Scaling Up: The Validation of Empirically Derived Scheduling Rules on NVIDIA GPUs

By
Joshua J. Bakita

Senior Honors Thesis
Department of Computer Science
University of North Carolina at Chapel Hill

May 2018

Approved: April 5th, 2018
F. Donelson Smith, Thesis Advisor
James H. Anderson, Second Reader
Diane Pozefsky, Director of Undergraduate Studies

Abstract— **Embedded systems augmented with graphics processing units (GPUs) are seeing increased use in safety-critical real-time systems such as autonomous vehicles. The current black-box and proprietary nature of these GPUs has made it difficult to determine their behavior in worst-case scenarios, threatening the safety of autonomous systems. In this work, we introduce a new automated validation framework to analyze GPU execution traces and determine if behavioral assumptions inferred from black-box experiments consistently match behavior of real-world devices. We find that the behaviors observed in prior work are consistent on a small scale, but the rules do not stretch to significantly older GPUs and struggle with complex GPU workloads.**

I. Introduction

Recent advancements in artificial intelligence and embedded computing have started to bring the revolution of self-driving vehicles closer to reality, but a multitude of unanswered questions still stand in their path to mass adoption. One key open question, *how good is good enough?* Recent fatalities [15, 19] have shown that the current standard of "good enough" falls short in more than one commercial system. To eliminate a subjective definition of "good enough", this paper envisions autonomous vehicle hardware eventually requiring *certification* for manufacturer, customer, and regulatory acceptance.

However, the hardware increasingly used in vehicles and labs today to meet size, weight, and power (SWaP) requirements utilizes proprietary architectures with vague and lacking public documentation. This presents a tremendous quandary to those attempting certification. This issue is exemplified in systems based on NVIDIA's Parker system-on-a-chip (SoC). This SoC powers NVIDIA's TX2 development board as well as the NVIDIA's DRIVE PX AutoChauffeur and AutoCruise boards marketed towards autonomous vehicles [10]. Known users of this platform include Tesla's Autopilot 2.0 system [8]. Due to the TX2's public availability, recent work [2, 21] has focused on that board as a representative for NVIDIA's other, more tightly controlled, Parker SoC-based boards.

These embedded platforms contain the majority of their raw computing power in their graphical processing units (GPUs), so it becomes essential to thoroughly understand how these devices behave when multiple general purpose GPU (GPGPU) workloads share a single GPU. Danger lies in justifying the use of these components in a self-driving vehicle without reasoning from fundamental behaviors [14]. Making simulation-based statistical assertions about the overall observed lack of failures of a self-driving vehicle system cannot carry over to the real world. Systems must either be statically, provably safe or allowed to drive for millions of representative miles without demonstrating any error [14]. The infeasibility of the latter option leaves us with the former, and returns us to the importance of understanding GPU behavior in these systems.

Our prior work attempted to solve this problem by forming rules of behavior for CUDA (a common GPGPU programming API for NVIDIA GPUs). Unfortunately, safety-critical systems could not yet rely on our rules. The rules were formulated from empirical observation, and rigorous applicability of the rules remained untested. Our rules also suffered from fragility and limited scope. As new GPU architectures appear almost every other year and new processors based on those architectures appear every few months, the possibility of rigorously testing all these devices by hand becomes vanishingly small. A field dominated by rapid and regular change needs some automated method to validate past results on new or more complicated devices.

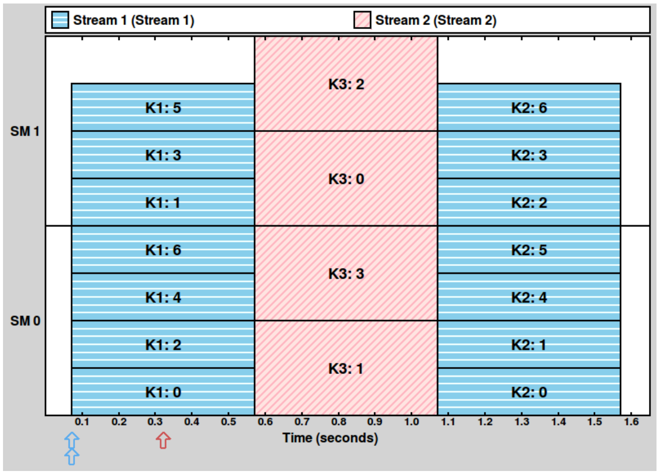


Fig. 1. EE queue benchmark trace on NIVIDA TX2

To emphasize this point, consider Fig. 1, which displays a GPU execution trace used in recent work [12]. Shaded rectangles represent GPU executions over time. The trace appears well-ordered and reasonably easy to step through by hand, given familiarity with prior work. Now take Fig. 2. This presents a trace from the same benchmark, but on a GPU with more compute capacity. A trained eye will pick out the subtly different rules in effect here, but even this simple modification tests the limits of empirical observation. Real-world executions can be far more complex. Fig. 3 provides an extreme example. It displays a trace from the execution of a randomly generated four-thread workload on a mainstream GPU. So many different interactions take place that even our graphing software struggles to cope.

No human can hand-validate what behavioral rules apply in traces like this. But comprehensive testing of our proposed rules requires the validation of these sorts of traces. To accomplish that, this paper introduces an automated rule-validation framework which provides a path to scalable, rigorous validation.

II. Background

Our solution builds on elements of NVIDIA GPUs, concepts from CUDA, and properties of a number of representative test platforms.

A. GPU and CUDA Fundamentals

A GPU represents a highly parallel co-processor. Traditionally used for fast 3D scene rasterisation, recent years have seen them used increasingly for GPGPU tasks. For the

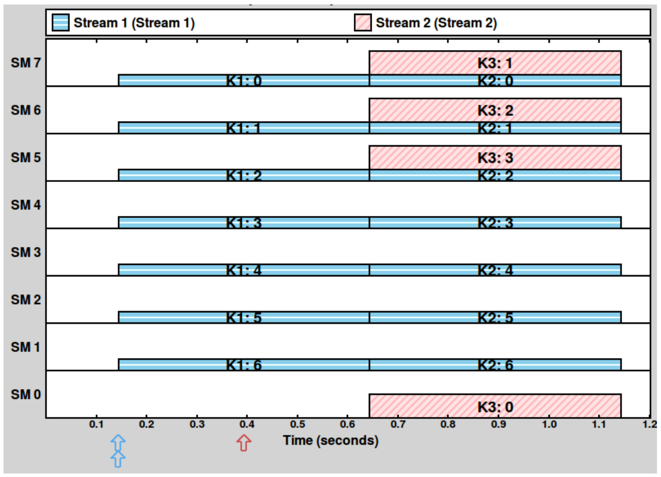


Fig. 2. EE queue benchmark trace on NIVIDA K5000

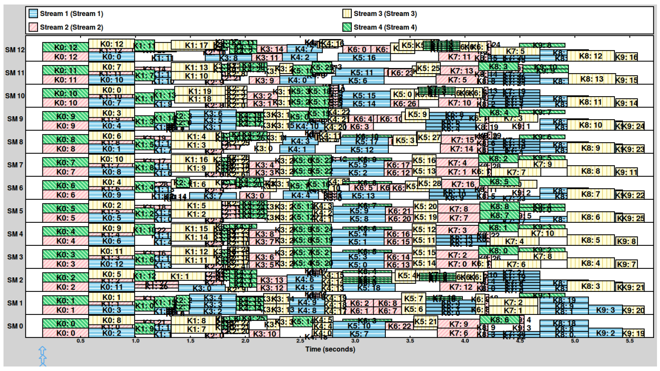


Fig. 3. Benchmark trace of random work on NVIDIA GTX 970

NVIDIA GPUs considered in this paper, CUDA provides the best API available to execute non-graphical computations on the GPU. Built as a C/C++ extension, valid CUDA code looks very similar to embedded C code. See Algorithm 1 for a loose example of vector addition in a CUDA program.

Some key GPU and CUDA terms used throughout this paper:

1) *CUDA Thread Block:* A group of GPU threads executing the same set of user-defined instructions in lockstep. This is the lowest-level GPU scheduling unit considered in this paper.
2) *CUDA Kernel:* A combination of instruction code and CUDA thread block specifications. Dispatched asynchronously by a user-space process.
3) *CUDA Stream:* A first-in-first-out (FIFO) work queue into which processes on the CPU can dispatch kernels.
4) *SM (Streaming Multiprocessor):* A subdivision of an NVIDIA GPU. Single thread blocks cannot be split across multiple SMs [11, p. 7].
5) *EE (Execution Engine) Queue:* A special internal queue of kernels that our past work has defined to exist between CUDA stream queues and the actual GPU. Fig. 6 illustrates its location in the execution flow.

Algorithm 1 Vector Addition Pseudocode from [21, p. 7].

```
kernel VECADD(A ptr to int, B: ptr to int, C: ptr to int)
    ▷ Calculate index based on built-in thread and block information
    i := blockDim.x * blockIdx.x + threadIdx.x
    C[i] := A[i] + B[i]
end kernel

procedure MAIN
    ▷ (i) Allocate GPU memory for arrays A, B, and C
    cudaMalloc(d_A)
    ...
    ▷ (ii) Copy data from CPU to GPU memory for arrays A and B
    cudaMemcpy(d_A, h_A)
    ...
    ▷ (iii) Launch the kernel
    vecAdd<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C)
    ▷ (iv) Copy results from GPU to CPU array C
    cudaMemcpy(h_C, d_C)
    ▷ (v) Free GPU memory for arrays A, B, and C
    cudaFree(d_A)
    ...
```

B. Test Platforms

To demonstrate the scalability and broad applicability of the framework presented in this paper, we test four generations of NVIDIA's graphics processors. The following list notes the specifications and release dates of the representatives from each generation:

Kepler	The Quadro K5000 discrete GPU (Nov. 2012) with 8 SMs.
Maxwell	The GeForce GTX 970 discrete GPU (Sep. 2014) with 13 SMs.
Pascal	The GeForce GTX 1070 discrete GPU (June 2016) with 15 SMs and the Jetson TX2 (March 2017) with 2 SMs.
Volta	The Titan V discrete GPU (Dec. 2017) with 80 SMs.

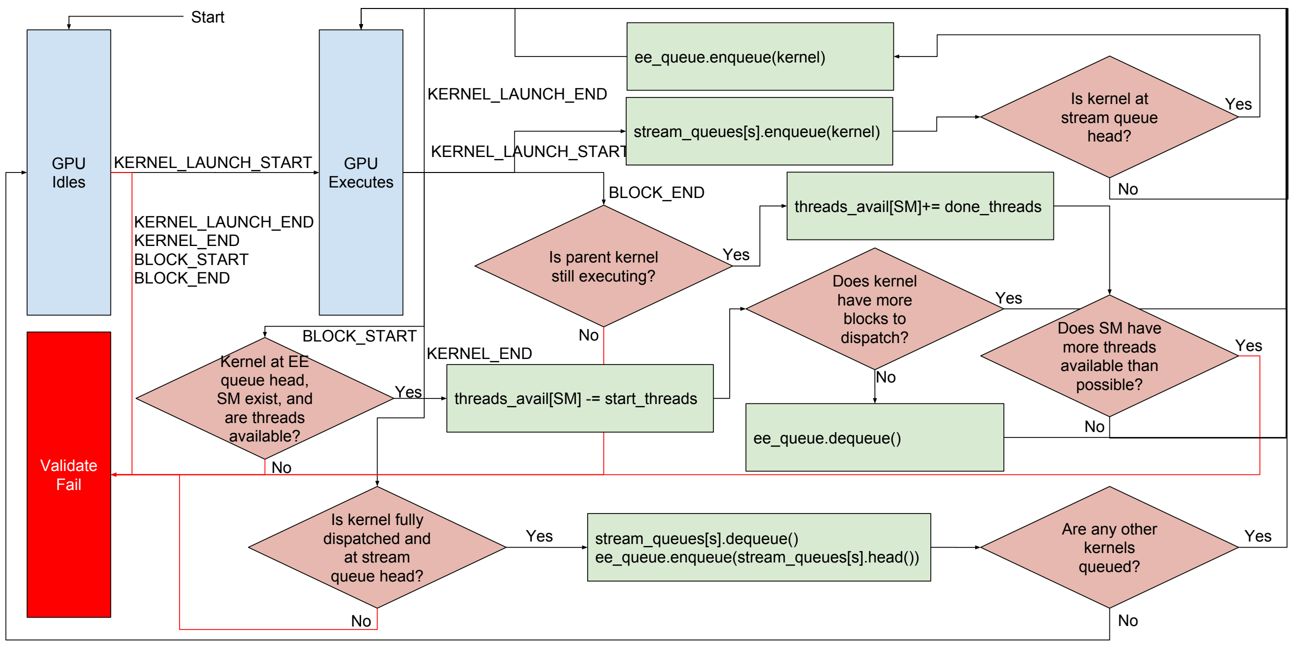
C. Related Work

Due to a lack of public documentation on how concurrent execution of GPU workloads behave, much past work in this area focuses on efficiently providing an exclusive locking mechanism for the GPU [5, 6, 7, 16, 17, 18, 20]. This effectively prevents any interference after lock acquisition (as processes only ever own an independent portion of the GPU) but this approach does not fully address worst-case execution times (WCETs). Tasks can still interfere with themselves.

Moving in a different direction, some work has also been done on increasing utilization at the cost of predictability. For example, Zhong et. al.'s scheduling approach showed increased utilization, but adds overhead and also does not account for potentially destructive interference between multiple concurrent GPU operations. [22]

Other work in [22, 4, 6, 9] attempts to expose more degrees of scheduling freedom in CUDA by using software to approximate preemptive hardware. Unfortunately, this research also does not concern itself with WCETs.

Our recent work addresses an orthogonal problem. It attempts to address WCETs in any GPU context by better understanding and leveraging existing undocumented GPU



Start
GPU Idles
KERNEL_LAUNCH_START
GPU Executes
KERNEL_LAUNCH_END
KERNEL_END
BLOCK_START
BLOCK_END
Validate Fail
ee_queue.enqueue(kernel)
stream_queues[s].enqueue(kernel)
Is kernel at stream queue head?
Yes
No
KERNEL_LAUNCH_END
KERNEL_LAUNCH_START
BLOCK_END
threads_avail[SM]+= done_threads
Is parent kernel still executing?
Yes
No
Does kernel have more blocks to dispatch?
Yes
No
Does SM have more threads available than possible?
Yes
No
BLOCK_START
Kernel at EE queue head, SM exist, and are threads available?
Yes
No
KERNEL_END
threads_avail[SM] -= start_threads
ee_queue.dequeue()
Is kernel fully dispatched and at stream queue head?
Yes
No
stream_queues[s].dequeue()
ee_queue.enqueue(stream_queues[s].head())
Are any other kernels queued?
Yes
No

Fig. 4. State machine for rules G1-G4, X1, and R2 (all-caps annotations denote what events trigger each transition)

hardware behaviors to enable real-time systems. Our group has postulated scheduling rules for the Jetson TX1 [12, 13] and Jetson TX2 [2] development boards while also discovering some more generally applicable behaviors and pitfalls in CUDA [21]. This paper expands on that work.

III. Experimental Approach

Beyond rules, our past work provides a GPU benchmarking framework that allows for sets of CUDA kernels to be run in a reproducible manner. This framework provides logs of the executions with events such as a kernel dispatch or thread block end marked with high-precision timestamps. Our prior work graphed these traces for visual empirical analysis, but the traces also provide all the information necessary to enable an automated validation framework.

A. Validator Design

We use a state machine to validate if these logged behaviors adhere to what we expect from our rules. We prefer the approach because it most easily lends itself to the problem. Some past work (namely GPGPU-Sim [3]) has applied a custom GPU simulator to confirm behaviors, but we find the inherent complexity of a full simulation too burdensome. Even the most recent simulators fall generations behind today's GPUs [1]. Our approach instead takes a subset of the events recorded from actual executions and uses each event to trigger state transitions.

B. Sourcing Traces

To obtain the event series which will trigger these transitions, we parse a selection of the information logged by the benchmarking framework and sort events by timestamp. The current version of the validation tool focuses on the following events:

- Kernel launch start
- Kernel launch end
- Kernel end[1]
- Thread block start
- Thread block end

During parsing, we extract and attach contextual data about each event. For kernel events, that includes a list of child thread blocks and the ID of the CUDA stream submitted to. For thread block events, context includes the number of threads in the block, the parent kernel, and the SM used.

C. Building the State Machine

After preparing the series of events, the actual state machine can proceed. Building off the scheduling rules postulated in our recent work [2], the constructed state machine appears as a flow chart in Fig. 4. This machine only validates a core, always-applicable subset of the recently published rules (6 of 16 rules). It covers only the "general scheduling rules" and a couple of closely connected ones. These chosen rules and labels have been reproduced from [2] in the following list:

G1 "A copy operation or kernel is enqueued on the [CUDA] stream queue for its stream when the associated CUDA API function (memory transfer or kernel launch) is invoked."

G2 "A kernel is enqueued on the EE queue when it reaches the head of its [CUDA] stream queue."

G3 "A kernel at the head of the EE queue is dequeued from that queue once it becomes fully dispatched."

[1]Pseudo-event; sometimes it is undesirable for a benchmark to perform a `cudaStreamSyncronize` to retrieve the actual kernel execution end time. In those cases, the parser uses the end time of the last thread block in the kernel as a substitute.

G4 "A kernel is dequeued from its [CUDA] stream queue once all of its blocks complete execution."

X1 "Only blocks of the kernel at the head of the EE queue are eligible to be assigned."

R2 "A block of the kernel at the head of the EE queue is eligible to be assigned only if there are sufficient thread resources available on some SM."

Validation proceeds by processing execution events in chronological order. At each event, state updates and a validity check can occur on the path between states. Fig. 4 represents events in all-caps, states as vertical rectangles, updates with horizontal rectangles, and checks as diamonds. Validation succeeds or fails dependent on entry into the red, terminal failure state. Our Python implementation of this state machine and the event stream parser is open-source software available online[2].

IV. Evaluation

We evaluated our framework by assuring that it properly categorized known good traces and known bad traces.

A. Validating Base Rules

We demonstrated that known correct traces do not falsely enter the validation failure state by applying the convenient fact that our selected scheduling rules mirror those first discussed in one of our papers from last year [12]. In that paper, we provided clear benchmark configurations to demonstrate each scheduling rule in action. For this paper, we ran those configurations again and confirmed that our automated validation of each of their traces succeeded. That verified that no single proper rule behavior would be flagged as incorrect by our framework.

We demonstrated that rule violations produce validation failures in practice by testing specifically crafted invalid traces. Some example modifications to previously correct traces were swapping the order that two thread blocks execute in, adding just one too many threads to a block, or creating timing abnormalities. By testing all of the possible ways that each individual rule could fail, we assured that all real failures or combinations of multiple failures will be caught by the framework. (To examine the exact violations added, the `tests/bad` directory in our online code repository contains all of these tests.) Importantly, these invalid traces all originate from static modifications to previously generated valid traces - we never expect the GPU to directly generate an invalid trace.

B. Results on Maxwell, Pascal, and Volta

After using this approach to confirm that our validator behaves as expected for the specific platform analyzed by hand in our past work (the TX2), we expanded tests to cover all of the platforms detailed in Sec. II-B. We found our scheduling rules to be broadly applicable to GPUs running NIVIDA's Maxwell, Pascal, and Volta architectures. This encompasses all major NVIDIA GPUs released since late 2014. However, we ran into unexpected results when attempting to validate large, randomly generated workloads such as the one demonstrated in Fig. 3.

[2]See `https://github.com/JoshuaJB/cuda_scheduling_validator_mirror`

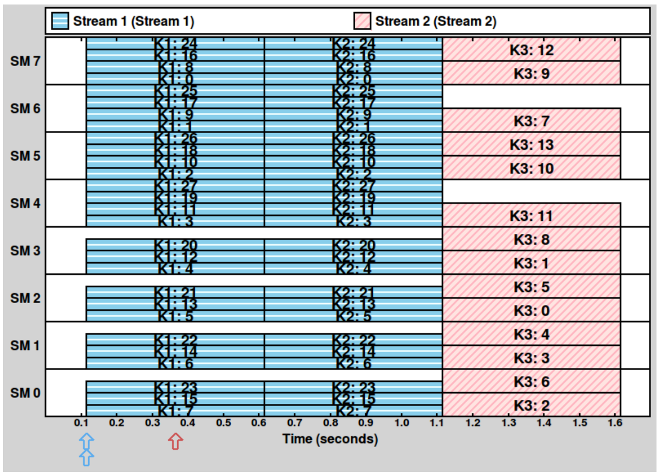


Fig. 5. EE queue benchmark trace on NVIDIA K5000 with thread block count adjusted to saturate the GPU

For example, on our GTX 970, only about 13% of 2,000 randomly generated 40-kernel tests passed validation. Upon further inspection, we found that the framework was correct; there appeared to be subtle violations of rule X1 (that only the head of the EE queue should be eligible for dispatching) recorded in the benchmark logs. However, the extent of this incorrect ordering never appeared to be more than a few microseconds. We currently hypothesize that these "violations" are merely figments of our current, accuracy-limited time-stamping approach.

Specifically, CUDA does not provide thread-block-level start or stop timestamps, so our benchmarking framework must instead obtain these by reading a global GPU time register immediately on start and before end inside each thread block. We believe that momentary stalls or propagation delays may cause these reads to sometimes not perfectly correspond to actual block start and end times. Preliminary investigation into the traces that failed validation have found support for this hypothesis, but we hope to further analyze and clarify this behavior in future work.

C. Results on Kepler

While we found the rules seem to apply to the three-most-recent architectures considered, the older Kepler architecture behaved rather differently. The framework revealed that a rule violation occurred during validation of the trace from the benchmark designed to demonstrate rule G2 in action.

During the subsequent empirical investigation, it became clear that the validator correctly detected a rule violation. Kepler architecture GPUs do not follow the same rules as their successors. This peculiarity brings us back to Fig. 1 and Fig. 2.

Each graph plots GPU time on the horizontal axis for each SM plotted on the vertical axis. Every rectangle in the plot area represents a thread block running over some time

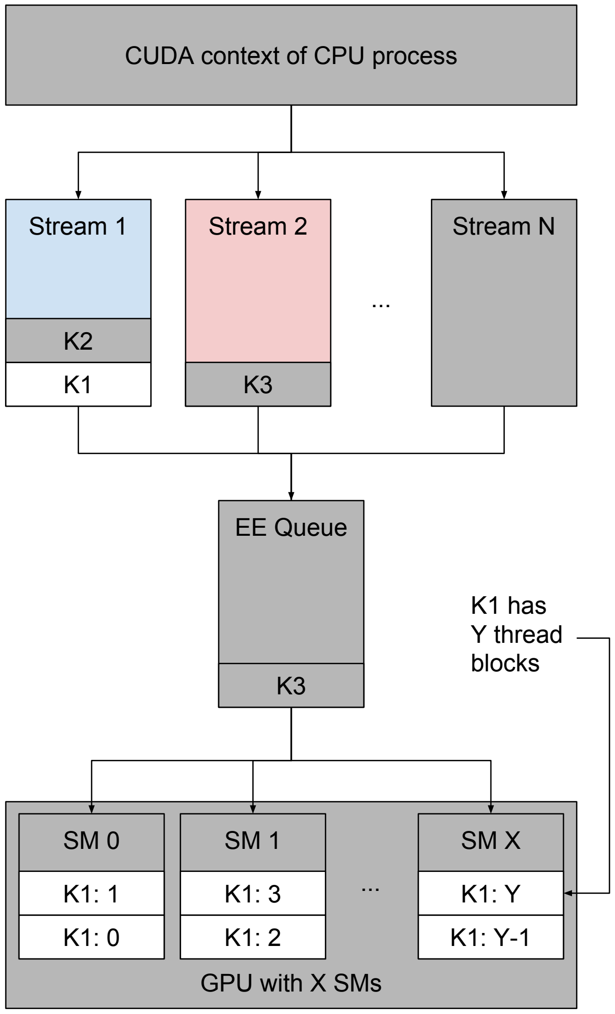


Fig. 6. Example of expected queue states for Fig. 1, Fig. 2, and Fig. 5 at time point 0.4s

period. The digit immediately prior to the colon in the label on each block indicates the block's kernel affiliation, and the number immediately after indicates the unique identification number of this thread block among all the kernel's thread blocks. Different colors indicate different CUDA streams, and the colored arrows along the horizontal axis indicate when kernels are released from our CPU process.

Now consider the simple plot presented in Figure Fig. 1 generated from a benchmark trace on the Pascal-based TX2. To walk through the series of events represented by this plot, kernels K1 and K2 release into stream 1 before 0.1s. At this point, K1 and K2 are in the stream 1 queue and K2 is in the EE queue. K1 quickly dispatches all of its blocks, nearly occupies all of the GPU's resources, and leaves the EE queue. K3 then releases shortly after the 0.3s mark and immediately moves to the head of the EE queue. It goes on the EE queue before K2 because rule G4 has kept K2 blocked behind K1 in the stream 1 queue. (Fig. 6 illustrates this point in time.) Once K1 completes execution around 0.55s, it leaves stream 1 and allows K2 to enqueue on the EE queue behind K3. K3 then fully dispatches, saturates the GPU, and leaves the EE queue. At K3's completion point around 1.05s, K2 then has space for at least one of its thread blocks, begins execution, and runs to completion. In this plot, nothing unexpected occurs. All the rules hold and operate correctly.

However, consider Fig. 2. Generated from the same benchmark configuration as Fig. 1, but on the Quadro K5000 Kepler-based GPU. A similar pattern emerges up to just before timestamp 0.4s. As in the prior example, K3 is expected to be on the EE queue and K2 is expected to be blocked in the stream 1 queue. (See Fig. 6.) That is not what we observe. If K3 were immediately moved to the head of the EE queue as we expect, it should near-immediately start on release due to more than enough capacity available for at least one thread block. What is occurring becomes clearer when examining the execution trace timestamps by hand. At a nanosecond level, K2 starts before K3. This indicates that both rules G2 and G4 do not properly apply on Kepler GPUs.

The violation becomes much clearer with scrutiny of Fig. 5. The trace plotted here comes from the same benchmark configuration used in Fig. 1 and Fig. 2, but with the number of thread blocks scaled up to compensate for the increased capacity of the K5000. The flipped order of execution of K2 and K3 becomes obvious in comparison to Fig. 1.

No head-of-line blocking on stream queues forms a potential explanation for this behavior. In that scenario, the GPU would immediately move kernels to the EE queue as they are received. More specifically, *rule G4 would no longer apply* and rule G2 would change to the following:

G2 (Kepler) A kernel is *dequeued from its stream queue* and enqueued on the EE queue when it reaches the head of its stream queue.

In essence, all the stream queues become aliases to only one single hardware queue. One can verify that these rules for Kepler work in at least some cases by stepping through Fig. 2 and Fig. 5. Each figure support our hypothesis by behaving as expected under the proposed rule variation.

V. Conclusion

A solid understanding of hardware scheduling behavior forms the essential foundation for any safety-critical system. To meet SWaP requirements, GPUs have emerged as one of the premier compute accelerators used in these platforms. Unfortunately, sufficient low-level documentation for these accelerators has not been forthcoming. Past solutions to this uncertainty have precluded parallelism via locking or introduced overheads without addressing questions about interference. Our recent work to cast light on GPU behavior rules has heavily depended on empirical observation. That approach quickly proves impractical and insufficient on large GPUs or in complicated test programs.

Our solution addresses that problem via an automated validation framework. By minimizing human input, our framework enables rigorous validation of scheduling rules across a multitude of complex platforms and workloads without the limitations of human error and inefficiency.

Future work could expand the state machine used for validation in this work to include the rules for priority streams, the NULL stream, copy operations, shared-memory blocking, and other yet-to-be codified rules. In the more distant future, one hopes that this framework could be modified to support traces from NVIDIA's native `nvprof` profiler, and thus be used to validate rule authority on execution traces from any CUDA program rather than just logs from the benchmarking framework.

Separately, we hope to further explore the irregularities causing complex tasks to fail validation. While we are reasonably confident that these unexpected results are being triggered by inaccurate timing information, we would like to more rigorously confirm that there are no fundamental flaws in our rules.

However, the framework developed in this paper should enable work to proceed faster than before. We need a comprehensive understanding of hardware to build safe autonomy, and this framework helps accelerate the assembly of that core foundation.

References

[1] Tor Aamodt. GPGPU-Sim. Online at `http://www.gpgpu-sim.org`.

[2] T. Amert, N. Otterness, M. Yang, J. Anderson, and F. D. Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *RTSS 2017*.

[3] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed GPU simulator. In *ISPASS 2009*.

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

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

[6] 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*.

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

[8] Fred Lambert. Look inside Tesla's onboard Nvidia supercomputer for self-driving. *Electrek*, 2017.

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

[10] NVIDIA. Autonomous car development platform from nvidia. Online at `https://www.nvidia.com/en-us/self-driving-cars/drive-px/`.

[11] NVIDIA. Fermi architecture whitepaper. Online at `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[12] N. Otterness, M. Yang, T. Amert, J. Anderson, and F.D. Smith. Inferring the scheduling policies of an embedded CUDA GPU. In *OSPERT '17*.

[13] 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*.

[14] S. Shalev-Shwartz, S. Shammah, and A. Shashua. On a Formal Model of Safe and Scalable Self-driving Cars. *ArXiv e-prints*, August 2017.

[15] Tesla. An update on last week's accident. Online at `https://www.tesla.com/blog/update-last-week%E2%80%99s-accident`, 2018.

[16] U. Verner, A. Mendelson, and A. Schuster. Batch method for efficient resource sharing in real-time multi-GPU systems. In *ICDCN '14*.

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

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

[19] Daisuke Wakabayashi. Self-driving uber car kills pedestrian in Arizona, where robots roam. *New York Times*, 2018.

[20] 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*.

[21] M. Yang, N. Otterness, T. Amert, J. Bakita, J. Anderson, and F.D. Smith. Avoiding pitfalls when using nvidia gpus for real-time tasks in autonomous systems. In *ECRTS '18 (to appear)*.

[22] 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.