Inferring Scheduling Policies of an Embedded CUDA GPU[*]

Nathan Otterness, Ming Yang, Tanya Amert, James H. Anderson, F. Donelson Smith
[1]Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

Embedded systems augmented with graphics processing units (GPUs) are seeing increased use in safety-critical real-time systems such as autonomous vehicles. Due to monetary cost requirements along with size, weight, and power (SWaP) constraints, embedded GPUs are often computationally impoverished compared to those used in non-embedded systems. In order to maximize performance on these impoverished GPUs, we examine co-scheduling*: allowing multiple applications concurrent access to a GPU. In this work, we use a new benchmarking framework to examine internal scheduling policies of the black-box hardware and software used to co-schedule GPU tasks on the NVIDIA Jetson TX1.*

1 Introduction

Fueled largely by the burgeoning autonomous vehicle industry, the demands being made of safety-critical embedded computers are growing at unprecedented rates. The monetary cost requirements and size, weight, and power (SWaP) constraints placed on embedded systems have resulted in traditional microprocessors being hard-pressed to provide the computing capacity needed for computation- and data-intensive tasks, such as analyzing multiple video feeds. To overcome the limits of traditional microprocessors, developers of autonomous vehicles are increasingly turning to specialized hardware such as graphics processing units (GPUs).

GPU manufacturers such as NVIDIA are embracing this new use case, as evidenced by offerings such as the Jetson TX1: a GPU-augmented single-board computer expressly designed for embedded development [7]. Such a platform meets the financial, SWaP, and computational requirements of modern embedded systems. Unfortunately, less attention has been given to the *safety-critical* aspects of autonomous systems, as mainstream GPU manufacturers have not provided key information needed for *certification*.

On one hand, this is not unexpected given typical GPU use cases: gaming and, increasingly often, throughput-oriented high-performance computing. On the other hand, information such as cache replacement policies, DRAM organization, and job scheduling are essential for the accurate calculation and verification of safety-critical temporal properties. In this paper, we present a new experimental framework and some results illuminating one of these topics: a selection of the GPU's scheduling rules. A framework like ours is necessary for evaluating behavior of the GPU's black-box components, which includes hardware, closed-source drivers, and user-level libraries.

Prior work and GPU co-scheduling. Due to the black-box behavior of most GPUs, a significant body of prior work in real-time GPU management has chosen to enforce exclusive GPU access [2, 3, 4, 11, 12, 13]. These works, which only allow a single task to execute at a time on a GPU, incur capacity loss if a task does not require all GPU resources. This may be acceptable on multi-GPU systems, but, on less-capable embedded GPUs, all possible processing cycles should be available to maximize performance. Other works have focused on subdividing GPU jobs into smaller, more manageable chunks to improve schedulability [1, 3, 5, 15]. Of particular note is a framework called *Kernelet* [14], which subdivides GPU tasks into smaller sub-tasks that can be co-scheduled. *Kernelet*, however, does not provide an in-depth investigation into how co-scheduled tasks actually behave aside from the observation that co-scheduling can lead to performance benefits.

Prior work by our group investigated co-scheduling GPU operations issued by separate CPU processes [8, 9]. In brief, this work found that GPU operations requested from separate CPU processes were co-scheduled via *multiprogramming*, where the GPU would dedicate all resources to a portion of a single operation and allow this portion to complete before switching to a portion of a different operation. Given this behavior, GPU operations from different processes are, in a sense, never executed concurrently because operations from different processes never have threads assigned to the GPU at the same time. This limitation does not, however, apply to GPU operations issued from multiple CPU threads within a single address space. Because execution from a single address is necessary to enable a GPU to truly execute different operations concurrently, this context is our focus in this paper.

Contributions. In this work, we present a new framework designed to enable observing the way GPU jobs are scheduled. We use these observations to infer a selected subset of (to our knowledge) undocumented GPU scheduling policies for the NVIDIA Jetson TX1.

Organization. In Sec. 2 of this paper, we describe our test platform and introduce terminology essential when describing GPU scheduling. We then describe our experimental framework in Sec. 3, detail the scheduling policies we infer

*Work supported by NSF grants CNS 1409175, CPS 1446631, and CNS 1563845, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and funding from General Motors.

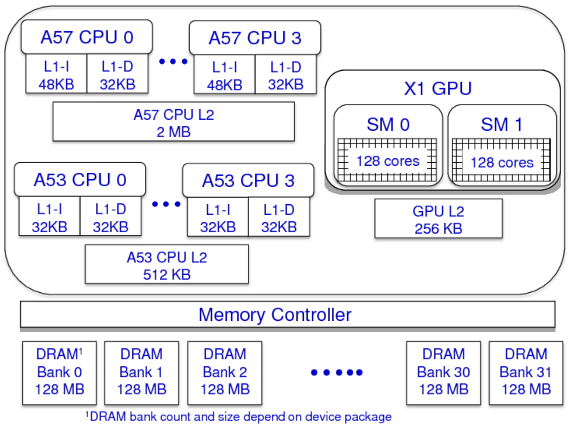

Figure 1: Jetson TX1 architecture.

in Sec. 4, and report our results in Sec. 5. Finally, we discuss the future direction of this research and conclude in Sec. 6.

2 Background

Our test platform. We carried out our experiments on the NVIDIA Jetson TX1, a single-board computer with a quad-core 64-bit ARM CPU, an integrated CUDA-capable GPU, and 4 GB of DRAM shared between the GPU and CPU. As mentioned in Sec. 1, this platform is relatively inexpensive, accessible, and geared towards embedded development. Fig. 1 provides a high-level overview of the TX1.

CUDA programming basics. GPUs can be viewed as co-processors that carry out work requested by a CPU process. Our experiments focus on CUDA, which is an API used to interact with NVIDIA GPUs. Requests made to the GPU via CUDA typically complete asynchronously, meaning that a single CPU process can enqueue multiple requests and must explicitly wait for requests to complete.

We supply the following brief list of CUDA terminology along with Fig. 2 to provide a few necessary definitions:

- *CUDA kernel*: A section of code that runs on the GPU.
- *Thread block (block)*: A collection of GPU threads that all execute concurrently and run the same instructions, but operate on different portions of data. The number of threads in a block and the number of blocks associated with a CUDA kernel are specified at runtime.
- *Streaming Multiprocessor (SM)*: The individual cores in a CUDA-capable GPU are partitioned into SMs. On the TX1, up to 2,048 threads can be assigned to an SM. Threads within a single block will never simultaneously execute on different SMs.
- *CUDA Stream (stream)*: A FIFO queue of CUDA kernels and memory-transfer operations to be run by the GPU. A single CPU process or thread can attempt to issue concurrent GPU operations by placing them in multiple streams.

Fig. 2 summarizes the hierarchy of how CUDA programs, kernels, and thread blocks are related. A CUDA program

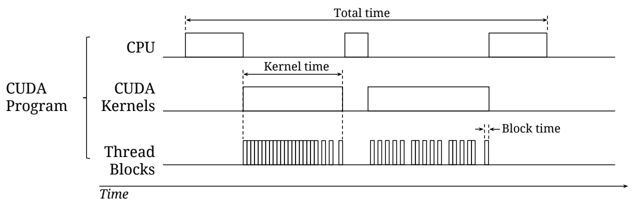

Figure 2: Diagram illustrating the relation between CUDA programs, kernels, and thread blocks.

consists of CPU code that invokes GPU code, which in turn is contained in CUDA kernels. The execution of a kernel requires running a programmer-specified number of thread blocks on the GPU. Not shown in Fig. 2 is the fact that multiple thread blocks from the same kernel can execute concurrently if sufficient GPU resources exist. We refer to the time taken to execute a single thread block as *block time*, the time taken from the invocation to completion of a CUDA kernel as *kernel time*, and the time taken by an entire CUDA program (including CPU portions) as *total time*.

A full understanding of CUDA is not necessary to understand this paper, and details about issues of relevance to us will be given in Sec. 3. In addition, this work focuses exclusively on scheduling GPU code issued by a single CPU process, so several important issues are beyond the scope of this paper (such as GPU memory management). We refer readers to one of our prior works [9] where we consider some of these questions in more detail.

3 Experimental Approach

In this section, we provide an overview of our experimental approach. We begin by delving into some open questions about scheduling on CUDA-capable GPUs. Afterwards, we describe the experimental framework we created for submitting short, handcrafted scenarios to the GPU and monitoring the GPU scheduler's behavior.

Documented and undocumented CUDA scheduling behavior. The official CUDA documentation contains almost no information about how CUDA kernels are scheduled, apart from the facts that kernels within a CUDA stream complete sequentially and that kernels from different streams *may* run concurrently.[1] However, the exact conditions for when kernels from different streams *will* run concurrently, or the default ordering of kernels from different streams, is not explicitly stated and is likely to be hardware-dependent. One semi-official presentation from 2011 [10] gives slightly more detail, and states that kernels from different streams are placed into a single internal queue in issue order, and that the head of the internal queue *may* be allowed to run concurrently with other kernels if sufficient resources exist. However, this talk covered an older GPU architecture; notably, newer NVIDIA GPUs contain multiple internal queues [6]. Fur-

[1]For example, this is the description of streams given in Sec. 9.1.2 of the *Best Practices Guide* for CUDA version 8.0.61.

thermore, our own prior work found that aspects of CUDA programs as fundamental as memory access APIs were subject to undocumented changes between software updates [9].

In brief, the high-level documentation and evolving GPU architecture have left us with the following two questions about the Jetson TX1's GPU scheduling: First, under what conditions will two kernels from different streams be scheduled concurrently? Second, if multiple streams have pending kernels that cannot be scheduled concurrently, how are the kernels from different streams prioritized?

A new framework for examining GPU scheduling. To answer these questions, we designed a new testing framework that enables scenarios to be set up in which the issue order and resource requirements of GPU kernels can be carefully controlled.[2] Additionally, we wanted the framework to gather detailed scheduling information, have a modular interface for supporting different GPU workloads, and have inputs and outputs that facilitate scripting. The current framework consists of approximately 2,700 lines of C and CUDA code and is available online.[3]

Our framework is used by providing a configuration file describing a particular scenario. In this paper, we use the term *task* to refer to a CPU thread that issues GPU work. All tasks in a scenario would share a single address space. Configuration files specify how many tasks should run, how many kernels each task submits to the GPU, how long each kernel should run, the number of threads per block, and the total number of thread blocks per kernel invocation. Release order is configured by specifying an amount of time each task must sleep before issuing GPU kernels.

After a scenario completes execution, the framework produces one output file per task, each of which contains a list of the start and end times for every block in every kernel submitted by that task. Additionally, the framework reports the ID of the SM on which each block ran. By combining this block-level information from all kernels and tasks in the scenario, we can obtain a complete view of how the scenario was scheduled.

In order to facilitate scripting, both configuration and output files use the JSON file format, which is a commonly supported plain-text format for serializing hierarchical information. All block timing and SM IDs are recorded on the GPU itself, by reading the `globaltimer` and `smid` registers available to CUDA kernel code. Our observations only depend on the relative ordering of GPU times, which eliminates the need to synchronize CPU and GPU time. The `globaltimer` register, which maintains a count of nanoseconds, was also used to implement our primary test kernel, which spins in a busy loop until a user-specified number of nanoseconds has elapsed. The results we present in Sec. 5 still hold for other CUDA kernels, but most of our experiments involved the simple busy-waiting kernel so that the we could easily control the kernels' execution times with minimal interference from co-scheduled work.

[2]Among other things, this requires our framework to be the only GPU-using process. We experienced confusing and inconsistent results in some early experiments, and later found that this was due to not disabling the X window server. We did not have displays connected to the systems, so we did not realize that X would start by default and consume GPU resources.

[3]`https://github.com/yalue/cuda_scheduling_examiner_mirror`

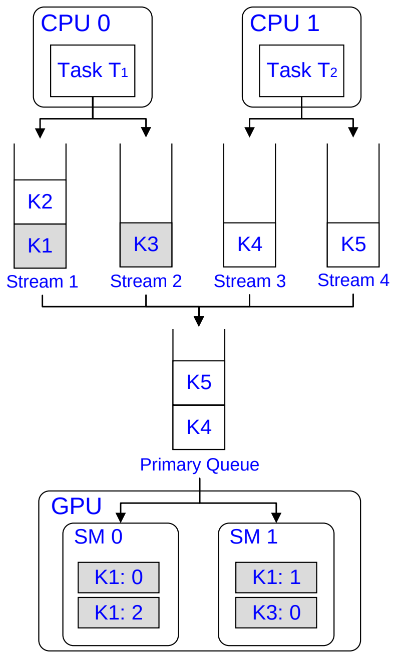

Figure 3: Structure of Streams and Primary Queue.

4 GPU Scheduling Rules

In this section, we present rules that, to our knowledge, govern how the TX1's GPU scheduler assigns work from multiple streams within a single CPU process to the GPU. For this set of rules, the GPU scheduler consists of one FIFO *primary queue* (per address space), and, as described in Sec. 2, one FIFO queue per CUDA stream.[4] This layout is depicted in Fig. 3, which is explained in detail after presenting the rules below.

According to our observations using CUDA version 8.0 and simple workloads being submitted from a single address space, 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.

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

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

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

Summary of GPU scheduling rules. Rules A and D restate the rule mentioned in Sec. 2 that kernels submitted to a single stream are always handled 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 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. We provided Fig. 3, which we next describe in detail, as a visual example of these rules' applications.

Example of GPU scheduling rules. In Fig. 3, two tasks each use two separate streams to submit kernels to the GPU. In total, these two tasks submit five kernels, labeled K1-K5. Each kernel may have multiple blocks, so kernel K1's i^{th} block is labeled "K1: i," and K3's single block is labeled similarly. 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).

GPU resource requirements. Rule B2 encompasses in itself a fairly complex set of constraints. In official documentation, the factors that determine the GPU resource requirements of a CUDA kernel are all condensed into a single metric known as *occupancy*.[5] A kernel invocation's occupancy is based on which of three GPU resources will be most constrained by that particular kernel. The GPU resources considered in the occupancy calculation are GPU threads, GPU registers, and shared memory.[6] CUDA GPUs have other limits on execution in addition to occupancy, such as a global maximum number of kernels per GPU, but these limits are usually fairly high. In this work, we focused on determining rules governing thread resource requirements, but our experimental approach could also be used to investigate other occupancy-related restrictions.

When considering GPU resource requirements such as threads, it is imperative to remember that a CUDA GPU is organized into SMs. As described in Sec. 2, each SM is associated with a cluster of CUDA cores, and groups of threads from CUDA kernels are assigned to SMs. On the TX1, the maximum number of threads that can be concurrently assigned to a single SM is 2,048.[7] Thread blocks are always fully assigned to a single SM, so if only 512 threads are available on each of the TX1's two SMs, an incoming block of 1,024 threads cannot be scheduled. We show scenarios where this behavior can lead to unnecessary blocking at the end of the next section, after our experiments to validate the set of scheduling rules.

5 Evaluation

In this section, we present a sample of experimental results that illustrate each of the rules in Sec. 4. All of these experiments were carried out using the experimental framework described in Sec. 3.

Interpreting the plots. For each experiment, we present one or more plots showing the time at which thread blocks were assigned to one of the TX1's two SMs. In these plots, each thread block is represented by a rectangle, with the left edge corresponding to the block's start time on the horizontal axis, the right edge corresponding to its end time, and the height proportional to the number of threads in the block. Blocks are individually labeled with their associated kernel followed by their block number. The plots are subdivided into upper and lower halves representing the two available SMs, and blocks are located in the half corresponding to the SM on which they executed. Apart from SM assignment, the vertical ordering of blocks may be arbitrary. Finally, all blocks issued to the same stream will have identical shades and patterns within a single plot.

Simple experiments corroborating Rules B1, B2, and C. Our first, basic tests were carried out to simply verify that co-scheduling can occur when multiple kernels are submitted from different streams in a single address space, 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 indicated by Rules A and D is trivial in these cases. Results of this first set of experiments are represented in Figs. 4 and 5.

Of these first two experiments, Fig. 4 represents the simplest, optimal co-scheduling situation in which we released kernels K1 and K2 at time $t = 0s$ and kernels K3 and K4 at time $t = 0.25s$. Each kernel was released in a separate stream, configured to run for the same amount of time, and

[5]The official occupancy calculator can be found at `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`

[6]On a CUDA GPU, *shared memory* refers to a small region of low-latency memory through which GPU threads can communicate.

[7]This number can be calculated from the *Compute Capabilites* table in the CUDA Programming Guide.

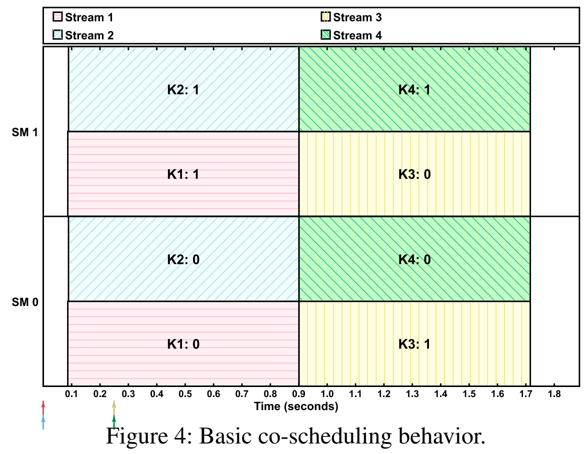

Figure 4: Basic co-scheduling behavior.

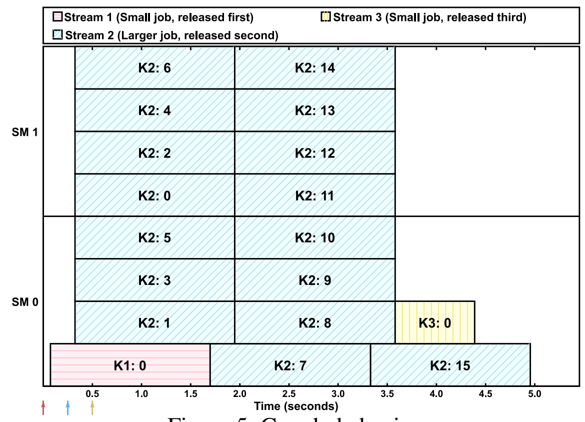

Figure 5: Greedy behavior.

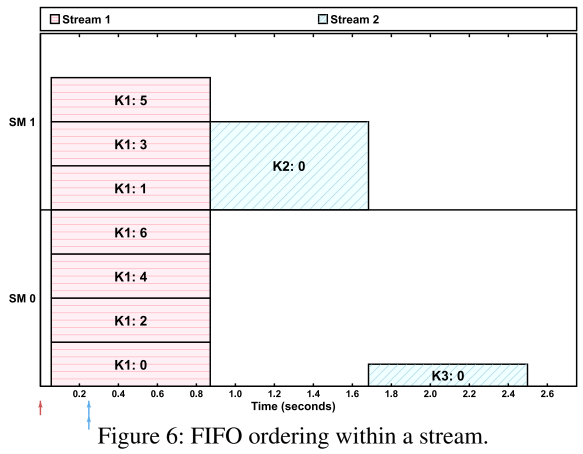

Figure 6: FIFO ordering within a stream.

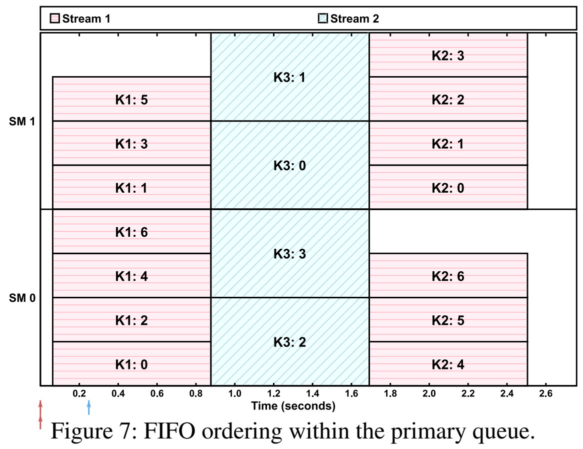

Figure 7: FIFO ordering within the primary queue.

required two blocks of 1,024 threads. The kernels that were released first, K1 and K2, were co-scheduled due to Rule B2 because 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. K3 and K4 could not commence execution until one of the first two kernels completed, freeing thread resources.

The second experiment, depicted in Fig. 5, illustrates the greedy behavior required by Rule C. Kernel K1, requiring few thread resources, was released at time $t = 0s$. Next, kernel K2 was released at time $t = 0.25s$ and began execution immediately. K2, however, required executing 16 blocks of 512 threads, which exceeded the GPU's capacity. Kernel K3, requiring few thread resources, was released at time $t = 0.5s$, but the scheduler did not allow it to execute until K2 had no blocks left to assign to the GPU. In accordance with Rule C, K3 was able to reach the head of the primary queue and begin executing while the final block of K2 was still completing.

Experiments corroborating Rules A and D. Our first set of experiments 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 Figs. 6 and 7.

Fig. 6 contains an example of how kernels within a single stream are executed in FIFO order. In this figure, kernels K2 and K3 were issued to a single stream, and, in accordance with Rules A and D, K3 did not begin execution until after K2 completed. Furthermore, K2 required too many resources to execute concurrently with K1, even though K1 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.

We provide Fig. 7 as a second illustration of Rules A and D. Unlike in Fig. 6, the kernels in Fig. 7 were executed in a different order from that in which they were issued. Kernels K1 and K2 were issued back-to-back at time $t = 0s$ into the same stream, and kernel K3 was issued into a separate stream at time $t = 0.25s$. Even though K2 was issued first, K3 executed before K2 because Rule D prevented K2 from reaching the head of its stream until K1 completed. Kernel K3, on the other hand, reached the head of its stream and entered the primary queue as soon as it was submitted.

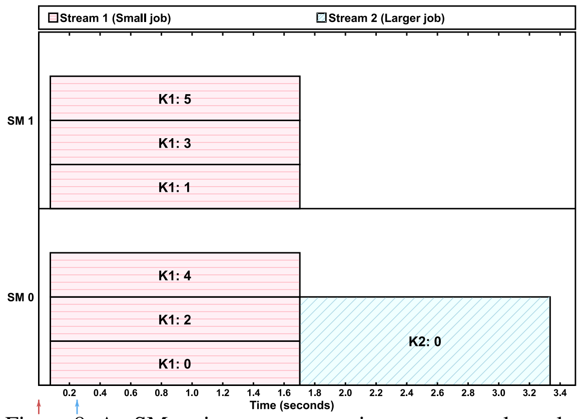

Figure 8: An SM assignment preventing concurrent kernels.

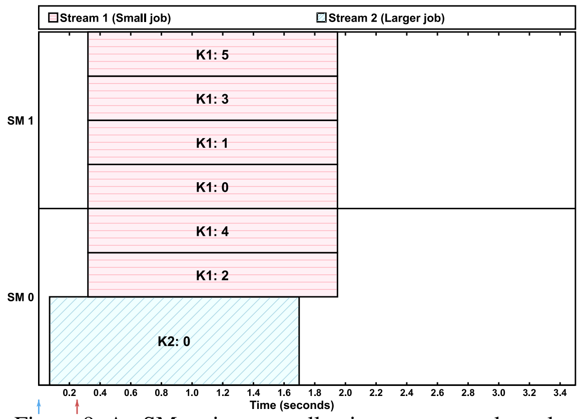

Figure 9: An SM assignment allowing concurrent kernels.

The impact of SMs on resource constraints. So far, our experiments focused on illustrating the behaviors caused by the scheduling rules, but, as noted in Sec. 4, Rule B2 about resource constraints requires considering the GPU as a set of SMs. This, along with the restriction that threads from a single block cannot be split across multiple SMs, prevents concurrency in some situations. We conducted one final experiment both to illustrate one such situation, and to demonstrate how concurrency can be improved with minor changes to issue order.

The first of two related scenarios is presented in Fig. 8. In this figure, kernel K1, requiring 6 blocks of 512 threads, was released at time $t = 0s$. Since nothing else was currently executing, the GPU scheduler evenly distributed K1's six blocks across the TX1's two SMs, leaving only 512 unassigned threads remaining on each SM. This meant that when K2, requiring a single block of 1,024 threads, was released at time $t = 0.25s$, it had to wait because neither SM could hold 1,024 threads.

Fig. 9 contains the same two kernels as Fig. 8, but with the two kernels released in the opposite order. Here, K2's single larger block was assigned to SM 0, and the GPU scheduler distributed K1's six blocks to fill up all remaining thread resources. 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. We hope to infer a set of rules describing exactly how blocks are assigned to SMs in future work, but for now it remains an open question.

6 Conclusion

In this work, we presented only part of an ongoing effort to force some undocumented GPU hardware features into the open. We eventually hope to expand the set of rules presented here to a point where it is possible to draw broader conclusions about schedulability for task systems that share a single GPU. The evaluation in Sec. 5 already contains examples of non-work-conserving scheduling that can be predicted or even mitigated with slight foreknowledge about a task system, coupled with what we now know about the GPU scheduler.

We hope that GPU manufacturers come to realize that transparency is a valuable feature in an embedded system, but the reality is that developers of autonomous systems are not willing to wait. Therefore, it is our duty to not only demand greater openness, but also to work towards making these systems safer for those who are already using them.

References

[1] C. Basaran and K. Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *ECRTS '12*.

[2] G. Elliott, B. Ward, and J. Anderson. GPUSync: A framework for real-time GPU management. In *RTSS '13*.

[3] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *RTSS '11*.

[4] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX ATC '11*.

[5] H. Lee and M. Abdullah Al Faruque. Run-time scheduling framework for event-driven applications on a GPU-based embedded system. In *TCAD '16*.

[6] P. Messmer. Unleash legacy MPI codes with kepler's hyper-q. Online at `https://blogs.nvidia.com/blog/2012/08/23/unleash-legacy-mpi-codes-with-keplers-hyper-q/`, 2012.

[7] NVIDIA. Embedded systems solutions from nvidia. Online at `http://www.nvidia.com/object/embedded-systems.html`.

[8] N. Otterness, V. Miller, M. Yang, J. Anderson, F.D. Smith, and S. Wang. GPU sharing for image processing in embedded real-time systems. In *OSPERT '16*.

[9] N. Otterness, M. Yang, S. Rust, E. Park, J. Anderson, F.D. Smith, A. Berg, and S. Wang. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *RTAS '17*.

[10] S. Rennich. Webinar: CUDA C/C++ streams and concurrency. Online at `https://developer.nvidia.com/gpu-computing-webinars`, 2011.

[11] U. Verner, A. Mendelson, and A. Schuster. Scheduling periodic real-time communication in multi-GPU systems. In *ICCCN '14*.

[12] U. Verner, A. Mendelson, and A. Schuster. Scheduling processing of real-time data streams on heterogeneous multi-GPU systems. In *SYSTOR '12*.

[13] Y. Xu, R. Wang, T. Li, M. Song, L. Gao, Z. Luan, and D. Qian. Scheduling tasks with mixed timing constraints in GPU-powered real-time systems. In *ICS '16*.

[14] J. Zhong and B. He. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25:15221532, 2014.

[15] H. Zhou, G. Tong, and C. Liu. GPES: A preemptive execution system for GPGPU computing. In *RTAS '15*.