Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems*

Ming Yang
The University of North Carolina at Chapel Hill, USA
yang@cs.unc.edu

Nathan Otterness
The University of North Carolina at Chapel Hill, USA
otternes@cs.unc.edu

Tanya Amert
The University of North Carolina at Chapel Hill, USA
tamert@cs.unc.edu

Joshua Bakita
The University of North Carolina at Chapel Hill, USA
jbakita@cs.unc.edu

James H. Anderson
The University of North Carolina at Chapel Hill, USA
anderson@cs.unc.edu

F. Donelson Smith
The University of North Carolina at Chapel Hill, USA
smithfd@cs.unc.edu

Abstract

NVIDIA's CUDA API has enabled GPUs to be used as computing accelerators across a wide range of applications. This has resulted in performance gains in many application domains, but the underlying GPU hardware and software are subject to many non-obvious pitfalls. This is particularly problematic for safety-critical systems, where worst-case behaviors must be taken into account. While such behaviors were not a key concern for earlier CUDA users, the usage of GPUs in autonomous vehicles has taken CUDA programs out of the sole domain of computer-vision and machine-learning experts and into safety-critical processing pipelines. Certification is necessary in this new domain, which is problematic because GPU software may have been developed without any regard for worst-case behaviors. Pitfalls when using CUDA in real-time autonomous systems can result from the lack of specifics in official documentation, and developers of GPU software not being aware of the implications of their design choices with regards to real-time requirements. This paper focuses on the particular challenges facing the real-time community when utilizing CUDA-enabled GPUs for autonomous applications, and best practices for applying real-time safety-critical principles.

2012 ACM Subject Classification Computer systems organization → Heterogeneous (hybrid) systems, Computer systems organization → Embedded software, Computer systems organization → Real-time systems, Computer systems organization → Embedded and cyber-physical systems, Software and its engineering → Scheduling, Software and its engineering → Concurrency control, Software and its engineering → Process synchronization

* Work supported by NSF grants CNS 1409175, CPS 1446631, CNS 1563845, and CNS 1717589, ARO grant W911NF-17-1-0294, and funding from General Motors.

© Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H. Anderson, and F. Donelson Smith;
licensed under Creative Commons License CC-BY
30th Euromicro Conference on Real-Time Systems (ECRTS 2018).
Editor: Sebastian Altmeyer; Article No. 20; pp. 20:1–20:21

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Keywords and phrases real-time systems, graphics processing units, scheduling algorithms, parallel computing, embedded software

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.20

1 Introduction

A fundamental shift is reshaping how real-time analysis is applied in all forms of autonomous systems (*e.g.*, UAVs, robotics, and, especially, self-driving automobiles). These systems are increasingly dependent on escalating computational requirements for various applications based on machine learning (ML). Examples include computer-vision applications that recognize people and objects in high-bit-rate streams from multiple video cameras, and applications that process 3-D models of the surrounding environment from high-volume streams of LIDAR data. These and other ML applications in autonomous vehicles have prompted the adoption of specialized computing accelerators to match computational demands. Graphics processing units (GPUs) are among the most prominent and accessible of these specialized accelerators because of their high-throughput performance. While high throughput is necessary for ML applications based on multiple streams of sensor inputs, it alone is not sufficient. *Safe operation of autonomous vehicles also requires temporal correctness from GPU-using tasks*—this is where real-time analysis becomes essential for autonomous systems.

Why there is a problem. Unfortunately, GPUs present many challenges, so modeling, analyzing, and certifying a safety-critical autonomous system using GPUs is currently beyond the state-of-the-art. One reason is that GPUs are fundamentally different from CPUs. Real-time analysis is based on well-understood scheduling algorithms that allocate CPU capacity. In contrast, GPU hardware and software together implement GPU-specific scheduling algorithms that are proprietary, opaque, and can change without notice. Modeling and analysis efforts under these conditions are subject to many pitfalls when applied to real-time safety-critical workloads in GPU-using autonomous systems.

Focus of this paper. Our motivation for this work is to provide guidance, recommendations, and warnings about numerous pitfalls to both research and implementation practitioners. We have found that writing programs for real-time tasks that combine CPU and GPU computations is harder than we first thought. Based on several years of study, experimentation, and experience with GPU programming, we are presenting here a compendium of specific issues that are essential background for developing task systems where real-time design meets GPUs.

Choice of GPU platforms. We base our findings on our experiences with NVIDIA GPUs for a number of reasons. The most salient reason is that *NVIDIA GPUs are in cars on the road today.* Further, NVIDIA has positioned itself as a market leader in automotive applications. For example, NVIDIA's "Jetson" line of embedded platforms specifically targets autonomous systems, and is marketed as "the embedded platform for autonomous everything" [21]. Three generations of the Jetson series of embedded single-board computers have been produced by NVIDIA; the TK1, TX1, and TX2. NVIDIA also markets a higher-performance line of embedded platforms, the "Drive PX" series, which includes multiple models such as the Drive PX2, Drive PX Xavier, and Drive PX Pegasus.

NVIDIA GPUs serve as an exemplar of the push for throughput over predictability in GPUs . Recent developments in the NVIDIA GPU ecosystem are focused on improving ML applications, especially those for autonomous driving. Most of these improvements center around increasing throughput or reducing execution latency, but little, if any, attention

has been paid to requirements of the real-time tasks used in autonomous systems. This lack of attention is evident in the sparse efforts by NVIDIA to improve or document GPU scheduling behavior or improve the predictability of GPU execution times.

1.1 Contributions

The major contribution of this paper lies in discussing pitfalls for real-time GPU usage of relevance to both those conducting research on autonomous systems and those who design and build them. These pitfalls fall within three categories:

Synchronization and blocking. In any task consisting of a combination of CPU and GPU computations, there are necessary synchronization points (*e.g.*, a CPU program needs to wait until a GPU has produced a result). Synchronization inherently leads to blocking terms in scheduling analysis. Unfortunately, we have learned that why and when synchronization blocking occurs in a GPU-using task is not straightforward to determine. Further, some forms of synchronization can lead to significant capacity loss on both CPUs and GPUs. We have constructed experiments that expose these synchronization effects and carefully describe them along with a list of specific pitfalls the unwary programmer may encounter. This contribution is fully presented in Sec. 3.

GPU concurrency. We have realized that there is a fundamental trade-off that exists for designing real-time tasks that use a GPU. A conventional choice is to write and execute the task program as an operating system (OS) process in its own non-shared address space. This provides cross-task memory isolation. If this choice is used, however, the NVIDIA GPU programming environment (described in Sec. 2) does not permit any concurrent computations on the GPU even if sufficient GPU resources are available. Depending on how GPU programs are organized and written, this can lead to capacity loss on the GPU. The alternate choice is to write and execute a task as a schedulable thread that shares a process address space with other task threads. Cross-task memory isolation is lost, but the GPU programming environment provides mechanisms that allow concurrent computations on the GPU. NVIDIA provides a third option with a middleware environment that is claimed to provide the best of both choices—memory isolation with concurrency enabled. We have performed a case study using algorithms that are exemplars for computer-vision tasks in autonomous vehicles to evaluate these trade-off options. The results and guidelines are fully presented in Sec. 4.

CUDA programming perils. Our research has necessarily involved constructing many thousands of lines of GPU programming for performing experiments. This experience has been especially enlightening about the perils one can encounter in programming for NVIDIA GPUs. The perils span a spectrum of pain ranging from simple documentation errors to functions that default in strange ways, to programming "gotchas." We present a list of perils with descriptions and examples of the ones most likely to cause problems in Sec. 5.

Value for autonomous systems. We believe that this paper will help bridge the gap between research and implementation in autonomous systems. For example, real-time researchers may not be familiar with GPU programming for applications of ML and other forms of AI used in real-time tasks. Likewise, programmers responsible for implementations are given little guidance about creating GPU-using task systems amenable to real-time analysis. We provide the necessary understanding required to apply GPUs in real-time tasks while avoiding numerous hidden pitfalls. We also expose GPU-related issues that must be mitigated for real-time guarantees to be possible in autonomous systems. We further believe that the fundamental issues presented herein are relevant to any real-time application using computational accelerators, and likely hold for other manufacturers' GPUs, digital signal

processors (DSPs), or FPGAs.

1.2 Related Work

Treating GPUs as non-shared devices has been a consistent theme in much of the prior research on GPU scheduling for real-time systems. More predictable execution times result from restricting access to the entire GPU (or its independent execution and data movement components) to a single task at a time [11, 15, 16, 27, 28, 29, 31].

Other prior research takes a slightly different approach and improves schedulability by simulating preemptive execution [3, 15, 17, 33]. These designs typically split GPU computations into smaller fragments, which can be individually scheduled and preempted. One of these frameworks, called Kernelet [32], even allows GPU sharing as a means to improve utilization, but interference effects caused by sharing are not addressed.

The decision to treat GPUs as non-shared devices is largely motivated by a perceived need to work with a greatly simplified *model* of GPU execution (resulting, we believe, primarily from a lack of information from GPU manufacturers). If GPU scheduling behavior is an opaque "black box," it is a rational conclusion that sharing must be avoided because execution ordering and interference effects *cannot be known*. Our research is motivated, however, by an observation that GPU sharing will become essential for effectively utilizing less-capable embedded GPUs. Our research goal is to enable the modeling and analysis of a combined CPU+GPU scheduling framework that allows real-time tasks to share multicore CPUs and one or more GPUs.

We began our research by experimentally investigating the impacts of GPU sharing on the NVIDIA Jetson TK1 [24] and TX1 [25]. In these studies, we focused on GPU sharing by CPU processes (tasks) that have separate address spaces. We found that sharing in this context happens only through round-robin time-sliced multiplexing of GPU computations onto the GPU execution hardware. This multiplexing form of scheduling presents many challenges for modeling and analysis. In later work, we experimentally investigated GPU sharing by CPU tasks that share an address space (threads) on both the TX1 [26] and the more-capable TX2 [1]. In these studies, we found that truly concurrent sharing can indeed occur and deduced rules the GPU uses to schedule execution.

The work summarized so far was all directed at scheduling real-time tasks that use a GPU for parts of their executions. Other work has focused on timing analysis for GPU workloads [4, 5, 6, 7, 8], techniques for remedying performance bottlenecks [13], direct I/O communication [2], and techniques for managing or evaluating GPU hardware resources, including the cache and DRAM [9, 10, 12, 14, 18, 19, 30].

2 Background

In this section, we provide background information on the NVIDIA GPUs used in this research. The CUDA programming framework is described, and a simple example of a CUDA program is explained.

2.1 CUDA-Enabled Devices

The work presented here refers to the Kepler, Maxwell, Pascal, and Volta architectures of NVIDIA GPUs. NVIDIA introduced these four different generations of GPU architectures, in that order, within a time span of about five years (2012 - 2017)—a pace of change more

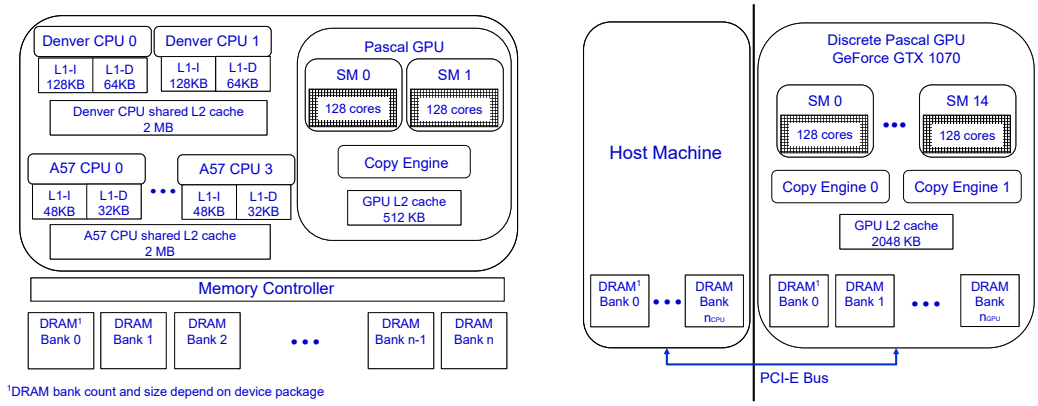

Figure 1 Jetson TX2 Architecture (left) and GeForce GTX 1070 Architecture (right)

rapid than normally seen in CPU generations. GPUs are programmed using the CUDA API, which is an NVIDIA-provided set of libraries and language extensions for C/C++.

We consider both *discrete* GPUs and *integrated* GPUs. An integrated GPU, such as the NVIDIA Jetson TX2 shown on the left in Fig. 1, is part of a System-On-Chip (SoC) implementation combined with conventional multicore CPUs. The SoC is packaged along with DRAM and external connectors as a small (approximately 7 inches square) single-board computer. The integrated GPU shares hardware resources, such as DRAM, with CPU cores. The TX2 runs the Linux operating system, with additional support from closed-source binary drivers provided by NVIDIA. The TX2's low size, weight, and power (SWaP) requirements and low price tag make it a good exemplar of GPU-enabled platforms intended for embedding in autonomous systems.

Fig. 1 (left) shows the high-level architecture of the TX2. The TX2 contains a six-core heterogeneous ARMv8 CPU, 8 GB of DRAM, and an integrated Pascal GPU. The TX2's GPU consists of two *streaming multiproccessors* (*SMs*), each comprised of 128 GPU cores. The SMs together can be logically viewed as an *execution engine* (*EE*). Additionally, there is a hardware *copy engine* (*CE*) that can copy data between memory regions allocated for CPU use and those allocated for GPU use. The integrated GPU has fewer GPU cores than found in typical high-end GPUs used for graphics, gaming, and high-performance computing applications. We are interested in exploiting any potential for sharing the TX2's GPU by multiple tasks so that its computing capacity is not unnecessarily wasted.

Shown on the right in Fig. 1 is the architecture of the GTX 1070, an example of a discrete GPU. Discrete GPUs consist only of the SMs and local device memory, typically packaged on an adapter card for mounting in a PCIe expansion slot on a computer motherboard. Like all discrete GPUs, the GTX 1070 does not share memory with the host CPU, instead using the PCIe bus to copy data to and from host memory. This GPU features many more SMs than the TX2, increasing the potential benefit attainable if shared among multiple tasks. It also has two CEs, and a larger cache.

Algorithm 1 Vector Addition Pseudocode.

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

5: procedure main
   ▷ (i) Allocate GPU memory for arrays A, B, and C
6:     cudaMalloc(d_A)
7:     ...
   ▷ (ii) Copy data from CPU to GPU memory for arrays A and B
8:     cudaMemcpy(d_A, h_A)
9:     ...
   ▷ (iii) Launch the kernel
10:    vecAdd<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C)
   ▷ (iv) Copy results from GPU to CPU array C
11:    cudaMemcpy(h_C, d_C)
   ▷ (v) Free GPU memory for arrays A, B, and C
12:    cudaFree(d_A)
13:    ...
```

2.2 Relevant CUDA Programming Fundamentals

A *CUDA program* runs as a task (process or thread) on a CPU and relies on a GPU for some part of its computational requirements.[1] The general structure of a CUDA program when it needs to interact with the GPU is as follows: **(i)** allocate memory for GPU use; **(ii)** copy input data from CPU memory to GPU memory; **(iii)** launch execution of a GPU program called a *kernel*[2] to process the data; **(iv)** copy the results from the GPU memory back to the CPU memory; **(v)** free unneeded memory.

CUDA kernels are written from the perspective of a single *GPU thread*. As an example, consider the CUDA program expressed in pseudocode in Algorithm 1. It uses the kernel vecAdd to add a single pair of elements per GPU thread, storing the sum in a corresponding location in an output array. Line 2 demonstrates the use of special global system-defined variables to determine the array element on which to operate. When the kernel executes, threads will run in lock-step with each thread performing the same operation simultaneously on different data. To avoid confusion with GPU threads, we will henceforth refer to CPU threads as *CPU tasks* (or just *tasks*).

A kernel is run on the GPU as a set of thread blocks that can be executed in any order. These thread blocks, or simply *blocks*, are each comprised of a number of threads. As seen in Line 10 of Algorithm 1, the number of blocks and threads per block are programmer-specified and can be set at runtime when a kernel is launched. The GPU scheduler uses these values to assign work to the SMs. *Blocks are the schedulable entities on the GPU.* All threads in a block are always executed on the same SM, and run non-preemptively until completion. A kernel completes when all threads in all blocks have exited.

We refer to kernels and memory-copy operations collectively as *GPU operations*. GPU operations are submitted to a GPU in *CUDA streams*. Operations within a stream are executed in FIFO order. By default, the *NULL stream* is used, but users can submit operations to multiple user-defined streams.[3] Kernels from different streams can run concurrently by

[1] Note that both CPU and GPU computations are specified in the same CUDA program.
[2] Unfortunate terminology, not to be confused with an OS kernel.
[3] CUDA documentation only guarantees that operations within a stream are executed in FIFO order,

sharing the GPU's cores if sufficient internal resources are available. Copy operations are handled by the GPU's CE and can be concurrent with kernel executions on the EE.

CUDA API calls can be synchronous or asynchronous; for many calls, a variant of both is available. For example, `cudaMemcpy` and `cudaMemcpyAsync` both copy data between regions of CPU memory and GPU memory, or between two regions of GPU memory, but `cudaMemcpyAsync` can return control to the calling CPU task before the copy is completed, whereas `cudaMemcpy` blocks the CPU task until the memory copy completes.

Kernel launches are always supposed to be asynchronous. The CUDA documentation[4] [23], however, uses a narrow definition of "asynchronous" that can be misleading. According to the documentation, "asynchronous library functions that return control to the host thread before the device completes the requested task." Notably, this definition does not imply that asynchronous API calls are *nonblocking* to the CPU. As noted in Sec. 3, we have found situations in which kernel launches still cause CPU blocking even if the API call returns before the requested kernel completes.

3 Synchronization and Blocking

CPU scheduling has been studied and well-understood for decades; in particular, real-time scheduling analysis of task systems is based on predictable scheduler and task behaviors. A worst-case execution time (WCET) for each task can be determined using clear specifications of the machine's architecture including the cache, bus, and DRAM operations. Incorporating GPUs into real-time analysis (as with all coprocessors), requires different models with new sets of issues to be considered. In this section, we discuss one set of issues that lead to a surprising number of pitfalls when CUDA GPUs are used: *synchronization*.

In prior work, we investigated the scheduling rules for kernels and copy operations in CUDA programs [1]. However, this investigation focused on a limited context where few CUDA operations beyond kernel launches and memory copies were used. In most real-world CUDA software, programmers will likely encounter (both intentionally and unintentionally) the need for synchronization between CPU and GPU operations. The added complexity of synchronization can result in utilization loss, potentially leading to unbounded response times in task sets with high utilization. In this section, we explore various forms of CPU-GPU synchronization and the resulting implications for real-time systems. We limit attention for now to CPU tasks that share a single Linux address space and create user-defined streams. As covered in detail in Sec. 4, this setup allows potential concurrency among operations on the GPU.

3.1 Overview of GPU Synchronization

Most developers are familiar with the concepts of synchronization in a CPU-only context where two or more tasks must communicate or coordinate their actions. Synchronization becomes more complicated when a CPU task must coordinate with programs executed on the GPU. The common case is that the CPU task must determine when data in GPU memory is safe to access (*e.g.*, copy back to CPU memory). This is accomplished using *GPU synchronization*, where the GPU must complete outstanding work and reach a *synchronization point*: a point in time when data access can safely occur. There are also other, less

but does not describe how operations from different streams are ordered.

[4] Specifically, Section 3.2.5.1 of the Programming Guide for CUDA version 9.1.85.

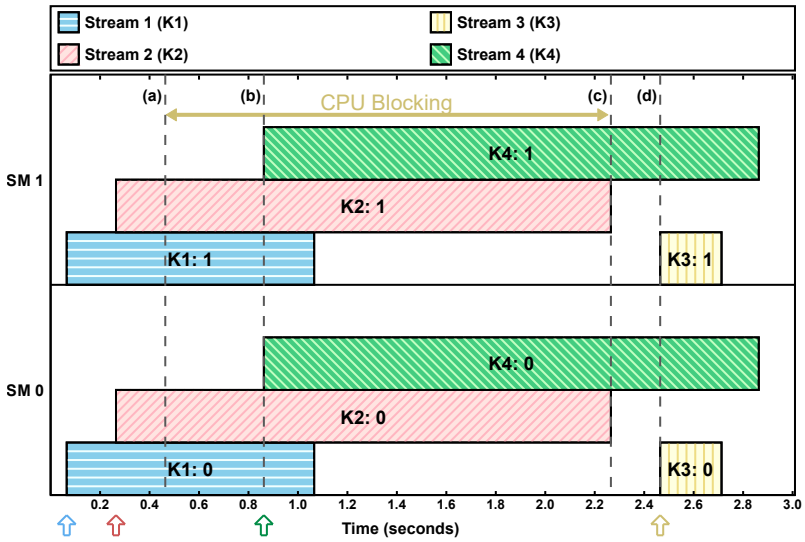

Figure 2 Explicit synchronization requested before K3, observed on the Jetson TX2.

common, cases when GPU synchronization is necessary.

In CUDA there are multiple ways to achieve GPU synchronization. They fall into two broad 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. We have uncovered in our research some unfortunate pitfalls relating to actual GPU synchronization behavior, especially with respect to *blocking*. So, while these may not be pitfalls for non-safety-critical applications, ignoring the effects of certain specific mechanisms for achieving synchronization would be perilous in a safety-critical system where blocking must be anticipated and accounted for in analysis.

3.1.1 Explicit Synchronization

Explicit synchronization refers to synchronization points that the CUDA programmer explicitly requests using the CUDA API. Explicit synchronization is typically used after a program has launched one or more asynchronous CUDA kernels or memory-transfer operations and must wait for computations to complete. In contrast to implicit synchronization, the sole purpose of explicit-synchronization functions is to block the calling CPU task until the GPU reaches a synchronization point.

The CUDA documentation[5] states that explicit synchronization will block the calling task until "all preceding commands" have completed. For example, if the API function `cudaDeviceSynchronize` is invoked, "preceding commands" may encompass all commands issued to the device from all CPU tasks. Other explicit-synchronization options, including `cudaStreamSynchronize`, will only block until preceding commands from a specified stream have completed.

We carried out experiments using our open-source framework[6] to investigate the specific behaviors of GPU synchronization on real GPU hardware. Fig. 2 shows the behavior of explicit synchronization observed in one such experiment. In Fig. 2 (also in Figs. 3 and 4),

[5] Section 3.2.5.5.3 of the Programming Guide for CUDA version 9.1.85.
[6] Available at `https://github.com/yalue/cuda_scheduling_examiner_mirror`.

each shaded rectangle corresponds to a separate thread block. The left and right endpoints of each rectangle correspond to the times at which the block started and completed execution, as measured on the GPU. Each rectangle's height represents its size in CUDA threads. Additionally, the vertical axis is subdivided by SM. The particular experiments presented in Figs. 2-4 were performed using the Jetson TX2, which features two SMs. Up to 2,048 CUDA threads can be assigned to a single SM at once.

The CUDA program executed to produce Fig. 2 consists of four CPU tasks all sharing a single address space. Each CPU task launched one kernel in a separate user-defined stream. Kernel launches were separated by a small amount of time. Each kernel consisted of two blocks of 512 threads, and the figure shows that one block from each kernel was scheduled on each SM. Each thread performed a busy-loop for a set amount of time.

An explicit-synchronization command, `cudaDeviceSynchronize`, was issued at time **(a)** by the CPU task responsible for launching kernel K3. This caused K3's CPU task to be blocked until the prior commands, the execution of kernels K1 and K2, had both completed at time **(c)**. This behavior is exactly what one would expect, given the description of explicit synchronization from official documentation. However, our experiments also uncovered Pitfall 1 for the unwary:

▶ **Pitfall 1.** *Explicit synchronization does not block future commands issued by other tasks.*

The fact that the launch of K4 by its CPU task was not blocked at time **(b)** is an example of this pitfall. Implicit synchronization, which we cover next, presents even more serious pitfalls.

3.1.2 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[7] states that "two commands from different streams cannot run concurrently if any one of the following operations is issued in-between them by the host thread:

1. A page-locked host memory allocation
2. A device memory allocation
3. A device memory set
4. A memory copy between two addresses to the same device memory
5. 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 2.** *Documented sources of implicit synchronization may not occur.*

[7] Section 3.2.5.5.4 of the Programming Guide for CUDA version 9.1.85.

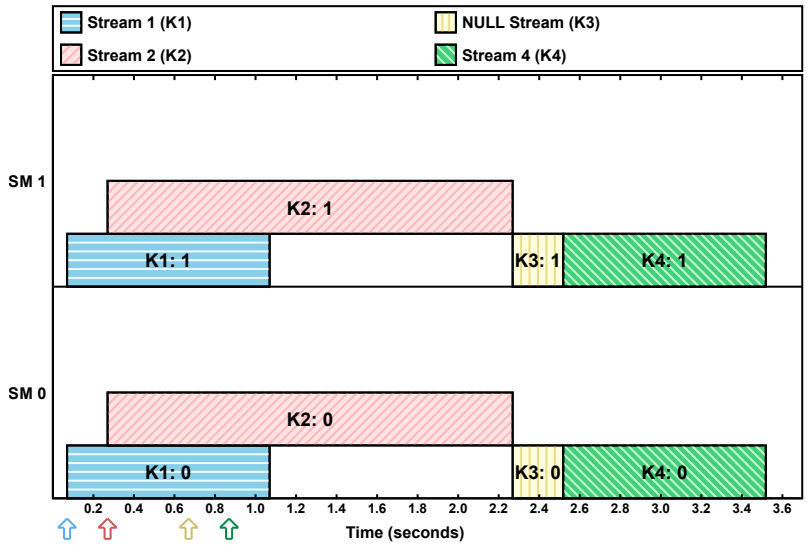

Figure 3 Implicit synchronization caused by launching kernel K3 in the NULL stream.

Pitfall 2 became apparent to us when, in all of our experiments, we never observed implicit synchronization as a result of a device-memory operation (allocation, set, or copy) or a page-locked host memory allocation. Our experiments covered the two most recent CUDA versions, 8.0 and 9.0, and the three most recent NVIDIA GPU architectures, Maxwell, Pascal, and Volta. This, of course, does not prove that implicit synchronization can *never* happen under such circumstances, but it does indicate that the documentation's statement that "two commands cannot run concurrently" is not a reliable rule. The only case (from this list) in which we did observe implicit synchronization was launching GPU operations in the NULL stream.

Fig. 3 shows a similar scenario to the one in Fig. 2, with one key difference: the CPU task for K3 did not call `cudaDeviceSynchronize` before K3 was launched, but instead launched 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, K4 is also prevented from running concurrently. Even though this loss of concurrency may be striking, it is notably explicitly documented, and can be used (or avoided) in a careful design for a real-time task.

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

▶ **Pitfall 4.** *Some CUDA API functions will block future, unrelated, CUDA tasks* ***on the CPU****.*

Fig. 4 shows the results of an experiment identical to the one in Fig. 2, but this time the call to `cudaDeviceSynchronize` at time **(a)** was replaced with a call to `cudaFree`, which was used to de-allocate memory on the GPU. Pitfalls 3 and 4 can be observed in this plot. The fact that this blocked the calling CPU thread until all prior GPU work had completed at time **(c)** indicates that `cudaFree` created implicit synchronization. Similar

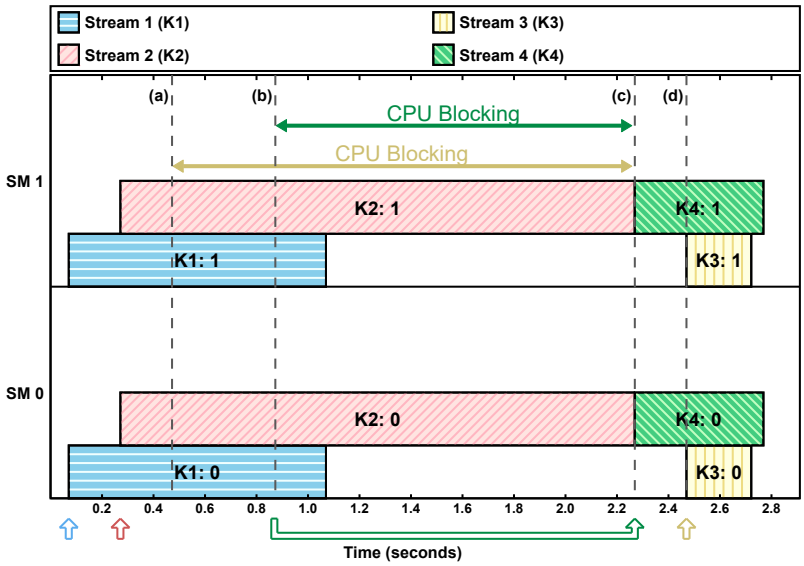

Figure 4 Implicit synchronization causing additional CPU blocking due to `cudaFree`.

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 requires 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. This reveals a pitfall that can harm real-time analysis that does not consider the fact that CPU tasks can experience blocking from GPU operations that are launched from unrelated tasks.

3.2 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. A more robust method would be to adopt middleware that handles such problems transparently.

Our experiments indicate that GPU synchronization does not extend across GPU-using tasks that are isolated in separate address spaces. If synchronization is the dominant limiting factor on schedulability, it may be desirable to place each task in a separate address space (OS process). As explained in the next section, this organization means that that CUDA kernels from different tasks can no longer execute concurrently, but it may still be beneficial overall if synchronization-related blocking is a greater limiting factor.

It turns out that NVIDIA may be aware of this issue. Even though it is not currently available for embedded platforms such as the TX2, NVIDIA does provide useful middleware for discrete GPUs: the *CUDA Multi-Process Service* (MPS). MPS allows kernels from multiple processes to execute concurrently on a single GPU, while maintaining the desirable

property that GPU synchronization from one process will not affect other processes. We explore the benefits of MPS further in Sec. 4.

4 Concurrency and Performance

In prior work, we investigated different GPU scheduling behavior when running GPU-using real-time task systems in two contexts: **(i)** each task has its own distinct address space, *i.e.*, it runs as an OS process, and **(ii)** all tasks belong to the same address space, *i.e.*, each task runs as a schedulable thread within a process. We refer to these two contexts as *process-based* and *thread-based* tasks, respectively.

While process-based tasks have the advantage of memory protection, they do not actually execute on the GPU concurrently; instead, GPU operations are multiprogrammed in a way that makes predictable scheduling of GPU-related resources difficult if not impossible to achieve [1]. When operations are multiprogrammed on a GPU, their execution times depend on contention for shared GPU resources, making it hard to bound a task's overall execution time. Additionally, concurrency among GPU operations may be important in order to avoid wasting GPU processing cycles, especially when a single kernel cannot fully utilize the GPU's resources. Although this may be avoided by running tasks with user-defined streams in a shared address space, a shared address space may actually *reduce* concurrency in task systems where tasks regularly interfere with each other via implicit synchronization (Sec. 3). Fortunately, NVIDIA provides a third option: middleware called the *Multi-Process Service* (*MPS*) [20].

4.1 Multi-Process Service (MPS)

MPS enables concurrent execution of GPU operations launched by independent CPU address spaces. It has the potential to combine the advantages of both thread- and process-based tasks. Programs written using the CUDA API require no changes to use MPS—if MPS is running, CUDA programs transparently issue requests to MPS rather than directly to a GPU. Official documentation reports that MPS operates as a server process with its own CUDA context, and that CUDA API requests are redirected from client processes to the MPS server. Because the server's CUDA context is effectively shared, GPU operations launched by separate processes can execute concurrently on a shared GPU, providing the benefits of thread-based tasks. However, MPS also continues to preserve the advantage of process-based tasks: separate processes will not block each other with implicit or explicit synchronization.

It is not clear from available documentation how MPS actually schedules GPU operations and whether the GPU scheduling rules revealed in prior work [1] are followed under MPS. For example, the documentation for MPS only mentions possible overlap between kernels and copy operations.[8] Given the documentation flaws discussed in Secs. 3 and 5.2, one could be skeptical of the veracity of this claim, so we verified experimentally that those scheduling rules are also followed under MPS. We omit from this paper the experimental methods used for verifying the scheduling rules; readers can refer to [1].

Maximizing the utilization of GPU resources using streams in thread-based tasks is suggested by NVIDIA's "Best Practices Guide" [22]. However, it would be unwise to simply take

[8] "MPS allows kernel and memory copy operations from different processes to overlap on the GPU, achieving higher utilization and shorter running times" [20].

	Multiple Process-based Tasks	Multiple Thread-based Tasks
Without MPS	MP	MT
With MPS	MP(MPS)	MT(MPS)

Table 1 Abbreviations used for our four experimental scenarios.

this recommendation at face value when choosing between MPS or a process- or thread-based task organization in a safety-critical system. Additionally, ***MPS is not yet supported on embedded ARM platforms*** like the Jetson TX2, so the other management systems are still necessary on some systems. Therefore, we conducted a case study on computer-vision software, demonstrating the performance differences among the available configurations.

4.2 Case Study of Computer-Vision Tasks

Our motivation primarily remains autonomous driving, so we chose to study algorithms for computer-vision tasks that provide functions commonly used for autonomous driving. In evaluating the results from this case study, we consider that the real-time tasks that use GPUs for autonomous driving may have multiple levels of criticality. Some may be safety-critical with hard deadlines and be provisioned for worst-case execution plus a margin for safety. Others may have only bounded tardiness requirements, or even be background work that can be provisioned for average-case execution.

We focus here on five programs from NVIDIA's provided sample code for VisionWorks:

- **Video Stabilization.** Smooths shaky video content. This is often a preprocessing step for a computer-vision pipeline.
- **Feature Tracking.** Tracks features between consecutive frames. This algorithm is used to track the positions of objects in a scene.
- **Motion Estimation.** Estimates the direction of moving pixels, which is fundamental to calculating trajectories of moving objects, *e.g.*, pedestrians and other vehicles.
- **Hough Transform.** (*Hough*) A feature-extraction algorithm; the provided sample detects circles and lines in images.
- **Stereo Matching.** Uses input from two cameras to generate depth information by matching features in both frames.

Methodology. We adapted NVIDIA's VisionWorks samples to be compatible with our open-source experimental framework.[9] These samples generally only use a single CUDA stream. We ran four instances of the same sample program in each experiment. We configured each instance to process 1,000 frames from a video sequence while recording per-frame response times. Our framework allows running each program instance in a shared address space (multiple thread-based tasks, MT) or in independent address spaces (multiple process-based tasks, MP), both with and without the MPS server active. This produces experiments for each algorithm in four different scenarios as summarized in Tbl. 1. Experimental results under MT(MPS) were always similar to MT with slight overheads caused by MPS, so we omit it in all of our results for clarity. We conducted these experiments on a Maxwell-architecture discrete GPU with CUDA 9.0. We briefly summarize results on other devices and different CUDA versions later.

[9] Again, `https://github.com/yalue/cuda_scheduling_examiner_mirror`.

VisionWorks Samples	Scenarios	Max	99^{th}%	90^{th}%	Mean	Median
Video Stabilization	MP	17.55	12.88	5.43	3.31	2.69
	MP (MPS)	36.73	**11.12**	**5.37**	**2.81**	**2.06**
	MT	**17.0**	13.87	8.94	4.72	3.63
Feature Tracking	MP	**5.64**	**3.87**	**1.45**	**1.08**	**0.96**
	MP (MPS)	14.73	6.04	1.51	1.31	1.09
	MT	31.11	20.86	11.51	4.68	2.68
Motion Estimation	MP	**28.64**	**21.25**	17.33	16.75	17.24
	MP (MPS)	33.05	22.66	**15.75**	**14.3**	**14.89**
	MT	42.86	26.12	16.53	15.07	15.14
Hough Transform	MP	**13.56**	**11.61**	7.28	5.68	5.7
	MP (MPS)	18.35	11.66	**6.44**	**3.74**	**3.18**
	MT	58.65	22.64	15.82	9.12	8.94
Stereo Matching	MP	75.13	50.54	30.42	24.14	24.77
	MP (MPS)	**59.73**	**45.05**	**26.87**	22.59	24.41
	MT	125.96	58.82	34.36	**20.75**	**18.95**

Table 2 Per-frame response time data (in milliseconds) of VisionWorks samples. The fastest scenario for each time metric is indicated by bold text.

Results. We show cumulative distribution function (CDF) and kernel density estimation (KDE)[10] plots of Hough and feature tracker as representatives in Figs. 5–8. The KDE curve was produced using the `Python` package `scipy.stats.gaussian_kde`. In both the CDF and KDE plots, each curve represents the recorded response-time data in an experimental scenario. For example, the curves labeled "x4 MP" in Figs. 5 and 6 represent the per-frame response time distributions where each of four Hough instances is run in a separate process. Result data for all five algorithms is summarized in Tbl. 2, which lists the maximum, 99^{th}-percentile, 90^{th}-percentile, mean, and median frame times for each scenario and algorithm.

▶ **Observation 1.** *MP(MPS) exhibits good average-case performance.*

Obs. 1 is supported by the data in Tbl. 2. 90^{th}-percentile, mean, and median performance under configuration MP(MPS) were consistently good with the top performance for three of the five algorithms. For Feature Tracking, MP was best in all metrics, and for Stereo Matching, MT had better mean and median performance. The results for average-case performance indicate that using MP(MPS) would likely be an attractive option for soft-real-time systems, *e.g.*, systems that can occasionally drop a video frame without compromising safety. We conjecture that the average-case performance advantage of MP(MPS) over MP in most cases is due to improved concurrency and lower GPU context-switching overheads.

Feature Tracking was the most notable exception to Obs. 1. In this case, MP was only slightly better than MP(MPS) when comparing the 90^{th}-percentile, mean, and median performance. We conducted additional experiments using NVIDIA's CUDA-profiling tool, `nvprof`, to gain some insight into this behavior. We found that Feature Tracking's overall execution time is heavily influenced by a large number of memory transfers, rather than

[10] KDE is a statistical method for estimating a continuous probability density function (PDF) from a set of discrete sample values.

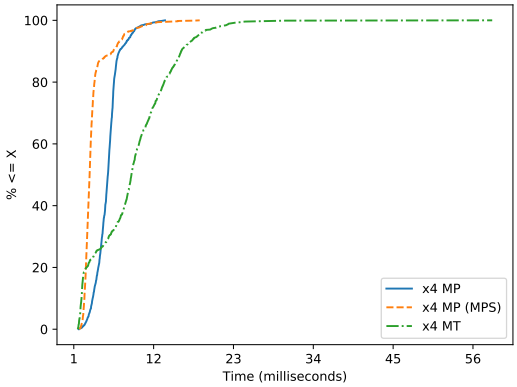

Figure 5 Per-frame response time CDFs for Hough.

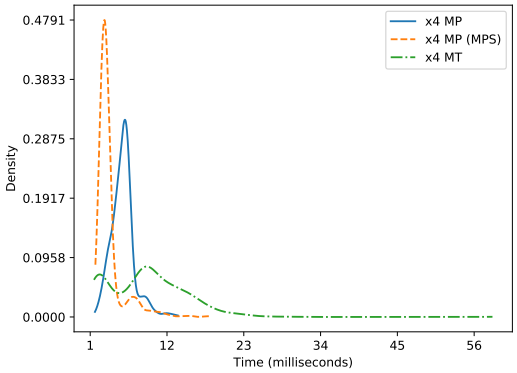

Figure 6 Per-frame response time KDEs for Hough.

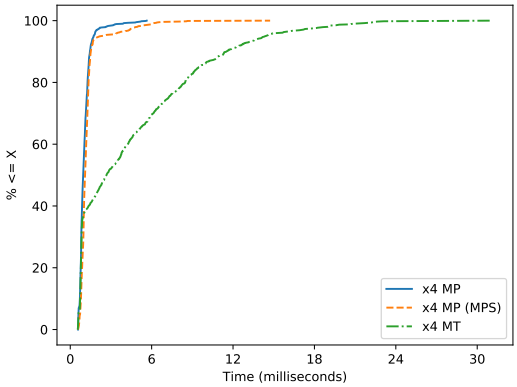

Figure 7 Per-frame response time CDFs for Feature Tracking.

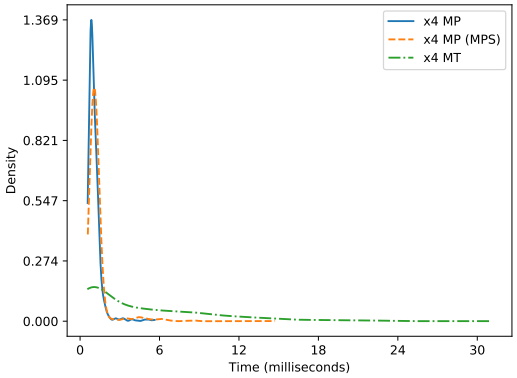

Figure 8 Per-frame response time KDEs for Feature Tracking.

CUDA kernel executions. This likely means that MPS only provides limited GPU concurrency benefits to Feature Tracking, which failed to outweigh other MPS-related overheads.

▶ **Observation 2.** *Worst-case and* 99^{th}*-percentile runtimes were typically better under MP.*

While MP(MPS) largely resulted in average-case improvements, Tbl. 2 shows three of our five applications (Feature Tracking, Motion Estimation, and Hough Transform) showed the smallest worst-case and 99^{th}-percentile execution times under MP. This indicates that MP may be a better option for certain task systems where worst-case performance is more important than average-case. Our results illustrate why the trade-offs between process-based and thread-based designs for tasks must be evaluated for individual algorithms.

▶ **Observation 3.** *MP and MP(MPS) exhibit more predictable execution times than MT.*

Obs. 3 is supported by Figs. 6 and 8, where the KDE shows a tight unimodal distribution for MP and MP(MPS) but not MT. A unimodal distribution function with little dispersion indicates that the response times exhibit low variance. MT, in contrast, shows both bimodal (in Fig. 6) and unimodal (in Fig. 8) distributions with significant dispersions (indicating high variance). Even if specific "spikes" are more difficult to observe in the corresponding CDF plots, the difference in response-time ranges are also apparent from the endpoints of the CDF curves in Figs. 5 and 7.

▶ **Observation 4.** *The MT configuration generally performed poorly.*

Obs. 4 is supported by Tbl. 2 and the plots. The only metrics where MT outperformed the other scenarios were the mean and median times for Stereo Matching, and worst-case response time for Video Stabilization (where MT was only slightly better than MP).

Other Results. In addition to the results presented above, we also conducted this case study using CUDA 8.0 on a Maxwell discrete GPU (GTX 860M) and CUDA 9.0 on Pascal discrete GPUs (GTX 1050 and GTX 1070). Even though we chose to omit tables of results from the other GPUs and CUDA versions in this paper, we made similar observations excepting that the performance of all configurations was better on a Pascal GPU. Additionally, the experimental results with CUDA 8.0 on the same Maxwell GPU stayed nearly identical to those using CUDA 9.0.

Summary. Our case study compared the impact of different GPU-sharing approaches on the performance of computer-vision algorithms. The results we obtained for these algorithms ran contrary to some of our observations regarding GPU concurrency from prior work [1, 26].

▶ **Pitfall 5.** *The suggestion from NVIDIA's documentation to exploit concurrency through user-defined streams may be of limited use for improving performance in thread-based tasks.*

We assumed that enabling concurrent GPU execution was of significant importance for limiting capacity loss in real-time workloads on embedded systems, and therefore fell victim to Pitfall 5. Instead, our results show that MT rarely outperforms tasks running as multiple processes, even without MPS. Additionally, any performance improvement via fine-tuned stream organization for MT can also be achieved with MP(MPS). That being said, even though enabling concurrency using MP(MPS) is generally beneficial, it unfortunately is not an option on ARM-based embedded platforms like the Jetson TX2. We would encourage NVIDIA to consider this shortcoming in hope that one day it may be addressed.

5 Perils of CUDA Programming for Real-Time Tasks

In the previous sections we presented several specific pitfalls in correctly designing and running CUDA programs for real-time tasks. Elements of both CUDA's design and documentation contribute to this ensemble of perils to avoid. In this section, we discuss some of the broader categories of pitfalls.

5.1 Synchronous Defaults

As hinted in Sec. 3, one of the primary pitfalls when designing a real-time task system that uses a GPU is that *all possible* blocking must be accounted for in analysis. Therefore, reducing the amount of blocking on both the CPU and GPU is essential. On the GPU, this requires issuing all CUDA operations to user-defined (non-NULL) streams, and carefully controlling the use of other API functions, like `cudaFree`, that cause blocking via implicit synchronization.

Even though it may seem like an easy task for a programmer to just specify a user-defined stream as opposed to the NULL stream, we note that 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 Listings 1 and 2, which present a particular example of Pitfall 6 below.

Listing 1 Causes implicit synchronization.

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

Listing 2 Correctly asynchronous.

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

▶ **Pitfall 6.** *`Async` CUDA functions use the GPU-synchronous NULL stream by default.*

Listing 1's call to `cudaMemsetAsync` is missing a final argument specifying a user-defined stream, which causes the NULL stream to be used by default. As pointed out in Sec. 3.1.2, NULL-stream usage causes implicit synchronization and hence blocking. This mistake is corrected in Listing 2. This specific mistake actually led to *months* of mystifyingly inconsistent results in our own experiments—despite our relatively deep experience examining the subtleties of CUDA behavior (note that these code snippets are parts of much larger listings). *Would an ML application developer catch such a mistake or appreciate its impact?* Note that NVIDIA's CUDA compiler does not catch this mistake because the compiler is based on the C++ programming language, which allows default arguments to functions.

Even though the examples in Listings 1 and 2 only use `cudaMemsetAsync`, Pitfall 6 applies to other CUDA API functions as well, such as `cudaMemcpyAsync`. The fact that the CUDA documentation indicates that these functions cause implicit synchronization, as discussed in Sec. 3 and Sec. 5.2, makes potential programmer errors even harder to notice in cases where synchronization is due to NULL-stream usage rather than memory operations.

To summarize this discussion, CUDA provides a brittle programming environment: difficult-to-spot mistakes can have profound consequences for real-time tasks.

5.2 Flawed Documentation

Another substantial danger stems from the inaccurate official documentation provided by NVIDIA. While function signatures and data structures seem to receive accurate (but often sparse) official documentation, scheduling and synchronization remain under-discussed. Our group's past work includes demystifying some scheduling rules [1]. In our work to demystify implicit synchronization (see definition in Sec. 3.1.2), however, we came across not only missing documentation, but incorrect documentation.

▶ **Pitfall 7.** *Observed CUDA behavior often diverges from what the documentation states or implies.*

Consider Tbl. 3. In all but one of the cases we investigated, the documentation claims implicit synchronization will occur when it does not. While this absence of synchronization may positively benefit performance, it also may cause incorrect timing analysis. Furthermore, program logic may be broken in the (albeit unlikely) case that the program relies on a function like `cudaMemsetAsync` to trigger GPU synchronization.

Unfortunately, the documentation also contains less-benign flaws. Take `cudaFree` and `cudaFreeHost` as an example. Our experiments in Sec. 3 found these functions to not only cause implicit synchronization, but block other CPU tasks from proceeding while `cudaFree` waits on the GPU. Much to our surprise, the documentation mentions neither of these side effects, leaving the reader to assume that these functions behave similarly to other CUDA functions and have no side effects.

Our experiments also revealed that `cudaMalloc` and `cudaMallocHost` may also cause cross-task CPU blocking in a similar manner to `cudaFree` in certain situations, even though

Source	Observed Behavior			Documented Behavior	
	Blocks Other CPU Tasks	Implicit Sync. (Sec. 3.1.2)	Caller Must Wait for GPU	Implicit Sync. (Sec. 3.1.2)	Caller Must Wait for GPU
`cudaDeviceSynchronize`	No	No	Yes	No	Yes
`cudaFree`	Yes	Yes	Yes	**No** (undoc.)	**No** (impl.)
`cudaFreeHost`	Yes	Yes	Yes	**No** (undoc.)	**No** (impl.)
`cudaMalloc`	?	No	No	**Yes**	No (impl.)
`cudaMallocHost`	?	No	No	**Yes**	No (impl.)
`cudaMemcpyAsync D-D`	No	No	No	**Yes**	No
`cudaMemcpyAsync D-H`	No	No	No	**Yes***	No
`cudaMemcpyAsync H-D`	No	No	No	**Yes***	No
`cudaMemset (sync.)`	No	Yes	No	Yes	No
`cudaMemsetAsync`	No	No	No	**Yes**	No
`cudaStreamSynchronize`	No	No	Yes	No	Yes

Table 3 Observed vs. documented synchronization sources in CUDA. For `cudaMemcpyAsync` we distinguish the direction of copy between device and host: (D-D) internal to GPU memory; (D-H) GPU memory to CPU memory; (H-D) CPU memory to GPU memory. *The documentation is contradictory for these instances, but the more detailed option indicates that these functions only cause synchronization if host memory is not page-locked. We were unable to observe this regardless of whether host memory was page-locked or not.

these functions do not trigger implicit synchronization. As we have not yet determined the specific causes for this behavior, this property is indicated by an entry of '?' in certain cells in Tbl. 3. In any case, we failed to find any mention of this variant of CPU blocking in the CUDA documentation, and investigating these functions remains an open topic that we plan to explore in future work.

An especially worrying pitfall is the following:

▶ **Pitfall 8.** *CUDA documentation can be contradictory.*

In one case, namely `cudaMemcpyAsync`, we discovered that the CUDA documentation actively contradicts itself. Section 3.2.5.1 of the CUDA Programming Guide states "The following device operations are asynchronous with respect to the host: ... Memory copies performed by functions that are suffixed with `Async`," but Section 2 of the CUDA Runtime API documentation states "For transfers from device memory to pageable host memory, [`cudaMemcpyAsync`] will return only once the copy has completed." This raises further doubts about the correctness of other parts of the CUDA documentation.

We note that the CUDA API contains 146 non-deprecated or compatibility-related functions, and we have only tested a small fraction of these in depth. Therefore, it is likely that our findings with Pitfalls 7 and 8 apply to other portions of the documentation that we have yet to observe.

5.3 Unknown Future

All of the pitfalls discussed in this paper, as well as the need to compare the alternatives considered in Sec. 4 empirically, can be attributed to a single overarching problem: the black-box nature of current GPU-enabled platforms means that *developers do not have a reliable model of GPU behavior*. Much of our group's prior work has focused on developing such a model. However, this highlights what is perhaps the most important pitfall:

▶ **Pitfall 9.** *What we learn about current black-box GPUs may not apply in the future.*

Despite the fact that we validated our experimental results on several of the most recent 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, but also may apply to performance characteristics like memory-access times, as we found in prior work [26].

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 Pitfall 9 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. Only then we can have a reliable GPU model upon which to base real-time analysis and certification. We hope that work such as ours signals to manufacturers like NVIDIA that greater openness is a desirable feature when marketing in safety-critical domains.

Unfortunately, there is little indication that NVIDIA plans to move towards open hardware or software in the immediate future. In the meantime, one of our continuing objectives is to produce tools, such as our experimental framework, that can be quickly adapted to new GPU hardware. So far, our tools have allowed us to quickly re-validate our prior results every time NVIDIA updates its black-box hardware or software.

6 Conclusion

Vehicles on the road today are already running highly complex GPU-accelerated applications. We anticipate a future where safety-critical autonomous vehicles must be certified, but this will require a change in the GPU-programming paradigm. Currently, computer-vision applications are developed with little guidance about how to achieve temporal safety. Even if a single programmer or application avoids some mistakes, it is increasingly difficult to avoid all of them, especially as applications and task systems grow in complexity. This necessitates work such as ours, which seeks to reduce the gap between computer-vision application developers and those responsible for certifying new systems' real-time safety.

With little openness in NVIDIA's hardware and software ecosystem, this paper contributes a list of potential pitfalls when developing CUDA applications for real-time systems. Reasons for these pitfalls include GPU synchronization, application performance, and problems with documentation. We uncovered these pitfalls via microbenchmark experiments, examining the performance of real-world computer-vision applications, and a careful reading of official GPU documentation. While there is no guarantee of stability in our observations as NVIDIA's hardware and software continues to evolve, we hope that our open-source experimental system will at least make it apparent when changes do occur.

This paper is part of an ongoing project with the aim of developing an abstract model of GPU execution. In the future, we plan to continue this investigation and eventually develop middleware capable of intercepting and reordering or delaying GPU operations. Our hope is that the control afforded by such middleware will enable us to produce reasonable analytical bounds on blocking and response times, while maintaining high GPU utilization wherever possible. However, even with better management, certifiable safety in the face of GPU sharing requires a *guarantee* that pitfalls including blocking due to GPU synchronization are controlled, which is only possible if developers of GPU-using software are aware of the consequences and how to avoid them. Fortunately, the best practices we have laid out herein should alleviate much of the strain on application developers on their first foray into real-time systems.

In addition to NVIDIA's GPU, we will also investigate other GPU implementations, *e.g.*, AMD's open-source GPU runtime and driver stack. Given the chances of modifying AMD's open-source implementation, we are interested in improving the real-time guarantees of AMD's GPUs and comparing them with NVIDIA's GPUs.

References

1. T. Amert, N. Otterness, M. Yang, J. Anderson, and F. D. Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *RTSS 2017*.
2. J. Aumiller, S. Brandt, S. Kato, and N. Rath. Supporting low-latency CPS using GPUs and direct I/O schemes. In *RTCSA '12*.
3. C. Basaran and K. Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *ECRTS '12*.
4. K. Berezovskyi, K. Bletsas, and B. Andersson. Makespan computation for GPU threads running on a single streaming multiprocessor. In *ECRTS '12*.
5. K. Berezovskyi, K. Bletsas, and S. Petters. Faster makespan estimation for GPU threads on a single streaming multiprocessor. In *ETFA '13*.
6. K. Berezovskyi, F. Guet, L. Santinelli, K. Bletsas, and E. Tovar. Measurement-based probabilistic timing analysis for graphics processor units. In *ARCS '16*.
7. K. Berezovskyi, L. Santinelli, K. Bletsas, and E. Tovar. WCET measurement-based and extreme value theory characterisation of CUDA kernels. In *RTNS '14*.
8. A. Betts and A. Donaldson. Estimating the WCET of GPU-accelerated applications using hybrid analysis. In *ECRTS '13*.
9. N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna. SiGAMMA: Server based integrated GPU arbitration mechanism for memory accesses. In *RTNS 2017*.
10. R. Cavicchioli, N. Capodieci, and M. Bertogna. Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms. In *RTNS 2017*.
11. G. Elliott, B. Ward, and J. Anderson. GPUSync: A framework for real-time GPU management. In *RTSS '13*.
12. B. Forsberg, A. Marongiu, and L. Benini. Gpuguard: Towards supporting a predictable execution model for heterogeneous SoC. In *DATE '17*.
13. A. Horga, S. Chattopadhyayb, P. Elesa, and Z. Peng. Systematic detection of memory related performance bottlenecks in GPGPU programs. In *JSA '16*.
14. P. Houdek, M. Sojka, and Z. Hanzálek. Towards predictable execution model on ARM-based heterogeneous platforms. In *ISIE '17*.
15. 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*.
16. S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX ATC '11*.
17. H. Lee and M. Abdullah Al Faruque. Run-time scheduling framework for event-driven applications on a GPU-based embedded system. In *TCAD '16*.
18. A. Li, G. van den Braak, A. Kumar, and H. Corporaal. Adaptive and transparent cache bypassing for GPUs. In *SIGHPC '15*.
19. X. Mei and X. Chu. Dissecting GPU memory hierarchy through microbenchmarking. In *TPDS '16*.
20. Multi-process service. Online at `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`.
21. NVIDIA. Embedded systems developer kits and modules. Online at `http://www.nvidia.com/object/embedded-systemsdev-kits-modules.html`.

22 NVIDIA. Best practices guide. Online at `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`, 2017.

23 NVIDIA. CUDA toolkit documentation v9.1.85. Online at `http://docs.nvidia.com/cuda/`, 2018.

24 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*.

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

26 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*.

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

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

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

30 H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS '10*.

31 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*.

32 J. Zhong and B. He. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25:1522–1532, 2014.

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