably linked with AMD's internal hardware-scheduling policies.

Compute-unit masking. As we covered in our discussion of Figure 2.3, CUs play an analogous role in AMD GPUs to SMs in NVIDIA GPUs. CU masking is a type of *spatial partitioning* technique, which has been a common target for prior real-time research on NVIDIA GPUs. Unfortunately, this prior work on NVIDIA (we give several examples in Section 2.6.2) relies on software "tricks" to assign tasks to SMs, most commonly requiring GPU kernel code to check SM assignments at runtime and immediately stop executing blocks that are assigned to incorrect SMs. This type of software workaround is not necessary for AMD GPUs, because CU masking is fully supported by hardware, and easily made transparent to user applications. We

[9]We originally learned many of these details in a private conversation with an AMD engineer, to whom we are extremely grateful. This conversation simplified our search for corresponding information in the publicly available material.

discuss specific usage of the CU-masking API in Section 4.2.3.1, but for now it is sufficient to know that an arbitrary CU mask can be associated with each HIP stream, and that any kernel submitted to the stream will be required to execute only on the set of CUs enabled by the mask.

4.2.1 Motivating Experiments

With foreknowledge of AMD GPU behavior, we can craft workloads that intentionally trigger highly destructive interference between competing tasks. Doing so serves a dual purpose: first, the challenge of explaining degenerate cases gives structure to our subsequent explanation of scheduling behavior. Second, the experiments concretely illustrate the magnitude of the impact scheduling behavior can have on response times. In this case, we intentionally apply "worst practices" in contrived scenarios, but the lack of documentation mentioned in Section 4.1 means that there is almost no relevant guidance from AMD on how to properly design GPU-sharing workloads. In other words, a naïve developer, not knowing to avoid these practices, may stumble into the same mistakes.

Microbenchmarking framework for AMD GPUs. To conduct our experiments, we dedicated considerable effort to porting our microbenchmarking framework described in Section 3.3 from CUDA to HIP.[10] Our HIP port of the microbenchmarking framework operates in a near-identical manner to the original CUDA version, but the transition to HIP unfortunately removes some useful features present in CUDA. For example, AMD GPU kernel code is unable to determine its own CU assignment at runtime, unlike CUDA kernel code, which can read the special `smid` register to determine its SM assignment. Additionally, AMD GPUs do not expose a `globaltimer` register, which we used in our CUDA-based experiments (*e.g.*, Figure 3.4) to measure block times in nanoseconds. In combination, these factors prevent us from generating the detailed visualizations of block-CU assignments in the manner of Figures 3.9, 3.11, and others from Chapter 3. While this is an unfortunate limitation on how we can present AMD GPU scheduling results, our HIP version of the framework does include one hardware-specific advantage over its CUDA counterpart: namely, the ability to specify a compute-unit mask for each microbenchmark instance. With this feature, our microbenchmarking framework was entirely sufficient for quickly configuring and running the experiments we conducted in this section.

[10]The source code for this HIP version of our test framework is available online at `https://github.com/yalue/hip_plugin_framework`.

4.2.1.1 Experimental Setup

All of the experiments in this section measure response times of matrix-multiply microbenchmarks. All tasks multiply two 1,024x1,024 square matrices, writing the output into a third matrix of the same size. Each GPU thread is responsible for computing one element in the results matrix (*i.e.*, each thread computes the dot product between a row from the first matrix and a column from the second matrix). All elements of each matrix are 32-bit floating-point numbers, randomly initialized to values between 0 and 1.

We chose to use a matrix-multiply workload for several reasons. First, each matrix multiplication requires a constant amount of computation and memory accesses, ideally resulting in low variability between kernel invocations. Second, matrix multiply is a relevant and common operation in many AI and graphics applications. Third, it is easy to design matrix-multiplication kernels that use differing thread block sizes without affecting the total amount of GPU computation required. To exploit this flexibility in our experiments, we designed a matrix-multiply plugin for our microbenchmarking framework that enabled us to easily configure different block sizes for each matrix-multiply instance.

Choice of competing tasks. Even though all of our tasks carry out multiplication of 1,024x1,024 matricies, we use the flexibility with respect to block size to define two different tasks:

- `MM1024`: Uses blocks of 1,024 threads (specifically, 2D blocks with dimensions of 32x32 threads). Since the matrix contains 1,024x1,024 elements, this task launches exactly 1,024 blocks.
- `MM256`: Uses blocks of 256 threads (in this case, 16x16 2D blocks). Covering the complete matrix therefore requires 4,096 blocks.

Once again, we stress that both `MM1024` and `MM256` carry out an identical number of floating-point computations, just under slightly different configurations. Most of our experiments consist of running a *measured task* and a single *competitor* at the same time on the GPU. We measure the response times of the measured task while it contends with the competitor for GPU resources.

We took several additional steps when launching experiments. We disabled graphics on the host system, to prevent graphics processing from affecting our measurements. In order to amplify contention for compute resources (as opposed to memory), we configured our tests to copy the input matrices to the GPU only once, at initialization time. Tasks using CU masking created their streams using HIP's `hipExtStreamCreateWithCUMask` function. At runtime, we configured competing tasks to run as

many multiplication iterations as possible within 60 seconds, as opposed to using a fixed number of iterations. (This is why the number of samples in Table 4.1 differs between tasks.) A fixed number of samples would require estimating a number of iterations for each competing task, risking outliers if the competitor ends too early.

4.2.1.2 "Anomalous" Results

Table 4.1 contains the results of all of our experiments, showing the response times for each possible measured task against each possible competitor. Table 4.1 also includes three partitioning configurations for each combination of measured task and competitor:

- *Full GPU Sharing*: Both the measured task and the competitor have unrestricted access to all CUs on the GPU.
- *Even Partitioning*: Both the measured task and the competitor were restricted to separate non-overlapping partitions containing half of the GPU's CUs.
- *Uneven Semi-Partitioning*: CU partitions were identical to the "Even Partitioning" case, but the measured task was allowed access to one additional CU in the competitor's partition.

Without an understanding of AMD scheduling internals, these results likely contain several surprises.

Observation 4.1 *Competing against* `MM256` *adversely affects* `MM1024`*'s performance more than any other configuration.*

Observation 4.1 is illustrated by comparing the "`MM1024` (vs. `MM256`)" section of Table 4.1 with the corresponding values in any other section. In particular, the "Full GPU Sharing" results, shown in bold, are the slowest under this configuration, with average-case response times more than double those against an identical `MM1024` competitor, and five times that of `MM1024` in isolation. We can check that this has almost no impact from `MM256`'s perspective by checking the "`MM256` (vs. `MM1024`)" portion of the table. Not only

Scenario	Partitioning	# Samples	Min	Max	Median	Arith. Mean	Std. Dev.
Isolated `MM1024`	N/A	18258	3.094	3.631	3.203	3.201	0.018
Isolated `MM256`	N/A	11712	4.499	5.588	5.034	5.034	0.100
`MM1024` (vs. `MM1024`)	Full GPU Sharing	9179	5.915	7.292	6.421	6.447	0.144
	Even Partitioning	8482	6.774	8.254	6.973	6.988	0.082
	Uneven Semi-Partitioning	789	61.986	98.380	73.402	76.011	5.888
`MM1024` (vs. `MM256`)	Full GPU Sharing	3811	**12.214**	**19.578**	**15.503**	**15.652**	1.114
	Even Partitioning	8477	6.784	7.630	6.944	6.991	0.133
	Uneven Semi-Partitioning	711	64.873	101.531	84.047	84.362	4.381
`MM256` (vs. `MM256`)	Full GPU Sharing	10383	5.115	7.932	5.552	5.687	0.367
	Even Partitioning	9269	6.126	7.031	6.316	6.386	0.182
	Uneven Semi-Partitioning	1064	55.535	56.928	56.311	56.310	0.171
`MM256` (vs. `MM1024`)	Full GPU Sharing	15539	3.240	6.923	3.564	3.770	0.557
	Even Partitioning	9361	6.085	7.610	6.256	6.318	0.167
	Uneven Semi-Partitioning	1079	55.137	56.028	55.549	55.550	0.114

All times are in milliseconds.

Table 4.1: Table of experimental results.

does `MM1024` not harm `MM256`'s performance, it slightly *improves* `MM256`'s response times over `MM256` in isolation.[11]

Observation 4.2 *Even partitioning protects `MM1024` against `MM256`, but otherwise moderately increases response times.*

Observation 4.2 is made evident by comparing the "Even Partitioning" lines in Table 4.1 against the "Full GPU Sharing" lines. In all cases except for `MM1024` vs. `MM256` (and the worst-case time of `MM256` vs. `MM256`), the response times when the full GPU is shared are at least one millisecond faster than the times when the competitors are partitioned. This is not particularly surprising for the common case; it makes sense that it will be faster to allow a kernel to occupy any CU as it becomes available across the entire GPU. However, partitioning's ability to protect a workload against an "evil" competitor is obvious when observing `MM1024`'s partitioned performance against `MM256`, where the improvement in the observed worst case is nearly 12 milliseconds. So, Observation 4.2 is not particularly surprising, and a classic example of a "real-time" tradeoff between overhead and predictability.

Observation 4.3 *Poor partitioning causes abysmal performance.*

[11] The material in Section 4.2.2 does not entirely explain this particular anomaly, but the improvement likely is due to a competitor's presence improving the performance of block-dispatching hardware. Two factors support this assumption. `MM256` launches four times the number of blocks as `MM1024`, meaning that speeding up block launches provides a stronger benefit to `MM256`. For example, the presence of the `MM1024` competitor may help keep some hardware components active, but, as we shall see in Section 4.2.2, it will cause minimal additional contention for resources against `MM256`. Second, even though `MM256`'s times are faster in this case than in isolation, it still is not as fast as `MM1024` in isolation. This indicates that `MM1024` still has some advantage arising from its block configuration, as it is otherwise identical to `MM256`.

The most surprising feature of Table 4.1 is undoubtedly the extreme increase in response times of "Uneven Semi-Partitioning," regardless of competitor choice (though `MM1024` vs. `MM256` is still the worst, especially in the average cases). Under this "partitioning" approach, the measured task still maintains sole ownership over all of the CUs it was allowed under "Even Partitioning," but is additionally granted one CU that is shared with the competitor's partition. In other words, one *additional* CU leads to response times around ten times slower than under full GPU sharing or with equal, non-overlapping partitions.[12]

Remarks on these results. We clearly demonstrated that a naïve application of CU masking is *dangerous*. With such extreme performance degradation, even someone only passingly familiar with GPU management should suspect that our uneven semi-partitioned setup is a "worst practice." Nonetheless, partitioning is also *essential*, as different combinations of GPU-sharing tasks (*i.e.*, `MM1024` vs. `MM256`) reveal that asymmetrically destructive interference is a real possibility. The interesting issue is, of course, not the results themselves, but the underlying causes. Why does simply reducing a kernel's block dimensions make it such a fierce competitor? Why can *adding* a compute unit, even a shared one, lead to a dramatic, nearly 14-fold increase in worst-case response time? Fortunately, answers to these questions become apparent with an understanding of AMD GPU scheduling internals.

4.2.2 Scheduling Compute Kernels on AMD GPUs

The effects from Section 4.2.1 turn out to mostly arise from hardware, but we also must explain how a kernel arrives at the hardware to begin with. This section covers this entire path, beginning with a description of the queuing structure used to issue kernel-launch commands to AMD GPUs.

4.2.2.1 Queue Handling in Userspace

Following our work on NVIDIA GPUs in Section 3.4, it should not come as a surprise that kernel-launch requests proceed through a hierarchy of queues before reaching AMD GPU hardware. Figure 4.2 depicts the paths this request may take. To help reduce the complexity of our later explanations (and to provide an easier introduction than Figure 4.2), we begin with a high-level outline of the steps involved:

1. A user program calls the `hipLaunchKernelGGL` API function to launch a kernel (*i.e.*, Figure 2.1).

[12] The competitor's response times are barely impacted by sharing one CU with the measured task. For brevity, we chose to exclude these measurements from Table 4.1.

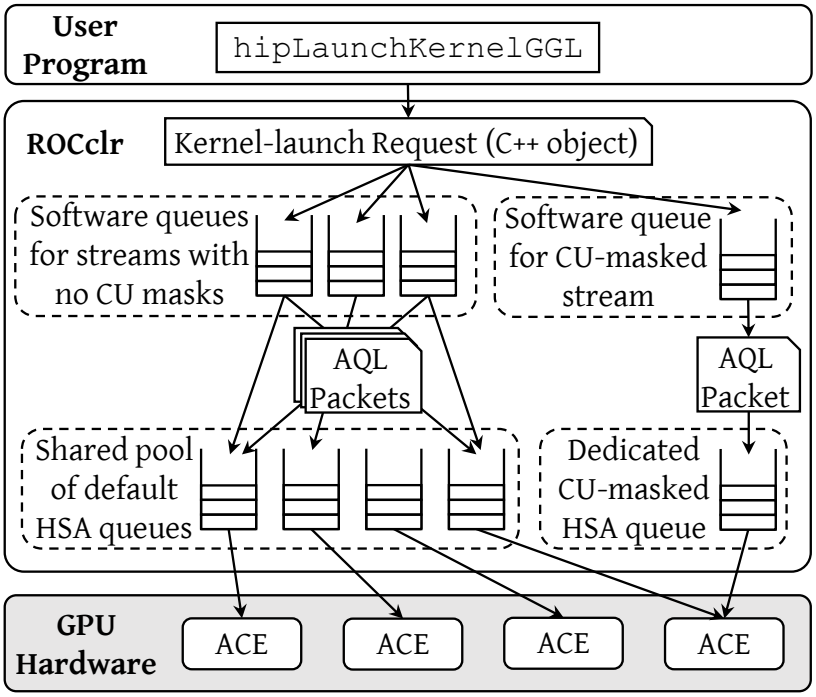


Figure 4.2: Paths through ROCm's queuing structure.

2. The HIP runtime inserts a kernel-launch command into a software queue managed by the ROCclr runtime library.

3. ROCclr converts the kernel-launch command into an AQL (architected queuing language) packet.

4. ROCclr inserts the AQL packet into an HSA queue.

5. In hardware, an asynchronous compute engine (ACE) processes HSA queues, assigning kernels to compute hardware.

A kernel's journey to the GPU's computational hardware begins with the `hipLaunchKernelGGL` API call, which is shown at the top of Figure 4.2 and responsible for enqueuing a kernel-launch request. A programmer's typical point of contact with the queuing structure is through HIP's "stream" interface introduced in Section 2.1. Briefly restated, a HIP stream is one of several arguments a programmer may specify when calling `hipLaunchKernelGGL`. Each HIP stream is backed by a software queue managed by ROCclr,[13] the backend runtime library used by HIP (see Figure 4.1). ROCclr stores the arguments to `hipLaunchKernelGGL` in a C++ object, then inserts this object into the software queue.

[13]This is largely defined in `platform/commandqueue.hpp` in ROCclr's source code.

HSA queues. Once a kernel-launch C++ object reaches the head of its software queue, ROCclr converts it into a kernel-dispatch AQL (Architected Queueing Language) packet. AQL packets are used to request single GPU operations, such as kernel launches or memory transfers.[14] In order to send the AQL packet to the GPU, ROCclr copies the AQL packet into an HSA queue. HSA queues are ring buffers of AQL packets, and are directly shared between the GPU and userspace memory. This direct memory sharing allows user programs to issue GPU commands without system calls (HSA Foundation 2018*b*, Sec. 2.5).

It may seem like an intuitive choice to back each ROCclr queue with a dedicated HSA queue, but Figure 4.2 likely already revealed that ROCclr's behavior is more complicated. ROCclr's software queues internally share a pool of HSA queues: one software queue may submit work to multiple different HSA queues, and each HSA queue may contain work from multiple software queues. Even though sharing HSA queues may occasionally prevent concurrent kernel launches, it will not break any of the ordering guarantees behind the top-level "stream" abstraction. ROCm employs a combination of hardware (*i.e.*, "barrier" AQL packets) and software mechanisms (*i.e.*, ROCclr's software queues) to enforce the in-order completion of commands from a single stream.

Figure 4.2 depicts fewer ROCclr software queues than HSA queues, but this is just to save space in the figure. In practice, using a shared pool of HSA queues is intended to reduce the total number of HSA queues created by an application; even if there are dozens of HIP streams, ROCclr will still use the same small pool of HSA queues. In the default configuration of ROCm 4.2, the pool is limited to four HSA queues.[15] Understanding the reason for this limitation, however, requires traveling farther down the scheduling hierarchy.

4.2.2.2 Assigning Queues to GPU Hardware

As mentioned previously, the contents of HSA queues (*i.e.* kernel-launch packets) can be directly shared between user applications and GPU hardware, so driver code is not necessary required when launching

[14] We do not cover memory-transfer requests further in this section, but they follow the same queuing structure as kernel launches. Ultimately, memory transfers are dispatched to hardware "DMA engines" (Bauman, Chalmers, Curtis, Freitag, Greathouse, Malaya, McDougall, Moe, van Oostrum and Wolfe 2019) rather than asynchronous compute engines.

[15] This, and related behavior can be observed by examining ROCclr's source code. For example, the `acquireQueue` function in `https://github.com/ROCm-Developer-Tools/ROCclr/blob/master/device/rocm/rocdevice.cpp` implements the functionality for selecting a single HSA queue from the pool of available queues.

any single kernel (this is why the driver is not shown in Figure 4.2). Even so, Linux's `amdgpu`[16] driver is required when initializing HSA queues and notifying the GPU of their existence. As such, driver code still maintains control over critical aspects of AMD GPU performance and reveals useful details about GPU scheduling internals, such as CU-masking specifics, discussed later in Section 4.2.3.1.

For now, though, we are concerned primarily with the functionality required to launch kernels. In the driver, this begins with the initialization of an HSA queue. Even here, a large portion of queue-creation logic is handled in ROCm's userspace code: the HSA API layer shown in Figure 4.1 is actually responsible for reserving the ring buffer for the HSA queue, setting up OS signals, *etc.*, via `mmap` and other standard Linux system calls.[17] Nonetheless, the driver must remain responsible for communicating this information to the hardware. Internally, it does so by populating a data structure called a *memory queue descriptor* (MQD), which includes the virtual address of the HSA queue's buffer, along with other metadata. MQDs are so named because they are allocated from GPU-accessible regions of CPU memory. In order for the GPU to actually start running work from the queues, however, MQDs must be assigned to *hardware queue descriptors* (HQDs) on the GPU itself.

In the default configuration for our test system, the `amdgpu` driver notifies the GPU about new queues by sending a *runlist* to the GPU—a buffer containing a list of all the MQDs on the system.[18] Interestingly, the act of "sending the runlist" itself requires writing the runlist to a special queue of GPU commands, known in driver code as the *HIQ* (HSA interface queue). The driver creates one HIQ for each GPU in the system, and, unlike HSA queues created in userspace, this queue of commands is mapped into kernelspace memory and is manually assigned to GPU hardware, allowing it to be initialized without needing to be part of the runlist itself.

Queues' arrival in hardware. Figure 4.3 gives a rough representation of the GPU hardware involved in compute workloads. As shown in Figure 4.2, *Asynchronous Compute Engines* (ACEs) are the hardware units responsible for processing the queues of AQL packets. Given its focus on kernels rather than queues, Figure 4.2 does not include the process by which MQDs are assigned to HQDs in the first place. Much of this

[16] When ROCm was first introduced, compute-specific code for AMD GPUs was instead in a separate `amdkfd` driver. It was merged into the `amdgpu` driver in of version 4.20 of the Linux kernel. While not particlularly relevant to this chapter's content, this distinction may be useful when consulting some of the older reference material we cite.

[17] For example, the `AqlQueue` constructor and related functions in the `ROCR-Runtime` library's `core/runtime/amd_aql_queue.cpp` source file are responsible for much of this low-level logic.

[18] As of Linux 5.14.0-rc3, source code for runlist construction is mostly contained in `drivers/gpu/drm/amd/amdkfd/kfd_packet_manager.c` in the Linux source tree.

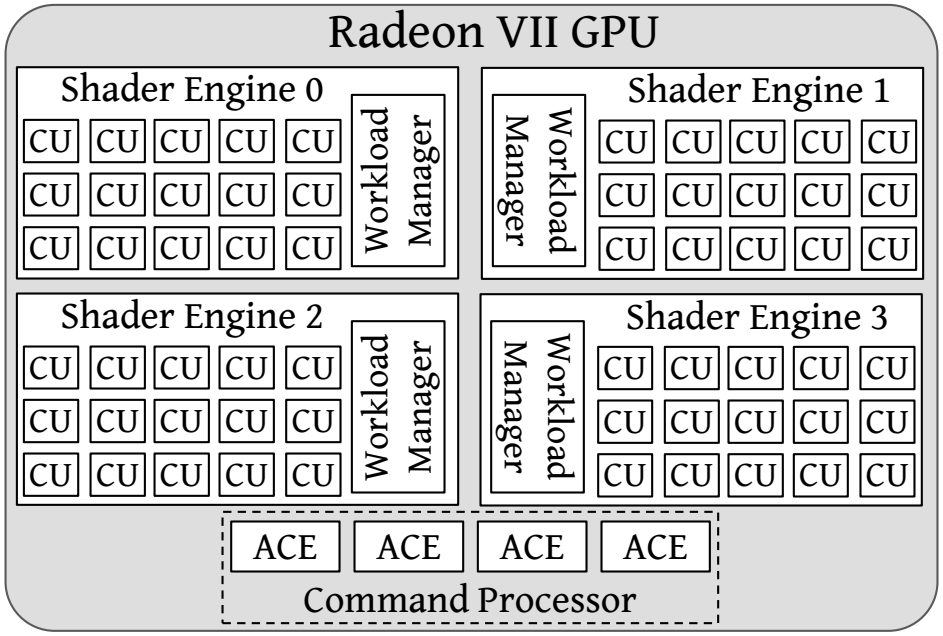


Figure 4.3: The Radeon VII's compute-related components.

process has little influence over the specific results shown in Section 4.2.1, but it still bears some explanation, as it ultimately illuminates the reason for ROCm's aforementioned attempt to reduce the number of active HSA queues.

Potential problem: hardware queue oversubscription. After receiving a runlist from the driver, firmware in the GPU's top-level *command processor* uses *hardware scheduling* (HWS) to assign MQDs to HQD "slots" in the ACEs.[19] The command processor contains four ACEs, and each ACE can support up to eight queues.[20] Knowing this, we can finally explain why ROCm attempts to limit the number of HSA queues used by any single application: the GPU hardware only supports up to 32 concurrent queues—eight queues on each of the four ACEs. Creating queues in excess of this limit leads to the GPU entering "oversubscription" behavior, which time-slices between the queues mapped to the 32 available slots. During oversubscription, queues are swapped without any sort of prioritization, meaning that queues with active work may even be swapped out of an HQD slot in favor of an empty queue.

[19]This is described in a comment in `drivers/gpu/drm/amd/include/kgd_kfd_interface.h` in the Linux 5.14.0-rc3 source tree.

[20]This can be confirmed in Linux 5.14 sources by observing where `num_pipe_per_mec` and `num_queues_per_pipe` are set in `drivers/gpu/drm/amd/amdgpu/gfx_v9_0.c`. Note that ACEs are typically called "pipes" in AMD's source code ("bridgman" 2016).

Naturally, this can lead to profoundly poor performance. To forestall any premature conclusions, though, oversubscription is not actually one of the bad practices through which we produced the data in Table 4.1. Puthoor, Tang, Gross and Beckmann (2018), AMD researchers, have already thoroughly investigated oversubscription in a prior publication, to which we refer readers interested in more details on the topic. Unfortunately, even Puthoor *et al.* resorted to a simulator when implementing alternative, more intelligent, approaches for multiplexing queues among the 32 HQD slots, leaving little chance that researchers unaffiliated with AMD could effectively conduct similar research on real hardware. Instead, in a real-time setting, we would recommend ensuring oversubscription is not a problem by disabling it: the `amdgpu` driver can easily be configured to enforce such a policy,[21] which will ultimately return errors to userspace if too many HSA queues are created.

How CU masks affect the queuing hierarchy. Internally, the GPU hardware associates a single CU mask with each HSA queue. If no mask is explicitly set, HSA queues default to allowing work to execute on any CU. Recall from before that HIP streams are backed by ROCclr software queues, which in turn submit work to a shared pool of HSA queues. The HSA queues in this shared pool are all created in the default configuration, and are therefore not suitable for use by any HIP stream that requires a non-default CU mask. In order to support CU masking, ROCclr follows a slightly different code path when handling a HIP stream with a CU mask, shown on the right side of Figure 4.2: it creates a separate HSA queue with the requested mask,[22] and uses the new HSA queue exclusively on behalf of the single stream.

4.2.2.3 Scheduling Thread Blocks

We now describe how a kernel at the head of an HSA queue gets assigned to computing hardware. Recall that thread blocks are the basic schedulable entity for GPU computations, so when kernel-dispatch AQL packets reach the heads of their queues, the question becomes how the GPU decides which blocks to run, and where to run them. Figure 4.4 essentially continues the kernel-launch process after the end of Figure 4.2, from the perspective of an ACE handling a single HSA queue. To simplify Figure 4.4, we only included

[21] This is configured by the `sched_policy` parameter to the `amdgpu` driver, defined in `drivers/gpu/drm/amd/amdgpu/amdgpu_drv.c`.

[22] This behavior can be observed in the `acquireQueue` function defined in ROCclr's `device/rocm/rocdevice.cpp` source file.

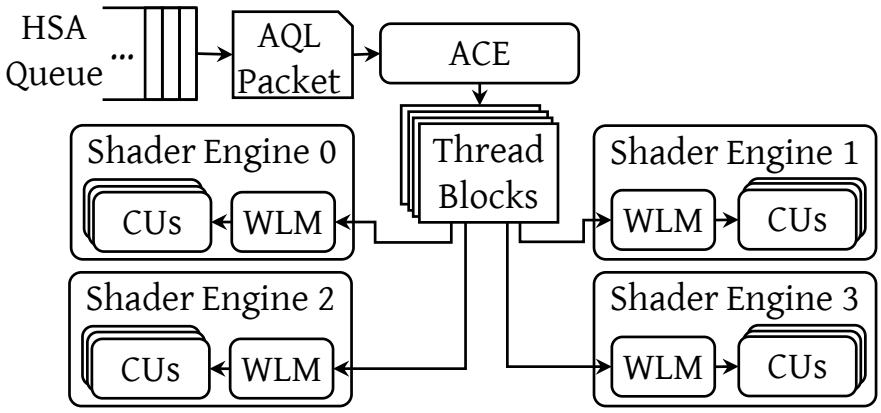


Figure 4.4: Hardware involved in dispatching blocks to CUs. (This figure abbreviates "Workload Manager" as WLM.)

a single HSA queue and a single ACE. If multiple HSA queues are assigned to the same ACE, the ACE alternates between dispatching packets from the head of each queue in a time-sliced round-robin fashion.

Dispatching blocks to shader engines. One of the more prominent features in Figures 4.3 and 4.4 is the division of the GPU's compute resources into four shader engines (SEs). As illustrated in Figure 4.4, the primary role of an ACE is to dispatch blocks from the kernel at the head of an HSA queue to the SEs. However, without a prior explanation of the reasons underlying certain design decisions, the ACE's behavior when dispatching blocks to SEs may seem bizarre. To forestall such confusion, we first describe a thread-ordering guarantee made by the HSA specification, which AMD implements in their GPU-compute architecture (HSA Foundation 2018*a*, Section 2.13).

The HSA specification states that it must be safe for GPU threads to *wait* for the completion of any GPU threads with a lower block index, *i.e.*, the value provided by `blockIdx.x` in Figure 2.1. Technically, block indices are three-dimensional tuples, so the HSA specification actually states its guarantee in terms of a block's *flattened ID*, which takes into account the fact that the special `blockIdx` variable is three-dimensional. For example, it must be safe for threads in block 1 (specifically, the block with a flattened ID of 1) to wait for threads in block 0 to complete, but it may be unsafe for a threads in block 0 to wait for block 1's completion—block 0 could occupy resources needed by block 1, preventing block 1 from ever starting to execute. A block-ordering guarantee has a practical application: prior work formally proves that it enables producer-consumer relationships between blocks in a single kernel (Sorensen, Evrard and Donaldson 2018).

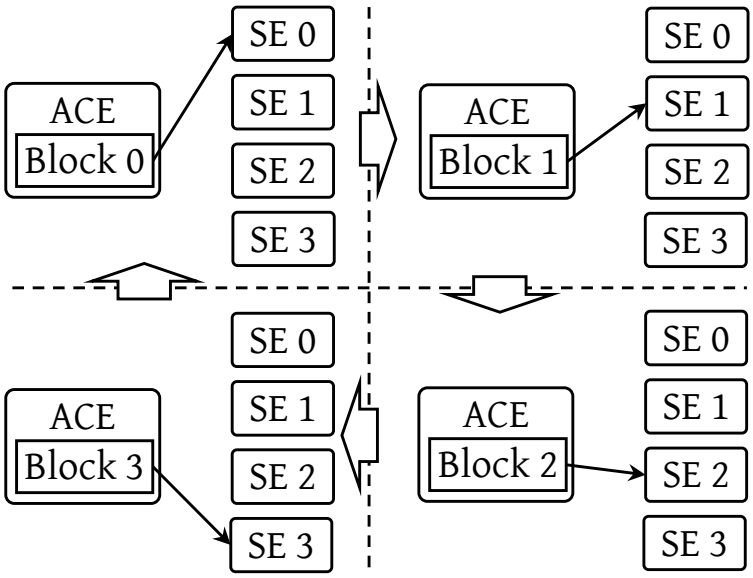


Figure 4.5: A simplified diagram of an ACE's behavior when dispatching consecutive blocks to SEs.

Even though our own experiments do not use such complicated kernel logic, the block-ordering guarantee plays an important role in scheduling, with significant performance ramifications.

AMD hardware enforces the block ordering when assigning blocks to SEs. The method is simple: ACEs *must* assign blocks to SEs in sequential order. For example, an ACE cannot assign block 1 to SE 1 until after it has assigned block 0 to SE 0.[23] Figure 4.5 illustrates this concept as the ACE dispatches four consecutive blocks to SEs. The cycle depicted in Figure 4.5 continues with block 5 being assigned to SE 0, and only ends after all blocks in the kernel have been dispatched. To use a metaphor: the block-dispatching behavior can be likened to a card game where a dealer is dealing cards to four players. As in most real-life card games, even one slow player may force the dealer to wait, slowing down the entire game! Nonetheless, it would ruin the game for the dealer to skip the slow player. When applying this to AMD GPUs, the "dealer" is the ACE, the four "players" correspond to the four SEs, and the "cards"are the blocks of a kernel. But what can cause an SE, the metaphorical "player," to be abnormally slow? The answer is intertwined both with CU masking, and with the behavior of the workload managers.

The role of workload managers. In order for an ACE to assign a block to an SE, the block must be assigned to a specific CU on that SE. As shown in Figures 4.3 and 4.4, assigning blocks to CUs is the job of

[23] Our only source for this claim remains private correspondence, which indicated that hardware enforces this rule using a "baton-passing" mechanism between the SEs. Despite the lack of additional external support for this claim, it is certainly well-supported by our experiments, *i.e.*, Table 4.1 or Figure 4.9.

a piece of per-SE hardware called the *workload manager* (Bauman *et al.* 2019). Each workload manager has four dedicated "slots" for staging incoming blocks: one slot dedicated to each of the GPU's four ACEs. This design means that activity from one ACE cannot prevent another ACE from accessing the workload manager. Ideally, workload managers will assign blocks from each of these four slots to CUs in a round-robin manner, prioritizing assigning a block to a CU from whichever kernel least recently had a block assigned.[24] This behavior changes, however, if no CU has sufficient available resources for a block from a particular slot (the concept of *occupancy* discussed in Section 3.4 is essentially identical between NVIDIA and AMD). In a striking contrast with our Section 3.4.1 experiment on NVIDIA GPUs, AMD GPUs' workload managers will allow blocks with smaller resource requirements to *cut ahead* if possible. This finally allows us to explain why, in Section 4.2.1, `MM256` was always more destructive to `MM1024`'s performance than any other configuration: each CU only supports a limited number of threads, and the completion of a 256-thread `MM256` block does not release enough resources to allow a block of `MM1024` to run. Instead, it merely allows another `MM256` block to cut ahead again, causing the problem to continue until no more `MM256` blocks remain!

Figures 4.6 and 4.7 illustrate this behavior on real hardware. The timelines correspond to the same workloads from the "Full GPU Sharing" configurations from the "`MM1024` (vs. `MM1024`)" and "`MM1024` (vs. `MM256`)" portions of Table 4.1. In order to generate the timelines, we instrumented our kernel code to use the `clock64()` function to obtain the start and end GPU clock cycle for every block of threads (sadly, as mentioned, the convenient `globaltimer` register is not available on AMD GPUs). After a kernel completes, we copy the block start and end times to the CPU and use them to compute the number of active threads at each GPU clock cycle. The timelines in Figures 4.6, 4.7a, and 4.7b all cover a single kernel's execution for each respective task.

As expected, the kernel running in Figure 4.6 uses the GPU at near-full capacity for its entire duration, with an obvious plateau at 122,880 active threads corresponding to 2,048 threads running on each of the GPU's 60 CUs. The occasional spikes likely coincide with groups of blocks nearing the end of their execution, as, unlike on NVIDIA (Section 3.4.2), our Radeon VII allows new blocks to start as soon as resources start to become available, even if the entire preceding block has not yet completed. To avoid the high overhead for tracking the start and end times of each individual thread, we only record the start of the first thread and end

[24] Unfortunately, we also learned this from private conversation and, despite being hinted in a presentation from van Oostrum, Chalmers, McDougall, Bauman, Curtis, Malaya and Wolfe (2019), we were not able to find clear corroborating evidence in published material. Nonetheless, this claim is supported by the observations in Figure 4.7a.

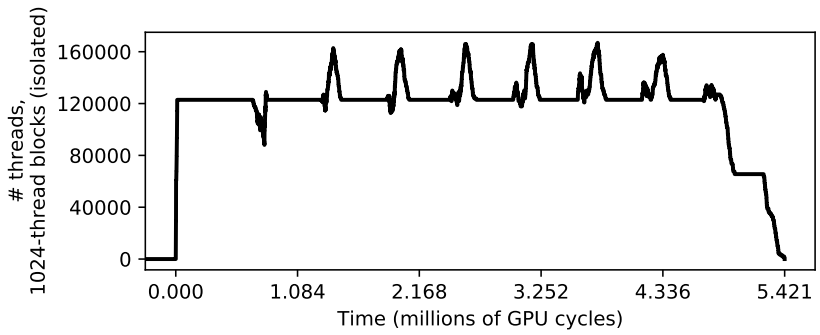


Figure 4.6: An isolated `MM1024` kernel.

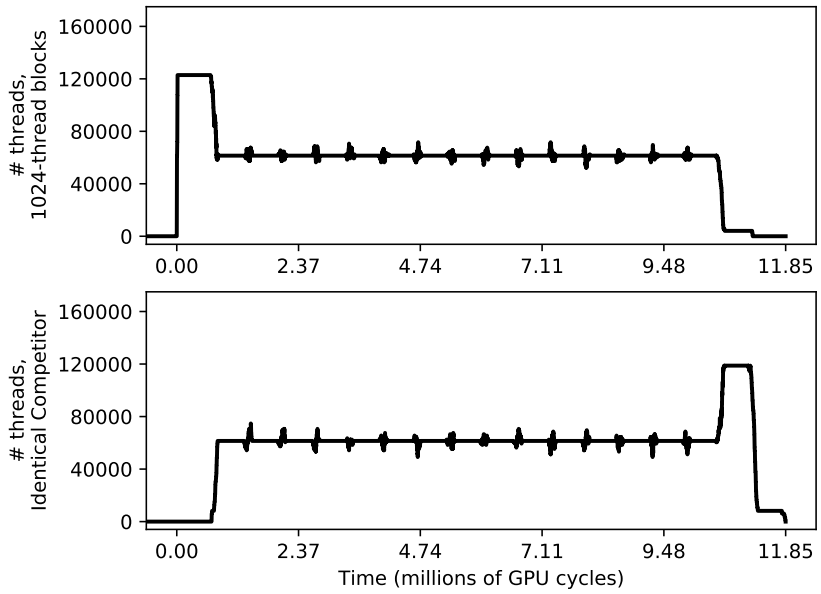
of the last thread in each block. However, we plot timelines as if all of a block's threads remain active so long as *any* thread in the block is active, leading to spikes where active thread or block counts are inaccurate. Fortunately, the behavior shown in Figure 4.7 is distinctive enough to be obvious even without perfectly tracking the active-thread count.

The cutting-ahead behavior is apparent when comparing Figures 4.7a and 4.7b. When two identical `MM1024` instances contend for GPU resources, Figure 4.7a shows that GPU computing capacity is divided evenly for the entire time that the two kernels overlap, corresponding to the workload managers dispatching blocks evenly from separate ACEs. Additionally, when the kernels do not overlap, the sole running kernel uses the full capacity. The contrast provided by Figure 4.7b is striking, where `MM1024` competes against an `MM256` kernel. Shortly after `MM256`'s kernel begins, it has taken sole control of virtually all GPU resources, with `MM1024` making practically no progress in the meantime.

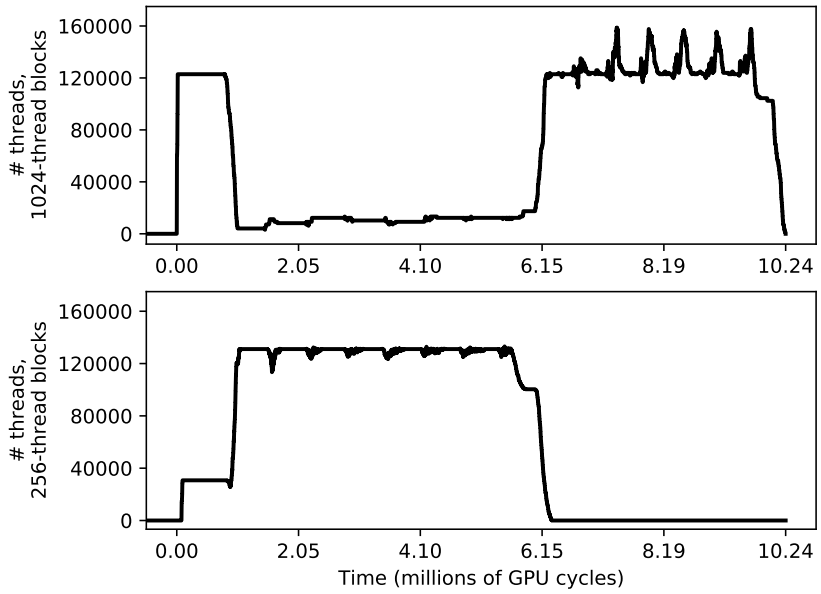
4.2.2.4 Explanation of the Worst Practices in Section 4.2.1

"Cutting ahead" explains the destructive interference that `MM256` causes against `MM1024`, but the terrible performance of "Uneven Semi-Partitioning" in Section 4.2.1 is, perhaps unsurprisingly, better attributed to a poor choice of CU partitions.

With the prior explanation of block-SE distribution, this behavior is now easier to explain. Recall that the ACE will always distribute blocks to SEs in sequential order, and will never skip an SE. There is one exception to this rule: the ACE will skip an SE only if a CU mask disables *all* CUs on that SE. In other words, blocks are evenly distributed among the set of SEs for which *any* CUs are enabled. This approach to block-ordering enforcement can lead to extreme performance pitfalls: If only one CU is enabled on an SE, it is far more difficult for an ACE to assign a block to that particular SE, as it must wait for the single CU to be



(a) Two overlapping `MM1024` kernels.



(b) Overlapping `MM1024` and `MM256`.

Figure 4.7: Comparison between timelines of matrix-multiply thread blocks in different configurations.

available. On top of this, the ACE is also prevented from "skipping" the slow SE and assigning blocks to the other SEs in the meantime! We exploited this behavior for our Section 4.2.1 experiments, by designing a CU mask that enables only one CU on an SE. Naturally, performance becomes even worse when the single CU is shared with a kernel where blocks can cut ahead: this is precisely what happens in Table 4.1 under "Uneven Semi-Partitioning" when `MM1024` competes against `MM256`.

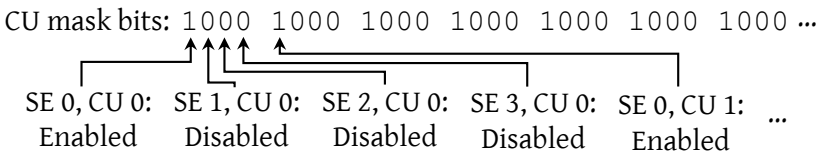


Figure 4.8: The mapping of CU mask bits to SEs.

4.2.3 Practical Considerations About AMD GPU Scheduling

In this section we give some guidance for using CU masking in practice.

4.2.3.1 Usage of the CU-Masking API

The primary practical method for using CU masking on AMD GPUs is through the HIP API, so the relevant HIP API details bear some more explanation. In order to specify a CU mask for a HIP stream, programmers must use the `hipExtStreamCreateWithCUMask` function (HIP offers no function to set a CU mask for an existing stream, due to the HSA-queue multiplexing discussed in Section 4.2.2.1). In HIP, CU masks are specified as bit vectors using 32-bit integers, where set bits indicate enabled CUs and clear bits indicate forbidden CUs.

The hardware, however, does not use the same "flat" CU mask that a HIP programmer specifies, and instead requires a separate CU mask for each SE. In fact, the `amdgpu` driver is responsible for transforming the single user-provided CU mask into the per-SE masks. By examining the driver code,[25] we can discover the specific mapping. Figure 4.8 shows how bits in a HIP CU mask relate to shader engines and CUs in the GPU. The pattern in Figure 4.8 is simple: every fourth bit maps to a different CU in the same SE. In the example mask in Figure 4.8, the first of every group of four bits is set, defining a partition consisting only of CUs on SE 0.

Knowing the mapping between HIP's CU masks and SEs in hardware allows partitioning tasks to specific SEs, but it is not clear if doing so has benefits over an approach that distributes CUs evenly across SEs. In order to evaluate the possible benefits and drawbacks of limiting partitions to specific SEs, we contrast two basic partitioning approaches:

[25] In the source tree for Linux 5.14.0, this is found in the `mqd_symmetrically_map_cu_mask` function in `drivers/gpu/drm/amd/amdkfd/kfd_mqd_manager.c`.

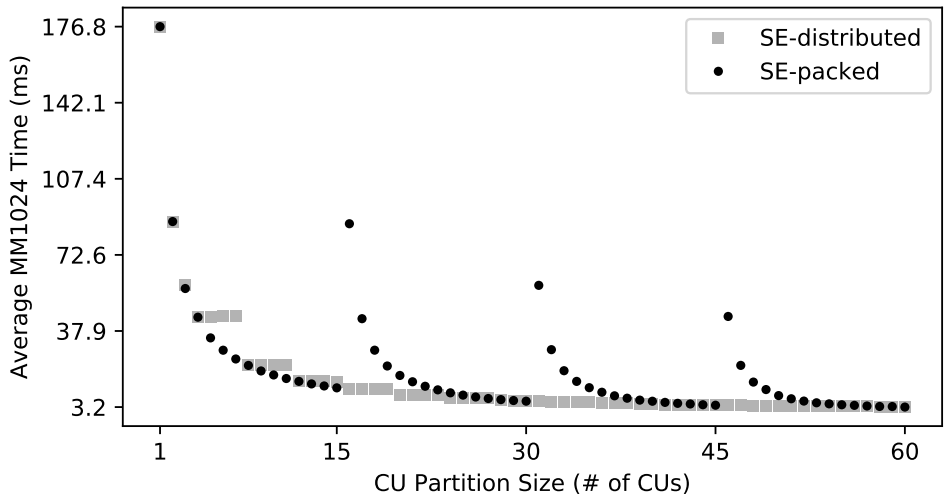


Figure 4.9: Performance of CU-masking strategies for varying partition sizes.

- *SE-packed*: Pack as many of the partition's CUs as possible into each single SE before starting to occupy CUs on an additional SE. This uses as few SEs as possible.
- *SE-distributed*: Distribute the partition's CUs across all SEs as evenly as possible. This implies that any partition containing four or more CUs will use all SEs.

Figure 4.9 shows the behavior of these two partitioning approaches when applied to an instance of `MM1024` running in isolation. For the most part, Figure 4.9 seems to show that there is little benefit to SE-packed partitioning: SE-packed visibly outperforms SE-distributed only for some partition sizes smaller than 15 CUs. Unsurprisingly, SE-packed performs far worse than SE-distributed when only one CU is enabled on an SE. The jumps in the SE-packed response times occur where expected: the Radeon VII has 15 CUs per SE, so SE-packed partitions with sizes of 16, 31, or 46 will end up occupying only a single CU on one of the SEs. Overall, SE-packed partitioning may look inferior in Figure 4.9, but Figure 4.9 is based on measurements taken without an important factor: contention.

CU partitioning in the presence of a competitor. We carried out a final experiment using an `MM1024` measured task facing an `MM256` competitor (as this results in the most destructive interference in the absence of partitioning). This experiment partially reuses data from Section 4.2.1, which used SE-packed partitioning. The main contribution from the new experiment is the inclusion of SE-distributed partitioning for contrast.

Table 4.2 shows the results of this experiment. For further illustration, Table 4.2 also includes the CU mask used by both the `MM1024` measured task and `MM256` competitor. For example, `MM1024`'s SE-packed

Description	MM1024 CU Mask	MM256 CU Mask	Min	Max	Median	Arith. Mean	Std. Dev.
Full GPU Sharing (Unpartitioned)	1111...1111	1111...1111	12.214	19.578	15.503	15.652	1.114
SE-packed, Equal Partitions	1010...1010	0101...0101	6.784	7.630	6.944	6.991	0.133
SE-packed, Unequal Partitions	1010...1011	0101...0101	64.873	101.531	84.047	84.362	4.381
SE-distributed, Equal Partitions	1111...0000	0000...1111	6.391	9.113	7.250	7.269	0.074
SE-distributed, Unequal Partitions	1111...0001	0000...1111	7.109	7.906	7.288	7.324	0.122

All times are in milliseconds.

Table 4.2: MM1024's response times in the presence of an MM256 competitor.

CU mask sets every even-numbered bit, starting with bit 0, meaning that MM1024 will occupy every CU on SEs 0 and 2, whereas MM256's SE-packed CU mask causes it to occupy every CU on SEs 1 and 3. As we did in Section 4.2.1, we also include "Unequal Partitions" cases in Table 4.2, produced by adding a single CU to MM1024's "Equal Partitions" CU masks.

Observation 4.4 *For some partition sizes, SE-packed partitioning is slightly better than SE- distributed partitioning.*

Observation 4.4 is seen when comparing the two "Equal Partitions" rows from Table 4.2. Both partitioning approaches ensure that MM256 does not share CUs with MM1024, and therefore sufficiently prevent the poor unpartitioned performance due to MM256 cutting ahead (shown for convenience in Table 4.2's first row). However, SE-packed partitioning exhibits slightly better response times than SE-distributed, likely due to two factors. First, SE-distributed partitioning is actually unable to assign an equal number of CUs to all SEs, as a 30-CU partition is not evenly divisible among four SEs. Second, SE-packed CU masks may prevent a small amount of contention for SE-wide resources such as workload managers.

Observation 4.5 *SE-distributed partitioning is vastly superior when a partition's size prevents it from occupying all CUs in an SE.*

Observation 4.5 is supported by comparing the "Unequal Partitions" lines in Table 4.2, where it should be apparent that it is highly undesirable to use an SE-packed partition containing 31 CUs. While it is hard to imagine a practical application where a task requires a partition containing exactly 31 CUs, one can certainly envision applications where greater flexibility in partition sizing could be useful. One such situation would be a task system requiring prioritization. For example, high-priority work may need a larger partition than low-priority work, but designating a full additional SE as "high-priority" could cause unacceptable adverse effects to low-priority performance. Instead, it could be better to allocate 40 CUs to high-priority

work, while low-priority tasks use the remaining 20 CUs. With these partition sizes, Figure 4.9 indicates that SE-distributed partitioning would be better than SE-packed partitioning for both high- and low-priority tasks, with low-priority work seeing a particular benefit given the gap between SE-distributed and SE-packed performance at 20 CUs.

4.3 Drawbacks and Pitfalls When Using AMD GPUs

As already shown in Section 4.2, AMD GPUs can behave in some counterintuitive ways and, despite using open-source software, face their own set of documentation woes. In this section, we return to some of the main pitfalls exposed in Section 4.2, and enumerate other non-behavioral difficulties that researchers targeting AMD GPUs are likely to face. Our intent in stating and describing pitfalls and workarounds is to provide guidance for future researchers who may, like us, end up facing questions about research platforms, or need to diagnose confusing performance characteristics of AMD GPU software.

4.3.1 Hardware-Related Behavioral Pitfalls

The material from Section 4.2 already covered several significant hardware-related pitfalls on AMD GPUs, but the discussion of each of these pitfalls was brief, as Section 4.2 focused mainly on *scheduling* rather than *developing real-time software*. Second, it is worth revisiting key pitfalls, as they all serve a dual purpose of highlighting key differences between the ways in which NVIDIA GPUs and AMD GPUs behave. In fact, the first pitfall we mention may not be particularly dangerous from a timing perspective, but has interesting implications for how sharing AMD and NVIDIA GPUs may need to be managed differently:

Pitfall 4.1 *Creating too many queues on AMD GPUs leads to "oversubscription," significantly harming performance.*

We mentioned Pitfall 4.1 in Section 4.2.2.2: AMD GPU hardware has only 32 "slots" to which queues of work may be concurrently assigned. As this did not impact our experiments from that section, we left further discussion to prior research from a different group (Puthoor *et al.* 2018), but oversubscription may remain an important consideration for real-time systems involving a larger number of independent tasks.

Pitfall 4.1 should not be a surprise when it does occur, as oversubscription can simply be disabled by configuring the `amdgpu` driver. With the driver configured to return errors rather than allow oversubscribed hardware, basic tests should quickly reveal oversubscription behavior, making it easy to avoid triggering by

mistake in deployed safety-critical systems. On the other hand, simply catching the mistake does not mean the problem is solved, and the limit of 32 queues[26] is deceptively small, especially with the default behavior of allocating a separate queue to each stream requiring a CU mask, and the fact that separate processes are unable to share queues.[27] These circumstances may even serve as motivation for designing multiple GPU tasks that share a single Linux process on AMD GPUs, similarly to how we structured our experiments using NVIDIA GPUs in Section 3.4. Ironically, this implies that the motivation for structuring tasks as threads within a single process may be the exact opposite on NVIDIA and AMD GPUs. On NVIDIA GPUs, multiple threads within a single process *enabled* more hardware sharing. On AMD GPUs, however, multiple threads in a process may be useful for *preventing* too much sharing.

Pitfall 4.2 *Users of CU masking must be cognizant of the mapping between CUs and SEs.*

We certainly do not count Pitfall 4.2 as a strike against AMD GPU usage, at least in competition with NVIDIA, as most consumer-grade NVIDIA GPUs offer no comparable hardware-partitioning support at all. Nonetheless, the conclusions from Section 4.2.2.4 make Pitfall 4.2 obvious—designing CU partitions on AMD GPUs can lead to disastrous performance in many circumstances. We also neglected to mention one complicating factor of Pitfall 4.2 in the previous section: different AMD GPUs may require different partitioning strategies, particularly as the number of CUs per SE may differ. Fortunately, unless AMD changes their CU-masking driver code, using SE-distributed partitioning as described in Section 4.2.3.1 makes it easier to avoid severe problems, at the cost of only slight reductions to efficiency in some cases.

Pitfall 4.3 *On AMD GPUs, smaller thread blocks can cut ahead of larger thread blocks.*

On one hand, AMD's decision to allow cutting ahead is understandable: it clearly allows greater hardware utilization in some situations where full utilization may not occur (such as our cutting-ahead experiment on NVIDIA in Section 3.4.1). On the other hand, the fact that thread blocks with smaller resource requirements can (and will) cut ahead of other thread blocks on AMD GPUs could easily lead to some tasks consistently experiencing starvation, as `MM1024` did in our Section 4.2.1 experiment. Pitfall 4.3 is made more insidious by the fact that it is not possible on NVIDIA hardware (Section 3.4.1): without knowing this pitfall, attempts to port real-time software from NVIDIA to AMD GPUs could easily result in unsafe changes to timing

[26]The limit on the number of queues may be even smaller in practice, as some queues may be reserved to support graphical operations.

[27]The `amdgpu` driver associates the address space of a Linux process with an address space on a GPU. This is required to prevent one process from corrupting another's GPU memory.

characteristics. It has been common practice for AMD to port CUDA software to HIP using automated tools[28] that, at least so far, do not take such concerns into account.

Pitfall 4.4 *AMD GPU code launched from separate Linux processes is not temporally isolated.*

The ACEs in an AMD GPU's command processor (shown in Figure 4.3) can and will concurrently schedule work from independent Linux processes. (As a reminder, our matrix-multiply tasks were independent processes in Section 4.2.1.) Purely from a hardware-utilization standpoint, this is a benefit—unlike NVIDIA GPUs, AMD does not require a service like MPS to allow true concurrency. In other circumstances, however, this behavior certainly qualifies as a pitfall, especially in conjunction with Pitfalls 4.1 and 4.3. For example, even aside from the cutting-ahead problems, other GPU hardware resources such as memory bandwidth may be adversely effected by some combinations of competing tasks. NVIDIA's default time-sliced management of inter-process GPU sharing avoids such interference, but enforcing similar time-sliced access on AMD would require additional management. This may even motivate reviving older real-time research targeting NVIDIA GPUs, such as the approaches listed in Section 2.6.1, to provide temporal isolation for AMD GPU sharing.

4.3.2 Software-Related Pitfalls

Like their NVIDIA counterparts, research targeting AMD GPUs will inevitably encounter difficulties stemming from the availability of documentation. However, much more common to AMD are issues stemming from software stability and available hardware support:

Pitfall 4.5 *ROCm only supports a limited number of discrete GPUs.*

Pitfall 4.6 *ROCm only supports the Linux operating system.*

Pitfall 4.5 is likely to be the first difficulty researchers unfamiliar with AMD GPUs are likely to encounter when using the platform. The ROCm software supports a very small number of GPUs, and support frequently lags behind the release of new GPUs. For example, the successor architecture to GCN, RDNA, was first released in 2019, but to this day, only two high-end RDNA GPUs support running ROCm, and ROCm only

[28] For example, the process for compiling PyTorch on AMD GPUs requires running a script to automatically convert CUDA source files to HIP: `https://github.com/pytorch/pytorch/blob/226a5e87f39564ecd8268c37350942c5129ae2b0/README.md?plain=1#L231`.

started supporting these GPUs nearly a year after the first RDNA-architecture GPU was released. At the time of writing, AMD's list of ROCm-supported GPUs only states that ten GPUs are "fully supported," and even these ten include several GPUs that are no longer in production.[29] Along with a lack of embedded offerings, this support stands in stark contrast with CUDA, which has been supported on virtually all NVIDIA GPUs from the past decade.

Pitfall 4.6 goes in hand with Pitfall 4.5. While not a barrier to using AMD GPUs as a research platform, ROCm's lack of widespread adoption can make it difficult to claim that any AMD-based GPU research is broadly applicable. Fortunately, Pitfall 4.6 may be somewhat mitigated by the rapid pace at which AMD publishes major updates to ROCm software, which, through a different lens, may be viewed as a pitfall of its own.

Pitfall 4.7 *ROCm software undergoes frequent, major changes.*

One can hardly fault AMD for the fact that ROCm continues to be under active development. The fact that continually updating the software is desirable for most users does not, however, negate the inconvenience for researchers. Our work using AMD GPUs started slightly over three years ago, with ROCm version 2.6. Since that time, our research has weathered multiple major and minor updates to ROCm's software. Many minor changes only added support for new hardware or improved stability, but other changes involved major restructuring. For example, prior to ROCm version 3.5, an entire "layer" of Figure 4.1 was different: rather than using LLVM and ROCclr to compile and run HIP code, HIP was backed by a separate compiler project, known as HCC (the Heterogenerous-Compute Compiler). HCC was deprecated in 2019, and entirely removed from ROCm shortly thereafter. Unsurprisingly, AMD's decision to replace an entire compiler and runtime layer of the software stack required reimplementing several of our own experimental modifications.

Eventually, we chose to avoid Pitfall 4.7 in the most straightforward manner: we stopped updating our ROCm installation. While our earlier efforts involved several versions of ROCm, the material in this dissertation, including Chapters 4 and 5, use ROCm version 4.2. Though ROCm has yet to undergo any changes as significant as the removal of HCC between versions 3.3 and 3.5, more large changes are yet to come. For example, even though ROCm version 5.1 (current at the time of writing) still does not support

[29] This can be seen by checking AMD's list of supported GPUs at `https://docs.amd.com/bundle/Hardware_and_Software_Reference_Guide/page/Hardware_and_Software_Support.html`.

Microsoft Windows, AMD's ongoing changes to some components of the software stack indicate that such support is under active development.[30]

4.4 Chapter Summary

In this chapter, we investigated the behavior of AMD GPUs, to establish whether they may serve as a viable alternative platform to NVIDIA in the context of real-time GPU sharing. In doing so, we provided information about AMD GPUs using a combination of public (but poorly advertised) information, first-party source code, and experimental evidence. Our primary goal was to identify situations where AMD GPU behavior may diverge from NVIDIA, but we also gave guidance on how to properly apply the CU-masking feature of AMD GPUs.

Establishing the "best" platform for any complex system will always involve a set of tradeoffs and personal preferences, but it is not necessary to prove that a system is the "best" in order for it to be a viable research platform. In a research setting, the question is still open as to whether the drawbacks discussed here for AMD GPUs are an acceptable trade for the benefits offered by an open-source GPU software stack and control over CU masking.

This chapter provided several clear demonstrations of the practical implications of knowing or failing to know key aspects of GPU-internal behavior. Without knowing this information, the risks of disastrous performance pitfalls make developing a reliable real-time system using AMD GPUs virtually impossible. But, from another view, *with* knowledge of this information, developers can expect predictable performance from their systems with a level of confidence that will not be present with a closed-source platform.

Acknowledgements. The material in this chapter was drawn from three papers published in the years 2020 through 2022. The content of Section 4.2 largely appears in two papers: Otterness and Anderson (2021) and Otterness and Anderson (2022). (The second of these papers is an extended version of the first, for journal publication.) The material in Section 4.3 and some of Section 4.1 was drawn from Otterness and Anderson (2020).

[30]For one of many such examples, this commit adds Windows support to the ROCclr library's compilation scripts: `https://github.com/ROCm-Developer-Tools/ROCclr/commit/df1449608e92c`.

CHAPTER 5: DEVELOPING SOFTWARE FOR REALISTIC REAL-TIME GPU EVALUATION

Much GPU-management research implicitly adopts an assumption from a long legacy of real-time research: that hardware-management techniques can be developed in a manner orthogonal to the applications being run. Unfortunately, as we stated in Chapter 1, control over hardware is only one part of the difficult new reality facing real-time GPU usage—of equal importance is the shift to complex neural-network-based applications.

5.1 The Behavior of Modern GPU-Accelerated Neural Networks

Over a decade ago, *AlexNet* (Krizhevsky, Sutskever and Hinton 2012), and the contemporaneous work by Ciresan, Meier, Masci, Gambardella and Schmidhuber (2011), were implemented from scratch using CUDA. Today, almost all neural-network-based computer-vision research is carried out using high-level neural-network development frameworks, such as those discussed in Section 2.3.2. While neural-network developers may occasionally write lower-level CUDA code to add specific functionality to high-level frameworks, current computer-vision research seems to have shifted focus away from the accelerated code itself, leaving further optimizations to other fields of study. Instead, new computer-vision and AI research tends instead to favor studying the structure of neural-network architectures at a higher level, while relying on the increasingly sophisticated collection of open-source libraries and tools to implement them. In this section, we examine the behavior of one of the most popular modern neural-network frameworks, *PyTorch*, and its behavior running a variant of the influential *MobileNetV1* neural network (Howard, Zhu, Chen, Kalenichenko, Wang, Weyand, Andreetto and Hartwig 2017).

5.1.1 Overview of PyTorch Execution

We already introduced PyTorch, and its popularity in the field of computer vision, in Section 2.3.2. PyTorch's complexity and popularity make it an ideal case study for modern GPU-accelerated applications. Regardless of opinions as to whether such applications *should* be included in safety-critical contexts, it is a

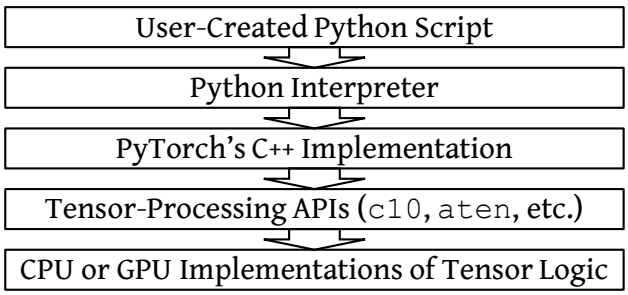


Figure 5.1: Basic representation of the layers of abstraction involved in a PyTorch application.

mistake for anybody claiming to do real-time GPU research to not have a modest understanding of the tool with which computer-vision researchers are developing the next generation of safety-critical applications. Understanding the entirety of of PyTorch's several hundred thousand lines of source code is beyond the scope of a single dissertation. Nonetheless, it remains within our capabilities to at least understand the layers of abstraction that arise in the framework.

Layers of abstraction. While the distinctions between abstractions in a single application are not as clear-cut as they are between the components in the visually similar Figure 4.1, we still provide Figure 5.1 as a visual reference for the components of PyTorch involved with launching GPU computations.

At the top of Figure 5.1 is the Python interpreter, obviously needed when executing Python scripts. The interesting part of Figure 5.1 begins when the user's script starts to interact with the PyTorch libraries. The bulk of PyTorch is implemented using the C++ programming language. Even though the PyTorch codebase includes many Python-language files, most of these are thin wrappers providing syntactic sugar around invocations of functions contained in native C++ libraries. Within PyTorch's C++ code base lie two prominent, interwoven tensor-processing libraries: `c10` and `aten`.[1] In PyTorch, almost every mathematical operation operates on one or more *tensor* arguments: multi-dimensional arrays of data. For example PyTorch uses its `Tensor` data type to hold neural-network weights, input data, *etc.* PyTorch is designed to make it easy for users to write both CPU-only and GPU-accelerated code with minimal changes. Implementing this feature requires the separation between the bottom two layers of Figure 5.1: the tensor-processing libraries provide a common interface backed by interchangable CPU-only and GPU-accelerated implementations.

[1]According to a PyTorch forum post (`https://discuss.pytorch.org/t/whats-the-difference-between-aten-and-c10/114034`), the `aten` library is older, and much of its functionality has been superseded by the newer `c10` code. However, the full integration of these two libraries remains incomplete, and both remain heavily used even in basic operations, as illustrated by the presence of both the `c10` and `at` namespaces in Figure 5.3.

```
import torch
a = torch.rand((1000, 1000, 100), device="cuda:0")
b = torch.rand((1000, 1000, 100), device="cuda:0")
c = a + b
```

Figure 5.2: Python code using PyTorch to perform GPU-accelerated addition of two tensors.

PyTorch scripts can easily combine or switch between CPU and GPU computations at runtime. We discuss this further in the context of the following example.

Using PyTorch for GPU-accelerated math. We provide Figure 5.2 to show why it can be so useful for developers to use high-level scripts, even for performance-sensitive work like neural networks. Figure 5.2 is a Python snippet that uses the PyTorch library (`torch`), to allocate two tensors containing 100 million ($1000 \times 1000 \times 100$) floating-point values each. The `torch.rand` function randomly initializes the contents of each tensor, and the `device="cuda:0"` argument indicates that the tensors are to be allocated in the memory of the first GPU on the system (index 0).[2] Finally, the `c = a + b` line instructs PyTorch to add the contents of the two "input" tensors, storing the results in a new tensor, `c`. Since both `a` and `b` are in GPU memory, PyTorch will automatically allocate space for `c` from GPU memory as well, and invoke a GPU kernel to perform the addition.

In contrast to the HIP or CUDA vector-add examples from Figures 2.1 or 2.2, the draw of writing code like Figure 5.2 is obvious. PyTorch allows developers to avoid writing kernel code or performing the memory-allocation and memory-transfer boilerplate present in Figures 2.1 or 2.2, which already omit some boilerplate like initializing vector contents. Being a minimal example, Figure 5.2 does not enqueue operations in a non-default HIP or CUDA stream, but other PyTorch API functions make it possible to do so with very little additional complexity. Finally, even though Figure 5.2 uses the GPU, converting it to perform computations on the CPU is trivial: removing the `device="cuda:0"` argument from the tensor allocations will cause PyTorch to place the tensors in CPU memory and perform addition on the CPU by default.

From Python scripts to the GPU. With an overview of the salient PyTorch components in Figure 5.1 and a simple code snippet in Figure 5.2, we are able to obtain a more precise picture of the abstractions involved when ultimately launching a kernel to carry out a single addition operation with PyTorch. By

[2]For compatibility reasons, PyTorch scripts use the term "cuda" to specify all GPU-accelerated operations, even ones using HIP on AMD GPUs.

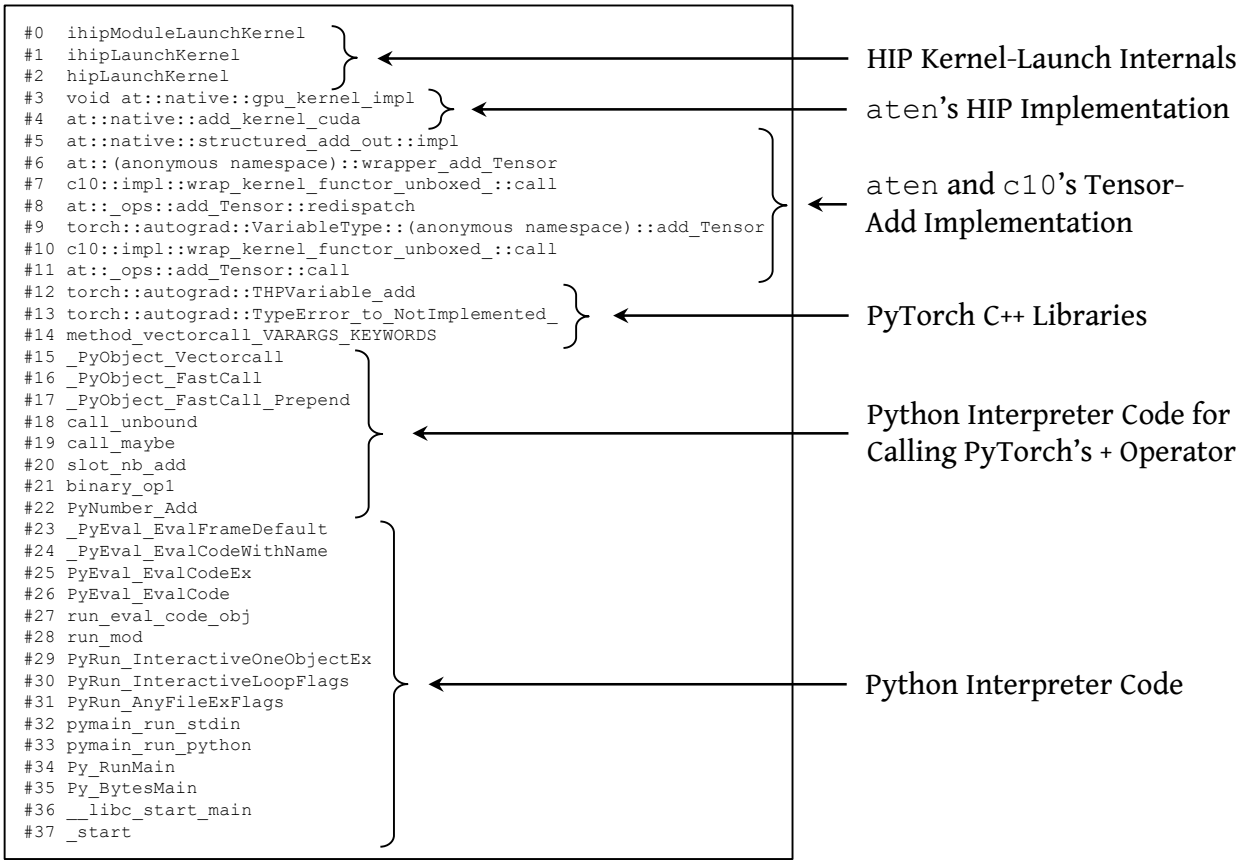


Figure 5.3: A concrete illustration of Figure 5.1: the call stack of functions leading to a kernel launch when PyTorch adds two tensors using the GPU. (This call stack grows upward, with the highest-level function, the Python interpreter's `_start` routine, appearing at the bottom.)

attaching the `gdb` debugger to the Python process executing the code in Figure 5.2, we obtained a backtrace of the C and C++ functions involved when PyTorch invokes the tensor-addition kernel. Figure 5.3 shows the stack of function calls, starting with the Python interpreter and ending with the HIP functions responsible for launching a kernel. Additionally, we annotated groups of functions in Figure 5.3 with the layers of Figure 5.1 to which they belong.

Some lessons. Considering our pursuit of "realistic" applications, it may seem counterproductive to start with an example as simple as a Python equivalent to the `VectorAdd` snippets from Chapter 2. However, there are some useful points to this exercise. First, it serves as a gentle introduction to the "full" path between PyTorch and GPU hardware. Granted, Figure 5.3 does not illustrate the entire relation between a Python script and GPU hardware, but the content we presented in Chapter 4 is sufficient for making the remaining

connections. In fact, one could envision a larger figure combining Figures 5.1 and 4.1: the bottom layer of Figure 5.1 would simply lead into to the HIP layer of Figure 4.1.

The second lesson to be learned from this simple example is more relevant to our true objective of evaluating complex applications from a timing perspective. Popular sentiment often assumes that Python scripts are inherently "slow," and with some good reasons. For example, attempting to add two "vectors" of 100 million numbers contained only in Python's built-in `list` type would be excruciatingly slow compared to even a basic C++ equivalent. However, Figures 5.1 and 5.3 indicate that the performance of Python itself is of little consequence: almost all of the relevant processing occurs in PyTorch's C++ code.

In fact, we conducted a simple experiment that found the overhead imposed by Python in executing Figure 5.2 is almost unnoticeable in the face of the time required to execute the GPU kernel. With slight modifications, we repeated the addition operation in Figure 5.2 1,000 times and recorded the time required for each iteration. Second, we used a `vector_add` plugin to the HIP port of our microbenchmarking framework, discussed in Sections 3.3 and 4.2.1. We configured our `vector_add` microbenchmark to perform an amount of GPU computation roughly equivalent to the amount required by the PyTorch script in Figure 5.2: add two vectors of 100,000,000 random floating-point numbers. As with our experiments in Chapter 4, we executed this code on a Radeon VII GPU. Surprisingly, this basic performance test favored PyTorch: the mean execution time of our extremely simple `vector_add` microbenchmark kernel took 1.8 milliseconds, while PyTorch's tensor-add took 1.6 milliseconds on average.

PyTorch: A short, qualitative, code review. We chose not investigate the small performance disparity from the previous paragraph any further,[3] but this is because the causes are irrelevant to the chief point of the exercise: if a "high level" framework such as PyTorch is capable of performance on par with an implementation that uses HIP directly, then it is unreasonable to assume that the amount of abstraction present in PyTorch applications *inherently* causes timing problems at this scale.

By the time we ran the aforementioned short "addition" experiment, the high quality of PyTorch's software engineering had already become a common theme in our exploration of the code base. Apart from the obvious effort that goes into writing optimized GPU kernel code for a popular, high-value framework, we found a similar level of effort in virtually all of the timing-relevant portions of PyTorch that we investigated.

[3]There are some obvious candidates for the disparity: PyTorch's multi-dimensional tensor addition may exhibit superior caching behavior, our microbenchmarks' kernel code may encounter unnecessary overheads due to our time measurements, PyTorch code may be more efficient due to loop unrolling, *etc.*

For example, we observed that PyTorch is capable of reusing pre-allocated buffers of GPU memory to reduce the number of necessary calls to `cudaMalloc` or `cudaFree` (or their HIP equivalents). With some additional tracing, we also found that PyTorch never seemed to make unnecessary copies to or from GPU memory, even when evaluating the more sophisticated neural network we introduce in the following section.

In summary, developing GPU-accelerated applications using a sophisticated framework is unlikely to *cause* unpredictable or degraded performance in itself. Even so, it can certainly make it easy to introduce complexities that do—we encounter one such example later in this chapter. Even this risk could be a worthwhile tradeoff if only we can convince software developers to repurpose the time they save by using high-level languages into carefully profiling or tracing their applications! Our PyTorch overview has a clear message towards current and future real-time GPU researchers as well: when developing case studies, it should not be necessary to shy away from "real" applications such as PyTorch in favor of microbenchmarks. The inherent overhead of the library is likely manageable, and the underlying GPU code will be well optimized.

5.1.2 Neural Network: Adaptable MobileNetV1

Recall that PyTorch is only a *framework*; in order to have a proper case study, we need an application that uses it. It is particularly fitting to focus on a neural network that was developed to explore timing tradeoffs: namely, *US-MobileNet V1*, first introduced by Yu and Huang (2019*b*).

Slimmable networks. For the most part, *US-MobileNet V1* is identical to the original *MobileNet* architecture introduced by Howard *et al.* (2017). The key distinction between the modified version and the original *MobileNet* is indicated by the *US* in the name of the modification—standing for *universally slimmable*. This refers to the manner in which the neural network enables trading between accuracy and computational cost.

The "universally slimmable" approach taken by Yu and Huang is applicable to many neural-network architectures, not just *MobileNet*.[4] As we discussed in Section 2.3, a neural network is organized into *layers* of neurons, where each subsequent layer processes the output of one or more preceding layers. The slimming implemented by Yu and Huang (2019*b*) is capable of disabling a portion of neurons in each of the network's

[4]For example, the *Slimmable Networks* authors successfully evaluated their techniques using *MobileNet V1*, *MobileNet V2*, *ShuffleNet*, *ResNet-50*, and *MNas-Net* across several publications (Yu *et al.* 2019, Yu and Huang 2019*b*, Yu and Huang 2019*a*).

layers. Making each layer "slimmer" in this way requires progressively fewer computations as more neurons are disabled.[5]

***US-MobileNet V1*'s width multiplier.** In *US-MobileNet V1*, the slimming is controlled by a variable called the *width multiplier*: a value between 0.0 and 1.0 specifying the ratio of neurons to enable. For example, a width multiplier of 0.5 enables only half of the neurons in the network, while a width multiplier of 1.0 enables all neurons in the network: resulting in the same computations as the full, original *MobileNet V1* network. It should not come as much of a surprise that a network with more neurons is more accurate but slower than a smaller one, but this in itself is not the point of Yu and Huang's research. The characteristic that sets their work apart is the ability to offer a range of accuracies and computational costs using only a single network, and even allowing these tradeoffs to be dynamically adjusted at runtime.

The ImageNet dataset. The *MobileNet V1* neural network, and by extension, *US-MobileNet V1*, perform image classification. As discussed in Section 2.3 this is a common problem in computer-vision: automatically assigning labels to an image. More specifically, the pre-trained version of *US-MobileNet V1*, provided by the paper's authors,[6] is designed to classify images from the *ImageNet* dataset: a very large collection of images with a wide variety of possible labels (Deng, Dong, Socher, Li, Li and Fei-Fei 2009). Figure 5.4, copied from the original *ImageNet* publication, shows a small sampling of these images and representative labels.

Benefits of US-MobileNet V1 as a real-time benchmark. *US-MobileNet V1* is an ideal benchmark for realistic evaluation of real-time GPU management for several reasons. It is suitably recent, with adaptable neural networks still being on the cutting edge of computer-vision research. It is nontrivial, well-known (at least, the underlying *MobileNet V1* architecture), and does not depend on external libraries aside from PyTorch itself. The most obvious benefit, however, is its adaptability. Having been developed for resource-constrained systems, adaptable neural networks are conceptually relevant to real-time systems, but they also have a very practical application when testing GPU-sharing systems. Without changing the network, or even the sizes or format of its inputs or outputs, such a benchmark allows us to simulate a wide range of computational

[5] Yu and Huang's contribution is more complicated than this sentence may make it seem: a large part of the effort from their work goes towards disabling less useful neurons before more useful ones. We direct readers interested in the topic to the original paper, or the preceding paper by the same group (Yu *et al.* 2019).

[6] A link to the pretrained network weights is provided in the README of the repository containing the network's PyTorch code: `https://github.com/JiahuiYu/slimmable_networks`.

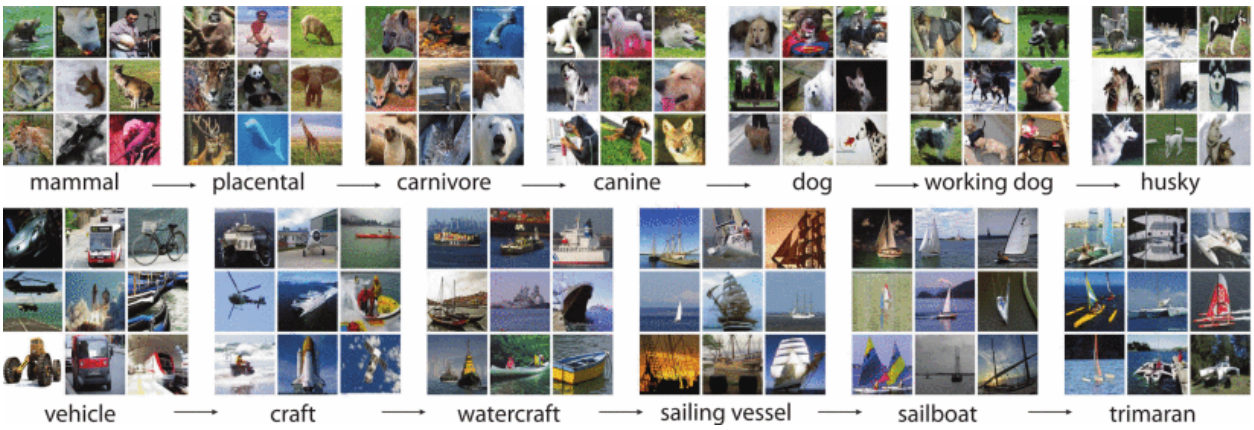


Figure 5.4: Example images from the *ImageNet* dataset along with their expected labels. This figure originally appears as Figure 1 in the original *ImageNet* publication by Deng *et al.* (2009).

requirements: making it possible to emulate a set of GPU-sharing tasks with different resource requirements using only a single application.

5.2 Runtime Behavior of a PyTorch Application

Now that we have both a framework to use and a neural network to execute, we can progress towards an application that can serve as a testbed for real-time GPU management. There is still plenty of relevant information to gather, however. In the remainder of this chapter, we focus on a "job" of a hypothetical *US-MobileNet V1* task: a single *forward pass* of the neural network, which processes a *batch* of one or more images, executes the network's layers, and produces classification results.

5.2.1 Experimental Setup

Handling data. In the form provided by the original authors, *US-MobileNet V1* loads input data (images and labels from the ImageNet dataset) on-demand from disk, preprocesses the images into the format expected by the neural network (224×224-pixel floating-point RGB images), and transfers the data to GPU memory. Our experiments focus on GPU management, so we avoided the overhead due to disk access and image-format conversions by preprocessing and loading a randomized set of images from the dataset into CPU memory prior to running any experiments. This was easy to do using the standard PyTorch API, which allows writing custom `DataSet` and `DataLoader` classes compatible with the rest of the library. While it would be equally simple and more performant to preload our random sampling of images into GPU memory,

we decided against doing so simply for the sake of greater realism—a realistic GPU-using application would likely need to involve at least some amount of data transfer.

Test platform. Our test platform in this chapter is identical to the one used in Chapter 4: a desktop computer containing a Radeon VII GPU, an Intel Xeon E5-2630 CPU, and 16 GB of DRAM. Our software platform consisted of version 5.17 of the Linux kernel, ROCm version 4.2, and PyTorch version 1.10.

We used AMD's `rocprof` tool to gather information about the kernels launched during each forward pass of *US-MobileNet V1*. Like NVIDIA's `nvprof`, `rocprof` is a command-line tool capable of recording a list of all GPU kernels a given application launches, along with the kernels' durations, block sizes, *etc.*

5.2.2 *US-MobileNet V1* GPU Kernel Performance

Our first experiments aimed to answer two fundamental questions: *how many kernels does* US-MobileNet V1 *launch?* and *how long do they take?* To obtain the necessary data, we monitored the performance of 100 forward passes of *US-MobileNet V1* using `rocprof`. The first of our two questions was easy to answer: *US-MobileNet V1* launches 91 kernels per forward pass if running at its full width. Interestingly, using a smaller width multiplier causes *US-MobileNet V1* to launch a larger number of kernels: 104 per forward pass. We assume that this behavior is due to implementation details of Yu and Huang's approach. As we shall see, however, overhead due to slimming is *usually* dominated by the faster execution of the slimmer network.

Answering the second question is more involved, as *US-MobileNet V1* launches far more kernels than the applications we considered in earlier chapters. First, we used the `rocprof` data to compute the average time taken by each of the 91 or 104 kernels individually. We next represented this list of individual kernel times as a CDF, and repeated the process for three different width multipliers. Figure 5.5 plots the results. Note that the endpoints of the plot's horizontal axis correspond to the fastest and slowest observed kernel times among all three width multipliers.

Observation 5.1 *Reducing* US-MobileNet V1*'s width multiplier consistently reduces the execution time of at least 70% of the network's GPU kernels.*

It should come as no surprise that narrower width multipliers speed up kernel execution in general; this only confirms that the faster performance of *US-MobileNet V1* at narrower widths extends to time spent executing on the GPU. Observation 5.1, illustrated by Figure 5.5, is more interesting than that, however. The curves only overlap near the bottom left, below $y = 30$, indicating that fewer than 30% of the kernels have

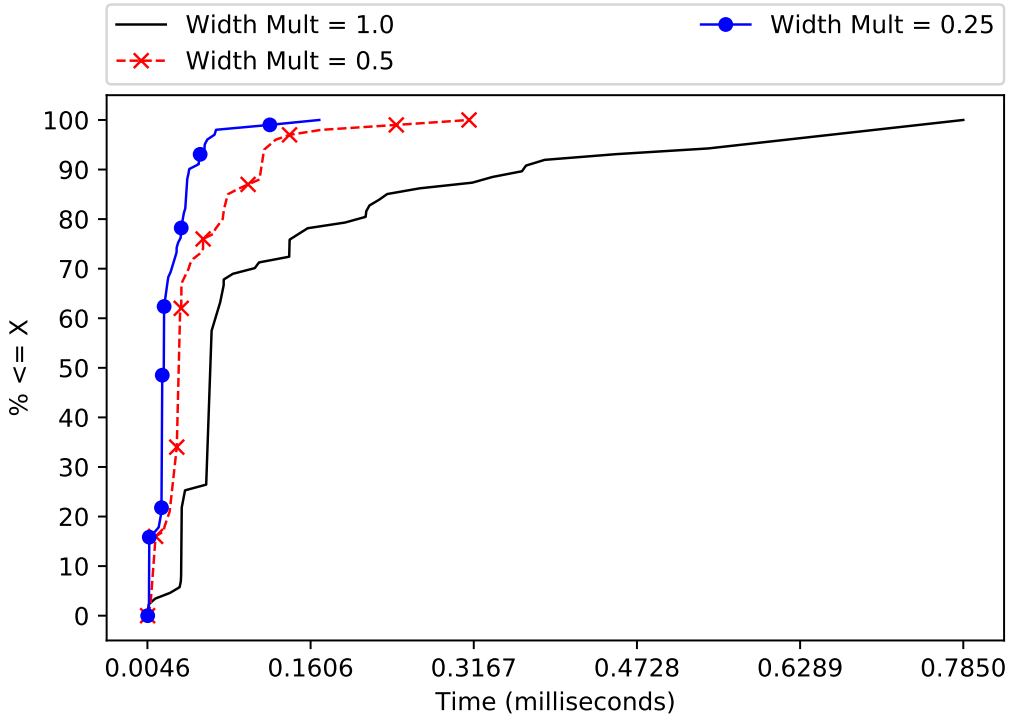


Figure 5.5: Distribution of the execution times of GPU kernels executed during a single forward pass of *US-MobileNet V1*, with varying width multipliers and a batch size of 32.

similar execution times regardless of the width multiplier. On top of this, this small number of kernels with similar execution times are the *fastest* kernels, executing for less than a tenth of a millisecond. The vast majority of the kernels, however, clearly speed up as the network slims. For example, the median-duration kernels, clearly speed up as the width multiplier decreases (median times are given by the curves' x positions where $y = 50\%$), and the slowest kernels see the greatest response-time improvements (worst-case times are given by the curves' x positions where $y = 100\%$).

The impact of batch size. While mentioned in passing earlier, many neural-network-based computer-vision applications, including *US-MobileNet V1*, enable users to configure a *batch size*: a variable number of images that a single forward pass processes in parallel. Using larger batch sizes can be especially helpful when training the network: allowing greater GPU parallelism at the expense of a larger memory footprint.[7] Similarly to choosing a width multiplier, choosing a batch size can also serve as a "knob" with which we can adjust the network's computational cost. Notably, increasing batch sizes does *not* cause *US-MobileNet V1* to launch a larger number of kernels, and instead simply increases the number of threads each kernel

[7]Batch sizes can also come into play with some training methods, such as *stochastic gradient descent*.

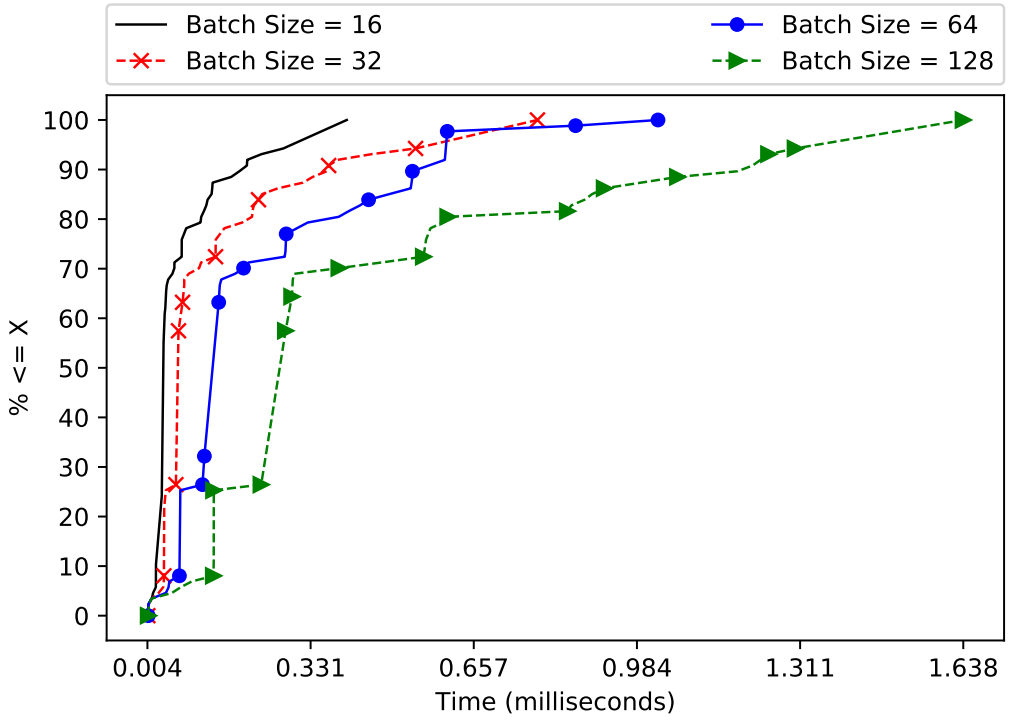


Figure 5.6: Distribution of the execution times of GPU kernels executed during a single forward pass of *US-MobileNet V1*, a width mutliplier of 1.0 and varying batch sizes.

executes to accommodate more input data.[8] This is naturally reflected in the execution time of each kernel, so we repeated our earlier experiment with width multipliers, this time varying batch size while keeping the network at full width. Figure 5.6 shows the resulting CDFs.

The behavior shown in Figure 5.6 is consistent with our expectations: execution times increase with batch size, but not entirely in a linear fashion. For example, the slowest kernel under a batch size of 64 executes for about slightly over one millisecond (the rightmost point in the "Batch Size = 64" curve of Figure 5.6), while the slowest kernel executing a batch of 128 inputs requires about 1.64 milliseconds (the rightmost point in Figure 5.6): processing twice as many input images without requiring twice the amount of time.

5.2.3 Response Times of *US-MobileNet V1* Jobs

Batch sizes and width multipliers clearly behave as expected with respect to kernel times: smaller batches and narrower widths reduce the amount of GPU computation required for a forward pass of *US-MobileNet*

[8]This is expected behavior in most neural networks, which seek to maintain flexibility with respect to batch size without imposing the overhead of additional kernel launches. Accommodating different batch sizes makes it easier to tune performance for different hardware (larger batch sizes may only be possible on GPUs with more memory), allows using a different number of inputs during training vs. evaluation, *etc.*

V1. However, kernel execution is only part of what would be required if *US-MobileNet V1* were a real-time *task*: a full *job* would also handle input and output data as well as various CPU operations. Put another way, our notion of a job is highly analogous to the "total time" illustrated in Figure 3.1 from Chapter 3. With the addition of CPU and data-transfer operations, the view of the application becomes more complicated, along with the batch-size and width-multiplier tradeoffs.

Figure 5.5 only used a batch size of 32, and Figure 5.6 only used a width multiplier of 1.0. We extended these experiments to include combinations of multiple batch sizes and width multipliers. Rather than plotting CDFs of these additional configurations, we chose to summarize their average response times in Table 5.1.

Observation 5.2 *Improvements to job times become decreasingly significant with smaller batch sizes and width multipliers.*

On one hand, Table 5.1 confirms that a trend from earlier results: progressively smaller batch sizes and width multipliers continue to speed up kernel execution. For example, Table 5.1 shows that a batch size of eight images and a width multiplier of 0.25 lead to all 104 kernels taking only 1.130 milliseconds in *total*. On the other hand, Observation 5.2 refers to the diminishing returns these reductions have on what an application designer is more likely to care about: the latency with which images are processed. For example, job times for batches of 8 or 16 inputs rarely differ by more than a millisecond, regardless of the width multiplier.

Observation 5.3 *The improvements to kernel time from smaller width multipliers do* not *always lead to faster job times.*

Extending our scrutiny to the job times required by small batches reveals Observation 5.3. For batches containing 32 or more inputs, reducing the network's width consistently reduces job times. However, consider the job times for a batch size of 8 images across the three width multipliers in Table 5.1: Rather than job times becoming faster, the time required to complete a batch of 8 images with a width multiplier of 0.25 is almost 0.6 milliseconds *slower* than with a width multiplier of 1.0. Recall from the start of Section 5.2.2 that the times listed in Table 5.1 are averages over 100 jobs, and, after observing these unexpected results, we re-ran the full set of experiments several times and found the results remained consistent. While we do not have the expertise in neural-network logic to pinpoint the cause of this slowdown in *US-MobileNet V1*'s code, the consistent reduction in GPU execution at slimmer widths leaves only CPU activity as the explanation. We have already encountered some observable logical differences, as evidenced by the full-width network

Batch Size	Width Multiplier	Mean Total Kernel Time (ms)	Mean Job Time (ms)	% Kernel Execution
8	1.00	3.422	14.373	23.8%
16	1.00	6.092	15.648	38.9%
32	1.00	11.496	22.960	50.1%
64	1.00	18.644	33.538	55.6%
128	1.00	36.643	56.601	64.7%
8	0.50	1.839	14.651	12.5%
16	0.50	3.081	14.716	20.9%
32	0.50	4.895	16.457	29.7%
64	0.50	9.076	22.703	40.0%
128	0.50	16.557	36.205	45.7%
8	0.25	1.130	14.945	7.6%
16	0.25	1.661	15.750	10.5%
32	0.25	2.670	16.381	16.3%
64	0.25	4.634	18.680	24.8%
128	0.25	8.583	27.545	31.2%

Table 5.1: *US-MobileNet V1*'s average total kernel times (as measured by `rocprof`) and overall job times with varying width multipliers and batch sizes. These measurements were taken prior to the performance improvements obtained by bypassing PyTorch's `DataParallel` wrapper.

requiring 91 kernel launches while narrower widths launch 104 kernels. While unfortunate, it is perhaps not a surprise that that these logical differences also involve at least a small amount of nontrivial CPU activity, but this is at least dominated by the improved GPU times when batch sizes are sufficiently large.

Observation 5.4 *In almost all of our configurations, GPU execution accounts for less than half of* US-MobileNet V1*'s job time.*

The impact of additional CPU overhead due to narrower width multipliers pales in comparison to the overhead apparent with Observation 5.4. According to Table 5.1, GPU execution accounts for only 7.6% of job time when executing *US-MobileNet V1* with a batch size of 8 and a width multiplier of 0.25. The only configurations in Table 5.1 where Observation 5.4 does *not* apply occur with a width multiplier of 1.0 and a batch size over 32, though even in these cases almost 20 milliseconds of job time cannot be accounted for by GPU execution.

One may assume that Observation 5.4 is due to the neural network carrying out intermediate CPU computations between GPU kernel activity. While an understandable guess, this is unfortunately not the case in our benchmark. We hinted in Section 5.1.1 that PyTorch does not transfer data between CPU and GPU memory unless necessary; this statement was in fact based on our traces of *US-MobileNet V1* using

`rocprof`. During each job, we only see three requests to transfer data to or from GPU memory: two requests to copy input images and labels to the GPU, and one request to copy results back to the CPU. This makes it highly unlikely that the CPU is responsible for intermediate computations, as intermediate results never seem to leave GPU memory. So, the question remains: if many jobs use less than half of their time on GPU computations, and intermediate math-heavy CPU operations seem unlikely, then where is all the time going?

5.2.4 Examining *US-MobileNet V1* Using *KUtrace*

While AMD's `rocprof` is useful for monitoring kernel launches and memory transfers, it is less useful for tracking down the cause of wasted CPU time. Fortunately, we can examine *US-MobileNet V1*'s behavior using a more powerful tool designed for such tasks: *KUtrace* (Sites 2021).

***KUtrace* overview.** *KUtrace* (Kernel-User trace) instruments the Linux kernel to record all transitions the system makes between kernelspace and userspace code, in addition to other events, such as interrupts. To maintain low overheads, *KUtrace* represents start/end pairs of events as 64-bit integers,[9] which it records in an in-memory buffer. The buffer's capacity is configurable, and typically sufficient for several seconds' worth of events. *KUtrace* is controlled using a userspace API, capable of enabling or disable tracing, saving the trace to a file, and manually inserting "marker" events into the trace. Given *KUtrace*'s internal limitation of representing events as 64-bit integers, *KUtrace* relies heavily on postprocessing to group events by Linux process, generate human-readable labels, *etc.*

In our situation, the most useful output produced by *KUtrace*'s postprocessing tools is a visual timeline of all system activity, across all Linux processes and CPU cores. We created a Python-language library to interact with *KUtrace*'s userspace API from within our PyTorch scripts, and used this library to record a trace of a single *US-MobileNet V1* job. Figure 5.7 shows the resulting timeline, with additional notation along the top indicating the approximate times during which GPU activity occurs.

Reading *KUtrace* timelines. *KUtrace* timelines, such as Figure 5.7, are standard HTML files that can be navigated using a web browser. Typically, two sets of timelines are visible, where events are grouped either by the CPU on which they were executed, or by the PID of their process or thread. For space, however, we cropped Figure 5.7 to only show the activity of two relevant threads from our PyTorch process. To

[9]For example, *KUtrace* packs both the enter and exit timestamps for a system call, interrupt, or trap handler into a single integer.

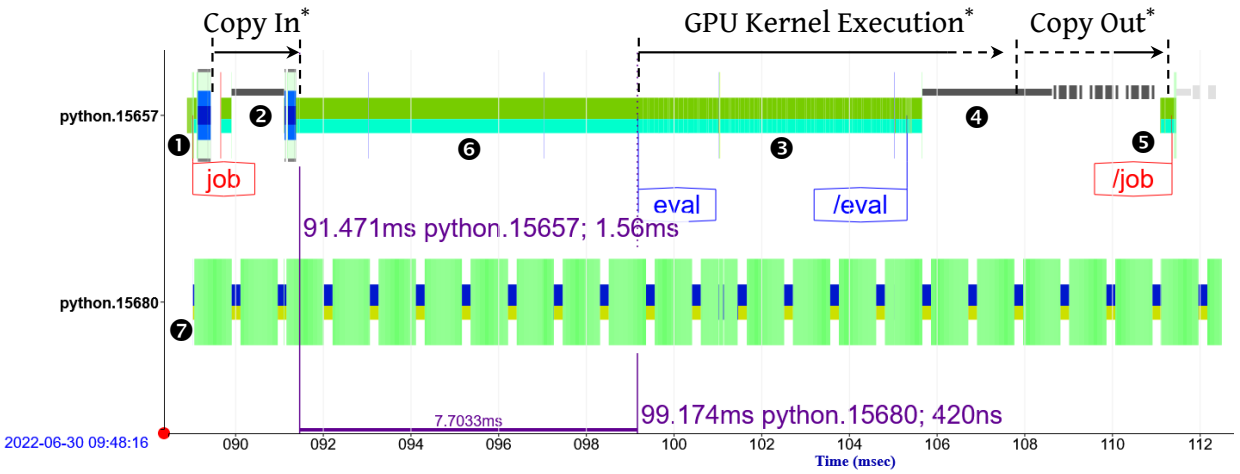


Figure 5.7: A visualization of PyTorch's CPU activity produced using *KUtrace*, encompassing the execution of a single *US-MobileNet V1* job, with a batch size of 32 and width multiplier of 1.0.

provide visual contrast, *KUtrace* timelines indicate CPU activity as two-colored bars: narrower bars represent userspace execution, and wider bars represent kernelspace execution. As an example, the timeline to the right of point ❼ in Figure 5.7 frequently alternates between userspace and kernelspace execution. The narrowest, gray bands, such as those appearing above points ❷ and ❹ in Figure 5.7, indicate spans of time during which the process is suspended. Manually inserted markers are labeled with text boxes appearing below the timeline in which the event occurred, such as the "job," "eval," "/eval," and "/job" labels in Figure 5.7.

Activity in the *US-MobileNet V1* trace. In addition to simply controlling when the trace starts and ends, we also used our *KUtrace* Python library to manually insert the four aforementioned markers from certain locations in our PyTorch scripts. The first of these labels, "job," indicates the start of a *US-MobileNet V1* job, shown at point ❶ in Figure 5.7. Shortly after the start of a job, point ❷ indicates the time when our input data, the batch of 32 images and expected labels, are copied into the GPU's memory. As we would hope, the main PyTorch thread is suspended while it waits for the memory transfer to complete, and the entire memory-copy procedure only lasts for roughly two milliseconds. (We confirmed using a combination of `rocprof` and additional *KUtrace* markers that all such input transfers only occur in the region of the timeline surrounding point ❷, annotated with the "Copy In" text.)

For the sake of explanation, we leave aside the span of the timeline indicated by point ❻ for now, and continue to the region annotated by point ❸. Point ❸ is surrounded by two marks, "eval" and "/eval," that

we inserted before and after the Python code responsible for executing the layers of the neural network.[10] To confirm that all GPU kernel launches occurred between the "eval" and "/eval" markers, we instrumented the underlying HIP userspace libraries to insert additional markers whenever a GPU kernel was launched. We ultimately chose to omit these per-kernel markers from Figure 5.7: they confirmed that all kernels were launched between the two "eval" markers as expected, but otherwise created significant visual noise.

Point ❹ indicates the span of time during which our script is waiting for resulting data to be copied from GPU to CPU memory. However, the suspension at point ❹ is clearly far longer than what we would expect for a small memory transfer, especially considering that copying an entire batch of input images to the GPU at point ❷ only took around two milliseconds. Fortunately, the length of this second suspension is easy to explain.

Recall that the prior paragraph only stated that all kernel *launches* appeared between the two points marked by "eval" and "/eval." Unless a CPU operation explicitly issues a synchronization request or depends on data from GPU computations, PyTorch will allow GPU work to proceed asynchronously. In the case of point ❹, the preceding kernel launches simply had yet to complete. Note that our "GPU Kernel Execution" annotation along the top of Figure 5.7 encompasses not only the region of CPU activity within the "eval" tags, but continues until some arbitrary point within the suspension. Without further tracing capable of aligning both CPU and GPU activity in time (*e.g.*, the `globaltimer` register we used to monitor NVIDIA GPU execution in Sections 3.2 and 3.3), it is hard to determine *exactly* where GPU kernel execution ends and the copy-out operation begins, but we do at least know that kernel execution must end before the resulting data can be copied to CPU memory.

The unaccounted-for time. With the end of the job at point ❺, we now return to the region of our timeline marked by point ❻. This span of time comprises nearly eight milliseconds of CPU execution (as indicated by the measurement near the bottom of Figure 5.7), over a third of the time taken by the entire job. CPU activity is no cause for suspicion in itself, but this particular region is suspicious due to *where* it appears: inserted between the end of the data transfer but before the start of the network's evaluation. As stated earlier, we have already preprocessed the input data, and all significant computation *should* be carried out by the neural network itself, in the region around point ❸—so what is the purpose of so much activity that has nothing to do with the the data transfer or the neural network's execution?

[10] For readers familiar with PyTorch, we added these markers at the start and end of the network module's `forward` function.

The extremely busy second Python thread. Even without identifying the region marked by point ❻ as a likely waste of time, Figure 5.7 contains a far more suspicious feature: the heavy activity in the second thread marked by ❼. Could this thread be causing a stall at point ❻, perhaps via some spin-based synchronization? After a lengthy investigation of this thread, we ultimately found it is unrelated to the activity at ❻, but it is still worth discussing for two reasons. First, it is illustrative of the relative lack of maturity still present in AMD's GPU-compute software (*i.e.*, Pitfall 4.7 in Section 4.3), and, second, it shows the value provided by fine-grained tracing, which makes the issue unmissable.[11]

Observation 5.5 *A bug in the* `amdgpu` *driver can cause ROCm to unnecessarily waste a full CPU core.*

After tracking down all of the threads created by ROCm, we arrived at Observation 5.5. We determined that the thread to the right of point ❼ in Figure 5.7 is created by ROCm's userspace runtime library to handle *hostcalls*. A hostcall is a *CPU* procedure invoked from *GPU* code, that does not require the GPU code to exit.[12] In other words, ROCm's userspace libraries create this thread to listen for requests coming from the GPU, and allow the thread to persist indefinitely, regardless of how frequently it is used. Normally, this would not be a problem, as the code run by the thread *ought* to be suspended if no events occur.[13] The reason for the thread continually spinning (in kernelspace code!) appears to be a bug in the `kfd_wait_on_events` ioctl in AMD's GPU driver for Linux 5.17.[14] For reasons we were unable to determine, this ioctl never actually suspends the caller while waiting for a signal from the GPU. Instead, it spins for some time before immediately returning to userspace, despite the signal having not occurred. This does not cause a logical error in the userspace code that issued the ioctl, because the semantics of the ioctl state that it may return spuriously, even under normal operation. Returning early, therefore, causes the userspace thread to assume it was spuriously woken up, and to call the ioctl again.

The thread to the right of point ❼ turned out to be a red herring with respect to the activity at point ❻, even though it is a clear hazard on its own for systems with limited energy or computational capacity. We could have omitted it from Figure 5.7 entirely, but chose to include it due to the interesting story and for

[11] Frankly, a tool such as UNIX `top` should be enough to raise red flags when every PyTorch process creates a thread that wastes an entire CPU core for no obvious reason.

[12] Among other things, ROCm uses hostcalls in its support of *GPU lambdas*, a feature of CUDA and HIP in which a C++ lambda function can be executed on the GPU, without the programmer needing to place kernel source code into a separate function.

[13] The loop of the hostcall thread can be seen in the ROCclr source code: `https://github.com/ROCm-Developer-Tools/ROCclr/blob/90f1f61a9d6c28ffd2f844dc773e921444752e47/device/devhostcall.cpp#L282`. Specifically, the `doorbell->Wait(...)` function should be able to suspend the calling thread.

[14] We have not checked whether this bug exists in other Linux versions, though 5.17 is still quite recent at the time of writing.

the sake of completion: we disabled all of PyTorch's multithreading support wherever possible, so the two timelines remaining in the figure encompass the entirety of PyTorch's CPU activity.

The casual waste enabled by high-level scripting. Eventually, additional tracing led us to identify the cause of the problem at point ❻: the time was spent in internal PyTorch code between the point at which our script requested a forward-pass of the neural network, and the actual start of the network's top-level `forward` function (the point marked by "eval" in Figure 5.7). This indicated that PyTorch was transparently invoking several wrapper functions instead of directly calling `forward`: an assumption we confirmed by printing a call trace from within the `forward` function itself. The call trace revealed the culprit: the version of *US-MobileNet V1* provided by the original authors encapsulates the entire neural network within the `DataParallel` helper class provided by PyTorch. This fact alone added nearly eight milliseconds of overhead to every forward pass of the neural network.

Ordinarily, the `DataParallel` wrapper class is used to simplify multi-GPU training. We never performed any training, but we understand the draw of applying a simple wrapper to transparently enable training a single network using multiple GPUs. However, for users like us, who do not have multi-GPU systems or are not interested in training, wasting up to half of a job's execution time is more problematic. We suspect that the original authors never even realized the performance penalty imposed by their blanket usage of the `DataParallel` wrapper: it occurs due to the inclusion of an unremarkable line of code,[15] has no impact on logical correctness, and the temporal problem only becomes apparent after a thorough investigation. This is a concrete illustration of a lesson we brought up in Section 5.1.1: while PyTorch may not be inherently slow, the ease with which time can be unnecessarily wasted is potentially disastrous in a safety-critical system, and a poor use of time and energy in any setting.

Solving the problem. Fixing the performance problem was easy after identifying its underlying cause: remove PyTorch's `DataParallel` wrapper from *US-MobileNet V1*. After doing so, we recreated the timeline using *KUtrace*; Figure 5.8 shows this new version.

The trace in Figure 5.8 now matches the traditional organization of a GPU application given in Section 2.1.1: exhibiting clear "copy-in," "execute," and "copy-out" phases at points ❷, ❸, and ❹. Furthermore, the location of the "eval" tags in Figure 5.8 indicate that the network starts executing immediately after

[15] The line in question can be seen in the original repository at `https://github.com/JiahuiYu/slimmable_networks/blob/5dc14d0357ccfc596d706281acdc8a5b0b66c6d6/train.py#L39`.

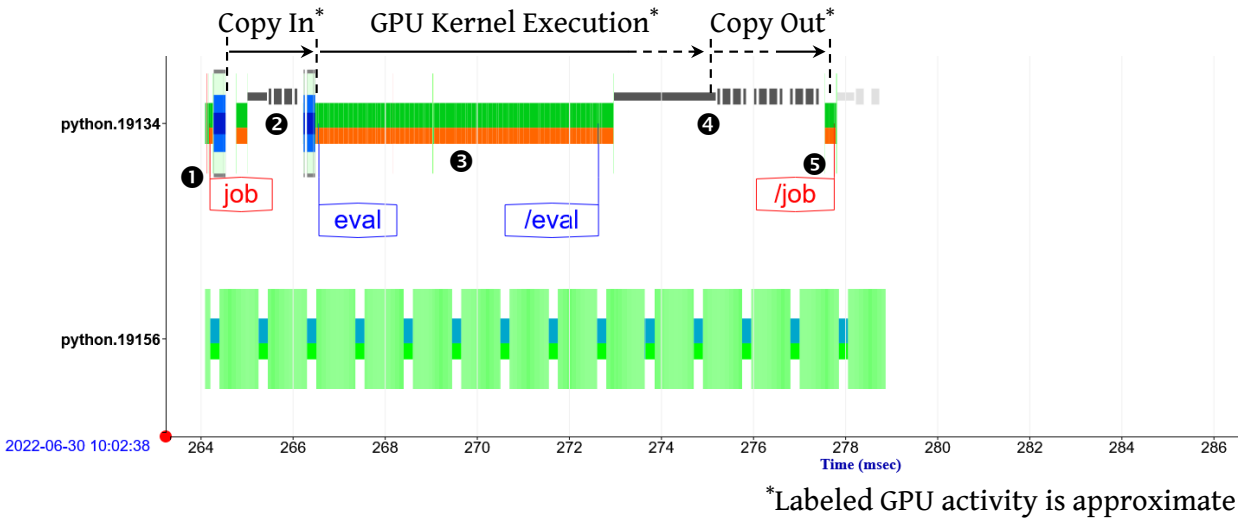


Figure 5.8: A visualization of PyTorch's CPU activity produced by *KUtrace*, showing the behavior of a *US-MobileNet V1* job after removing the `DataParallel` wrapper.

the input data is copied to the GPU, and the job completes approximately eight milliseconds faster, both consistent with our expectations.

Finally, we repeated the set of experiments with varying batch sizes and width multipliers that we used earlier to produce Table 5.1, but after applying the `DataParallel` fix. We collected these results into a new table: Table 5.2.

Observation 5.6 *Removing the* `DataParallel` *wrapper from* US-MobileNet V1 *reduced job times by nearly eight milliseconds regardless of the width multiplier or batch size.*

Observation 5.6 is supported by comparing the job times shown in Tables 5.1 and 5.2. Unsurprisingly, the fix had no significant impact on kernel times. Fixing the `DataParallel` issue also seems to have fixed or reduced the overhead mentioned in Observation 5.3: Table 5.2 no longer exhibits a significant increase in job times at narrower width multipliers. Similarly, Observation 5.4 is far less prominent after fixing the performance deficiency.

5.2.5 Revisiting Our Motivations: *US-MobileNet V1* as Real-Time Case Study

We conclude this section by clarifying one of our goals: we wish to make it easier for other real-time researchers to evaluate their work using modern, realistic applications like PyTorch. It is easy to find online articles containing instructions on how to use high-level frameworks to create and train neural networks, but

Batch Size	Width Multiplier	Mean Total Kernel Time (ms)	Mean Job Time (ms)	% Kernel Execution
8	1.00	3.419	7.200	47.5%
16	1.00	6.114	7.843	77.9%
32	1.00	10.783	13.680	78.8%
64	1.00	18.320	23.896	76.7%
128	1.00	36.744	47.733	77.0%
8	0.50	1.839	7.368	25.0%
16	0.50	3.086	7.855	39.3%
32	0.50	4.904	9.043	54.2%
64	0.50	9.079	13.981	64.9%
128	0.50	16.453	26.187	62.8%
8	0.25	1.131	6.909	16.4%
16	0.25	1.662	7.869	21.1%
32	0.25	2.670	8.949	29.8%
64	0.25	4.639	11.209	41.4%
128	0.25	8.589	18.063	47.5%

Table 5.2: *US-MobileNet V1*'s average total kernel times (as measured by `rocprof`) and overall job times with varying width multipliers and batch sizes. These measurements were taken after applying the performance fix of removing the `DataParallel` wrapper from *US-MobileNet V1*.

our goal is narrower: we assume that a real-time researcher is, like us, more likely to modify an existing application to produce a real-time task. In such a case, understanding PyTorch's underlying libraries and expected behavior is a higher concern than understanding topics such as neural-network training. Apart from the text of this chapter, we have also made the source code of our *US-MobileNet V1* application available online.[16]

Unfortunately, using complex software like PyTorch when evaluating GPU-management approaches remains quite uncommon in real-time literature. Many works (*e.g.*, our own work from Section 3.2, Basaran and Kang (2012), Lee and Al Faruque (2014), Lee and Al Faruque (2016), Jain *et al.* (2019), *etc.*) only evaluate GPU performance using a small number of microbenchmark tasks. More complex tasks structured as DAGs (directed acyclic graphs) have also seen some attention in real-time GPU research. Even in this case, however, it is unclear whether DAG-based approaches are capable of scaling to meet the needs of neural networks requiring hundreds of GPU kernels. For example, both Elliott (2015) and Yang, Amert, Yang, Otterness, Anderson, Smith and Wang (2018) contain evaluations using the DAG-based *OpenVX* computer-vision framework, but both only test real-world DAGs containing fewer than a dozen nodes.

[16] `https://github.com/yalue/slimmable_nets_rtbench`

Admittedly, some real-time GPU publications, such as Capodieci *et al.* (2018) or Yang, Wang, Bakita, Vu, Smith, Anderson and Frahm (2019) do use neural-network workloads in their evaluation. To our knowledge, however, none of these small handful of papers use PyTorch, despite the tool's overwhelming success in other fields.

We do not mean to imply at any of these aforementioned works are *incapable* of handling the demands of a high-level framework like PyTorch. In fact, none of the proposed frameworks appear (in our reading) to be fundamentally incapable of such a task. Regardless, the question of capability is superficial—moving from the concepts to actually running PyTorch code would largely be an implementation exercise (though certainly one that is more difficult in some cases than in others). Our point instead is this: *if these management techniques are capable of handling complex applications, then their evaluation should involve complex applications*. Anything less is shortselling the authors' own work, and hindering the adoption of our field's research into other GPU-using domains.

5.3 Chapter Summary

In the same way that Chapter 4 advocates for an alternative hardware platform on which real-time GPU research can be implemented, this chapter advocates for an alternative software platform with which the research can be evaluated.

In this chapter, we proposed one such evaluation platform: the *US-MobileNet V1* neural network using the PyTorch framework. In Section 5.1.1, we summarized the relevant structural components of PyTorch's code. Based on a simple comparison with a microbenchmark, we found that PyTorch's layers of abstraction are not inherently problematic, at least when contrasted with the GPU execution they can carry out. On the other hand, our work in Section 5.2.4 provided ample evidence that PyTorch applications' reputation for being "slow" is not without warrant, as the high-level design makes it easy to overlook mistakes capable of wasting a significant amount of time.

Apart from our lower-level investigations, Sections 5.2.2 and 5.2.3 present basic information about the kernels executed by *US-MobileNet V1*, and the amount of time it takes to execute on our system. In addition to revealing the performance pitfall we investigated in Section 5.2.4, these sections support our concluding argument: given that standard computer-vision software such as *MobileNet V1* can require executing over

a hundred GPU kernels, real-time GPU management should focus on supporting applications of similar magnitude.

CHAPTER 6: REAL-TIME MANAGEMENT FOR GPU-SHARING NEURAL NETWORKS

With the *US-MobileNet V1* benchmark developed in Chapter 5, we are now able to directly investigate an important question: what, if any, real-time GPU management systems are effective at improving the timing characteristics of modern neural-network-based applications?

6.1 Implementing GPU Spatial Partitioning and Locking Using ROCm and PyTorch

In this chapter, we use the ROCm software stack and our *US-MobileNet V1* benchmark to implement and test two prominent real-time GPU management approaches from prior work: *spatial partitioning* and *locking*. Spatial partitioning (*i.e.*, allocating non-overlapping sets of CUs to separate tasks) is an obvious candidate for testing given our AMD GPU's hardware support and our experience in the topic from Section 4.2. We do not experiment with fine-grained temporal partitioning, such as instruction-level preemption, due to platform limitations,[1] but we nonetheless implement coarse-grained temporal partitioning, in the form of *GPU locking*. We also supplement locking methods from prior work with our own extension: using k*-exclusion locking* to share a single GPU.

k-exclusion locks. Unlike typical mutual-exclusion (mutex) locks, which only allow a single task at a time to access a shared resource, k-exclusion locks allow up to k tasks to access the shared resource at once. (If k equals 1, then a k-exclusion lock behaves identically to a standard mutex lock.) Prior work used k-exclusion locks to allocate entire GPUs to tasks in multi-GPU systems (Elliott and Anderson 2011), but, to our knowledge, this chapter contains the first documented attempts at applying the technique to a single GPU. We are especially interested in combining k-exclusion locking with spatial partitioning: in the same way that Elliott and Anderson (2011) assigned entire GPUs to tasks, we can instead assign *partitions* of a single GPU to tasks, eliminating the possibility of expensive inter-GPU migrations present in the prior work.

[1] We conducted some preliminary experiments with hardware preemption support on our Radeon VII GPU, but decided against pursuing the topic further due to high overheads.

6.1.1 Implementing Spatial Partitioning

Given the hardware support, using spatial partitioning on AMD GPUs requires fairly little work. Users can create a stream with a CU mask using the `hipExtStreamCreateWithCUMask` function, and any kernels submitted to the stream only use CUs enabled by the mask (we discussed this in more detail in Section 4.2.3.1). Our experiments, however, required this functionality to be accessible from within Python scripts, and to be compatible with PyTorch's own stream API.

Addressing the first of these two requirements, accessing `hipExtStreamCreateWithCUMask` from within Python scripts, was fairly straightforward. We created a native Python library, that calls the function from HIP's C-language API and returns the resulting `hipStream_t` handle to the Python code. (We used similar native Python extensions to access the *KUtrace* API in Section 5.2.4.)

Addressing the second requirement, using the stream handle within PyTorch code, was slightly more difficult. Typically, PyTorch's API for managing CUDA or HIP streams uses opaque data structures, enabling PyTorch to reuse a small pool of pre-allocated streams. PyTorch ostensibly supports using what it calls "external streams," which are opaque objects wrapping `hipStream_t` handles allocated by libraries other than PyTorch. Disappointingly, despite being included in PyTorch's documentation, implementation of this feature was incomplete at the time we conducted our experiments; attempting to use it would cause errors. We successfully worked around the problem by directly modifying PyTorch's source code: in addition to some trivial additions to `switch` statements, our fix required creating a cache of existing PyTorch "stream" objects, preventing PyTorch from allocating new objects when external stream handles were reused.[2]

With these two fixes, we were able to spatially partition PyTorch applications on the GPU as follows:

1. Use our Python wrapper to call `hipExtStreamCreateWithCUMask`. Our wrapper function takes a CU mask as a hexadecimal string and returns a handle to a HIP stream.

2. Use PyTorch's API (including our fixes) to create an `ExternalStream` object using the HIP stream handle obtained in Step 1.

3. Use PyTorch's existing `torch.cuda.Stream(...)` function to prompt any following GPU kernel launches to use the `ExternalStream` created in Step 2.

[2]Support for external streams has since been fixed in the upstream PyTorch respository, but our own fix for PyTorch 1.2 is still available online: `https://github.com/yalue/rocm_pytorch`.

4. Execute the remainder of the *US-MobileNet V1* code as normal.

6.1.2 Implementing k-Exclusion Locking

Due to queue-oversubscription issues (discussed in more detail in Section 6.2.1.1), we were effectively limited in our experiments to running at most four *US-MobileNet V1* instances. Even with each of these instances creating a buggy CPU-occupying thread (see Section 5.2.4), our test system had 16 CPU cores, ensuring sufficient capacity for all four *US-MobileNet V1* instances. Our assumption that CPU availability is never a problem for our task systems enabled us to simplify the k-exclusion approach taken by prior work, which used complex priority-inheritance mechanisms to determine CPU scheduling (Elliott and Anderson 2011). We implemented k-exclusion locking using a Linux kernel module, exposing `acquire`, `release`, and `set_k` ioctls. Internally, our kernel module maintains a list of k "slots," each of which can be occupied by a separate process.

To obtain the lock, a task issues the `acquire` ioctl, which places it into a FIFO `wait_queue` and suspends it until one of the k slots becomes available. The `release` ioctl, in turn, causes a lock-holding task to relinquish its own slot. Additionally, `release` removes a waiting process from the head of the wait queue, assigns it to the newly empty lock slot, and wakes it up. Even though we assume our system always has sufficient CPU capacity, we nonetheless boost a lock-holding task's priority to ensure it is not preempted by spurious background work. When a task acquires the lock, our kernel module assigns it to Linux's `SCHED_FIFO` priority level, and reverts it to `SCHED_NORMAL` priority after it releases the lock.

Finally, the `set_k` ioctl can be called by a manager process to change the number of available slots (this is only possible if no processes currently hold the lock). Technically, a count of lock holders is sufficient for a standard k-exclusion implementation; maintaining a list of slots is an extension sometimes known as *slotted k-exclusion* (Attiya, Bar-Noy, Dolev, Peleg and Reischuk 1990) or *k-assignment* (Anderson and Moir 1997). In our case, we require the slot-based extension for an additional purpose: determining GPU partition assignments.

6.1.3 Combining k-Exclusion Locking and Spatial Partitioning

Upon returning, our `acquire` ioctl returns the index of the slot it occupies to the calling process. When combining locking and CU partitioning, the calling process uses this slot index to determine the task's GPU

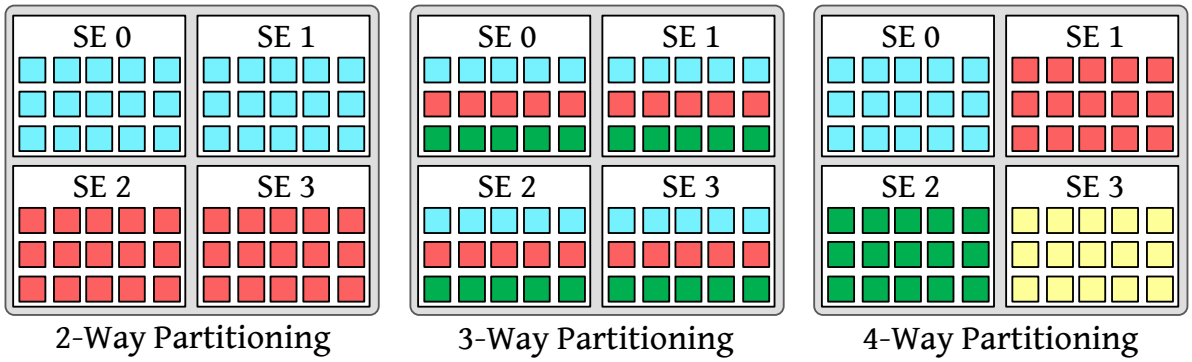


Figure 6.1: The mapping of partitions to CUs and SEs for various partition sizes. CUs are represented by the colored squares within the SEs, and colored based on partition assignment.

partition assignment. In order to quickly switch between partitions on a per-job basis, each task creates a list containing k of PyTorch's `ExternalStream` objects at initialization time (following the steps from Section 6.1.1). Each of these k streams uses a non-overlapping CU mask, containing $60/k$ CUs (due to the Radevon VII's 60 CUs). After acquiring the k-exclusion lock, the PyTorch script simply starts using the `ExternalStream` object at the same index as its lock slot. Creating the streams once, at initialization time, is essential: HIP's stream-creation process is high-overhead (see Section 4.2.2.1), but instructing PyTorch to start submitting kernels to a different stream is comparatively negligible.

CU masks for different partition sizes. One of the clearest lessons from Section 4.2 of Chapter 4 is that even masks with identical numbers of enabled CUs can have extremely different behavior. Fortunately, our Radeon VII's 60 CUs are divisible into several evenly-sized partitions that do not fall into the performance pitfalls. We made use of both SE-distributed and SE-packed partitions, depending on partition size. Being limited to at most four concurrent tasks means that we only needed to define three different partition layouts, which we show in Figure 6.1, and list below:

- *Two-way partitioning*: When dividing the GPU into two partitions, we used the SE-packed partitioning scheme to force each of the two partitions to fully occupy two of the GPU's four shader engines. This configuration is represented by the leftmost image in Figure 6.1. Our experiments in Section 4.2.3.1 showed that this led to slight performance benefits, likely due to the fact that 30 SMs can entirely packed into two SEs, but not evenly distributed across four SEs.

- *Three-way partitioning*: When dividing the GPU into three partitions, we used the SE-distributed partitioning scheme so that each partition occupied five SMs on each of the four SEs. This choice was

in line with Figure 4.9 from Section 4.2.3.1 (as well as earlier explanations in Chapter 4), which clearly shows that SE-distributed partitions are faster than SE-packed partitions when using 20 CUs. This configuration is shown by the middle image in Figure 6.1.

- *Four-way partitioning*: When dividing the GPU into four partitions, we used SE-packed partitioning to designate each shader engine as a separate partition. As was the case for two-way partitioning, four partitions map easily onto the GPU's four SEs, while it would be impossible to evenly distribute the 15 SMs among four shader engines in a SE-distributed scheme. This configuration is represented by the rightmost image in Figure 6.1.

6.2 Real-Time GPU Management of Multiple, Identical Neural-Network Applications

With our *US-MobileNet V1* benchmark and a handful of management techniques, we are faced with a large number both of potential task parameters and different management configurations. While our experiment's limited size makes us hesitant to extrapolate our results to broader contexts, we found lock-based GPU management to be entirely detrimental in our constrained subset of *US-MobileNet V1* tasks. The experiments in this section provide evidence in support of this conclusion.

6.2.1 Experimental Setup

Unless otherwise noted, the experiments in this section involve four concurrent *US-MobileNet V1* instances, sharing a single GPU. Sections 6.2.1.1 through 6.2.1.3 provide remaining details about how we set up the experiment and the scenarios we evaluated.

6.2.1.1 Overcoming Practical Limitations to Multiple PyTorch Instances

Before discussing the specific task configurations or management techniques, it is worth stating that the simple act of starting four concurrent *US-MobileNet V1* instances requires overcoming a few hurdles.

Avoiding queue oversubscription. When using ROCm, it is remarkably easy to encounter Pitfall 4.1 from Section 4.3, unless specific steps are taken to mitigate it. As stated in Section 4.2, AMD GPUs such as the Radeon VII are limited to 32 hardware queues before entering "oversubscription" behavior, with severe performance degradation. Additionally, on our system, eight of the 32 queues are reserved for graphical work, leaving only 24 for computational work such as HIP streams. As stated in Sections 4.2.2.2 and 6.1.1, both

internal ROCm and PyTorch libraries create pools of streams (or lower-level queues), which can encroach on this limit even if we never require more than one stream. In ROCm's case, we limited the size of the pool of shared HSA queues using an environment variable.[3] For PyTorch, however, the size of the pool of shared streams can only be changed by modifying the source code itself. Fortunately, this was also an easy fix for us, because we were already modifying the relevant stream-related source files as part of our spatial-partitioning implementation described in Section 6.1.1.

Even while limiting each process to a single stream per shared pool, our k-exclusion partitioning approach requires creating k additional streams per process, each of which uses a separate CU mask, meaning they are required to occupy separate hardware queues (see Section 4.2.2.2). In other words, each of our *US-MobileNet V1* tasks requires up to five queues: one default queue in addition to another four queues, one per partition. This effectively limited us to four concurrent tasks: four tasks require a total of 20 hardware queues, but five tasks would require 25: exceeding the limit of 32 after accounting for the eight graphical queues.

Memory limitations. In Section 5.2.1 of the previous chapter, we explained our choice to pre-load a sample of *US-MobileNet V1*'s input data into CPU memory: avoiding unpredictable and slow disk-access times while running the application. Unfortunately, maintaining a separate in-memory buffer of input data for multiple *US-MobileNet V1* instances can quickly exhaust system memory. Rather than attempting to limit our input dataset to a tiny size, we instead opted to load the dataset into a region of shared memory, to which each task was granted read-only access. This is possible using the `mmap` function present in Python's standard library, in combination with PyTorch's capability to load data from plain `mmap`-ed buffers. Pre-loading input data into shared memory has the added benefit of speeding up our experiments: when multiple *US-MobileNet V1* instances are running concurrently, we only need to wait for the dataset to be loaded into memory once.

Warming up and synchronization. Loading data from disk into CPU memory is only one part of the initialization process when measuring the performance of GPU applications: initializing ROCm libraries and transferring kernel code to GPU memory also costs a nontrivial amount of time. This is not unique to GPU applications, and the practice of running one or more "warmup" iterations (*i.e.*, to ensure code is loaded, libraries are initialized, *etc.*) is common in many benchmarking domains. Still, this is particularly important in our case, as the first iteration of a *US-MobileNet V1* forward pass takes nearly eight *seconds* on our ROCm test platform. Therefore, in all of our experiments we uses a kernel-supported barrier-synchronization

[3]Specifically, ROCm allows users to control the size of the pool by setting the `GPU_MAX_HW_QUEUES` variable.

operation to ensure that all of our *US-MobileNet V1* tasks have completed two warmup iterations before making any timing measurements.

Avoiding repeated ROCm initialization. We mentioned in the context of our microbenchmarking framework in Section 3.3 that we needed to avoid initializing CUDA in a parent process, as the act of creating a child using the `fork` system call would cause the CUDA context to be copied. The same constraint, unfortunately, remains in place for ROCm processes, including in the context of Python and PyTorch. While this requires some attention, it is at least an easy problem to work around: we simply avoid interacting with any code that requires initializing ROCm (*i.e.*, the PyTorch libraries) outside of child processes.

6.2.1.2 Task Parameters

We varied the batch size and width multipliers of *US-MobileNet V1* to define three possible tasks with different computational requirements: `small`, `medium`, and `large`.

- `small`: A single job of this task classifies a batch of eight input images, using a width multiplier of 0.25. This task uses the smallest parameters we measured to produce Table 5.2 in Section 5.2.4: in isolation, we expect jobs of the `small` task to complete in 6.9 milliseconds on average, including 1.1 milliseconds of GPU kernel execution (about 16% of its job time).

- `medium`: A single job of this task classifies a batch of 32 input images, using a width multiplier of 0.5. According to Table 5.2, jobs of this task should only take approximately 9 milliseconds in isolation, roughly only 2 milliseconds longer than jobs of `small`. However, `medium` jobs spend a significantly larger portion of their time performing GPU execution: about 4.9 milliseconds, or 54% the average job.

- `large`: A single job of this task classifies a batch of 64 input images, using a width multiplier of 1.0. According to Table 5.2, jobs of this task require about 23.9 milliseconds to complete on average, when executing in isolation. Additionally, `large` jobs execute on the GPU for 18.3 milliseconds on average, about 76% of each job's duration.

6.2.1.3 Management Techniques

We tested five GPU-management techniques in this round of experiments, using k-exclusion locking with different k values, and different partition sizes.

- *Unmanaged*: Allow all four competing tasks to access all CUs on the GPU with no additional restrictions. This is most similar to the unmanaged "co-scheduling" we investigated using microbenchmarks in Section 3.2.

- *Exclusive Locking*: Allow only a single job to execute on the GPU at a time. The single job has full access to the entire GPU. This is intended to mimic earlier work such as Elliott, Ward and Anderson (2013), even though, unlike the prior work, our system only contains a single GPU.

- *2-Exclusion Locking*: Allow at most two jobs to execute at a time, with no partitioning of the GPU's CUs.

- *2-Exclusive Locking, with Partitioning*: This allows at most two jobs to execute at a time, but the two active jobs execute on separate GPU partitions, in the manner described in Section 6.1.3.

- *4-Way Partitioning*: Each job is assigned to one of four equally sized, non-overlapping partitions of the GPU's CUs. With four partitions and four tasks, locking is unnecessary, as each of the tasks can always be assigned to the same partition.

Locking granularity. For our first set of experiments, we only consider locking at job-level granularity: in scenarios requiring locking, tasks acquire the lock at the start of each job and release the lock when the job ends. Later, we also consider locking on a per-kernel basis in Section 6.2.2.1.

6.2.2 Lock-Based GPU Management: Results

Our overall experiment required running fifteen scenarios: one scenario for each possible combination of task size from Section 6.2.1.2 and management technique from Section 6.2.1.3. In executing a single scenario, we ran four identical instances of the selected task size and recorded job times until a full minute had elapsed (not counting initialization time). After each scenario completed, we collected statistics summarizing all job times across all four task instances. Table 6.1 contains these results.

Observation 6.1 *When considering the combined job times of four concurrent tasks, most GPU management approaches result in degraded performance.*

It is hard to miss Observation 6.1: our attempted GPU-management techniques only served to harm performance in almost every statistical category and for every task size. There are two notable exceptions:

4-Way Partitioning slightly sped up the average-case times for `medium` and `large` tasks, and 2-Exclusion locking (without partitioning) resulted in slight improvements to average-case performance for `large` tasks. Even so, in these cases where additional management led to improved average-case performance, the difference was always within a single standard deviation of the "Unmanaged" performance, with notable drawbacks in other metrics.

Observation 6.2 *Exclusive locking always produced the slowest worst-case and average-case performance.*

Observation 6.2 is apparent by examining the "Exclusive Locking" rows of Table 6.1. This observation indicates that loss of potential GPU capacity is particularly destructive in neural-network applications. The magnitude of the slowdown due to exclusive locking decreases with larger tasks, an unsurprising result given the fact that tasks requiring more GPU resources waste less capacity to begin with. For example, according to the list in Section 6.2.1.2, `small` tasks only spend around 16% of each job executing GPU kernels in isolation, so locking the GPU for the entire duration of a `small` job potentially leaves the GPU unused for over 80% of each job. On the other hand, `large` tasks spend nearly 76% of their jobs executing kernels, making the capacity loss due to exclusive locking proportionally less extreme (though still more costly than less-restrictive locking options).

Observation 6.3 *The "Unamanged" technique always resulted in the lowest worst-case times.*

Observation 6.3 is supported by the contents of the "Max" column in Table 6.1, containing the worst observed job time from the 60 seconds of execution, across all four tasks. No matter the sizes of the contending tasks, the "Unmanaged" technique always produced the lowest worst-case time: an unfortunate indictment of such management for *this particular type of application*. Put another way: our experiments never encountered a situation where interference due to sharing GPU hardware was destructive enough (in the worst case) to overcome the capacity loss imposed by additional management.

Observation 6.4 *4-way partitioning improved average-case response times for* `medium` *and* `large` *jobs.*

In an interesting contrast to Observation 6.3, Observation 6.4 indicates that destructive interference is common enough that *some* management is capable of improving average-case times, at least for larger tasks.

Task Sizes	Management Technique	Min (ms)	Max (ms)	Arithmetic Mean (ms)	Standard Deviation
small	Unmanaged	6.68	13.64	7.86	1.33
	Exclusive Locking	9.00	48.99	39.01	2.91
	2-Exclusion Locking	11.40	28.99	19.18	2.50
	2-Excl. w/ Partitioning	13.79	38.71	19.09	2.42
	4-Way Partitioning	6.74	19.89	8.53	1.43
medium	Unmanaged	8.28	25.30	18.53	3.45
	Exclusive Locking	11.41	59.49	45.38	3.63
	2-Exclusion Locking	13.88	35.96	24.05	2.77
	2-Excl. w/ Partitioning	17.28	42.28	23.81	2.73
	4-Way Partitioning	15.14	46.72	17.72	1.96
large	Unmanaged	75.51	88.23	83.43	2.11
	Exclusive Locking	24.37	124.09	96.09	2.97
	2-Exclusion Locking	41.40	94.71	82.78	1.98
	2-Excl. w/ Partitioning	49.55	98.69	86.39	2.06
	4-Way Partitioning	66.10	110.80	80.80	4.30

Table 6.1: Average job times when four identical tasks share the GPU, under varying sizes and management techniques.

Observation 6.4 is supported by the mean times under 4-Way partitioning in Table 6.1, which, for `medium` and `large` tasks, led to the fastest job times by a slight margin. We also provide Figure 6.2 for a more thorough view of the impact different management techniques have on `medium` jobs. The leftmost two curves in `medium` serve to confirm what we learned from the table: both Unamaged sharing and 4-way partitioning have similar average job times, but the distribution of times is slightly in favor of the partitioned approach.

The fact that the benefit of 4-Way Partitioning did not hold for `small` tasks indicates that partitioning is more helpful for tasks requiring a larger amount of GPU work—jobs with greater potential both to cause and receive interference. As for the reason why 4-way partitioning's benefits do not extend to worst-case times, we can only speculate. For example, despite its average times being the fastest under partitioning, `large` also experienced its second-worst worst-case time under partitioning. This indicates a rare but potentially costly source of interference that is shared among the entire GPU, but the rarity of such outliers makes them harder to investigate. Both Table 6.1 and Figure 6.2 make it clear that the benefit to average-case times is rather minor: less than a single standard deviation away from the Unmanaged times for both `medium` and `large` tasks.

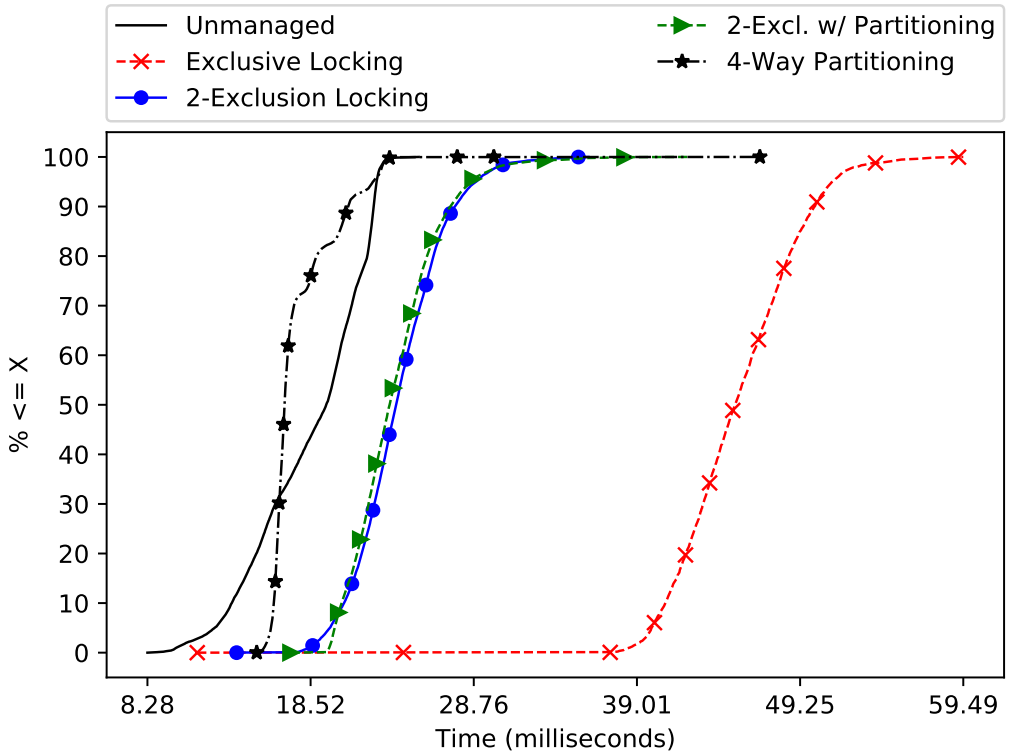


Figure 6.2: CDFs of `medium` job times while sharing the GPU with three other `medium` instances, under varying management strategies.

6.2.2.1 Per-Job or Per-Kernel Locking?

While easier to implement, one may reasonably question whether the coarse-grained approach of holding a GPU lock for the duration of a job is more effective than alternative locking approaches, *i.e.*, holding the lock for shorter amounts of time, but acquiring and releasing the lock multiple times per job. Of these, the most straightforward option is requiring each task to hold a lock prior to launching a single GPU kernel, and to release the lock when each kernel completes. Rather than replicating the full set of experiments used for Table 6.1, we evaluated per-kernel and per-job locking using a much narrower experiment: sharing the GPU between up to two `medium` tasks.

Implementing per-kernel locking. Access to the underlying ROCm libraries makes it relatively straightforward to implement per-kernel locking. For this, we modified HIP's internal kernel-launch function, `ihipModuleLaunchKernel`,[4] to acquire a lock prior to enqueueing kernel-launch commands. Unfortunately, per-kernel locking cannot allow multiple asnychronous kernel launches from a single task, as this

[4]Source code for this, and our other ROCm modifications, is available online: `https://github.com/yalue/rocm_mega_repo`.

With Competitor?	Management	Min (ms)	Max (ms)	Arithmetic Mean (ms)	Standard Deviation
No	Unmanaged	8.81	14.68	9.74	0.60
	Per-Kernel Locking	15.29	22.09	16.88	0.85
	Per-Job Locking	8.87	15.27	9.85	0.59
Yes	Unmanaged	8.57	16.84	11.37	1.21
	Per-Kernel Locking	16.47	27.38	20.38	1.77
	Per-Job Locking	13.52	29.98	23.03	1.97

Table 6.2: Response times of a `medium` task, both with and without a single additional `medium` competitor, contrasting per-kernel and per-job locking approaches.

could allow a task to "extend" the time it holds the lock indefinitely—degenerating into per-job locking. Therefore, we also required `ihipModuleLaunchKernel` to wait for the kernel to complete and release the lock before returning. In other words, per-kernel locking *prevents asynchronous kernel launches*. An ideal solution would require locks to be acquired and released immediately at the start or end of GPU kernel code, meaning that kernels could be enqueued ahead of time, but would still not concurrently execute on GPU hardware. Unfortunately, to the best of our knowledge, neither AMD nor NVIDIA GPUs provide mechanisms for accomplishing this. Short of destroying a context entirely, we are unable to prevent a kernel's execution after writing the kernel-launch request into a hardware queue (*i.e.*, after writing the kernel-launch AQL packet into an HSA queue, as described in Section 4.2.2.1).

Per-kernel and per-job locking experiment. As with the set of experiments at the start of Section 6.2.2, we once again configured tasks to run jobs for 60 seconds (not including warm-up iterations), during which we recorded response times. Table 6.2 summarizes the resulting measurements.

Observation 6.5 *Per-kernel locking causes high overheads for tasks running in isolation.*

Observation 6.5 is supported by the top half of Table 6.2, which summarizes the overhead a single `medium` task experiences when the different forms of locking are enabled. Unsurprisingly, per-job locking causes very little overhead when there is no contention for the GPU, slowing job times by only a few hundredths of a millisecond over their Unmanaged counterparts. The consequences of using per-kernel locking are stark: causing job times to increase by nearly seven milliseconds in the minimum, maximum, and average cases.[5] This illustrates the fact that preventing asynchronous kernels can result in extreme capacity

[5]In order to eliminate the possibility of these results being due to locking overheads, we ran a separate experiment in which we performed per-job locking, but replaced the single lock-acquire `ioctl` at the start of each job with a loop that released and re-acquired the lock 104 times. This led to no obvious increase in job times over the normal Per-Job Locking results in Table 6.2.

loss: being unable to enque subsequent kernels as prior kernels complete appears to significantly reduce hardware efficiency, especially for *US-MobileNet V1* tasks requiring over 100 kernels.

Observation 6.6 *Under contention, per-kernel locking wastes less capacity than per-job locking, but is still far slower than Unmanaged GPU sharing.*

The lower half of Table 6.2 summarizes the job times when two `medium` tasks contend for the Radeon VII. The results in this half of the table support Observation 6.6: despite the consequences to job times in isolation, the finer-grained per-kernel locking clearly improves GPU utilization compared to per-job locking, speeding up both worst- and average-case job times by roughly three milliseconds. Even so, jobs' performance under per-kernel locking is nowhere near as fast as the speeds possible when the jobs are allowed to access the full GPU with no additional restrictions.

Observation 6.7 *Per-kernel exclusive locking between* two *contending* `medium` *tasks leads to slower per-job performance than when* four *tasks of the same size are managed using 4-Way Partitioning.*

The final observation we take from this experiment serves as yet another strong condemnation of applying exclusive GPU locking to neural networks. In Table 6.1, we see that *four* `medium` tasks have average job times of 17.72 milliseconds when partitioned to four separate sets of CUs, and average job times of 18.53 milliseconds when left unmanaged. In contrast, we see in Table 6.2 that both per-kernel and per-job locking lead to slower average job times, *despite running half the number of concurrent tasks*.

Per-kernel locking: conclusions. While this short experiment indicated that per-kernel locking may offer some benefits over per-job locking when the GPU is contested, the overhead imposed on isolated tasks is hard to justify. More importantly, our experiments with per-kernel locking do nothing to dispel Observation 6.2: in our experiments, *any* sort of exclusive GPU locking seems to be far worse than partitioning or no management at all.

6.2.3 Summary of Results

Without any reason to prioritize results from one task over another, our negative conclusions pertaining to GPU locking may not be surprising: if maximizing system-wide throughput is a primary goal, then additional overheads will only be detrimental.

Observation 6.3, however, poses a more inconvenient truth for would-be real-time GPU management approaches: not only are most approaches not beneficial from an average-case standpoint, they also serve no benefit from a worst-case perspective, likely as additional overheads harm worst-case times more than the interference they seek to prevent.

None of this is to say that contention for GPU hardware is not a problem at all. In fact, our results for four-way partitioning specifically refute such a claim, leading to slight performance improvements over unmanaged GPU sharing, even when evaluated solely from the standpoint of throughput. As we shall discuss in the following section, GPU partitioning becomes even more useful when we wish to prioritize the timing predictability of a single task.

What about NVIDIA GPUs? Given prior work's positive portrayal of lock-based management using NVIDIA GPUs, one may be tempted to believe that this section's negative results are platform-specific. While we admittedly only ran this set of experiments on an AMD GPU, we have good reasons to believe that similar performance degradation due to locking would apply in equal, if not greater, measure on modern NVIDIA GPUs, especially in embedded cases where MPS cannot be used. As we learned in Section 3.2, NVIDIA GPUs *already* co-schedule multiple processes using time slicing, meaning that external lock-based management will only impose additional blocking without the potential to reduce hardware interference.

Is this actually a realistic application? Another tempting rebuttal to the results in this section may be to claim that our *US-MobileNet V1* benchmark is an unrealistically demanding application compared to the microbenchmarks favored by prior work, or compared to whatever processing pipelines currently serve to control experimental autonomous systems. Unfortunately, such a claim is far from the truth; if anything, our *US-MobileNet V1* tasks are *less* sophisticated than many real-world neural networks, much less full computer-vision processing pipelines (as stated in Section 2.3, image-classification algorithms such as *US-MobileNet V1* only serve as one of many components in such a system).

US-MobileNet V1 is actually designed to be a *small* neural network. The entire point of the underlying *MobileNet V1* architecture was to produce a neural network capable of running on edge-computing, *mobile* devices (Howard *et al.* 2017), hence the name. Our experiments found that additional management performed poorly even when running the network at a width multiplier of 0.25—at which it only correctly classifies approximately 50% of its input images (Yu and Huang 2019*b*). For comparison, the most accurate neural network at the time of writing classifies 91% of the ImageNet dataset correctly, and is approximately 1,000

times larger than *US-MobileNet V1* running at full width (Yu, Wang, Vasudevan, Yeung, Seyedhosseini and Wu 2022). Certainly, these large, modern networks were not designed for embedded use cases, but the trend is clear: *MobileNet V1* was considered small from the time of its creation. After five years of hardware improvements and expanding applications, if it fails to be representative of modern use cases, it is *due to not being complex enough*.

6.3 Using Partitioning to Protect High-Priority Tasks

Our experiments in Section 6.2 that lock-based GPU management is unlikely to lead to any sort of improvement for neural-network applications such as *US-MobileNet V1*, but this conclusion had two caveats. First, the results considered system-wide performance, counting job times from all tasks with equal priority. Second, Section 6.2 found one GPU-management option remained mostly beneficial: *spatial partitioning*. In this section, we combine these two factors to examine another common target of prior real-time GPU research: *prioritizing* some tasks over others.

6.3.1 Prioritizing *US-MobileNet V1* Tasks Using CU Masking

A successful prioritization approach for real-time GPU management ought to not only improve the average-case response times of a high-priority task, but also the task's worst-case times. We restructured several aspects of our GPU-sharing experiment from Section 6.2 to design a new set of scenarios in which we measure the performance of a single *measured task* when facing contention from several competitors of differing sizes.

Experimental setup. For this new set of experiments, we arbitrarily chose a `medium` task to serve as our measured "high-priority" task. Our experiment involved 16 scenarios in total, using four partition sizes and four different sets of competitors. For competitors, we tested the measured task in isolation, or in contention with three instances of `small`, `medium`, or `large` tasks. As for partition sizes, our experiments either allowed the measured task to share the full GPU with competitors, or assigned it to a partition containing 15, 20, or 30 CUs. In the partitioned cases, we assigned the non-measured competitors to the remaining CUs, and followed the strategies outlined in Section 6.1.3 for producing CU masks of the correct sizes.

Competitors	Partition Size (CUs)	Min (ms)	Max (ms)	Arithmetic Mean (ms)	Stdandard Deviation
None	15	13.874	21.260	14.043	0.233
	20	11.428	21.661	11.591	0.277
	30	9.405	18.102	9.596	0.423
	Unpartitioned	8.233	17.844	8.994	0.572
`small`	15	14.044	22.174	14.886	0.441
	20	11.554	20.799	12.377	0.489
	30	9.458	21.218	10.602	1.170
	Unpartitioned	8.313	22.686	10.093	1.507
`medium`	15	15.360	28.716	18.920	1.745
	20	11.936	26.215	15.112	1.399
	30	9.360	23.154	12.042	1.438
	Unpartitioned	9.390	25.486	14.791	1.733
`large`	15	16.792	31.922	18.702	1.123
	20	13.577	32.797	15.018	0.978
	30	10.387	20.021	11.758	1.367
	Unpartitioned	8.767	52.856	24.921	4.158

Table 6.3: Table of job times for a `medium` neural network (using a batch size of 32 and a width multiplier of 0.5).

As with all of our experiments in this chapter, we allowed all tasks in each scenario to run for 60 seconds, not including warm-up time. Unlike in other experiments, however, we only collected job-time measurements for the single `medium` measured task. Table 6.3 summarizes the results.

Observation 6.8 *Partitions of any size improved the measured task's average-case performance in the presence of* `large` *competitors.*

Observation 6.9 *Partitions of any size improved the measured task's worst-case performance in the presence of* `large` *competitors.*

Observations 6.8 and 6.9 are supported by the bottom portion of Table 6.3. Additionally, these two observations are supported by Figure 6.3, which contains CDF plots for the portion of Table 6.3 corresponding to `large` competitors. Even when restricted to a partition containing 15 CUs, the measured task still exhibited faster average and worst-case times than when sharing the entirety of the GPU with `large`. While Figure 6.3 contains some of the same information as the bottom four rows of Table 6.3, the CDF plot gives a clear visual representation of the benefits partitioning can have to the *predictability* of the measured task's response times. Not only are all of the "Partitioned" curves to the left of the "Unmanaged" curve, indicating faster

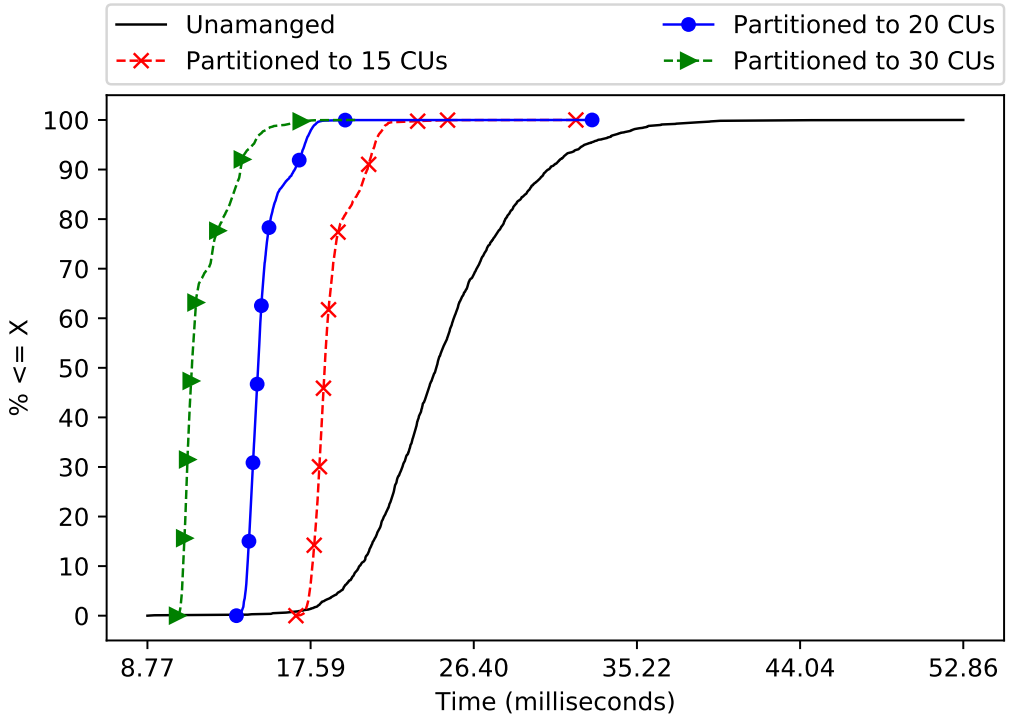


Figure 6.3: CDFs of a `medium` task's response times when competing against three `large` competitors, using different partitioning schemes.

performance, the curves are also much narrower horizontally, indicating a narrower distribution of response times with a thin tail containing few outliers.

Observation 6.10 *Partitioning becomes less effective or detrimental when facing small amounts of contention.*

Observation 6.10 is supported by the sections of Table 6.3 showing the performance against `small` and `medium` competitors. Against three `medium` competitors, our `medium` measured task only fared better than its Unpartitioned performance when it was allocated 30 CUs: half of the entire GPU. This performance indicates that even three `medium` competitors do not cause enough contention for GPU hardware to overcome the downsides associated with large reductions to the number of CUs available to the measured task. Observation 6.10 is even more apparent when considering the average-case performance of the measured task against `small` competitors, where average Unpartitioned performance was faster than any partitioning size.

6.3.2 Summary of CU-Partitioning Experiments

Unlike our experiments in Section 6.2, our spatial-partitioning experiments were mostly positive. Clearly, spatial partitioning remains an effective method for reducing harmful hardware contention without inducing unnecessary overhead, even when applied to applications such as *US-MobileNet V1*. While we observed greater efficacy with higher amounts of contention and relatively little benefit for smaller task sizes, we do not find results such as Observation 6.10 to be particularly discouraging. Our attitude in this matter stems from the same information with which we concluded Section 6.2.3: we intentionally erred on the side of keeping our *US-MobileNet V1* benchmark small and simple. Real-world neural networks, such as those likely to be employed in a safety-critical autonomous system, are likely to be larger themselves and subject to greater amounts of contention from other tasks in a system consisting of multiple software components. In such a situation, our experiments from this section indicate that GPU spatial partitioning is still a management technique well worth pursuing.

6.4 Chapter Summary

In Chapter 1, we claimed that "simplistic models of GPU behavior and restrictive requirements placed on GPU software are unable to handle modern GPU-accelerated AI." In this chapter, we used our platform knowledge from Chapter 4 along with the *US-MobileNet V1* application from Chapter 5 to finally provide clear support of this claim. In Section 6.2, we found that capacity loss is a significant concern: in our experiments, the additional blocking caused by lock-based approaches far overshadowed any benefits due to a reduction in contention for GPU hardware.

Apart from the disappointing conclusions in Section 6.2, in Section 6.3 we found spatial partitioning to be quite effective if one's goal is not maximizing GPU throughput, but protecting a single task from interference due to larger competitors.

Logical correctness and throughput. While most of the material presented in this dissertation remains concerned with the timing behavior of GPU applications, it would be remiss to omit discussing an important point last mentioned in Chapter 1: temporal correctness is not the only requirement from AI applications. Useful results, naturally, must also be *logically correct*. Readers may have noticed this chapter's primary focus on *average-case* performance, which often falls outside the purview of real-time research, being dismissed as "real fast." We do not dispute the utility of bounding tail latencies or reducing the number

of temporal outliers, but sacrificing too much throughput in favor of these other concerns is also unlikely to make a neural-network-based system safer. Our discussion of Figure 1.1 in Chapter 1 implied that, for most neural networks, greater throughput *is* greater safety. Ultimately, a safety-critical neural network must maximize the probability of producing a logically correct result within a given window of time—a goal served equally well, if not better, by focusing on "real-fast" performance. Therefore, this chapter's increased focus on throughput still serves the same goal as all other real-time research: producing safer software.

CHAPTER 7: CONCLUSION

It seems contradictory to claim that any system is "safety critical" when it is built using hardware with poorly understood behavior, or when its software lacks any guarantee of logical correctness. Heedless of this fact, developers of safety-critical systems such as autonomous vehicles seem to be guided by the principle that "perfect is the enemy of good," staking human lives on statistical properties of software with well-known shortcomings. Short of major catastrophes, our criticisms remain unlikely to halt or pause development of such systems, leaving us with two options for improving safety: first, we can educate the relevant research communities on how to avoid certain risks, and, second, we can develop middleware systems capable of mitigating the risks in a transparent manner.

This dissertation contributes to both options. GPUs play a pivotal role in modern safety-critical systems, providing much-needed acceleration for artificial-intelligence and computer-vision computations. First, by providing additional information about GPU behavior, we both educate users about inherent risks and build the capability to produce models with which timing behavior can be predicted. Second, by developing and testing a neural-network-based real-time benchmark, we support accurate evaluation of past and future real-time GPU-management techniques, and promote easier integration with high-level applications.

7.1 Summary of Results

In support of our thesis statement in Section 1.4 of Chapter 1, we made the following contributions:

Measuring the performance impact of unmanaged co-scheduling on NVIDIA GPUs. In Section 3.2, we investigated whether GPU co-scheduling can maintain temporal predictability while reducing capacity loss, even without additional management. We co-scheduled multiple microbenchmark applications on a single, embedded NVIDIA GPU in several experiments, and found promising results: co-scheduling improved system throughput with no obvious negative impacts on temporal predictability. On the other hand, these results relied on NVIDIA's default time-sliced management of multiple contexts, which prevents inter-process

concurrent execution on the GPU hardware. This motivated our subsequent contributions, which sought to enable true concurrent usage of GPU hardware by multiple independent tasks.

Developing a microbenchmarking framework for investigating GPU queueing behavior. In Section 3.3, we developed and explained the need for a microbenchmarking framework, capable of carrying out the black-box experiments we used to discern scheduling behavior on NVIDIA GPUs. We later ported the framework to use AMD's software stack, and used it for our AMD-based experiments in Section 4.2 of Chapter 4.

Discerning queueing rules governing NVIDIA GPU sharing. In Section 3.4, we discerned several fundamental rules governing the order in which kernels submitted to different CUDA streams execute on an NVIDIA GPU. Absent confounding factors such as stream priorities or implicit synchronization, we discovered that all kernels from separate streams enter a single per-context "primary" queue based on the order in which they reach the head of their stream queues. From the single primary queue, kernels can begin executing in FIFO order as long as sufficient GPU computational capacity is available.

Investigating synchronization pitfalls with NVIDIA GPUs. In Section 3.5, we investigated several causes of *synchronization* in NVIDIA GPUs, which can stall CPU or GPU activity while waiting for ongoing GPU work to complete. In addition to documenting behavioral differences of different types of synchronization, we found additional pitfalls including the potential for blocking "asynchronous" CPU tasks and discrepancies between documented and actual behavior.

Documenting internal AMD GPU scheduling behavior. In Section 4.2, we documented the internal mechanisms responsible for scheduling computational kernels on AMD's GCN-architecture GPUs. Despite having an open-source software stack, documentation pertaining to AMD GPUs' internal behavior remains hard to find. We combined several sources of information with white-box experiments to contrive an example workload that intentionally used AMD-specific "worst practices" to achieve surprisingly poor performance in simple matrix-multiplication tasks. By explaining the bad behavior, we not only clarified how the specific problems can be avoided, but also explained the entire process by which GPU kernels and thread blocks are assigned to AMD hardware.

Providing guidance on using CU masking to spatially partition AMD GPUs. In Section 4.2.3.1, we focused specifically on the AMD-specific CU-masking API for spatially partitioning the GPU's computational resources. CU masking plays an integral role in kernel scheduling on AMD GPUs as a whole, and was one of

the methods with which we intentionally triggered the poor performance mentioned in the previous paragraph. AMD's official documentation includes little, if any, information on the impact certain CU-mask choices can have on performance, so our guidance on correct CU-mask usage is a key contribution of Chapter 4. We identified two CU-masking approaches, *SE-distributed* and *SE-packed*, each of which may be more or less effective depending on partition sizes.

Producing a real-time benchmark using a modern PyTorch neural network. In Chapter 5, we addressed what we view as a key need in real-time GPU research: relevant benchmarks for neural networks, the flagship applications of modern GPU programming. To produce a suitable benchmark, we adapted an existing image-classification neural network: *US-MobileNet V1*. Despite being considered a "small" network by many standards, *US-MobileNet V1* still launches over 100 GPU kernels, and can take tens of milliseconds per job (depending on input sizes and accuracy settings). Our investigation of *US-MobileNet V1* also included the PyTorch neural-network programming framework in which it was developed. Our efforts in Chapter 5 encountered the pitfall that is nearly synonymous with high-level programming: unintentionally wasted computational capacity. We used *KUtrace*, a tool that traces system-wide CPU activity on Linux, to identify and fix some performance mistakes in *US-MobileNet V1*, leading to faster response times as well as an ancillary observation: high-level frameworks like PyTorch are not inherently unsuitable for real-time work, but they can certainly make timing problems easy to introduce.

Evaluating locking- and partitioning-based management of neural-network applications. In Chapter 6, we revisited two GPU-management approaches from prior real-time research: *locking* and *spatial partitioning*. Our management implementations relied on our findings about AMD GPU hardware and software from Chapter 4, and our evaluation was based on the *US-MobileNet V1* application we developed in Chapter 5. Our evaluation found that lock-based management incurs significant capacity loss when used with *US-MobileNet V1*, more than negating any benefits of reduced hardware contention. On the other hand, we found spatial partitioning to be beneficial, both improving throughput and reducing worst-case response times, especially for tasks more sensitive to interference due to heavy GPU utilization.

7.2 Future Work

Our work includes several possible avenues for future developments.

Expanding the availability of real-time neural-network benchmarks. Recent years have seen some encouraging progress in the development of realistic, neural-network-based GPU benchmarks. For example, the *Tango DNN Suite* (Karki, Keshava, Shivakumar, Skow, Hegde and Jeon 2019) is one such effort that produced a C-language implementation of several neural-network architectures. Even so, a scan of *Tango*'s source code reveals a need for significant corrections before it is an appropriate real-time benchmark, such as restructuring logic into repeating jobs, or preloading the neural-network's weights into GPU memory prior to starting to execute kernels. A small number of prior real-time papers (Yang *et al.* 2019) use neural networks developed using the *DarkNet* framework,[1] which is conveniently implemented in C rather than Python. Still, we are unaware of any real-time GPU research apart from our own that evaluates management techniques using a "big" framework such as PyTorch or Tensorflow. While high-level frameworks may not be ideal to use in deployed embedded products, improved support for them in real-time research would still serve to promote integration with other disciplines, where such tools remain incredibly popular.

Performance optimizations and detailed tracing. While high-performance computing research continues to thrive, issues such as the ones we discuss in Chapter 5 persist, even in relatively popular applications such as *US-MobileNet V1*. The fact that our trace-based analysis easily uncovered the fact that *US-MobileNet V1* causes ROCm's internal libraries to waste an entire CPU core suggests there are plenty of low-hanging performance fixes for researchers willing to perform similar investigations. Future work applying *KUtrace* or other tracing tools can identify and fix timing pitfalls in other other deep-learning software frameworks or GPU drivers.

Combining spatial partitioning with fine-grained temporal partitioning. Combining fine-grained temporal partitioning, such as hardware-supported preemption, with spatial partitioning techniques remains a promising area for future work. For example, an ideal GPU-management platform could combine the deadline-based GPU scheduling by Capodieci *et al.* (2018) with the hardware-partitioning work by Jain *et al.* (2019). In the future, we expect such combined approaches to be possible on both NVIDIA and AMD GPUs, so long as NVIDIA's MIG partitioning remains available and AMD's hardware-preemption support becomes sufficiently performant with future GPUs.

[1] `https://pjreddie.com/darknet/`

GPU-management possibilities enabled by NVIDIA's new open-source Linux driver. NVIDIA recently released an open-source GPU driver for Linux,[2] which sadly arrived too late to factor into our NVIDIA-related work from Chapter 3. In future work, an open-source driver should enable GPU-management techniques that were previously only possible with the help of NVIDIA collaboration, such as the work by Capodieci *et al.* mentioned in the previous paragraph.

7.3 Other Related Work

We close by acknowledging some of our non-GPU-related research, which was not included in this dissertation. Along with several collaborators, we also conducted a series of projects aimed at enabling mixed-criticality scheduling in combination with CPU, DRAM, and cache partitioning on an embedded ARM-based platform. In one publication, we proposed several methods to allow data sharing in contexts where concurrent access to shared-memory buffers may break temporal-isolation guarantees.[3] In follow-up work, we extended our data-sharing techniques to executable code contained in shared libraries.[4] Next, we proposed methods for handling *mode changes*: alternating between different task systems without breaking isolation guarantees.[5] Finally, we proposed methods for enabling memory-mapped device I/O on our platform, while maintaining memory and cache partitioning.[6]

[2] `https://github.com/NVIDIA/nvidia-installer`

[3] This work appears in Chisholm, Kim, Ward, Otterness, Anderson and Smith (2016). *Reconciling the Tension Between Hardware Isolation and Data Sharing in Mixed-Criticality, Multicore Systems*. IEEE Real-Time Systems Symposium (RTSS).

[4] This work appears in Kim, Chisholm, Otterness, Anderson and Smith (2017). *Allowing Shared Libraries While Supporting Hardware Isolation in Multicore Real-Time Systems*. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).

[5] This work appears in Chisholm, Kim, Tang, Otterness, Anderson, Smith and Porter (2017). *Supporting Mode Changes While Providing Hardware Isolation in Mixed-Criticality Multicore Systems*. International Conference on Real-Time Networks and Systems (RTNS).

[6] This work appears in Kim, Tang, Otterness, Anderson, Smith and Porter (2020). *Supporting I/O and IPC via Fine-Grained OS Isolation for Mixed-Criticality Real-Time Tasks*. Springer Real-Time Systems journal.

BIBLIOGRAPHY

Abadi, Martín, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard *et al.*. 2016. Tensorflow: A System for Large-scale Machine Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Abodo, Franklin, Robert Rittmuller, Brian Sumner and Andrew Berthaume. 2018. Detecting Work Zones in SHRP 2 NDS Videos Using Deep Learning Based Computer Vision. In *IEEE International Conference on Machine Learning and Applications (ICMLA)*.

Ali, Waqar and Heechul Yun. 2018. Protecting Real-Time GPU Kernels on Integrated CPU-GPU SoC Platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*.

AMD Corporation. 2011. "AMD Graphics Core Next (GCN) Architecture." Online at `https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf`, accessed September 2019.

AMD Corporation. 2019. "Introducing RDNA Architecture." Online at `https://www.amd.com/system/files/documents/rdna-whitepaper.pdf`, accessed June 2022.

AMD Corporation. 2021. "HIP: C++ Heterogeneous-Compute Interface for Portability." Online at `https://github.com/ROCm-Developer-Tools/HIP`.

AMD Corporation. 2022. "ROCm, a New Era in Open GPU Computing." Online at `https://rocmdocs.amd.com/en/latest/`.

Amert, Tanya and James H. Anderson. 2021. CUPiD$^{\mathrm{RT}}$: Detecting Improper GPU Usage in Real-Time Applications. In *IEEE International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE.

Amert, Tanya, Nathan Otterness, James H. Anderson and F. Donelson Smith. 2017. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *IEEE Real-Time Systems Symposium (RTSS)*.

Anderson, James H and Mark Moir. 1997. "Using Local-Spin k-Exclusion Algorithms to Improve Wait-Free Object Implementations." *Distributed Computing* 11(1):1–20.

Attiya, Hagit, Amotz Bar-Noy, Danny Dolev, David Peleg and Rüdiger Reischuk. 1990. "Renaming in an Asynchronous Environment." *Journal of the ACM (JACM)* 37(3):524–548.

Basaran, Can and Kyoung-Don Kang. 2012. Supporting Preemptive Task Executions and Memory Copies in GPGPUs. In *Euromicro Conference on Real-Time Systems (ECRTS)*.

Bauman, Paul, Noel Chalmers, Nick Curtis, Chip Freitag, Joe Greathouse, Nicholas Malaya, Damon McDougall, Scott Moe, René van Oostrum and Noah Wolfe. 2019. "Introduction to AMD GPU Programming with HIP." Presentation at Oak Ridge National Laboratory. Online at: `https://www.olcf.ornl.gov/calendar/intro-to-amd-gpu-programming-with-hip/`.

Benchoff, Brian. 2017. "Hands-On Nvidia Jetson TX2: Fast Processing For Embedded Devices." Online at `https://hackaday.com/2017/03/14/hands-on-nvidia-jetson-tx2-fast-processing-for-embedded-devices/`.

Berezovskyi, Kostiantyn, Konstantinos Bletsas and Stefan M Petters. 2013. Faster Makespan Estimation for GPU Threads on a Single Streaming Multiprocessor. In *IEEE Conference on Emerging Technologies & Factory Automation (ETFA)*.

"bridgman". 2016. "amdgpu questions." Phoronix Forums. Online at `https://www.phoronix.com/forums/forum/linux-graphics-x-org-drivers/open-source-amd-linux/856534-amdgpu-questions?p=857850\#post857850`. Accessed in 2020.

Capodieci, Nicola, Roberto Cavicchioli, Marko Bertogna and Aingara Paramakuru. 2018. Deadline-based Scheduling for GPU with Preemption Support. In *IEEE Real-Time Systems Symposium (RTSS)*.

Capodieci, Nicola, Roberto Cavicchioli, Paolo Valente and Marko Bertogna. 2017. Sigamma: Server-Based Integrated GPU Arbitration Mechanism for Memory Accesses. In *International Conference on Real-Time Networks and Systems (RTNS)*.

Chen, Guoyang, Yue Zhao, Xipeng Shen and Huiyang Zhou. 2017. Effisha: A Software Framework for Enabling Effficient Preemptive Scheduling of GPU. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

Chisholm, Micaiah, Namhoon Kim, Bryan C Ward, Nathan Otterness, James H Anderson and F Donelson Smith. 2016. Reconciling the Tension Between Hardware Isolation and Data Sharing in Mixed-Criticality, Multicore Systems. In *IEEE Real-Time Systems Symposium (RTSS)*.

Chisholm, Micaiah, Namhoon Kim, Stephen Tang, Nathan Otterness, James H Anderson, F Donelson Smith and Donald E Porter. 2017. Supporting Mode Changes While Providing Hardware Isolation in Mixed-Criticality Multicore Systems. In *International Conference on Real-Time Networks and Systems (RTNS)*.

Ciresan, Dan Claudiu, Ueli Meier, Jonathan Masci, Luca Maria Gambardella and Jürgen Schmidhuber. 2011. Flexible, High Performance Convolutional Neural Networks for Image Classification. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li and Li Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Elliott, Glenn. 2015. Real-time Scheduling for GPUs With Applications in Advanced Automotive Systems PhD thesis The University of North Carolina at Chapel Hill.

Elliott, Glenn A, Bryan C Ward and James H. Anderson. 2013. GPUSync: A Framework for Real-time GPU Management. In *IEEE Real-Time Systems Symposium (RTSS)*.

Elliott, Glenn and James H. Anderson. 2011. An Optimal k-Exclusion Real-Time Locking Protocol Motivated by Multi-GPU Systems. In *International Conference on Real-Time Networks and Systems (RTNS)*.

Forsberg, Björn, Andrea Marongiu and Luca Benini. 2017. GPUguard: Towards Supporting a Predictable Execution Model for Heterogeneous SoC. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*.

Franklin, Dustin. 2017. "NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge." Online at `https://developer.nvidia.com/blog/jetson-tx2-delivers-twice-intelligence-edge/`.

Fujii, Yusuke, Takuya Azumi, Nobuhiko Nishio and Shinpei Kato. 2013. Exploring Microcontrollers in GPUs. In *Asia-Pacific Workshop on Systems (APSys)*.

Han, Xian-Feng, Hamid Laga and Mohammed Bennamoun. 2019. "Image-Based 3D Object Reconstruction: State-of-the-art and Trends in the Deep Learning Era." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43(5):1578–1604.

He, Horace. 2019. "The State of Machine Learning Frameworks in 2019." Online at `https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/`.

Howard, Andrew G., Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto and Adam Hartwig. 2017. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.".

HSA Foundation. 2018*a*. "HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer, and Object Format (BRIG), Version 1.2." Online at `http://hsa.glossner.org/wp-content/uploads/2021/02/HSA-PRM-1.2.pdf`.

HSA Foundation. 2018*b*. "HSA Runtime Programmer's Reference Manual, Version 1.2." Online at `http://hsa.glossner.org/wp-content/uploads/2021/02/HSA-Runtime-1.2.pdf`.

Jain, Saksham, Iljoo Baek, Shige Wang and Ragunathan (Raj) Rajkumar. 2019. Fractional GPUs: Software-based Compute and Memory Bandwidth Reservation for GPUs. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Janzén, Johan, David Black-Schaffer and Andra Hugo. 2016. Partitioning GPUs for Improved Scalability. In *IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*.

Jia, Yangqing, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama and Trevor Darrell. 2014. "Caffe: Convolutional Architecture for Fast Feature Embedding." *arXiv preprint arXiv:1408.5093* .

Jia, Zhe, Marco Maggioni, Benjamin Staiger and Daniele Paolo Scarpazza. 2018. "Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking." *CoRR* abs/1804.06826.
URL: *http://arxiv.org/abs/1804.06826*

Jia, Zhe, Marco Maggioni, Jeffrey Smith and Daniele Paolo Scarpazza. 2019. "Dissecting the NVidia Turing T4 GPU via Microbenchmarking." *CoRR* abs/1903.07486.
URL: *http://arxiv.org/abs/1903.07486*

Karki, Aajna, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde and Hyeran Jeon. 2019. Tango: A Deep Neural Network Benchmark Suite for Various Accelerators. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

Karumbunathan, Leela S. 2022. "NVIDIA Jetson AGX Orin Series: A Giant Leap Forward for Robotics and Edge AI Applications (Technical Brief)." Online at `https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf`.

Kato, Shinpei, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa and Ragunathan Rajkumar. 2011. RGEM: A Responsive GPGPU Execution Model for Runtime Engines. In *IEEE Real-Time Systems Symposium (RTSS)*.

Kato, Shinpei, Karthik Lakshmanan, Raj Rajkumar and Yutaka Ishikawa. 2011. TimeGraph: GPU Scheduling for Real-time Multi-tasking Environments. In *USENIX ATC*.

Kato, Shinpei, Michael McThrow, Carlos Maltzahn and Scott Brandt. 2012. Gdev: First-class GPU Resource Management in the Operating System. In *USENIX Annual Technical Conference (ATC)*.

Khronos Group. 2020. "OpenCL: Open Standard for Parallel Programming of Heterogeneous Systems." Online at `https://www.khronos.org/opencl/`.

Khronos Group. 2021. "OpenGL: The Industry Standard for High Performance Graphics." Online at `https://www.opengl.org/`.

Khronos Vulkan Working Group. 2022. "Vulkan 1.3.212 - A Specification." Available online from `https://www.khronos.org/registry/vulkan/`.

Kim, Eunwoo, Chanho Ahn and Songhwai Oh. 2018. NestedNet: Learning Nested Sparse Structures in Deep Neural Networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Kim, Hyoseung, Pratyush Patel, Shige Wang and Ragunathan Raj Rajkumar. 2017. A Server-based Approach for Predictable GPU Access Control. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*.

Kim, Namhoon, Micaiah Chisholm, Nathan Otterness, James H Anderson and F Donelson Smith. 2017. Allowing Shared Libraries While Supporting Hardware Isolation in Multicore Real-Time Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Kim, Namhoon, Stephen Tang, Nathan Otterness, James H Anderson, F Donelson Smith and Donald E Porter. 2020. "Supporting I/O and IPC via Fine-Grained OS Isolation for Mixed-Criticality Real-Time Tasks." *Real-Time Systems* 56(4):349–390.

Krizhevsky, Alex, Ilya Sutskever and Geoffrey E Hinton. 2012. "Imagenet Classification With Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems* 25:1097–1105.

Larabel, Michael. 2020. "The AMD Radeon Graphics Driver Makes Up Roughly 10.5% Of The Linux Kernel." *Phoronix.com* . Online at `https://www.phoronix.com/scan.php?page=news_item&px=Linux-5.9-AMDGPU-Stats`.

Lee, Haeseung and Mohammad Abdullah Al Faruque. 2014. GPU-EvR: Run-Time Event Based Teal-Time Scheduling Framework on GPGPU Platform. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE.

Lee, Haeseung and Mohammad Abdullah Al Faruque. 2016. "Run-time Scheduling Framework for Event-driven Applications on a GPU-based Embedded System." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* .

Lee, Hyeonsu, Jaehun Roh and Euiseong Seo. 2018. A GPU Kernel Transactionization Scheme for Preemptive Priority Scheduling. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Liang, Yun, Xiuhong Li and Xiaolong Xie. 2017. Exploring Cache Bypassing and Partitioning for Multi-tasking on GPUs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

Litjens, Geert, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen Awm Van Der Laak, Bram Van Ginneken and Clara I Sánchez. 2017. "A Survey on Deep Learning in Medical Image Analysis." *Medical Image Analysis* 42:60–88.

Liu, Chung Laung and James W Layland. 1973. "Scheduling Algorithms for Multiprogramming in a Hard-real-time Environment." *Journal of the ACM (JACM)* 20(1):46–61.

Mei, Xinxin and Xiaowen Chu. 2016. "Dissecting GPU Memory Hierarchy Through Microbenchmarking." *IEEE Transactions on Parallel and Distributed Systems (TPDS)* .

Mellor-Crummey, John M and Michael L Scott. 1991. "Algorithms for Scalable Synchronization on Shared-memory Multiprocessors." *ACM Transactions on Computer Systems (TOCS)* 9(1):21–65.

Memon, Jamshed, Maira Sami, Rizwan Ahmed Khan and Mueen Uddin. 2020. "Handwritten Optical Character Recognition (OCR): A Comprehensive Systematic Literature Review (SLR)." *IEEE Access* 8:142642–142668.

Mok, Aloysius Ka-Lau. 1983. Fundamental Design Problems of Distributed Systems for the Hard-real-time Environment PhD thesis Massachusetts Institute of Technology.

Mujtaba, Hassan. 2021. "NVIDIA & Intel GPU Market Share Increased While AMD Declined In Q2 2021, GPU Shipments Hit 37% Growth Year-Over-Year." Online at `https://wccftech.com/nvidia-intel-gpu-market-share-increased-while-amd-declined-in-q2-2021/`.

NVIDIA. 2021. "NVIDIA cuDNN." Online at `https://developer.nvidia.com/cudnn`.

NVIDIA. 2022. "CUDA Toolkit." Online at `https://developer.nvidia.com/cuda-toolkit`.

NVIDIA Corporation. 2014. "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210." Available online from `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf`.

NVIDIA Corporation. 2017. "NVIDIA Tesla V100 GPU Architecture." Online at `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`.

NVIDIA Corporation. 2020*a*. "Multi-Process Service." Online at `https://docs.nvidia.com/pdf/CUDA_Multi_Process_Service_Overview.pdf`.

NVIDIA Corporation. 2020*b*. "NVIDIA A100 Tensor Core GPU Architecture." Online at `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`.

NVIDIA Corporation. 2020*c*. "NVIDIA Multi-Instance GPU User Guide." Online at `https://docs.nvidia.com/datacenter/tesla/mig-user-guide/`.

NVIDIA Corporation. 2022*a*. "CUDA C++ Best Practices Guide." Online at `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`. Version 11.7.0.

NVIDIA Corporation. 2022*b*. "CUDA C Programming Guide." Online at https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Version 11.7.0.

NVIDIA Corporation. 2022*c*. "CUDA Occupancy Calculator." Online at https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html. Version 11.7.0.

NVIDIA Corporation. 2022*d*. "NVIDIA H100 Tensor Core GPU Architecture." Online at https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper.

Olmedo, Ignacio Sañudo, Nicola Capodieci, Jorge Luis Martínez, Andrea Marongiu and Marko Bertogna. 2020. Dissecting the CUDA Scheduling Hierarchy: A Performance and Predictability Perspective. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Otterness, Nathan and James H. Anderson. 2020. AMD GPUs as an Alternative to NVIDIA for Supporting Real-Time Workloads. In *Euromicro Conference on Real-Time Systems (ECRTS)*.

Otterness, Nathan and James H. Anderson. 2021. Exploring AMD GPU Scheduling Details by Experimenting With "Worst Practices". In *International Conference on Real-Time Networks and Systems (RTNS)*.

Otterness, Nathan and James H. Anderson. 2022. "Exploring AMD GPU Scheduling Details by Experimenting With "Worst Practices"." *Real-Time Systems* 58(2):105–133.

Otterness, Nathan, Ming Yang, Sarah Rust, Eunbyung Park, James Anderson, F.D. Smith, Alex Berg and Shige Wang. 2017. An Evaluation of the NVIDIA TX1 for Supporting Real-Time Computer-Vision Workloads. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Otterness, Nathan, Ming Yang, Tanya Amert, James Anderson and F. D. Smith. 2017. Inferring the Scheduling Policies of an Embedded CUDA GPU. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*.

Otterness, Nathan, Vance Miller, Ming Yang, James Anderson and F.D. Smith. 2016. GPU Sharing for Image Processing in Embedded Real-Time Systems. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga *et al.*. 2019. "PyTorch: An Imperative Style, High-performance Deep Learning Library." *Advances in Neural Information Processing Systems* 32:8026–8037.

Pellizzoni, Rodolfo, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo and Russell Kegley. 2011. A Predictable Execution Model for COTS-based Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Peres, Martin. 2013. Reverse Engineering Power Management on NVIDIA GPUs-Anatomy of an Autonomic-Ready System. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*.

Puthoor, Sooraj, Xulong Tang, Joseph Gross and Bradford M. Beckmann. 2018. Oversubscribed Command Queues in GPUs. In *ACM Workshop on General Purpose GPUs (GPGPU)*.

Reiff, Nathan. 2022. "How Nvidia Makes Money." Online at https://www.investopedia.com/how-nvidia-makes-money-4799532.

Saha, Sujan Kumar. 2018. Spatio-Temporal GPU Management for Real-Time Cyber-Physical Systems. Master's thesis UC Riverside.

Schmidhuber, Jürgen. 2015. "Deep Learning in Neural Networks: An Overview." *Neural networks* 61:85–117.

Sites, Richard L. 2021. *Understanding Software Dynamics*. Pearson Addison-Wesley.

Smith, Ryan and Ganesh T. S. 2014. "The NVIDIA GeForce GTX 750 Ti and GTX 750 Review: Maxwell Makes Its Move." Available online from `https://www.anandtech.com/show/7764/the-nvidia-geforce-gtx-750-ti-and-gtx-750-review-maxwell`.

Sorensen, Tyler, Hugues Evrard and Alastair F Donaldson. 2018. GPU Schedulers: How Fair is Fair Enough? In *International Conference on Concurrency Theory (CONCUR)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

van Oostrum, René, Noel Chalmers, Damon McDougall, Paul Bauman Bauman, Nicholas Curtis, Nicholas Malaya and Noah Wolfe. 2019. "AMD GPU Hardware Basics." Presentation at Oak Ridge National Laboratory. Online at: `https://www.olcf.ornl.gov/wp-content/uploads/2019/10/ORNL_Application_Readiness_Workshop-AMD_GPU_Basics.pdf`.

Verner, Uri, Assaf Schuster, Mark Silberstein and Avi Mendelson. 2012. Scheduling Processing of Real-time Data Streams on Heterogeneous Multi-GPU Systems. In *ACM International Systems and Storage Conference*.

Verner, Uri, Avi Mendelson and Assaf Schuster. 2014*a*. Batch Method for Efficient Resource Sharing in Real-time Multi-GPU Systems. In *International Conference on Distributed Computing and Networking*. Springer.

Verner, Uri, Avi Mendelson and Assaf Schuster. 2014*b*. Scheduling Periodic Real-time Communication in Multi-GPU Systems. In *IEEE International Conference on Computer Communication and Networks (ICCCN)*.

Vu, Thanh, Marc Eder, True Price and Jan-Michael Frahm. 2020. Any-Width Networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*.

Wang, Po-Han, Cheng-Hsuan Li and Chia-Lin Yang. 2016. Latency Sensitivity-based Cache Partitioning for Heterogeneous Multi-core Architecture. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*.

Yang, Ming. 2020. Sharing GPUs for Real-Time Autonomous-Driving Systems PhD thesis The University of North Carolina at Chapel Hill.

Yang, Ming, Nathan Otterness, Tanya Amert, Joshua Bakita, James H. Anderson and F Donelson Smith. 2018. Avoiding Pitfalls when using NVIDIA GPUs for Real-time Tasks in Autonomous Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*.

Yang, Ming, Shige Wang, Joshua Bakita, Thanh Vu, F Donelson Smith, James H. Anderson and Jan-Michael Frahm. 2019. Re-thinking CNN Frameworks for Time-sensitive Autonomous-driving Ppplications: Addressing an Industrial Challenge. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Yang, Ming, Tanya Amert, Kecheng Yang, Nathan Otterness, James H. Anderson, F. Donelson Smith and Shige Wang. 2018. Making OpenVX Really 'Real Time'. In *IEEE Real-Time Systems Symposium (RTSS)*.

Yu, Jiahui, Linjie Yang, Ning Xu, Jianchao Yang and Thomas Huang. 2019. Slimmable Neural Networks. In *International Conference on Learning Representations (ICLR)*.

Yu, Jiahui and Thomas Huang. 2019*a*. "AutoSlim: Towards One-Shot Architecture Search for Channel Numbers." *arXiv preprint arXiv:1903.11728* .

Yu, Jiahui and Thomas Huang. 2019*b*. Universally Slimmable Networks and Improved Training Techniques. In *IEEE/CVF International Conference on Computer Vision (ICCV)*.

Yu, Jiahui, Zirui Wang, Vijay Vasudevan, Legg Yeung, Mojtaba Seyedhosseini and Yonghui Wu. 2022. "CoCa: Contrastive Captioners Are Image-Text Foundation Models." *arXiv preprint arXiv:2205.01917* .

Yun, Heechul, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo and Lui Sha. 2013. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Zhong, Jianlong and Bingsheng He. 2013. "Kernelet: High-throughput GPU Kernel Executions with Dynamic Slicing and Scheduling." *IEEE Transactions on Parallel and Distributed Systems (TPDS)* .

Zhou, Husheng, Guangmo Tong and Cong Liu. 2015. GPES: A Preemptive Execution System for GPGPU Computing. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.