### Title

Methods for multitasking among real-time embedded compute tasks running on the GPU




### Journal

Concurrency and Computation Practice and Experience, 29(15)

### ISSN

1532-0626



### Authors

Muyan-Özçelik, Pınar
Owens, John D



### Publication Date

2017-08-10

### DOI

10.1002/cpe.4118

Peer reviewed

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* 2016; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

# Methods for Multitasking among Real-time Embedded Compute Tasks Running on the GPU

Pınar Muyan-Özçelik[1]*and John D. Owens[2]

[1]*California State University, Sacramento, CA, USA E-mail: pmuyan@csus.edu*
[2]*University of California, Davis, CA, USA E-mail: jowens@ece.ucdavis.edu*

SUMMARY

In this study, we provide an extensive survey on wide spectrum of scheduling methods for multitasking among GPU computing tasks. We then design several schedulers and explain in detail the selected methods we have developed to implement our scheduling strategies. Next, we compare the performance of schedulers on various workloads running on Fermi and Kepler architectures and arrive at the following major conclusions: (a) Small kernels benefit from running kernels concurrently. (b) The combination of small kernels, high-priority kernels with longer runtimes, and lower-priority kernels with shorter runtimes benefits from a CPU scheduler that dynamically changes kernel order on the Fermi architecture. (c) Due to limitations of existing GPU architectures, currently CPU schedulers outperform their GPU counterparts. We also provide results and observations obtained from implementing and evaluating our schedulers on the NVIDIA Jetson TX1 system-on-chip architecture. We observe that although TX1 has the newer Maxwell architecture, the mechanism used for scheduler timings behaves differently on TX1 compared to Kepler leading to incorrect timings. In this paper, we describe our methods that allow us to report correct timings for CPU schedulers running on TX1. Finally, we propose new research directions involving the investigation of additional scheduling strategies. Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: GPU computing; multitasking; real-time embedded tasks

## 1. INTRODUCTION

As Graphics Processing Units (GPUs) have become more programmable, we have increasingly used them for data-parallel applications beyond traditional graphics. The performance speedups provided by GPU computing make GPUs a good fit for data-parallel tasks, especially for those with time constraints such as real-time applications. GPUs also provide opportunities for embedded systems since they offer superior price-per-performance and power-per-performance [1]. It is common for embedded systems to run multiple data-parallel real-time tasks, concurrently. However, the current programming model of GPUs poses many challenges for multitasking of such tasks in an effective way. Hence, even though GPUs are a good fit for running real-time embedded tasks, our ability to utilize the full potential of GPUs in embedded systems requires developing effective multitasking strategies, our focus in this study.

Muyan-Özçelik and Owens [2] provide an overview of our system. This paper extends our prior work in three ways: (1) In our previous study, we provide a small-scale literature review of prior scheduling methods used for multitasking among GPU computing tasks. In this paper, we conduct an extensive survey on a wide spectrum of such scheduling methods. Hence, we

*Correspondence to: California State University, Sacramento, CA, USA E-mail: pmuyan@ecs.csus.edu

contribute to the GPU literature by indicating the shortcomings of these broad range of prior studies for running real-time embedded tasks and compare our schedulers to related approaches. (2) Our prior work introduces several methods that we have developed to construct our schedulers. Most of these methods are either new techniques proposed in this work or combine existing techniques in novel ways. In this paper, we provide in-depth descriptions of our selected methods so that future studies can utilize them as building blocks for implementing different scheduling strategies. (3) This paper further extends our previous work by providing results and observations obtained from implementing and evaluating our schedulers on the NVIDIA Jetson with the Tegra X1 [3] system-on-chip architecture. We observe that although TX1 has a Maxwell GPU and Maxwell is a successor to Kepler, the mechanism used for scheduler timings behaves differently on TX1 compared to the Kepler architecture. Hence, scheduler timings are reported incorrectly on TX1 unless we develop new methods. In this paper, we describe our methods that allow us to report correct timings for CPU schedulers running on TX1.

Embedded systems usually involve several data-parallel real-time tasks that need to run concurrently. For instance, the automotive computing domain involves tasks such as speed-limit-sign recognition, lane departure warning system, speech recognition, infotainment systems, etc. Similarly, mobile devices involve tasks such as augmented reality, face recognition, fingerprint unlocking, and so on. As the number of concurrent real-time data-parallel applications that embedded devices run increases, to take the full advantage of GPUs on these platforms, we need strategies that would allow GPUs to multitask among several real-time tasks.

The current programming model of GPUs poses several challenges for effectively managing workloads containing multiple concurrent data-parallel real-time tasks: (1) GPU computing research typically concentrates on high-performance computing applications performing only one demanding task at a time. Although current architectures provide some features that support running multiple GPU tasks at the same time, since they are not primarily designed for multitasking among real-time tasks, these features present several limitations for such a purpose. (2) GPUs have historically evolved to efficiently implement throughput-oriented applications such as graphics, so they are optimized for providing high throughput rather than low latency. However, real-time tasks may require either low latency or high throughput, or perhaps both. (3) Since GPUs are optimized for throughput rather than latency, they lack some important characteristics of real-time systems: having a time measure synchronized with the CPU, an ability to assign priorities, a preemption mechanism, and a fast interface to the CPU. This study aims to address the abovementioned challenges.

Commodity CPU-GPU systems lack support for performing hard-real-time tasks; these typically require extensive hardware and operating system (OS) features. Lacking this support prevents these systems from performing schedulability analysis and providing real-time guarantees. Thus, in this study, we focus on soft-real-time tasks, which use best-effort scheduling to meet real-time requirements.

The input to our system is a workload that consists of GPU computing tasks that belong to multiple real-time embedded applications. The tasks in this GPU workload specify different real-time requirements. The output of our system is several schedulers that perform multitasking among the workload tasks by adhering to the constraints specified in the real-time requirements. Since different workloads have different characteristics, we investigate the use of various schedulers that pursue alternative approaches instead of focusing on a specific scheduler. Hence, one of the important contributions of our study is that by considering the salient characteristics of GPU workloads, we design our schedulers using a variety of different scheduling strategies for multitasking among real-time embedded tasks.

To determine which scheduling strategy is more effective for a given workload and why, we evaluate and compare our schedulers that use alternative approaches using synthetic cases. To demonstrate a plausible scenario to which this study can be applied, we also provide an evaluation of our schedulers for a real-world case as described in Muyan-Özçelik and Owens [2]. For these experiments, we run our schedulers on the Fermi [4] and Kepler [5] architectures.

Another important contribution of our study is that based on our results, we highlight the shortcomings of current GPU architectures with regard to running multiple real-time tasks and



DOI: 10.1002/cpe

recommend new features that would allow better schedulers to be designed. We end our paper by proposing new research directions involving the investigation of additional scheduling strategies such as adding preemption capability to schedulers and exploring work division.

## 2. SURVEY

One of the important contributions of this paper is that we provide an extensive survey of scheduling methods used for multitasking among GPU tasks. As a result of our survey, we have seen that these methods use techniques that naturally fall into three major high-level categories: scheduling methods defined by applications, scheduling methods that use programming frameworks, and scheduling methods supported by operating systems/drivers.

### *2.1. Application-defined scheduling*

A scheduling method defined in an application involves a scheduling software written by developers, which runs either on the CPU or the GPU. In the literature, we see examples of application-defined schedulers running tasks in parallel on systems with GPUs. These studies perform multitasking in the context of dealing with out-of-core data [6], handling irregular workloads [7], constructing and running programmable rendering pipelines or applications [8], multitasking between two simultaneously running graphics and computation applications [9], constructing a general purpose ray tracing engine [10], allowing different CPU clients to share the resources of single GPU simultaneously [11], and so on. This previous work generally focuses on running one (complex) application at a time and does not target the real-time multi-application workloads that are the focus of this study.

In contrast, Elliott and Anderson [12] and Steinberger et al. [13] target real-time tasks. However, like most of the related prior work in this area [6,8], Elliott and Anderson focus on running multiple tasks on hybrid systems, which includes collections of CPUs, GPUs, special purpose hardware, etc., and schedule only some of the tasks to run on GPUs. Likewise, Softshell [13] splits work across multiple GPUs. On the other hand, we propose to schedule all the available data-parallel real-time work to run on a single GPU.

Studies performing multitasking on hybrid system usually run the scheduler on the CPU. Softshell, which involves work distribution between multiple GPUs, implements a small part of the scheduler on the CPU and the main part on the GPU. On the other hand, studies performing multitasking on a single GPU choose to run the scheduler on the CPU [9, 11] or on the GPU [7, 10]. CPU scheduling techniques usually execute one task at a time on the GPU. Since launching several concurrent kernels on a single GPU has only recently begun to be supported by new GPU architectures (e.g., Fermi [4]), we can say that CPU scheduling techniques tend to use the traditional GPU programming model. As an exception, Peters et al. [11] propose a CPU scheduling technique that relaxes the traditional GPU programming model and allows multiple tasks to be resident together on a single GPU using a persistent kernel. However, unlike our research, this study does not target real-time tasks. Having multiple concurrent tasks sharing the resources of a single GPU is also the central idea of GPU scheduling methods. To construct a framework that can provide this ability, these studies use a combination of several methods.

For instance, Tzeng et al. [7] use a combination of uber kernels, persistent threads [14], work queues, and warp-sized blocks. Parker et al. [10] use megakernels with state machine mechanism similar to the uber kernel/persistent thread approach used by Tzeng et al. In addition, Parker et al. use a just-in-time compiler, providing techniques to automatically combine the user-provided shader programs into megakernels. Peters et al. [11] advocate launching one persistent kernel consisting of a set of device functions, which again has a similar structure to uber/megakernels. As mentioned above, unlike Tzeng et al. and Parker et al., Peters et al. run the scheduling logic on the CPU. To control the execution of the persistent kernel from the CPU, Peters et al. use a method enabled by asynchronous memory transfers between the CPU and the GPU.



DOI: 10.1002/cpe

Since they improve the restricted communication between the CPU and the GPU, techniques utilizing asynchronous memory transfer are powerful methods for implementing CPU schedulers. In the current programming model, CPUs can send signals to GPUs, but cannot send interrupts. On the other hand, GPUs cannot signal or interrupt CPUs. Techniques enabled by asynchronous memory transfer allow us to mitigate these shortcomings. For instance, Stuart and Owens combine asynchronous memory transfer with the polling technique to allow the GPU to request work from the CPU [15]. In their later study, Stuart et al. [16] extend their technique using zero-copy memory and implement GPU-to-CPU callbacks. Zero-copy memory is also used by Steinberger et al. [13] to construct message buffers enabling communication between the CPU and the GPU.

In a recent study, Margiolas and O'Boyle [17] propose an approach that enables fair resource sharing control and software managed scheduling on accelerators. They use a just-in-time compiler (like Parker et al.) and a host runtime environment. Their approach is portable and transparent and requires no modifications or recompilation of existing systems. In addition, Margiolas and O'Boyle indicated that their system does not compromise security in favor of improved accelerator sharing.

### *2.2. Scheduling using programming frameworks*

Hardware and software vendors also give support for task parallelism by providing frameworks simplifying the process of adding concurrency to applications. Examples of such systems include Intel's Threading Building Blocks [18], Cilk [19], and Nulstein [20]; Microsoft's Concurrency Runtime [21] and Task Parallel Library [22]; and Apple's Grand Central Dispatch [23]. These frameworks abstract details that are difficult to manage for the developer such as scheduling of tasks to available processors. These systems are widely used in domains dealing with programs that can benefit from task parallelism and are intended to run on a variety of platforms with different processor topologies. Game development is one such domain. For instance, Werth [24] proposes using Nulstein or Threading Building Blocks for achieving task parallelism in game development. Although all the frameworks mentioned above target multi-core CPUs and do not work on GPUs, their design principles provide insights for building schedulers that can manage GPU tasks.

On the other hand, parallel programming frameworks such as OpenCL [25] and CUDA [26] target heterogeneous systems, including CPUs, GPUs, etc. The degree of task parallelism supported by these frameworks depends on the compute capability of the underlying GPU hardware. For instance, while performing some of the CUDA functions, the CPU and the GPU can execute concurrently. In addition, by using "streams" on GPUs with higher compute capabilities, CUDA can also allow overlap of GPU kernel execution with CPU/GPU data transfer as well as concurrent execution of multiple kernels on a single GPU. OpenCL provides similar functionalities by providing "execution queues", akin to streams. However, while these frameworks allow executing multiple kernels on high-end GPUs, they do not allow developers to specify scheduling policies such as resource allocation, priority assignment, etc. Hence, the techniques provided by these frameworks are by themselves not sufficient to perform multitasking among real-time tasks on a single GPU, which we target in this study.

Traditionally, implementation of real-time embedded applications are done using synchronous programming languages such as Esterel [27], Lustre [28], and Signal [29]. Synchronous models have been strong in safety-critical embedded control systems that can be found in aerospace and automotive domains. Most real-time systems naturally decompose into concurrent subcomponents. Implementing each subcomponent in a different program and making these programs communicate using operating system primitives have drawbacks due to the somewhat nondeterministic nature of standard operating system facilities such as interrupts. Some of these drawbacks include having little room for clean automatic system behavior analysis and no way of formally guaranteeing safety properties. Synchronous models aim to address these drawbacks by eliminating a need for OS schedulers, which in turn reduces system complexity [30].

Synchronous models provide deterministic system behavior and thus allow for formal verification of systems, which is very important for safety-critical real-time embedded applications. Determinism stems from synchronous concurrency, requiring all processes to execute in lock-step. Since synchronous models avoid interleaving, they are traditionally applied to centralized systems



DOI: 10.1002/cpe

such as circuits or single processors, and implementing them on asynchronous execution platforms is difficult. Tripakis et al. [31] address this shortcoming and propose extending synchronous models to asynchronous execution platforms and preserving deterministic semantics of synchronous models on these non-deterministic platforms to avoid mechanisms such as locks, semaphores, etc. They implement a synchronous model on distributed systems and mention that extension to other execution platforms or domains such as multicores or streaming applications might be possible. However, since we are not addressing safety-critical applications requiring formal verification (although it would be an interesting venue to explore), we are not pursuing the extension of these models to the GPU platform.

For implementation of hard real-time embedded systems, some programming frameworks, such as Atom [32], propose compile-time scheduling that provides guarantees of deterministic execution time and memory consumption. Providing these features simplifies the worst-case execution time analysis needed to be done for hard real-time applications. To accomplish compile-time scheduling, Atom enforces several programming restrictions (e.g., all variables declared at compile time, no looping constructs are provided, etc.). In addition, by employing guarded atomic actions, Atom enables highly concurrent programming without the need for mutex locking. Since applications implemented in Atom do not require mutex locking and run-time task scheduling (services traditionally provided by real-time operating systems), Atom eliminates the need and overhead of real-time operating systems for many embedded applications.

In this study, it is possible to use compile-time scheduling to provide cooperative multitasking (e.g., while breaking tasks into subtasks, we can break a while loop into small parts so that each piece would not take more than a certain amount of time to execute). However, due to the fact that the compiler resides at the lowest level of the programming stack, implementing scheduling techniques at this level is very complex. In addition, since it provides precise timing guarantees, compile-time scheduling is more beneficial for hard real-time systems rather than soft real-time tasks. In this study, we target soft real-time applications; hence, the benefits of using compiler-level scheduling techniques would not make up for their complexity.

### *2.3. Operating system and driver support for scheduling*

Since the only Operating System (OS)-level abstraction available in the current GPUs is the I/O control system call, the OS cannot schedule or manage the GPU. Instead, the GPU driver performs these tasks. The OS and the GPU driver work together to allow different applications to run at the same time on a single GPU. This model allows us to context-switch and share the GPU resources among multiple graphics or compute applications, or combinations of these two. However, the traditional GPU command scheduling model has weaknesses preventing us from performing an efficient multitasking among these applications. Two of these weaknesses include: (a) GPU computation and video memory resources are allocated on a first-come, first-serve basis without regards to any notion of fairness or priority. This may lead to starvation of some applications, which causes bursts (e.g., large variation in the frame rate). (b) Active processes lock access to the GPU and can issue as many GPU commands as they want, which may result in monopolization of resources.

OS support—like the Windows Display Driver Model (WDDM) [33] (introduced on Windows Vista) or the Direct Rendering Infrastructure (DRI) [34] (for X Windows systems)—tries to solve the problem of fairly allocating GPU resources, but cannot prevent applications from monopolizing resources. Due to the coarse granularity of GPU command scheduling and the lack of support for preventing monopolization of resources, we cannot use these frameworks for multitasking among the real-time tasks that we target in this study. By implementing more granular GPU command scheduling and eliminating hardware locks, GERM [35] provides more fair and efficient GPU resource allocation. When scheduling GPU commands among competing applications, GERM relies on the estimated GPU time/resource requirement of these commands. However, since this estimation technique is based on the number of bytes and the number of vertices of GPU commands, it is not applicable to the commands of compute tasks we target; hence, we cannot use this technique in our study. Finally, the context switch overhead involved in all these systems makes them unattractive for real-time tasks.



DOI: 10.1002/cpe

Kato et al. propose a real-time GPU scheduler called TimeGraph [36], which aims for GPU multitasking support similar to GERM. While fairness is a primary concern for GERM, TimeGraph focuses on prioritization and isolation among competing GPU applications. Focusing on prioritization and isolation protects important GPU workloads from the performance interference in the face of extreme workloads. To provide prioritization, it supports two scheduling policies, addressing the trade-off between response times and throughput. To provide isolation, it employs two resource reservation policies on the GPU, which allow different levels of quality of service at the expense of different levels of overhead. While GERM is spread across the device driver and user-space library, TimeGraph falls inside the device driver. Hence, unlike GERM, TimeGraph would not require major modifications for different runtime frameworks, e.g., OpenGL, CUDA, etc.

The developers of GERM propose that as more and more applications need to use GPU resources concurrently, GPUs may need an OS to manage its resources, just as CPUs do. Once such a technology is enabled, the methods used in real-time operating systems (RTOSs) will be available to make the GPU meet requirements of running real-time tasks. RTOSs propose advanced algorithms for scheduling so that the system can quickly and predictably respond by using cooperative or preemptive techniques.

If the critical section (the piece of code that has access to a shared resource) is short, RTOSs may allow programs to temporarily disable interrupts and run in kernel mode to prevent race conditions, instead of using semaphores or message passing schemes with high overheads. To further reduce the latency, RTOSs typically make the memory allocation as fast as possible by using techniques like the fixed-size-blocks algorithm and keeping interrupt handlers as short as possible.

Rossbach and Witchel [37] also indicate that managing GPU resources with the OS is desirable since user-space runtimes such as CUDA can support but cannot provide traditional OS guarantees such as fairness and isolation. They say that having OS support is especially important for the performance of interactive applications needing low latency and concurrency. Rossbach and Witchel add that running OS code on current GPU architectures is not possible since they have different instruction set architectures than CPUs and lack important features for supporting the OS such as interrupts. However, it is still possible to allow the OS to manage GPU resources by introducing different OS-level abstractions and interfaces for the GPUs, which integrate with existing user-space runtimes. Rossbach and Witchel show that these abstractions eliminate unnecessary data migration in the system and improve performance of interactive applications that utilize a GPU. Their PTask API [38] aggregates these abstractions and supports GPUs and other accelerator devices as first class computing resources. It promotes GPUs to a general-purpose, shared compute resource managed by the OS, which can then provide fairness and isolation.

Similar to PTask, Kato et al. propose Gdev [39], which allows the user space as well as the OS itself to use GPUs as first-class computing resources. Specifically, Gdev provides shared device memory functionality that allows GPU contexts to communicate with other contexts and enables GPU contexts to allocate memory exceeding the physical size of device memory. It also allows virtualizing the GPU into multiple logical GPUs to enhance isolation among working sets of multitasking systems. Gdev and PTask are both API-driven, but PTask exposes the input/output control system call to user-space programs, which could allow misbehaving tasks to abuse GPU resources. Hence, Kato et al. advocate that Gdev, which integrates runtime support into the OS, is a more reliable solution.

In a recent study, Wang et al. [40] propose Simultaneous Multikernel (SMK), a fine-grained dynamic sharing mechanism, that utilizes resources within a streaming multiprocessor by exploiting heterogeneity of different kernels. They evaluate their design using Parboil [41] benchmarks and GPGPU-Sim [42], a GPU simulator that they modify to support running multiple kernels on the same streaming multiprocessor. The fundamental principle of SMK is to co-execute kernels with compensating resource usage in the same streaming multiprocessor to achieve high utilization and efficiency. They propose several resource allocation strategies to improve system throughput while maintaining fairness.



DOI: 10.1002/cpe

### 2.4. Shortcomings of prior research

A handful of prior studies target GPU workloads with real-time tasks [12, 13, 36, 38]. However, these studies either do not provide plausible scenarios or perform multitasking among collaborative tasks. For instance, Elliott and Anderson [12] and Kato et al. [36] conduct their experiments on randomly generated task sets and OpenGL graphics benchmarks, respectively; therefore, they do not provide plausible scenarios. Although some other related studies provide examples from real-world scenarios, they usually perform multitasking among collaborative tasks. These tasks are part of one big application; hence, they usually have a common goal and a consumer/producer relationship. For instance, Steinberger et al. [13] show that their method accelerates selected computer graphics techniques, e.g., view-dependent mesh simplification. Likewise, Rossbach et al. [38] enable interactive applications such as a gestural interface.

Hence, by designing schedulers that take into account the goal of supporting multitasking among non-collaborative real-time tasks in the real-world, we allow scheduling of automotive computing tasks or other similar real-time embedded tasks (e.g., mobile or robotics tasks). To present a plausible scenario, we provide the results of our study that examines multitasking among tasks used for various automotive applications such as pedestrian detection, road sign recognition, etc. as described in Muyan-Özçelik and Owens [2].

Although previous research on multitasking among GPU tasks used different methods (e.g., GPU scheduling, CPU scheduling, persistent uber kernels, etc.), the comparison of these techniques to each other is an under-researched area. In our study, we take a comprehensive look at a variety of possible schedulers by comparing different scheduling strategies to each other.

To the best to our knowledge, there is no prior study utilizing the new features of GPU architectures, i.e., concurrent kernels and dynamic parallelism, that can be useful for multitasking among GPU tasks. Hence, one of the contributions of this study is that we investigate the use of both concurrent kernels and dynamic parallelism features for multitasking among real-time embedded tasks.

Understanding the needs of the two different types of real-time tasks, i.e., latency-oriented and throughput-oriented, and scheduling them in parallel, is a challenge. To prioritize tasks on the GPU, Kato et al. [36] provide two scheduling policies that address the trade-off between response times and throughput at the device-driver level. Our study contributes to the literature by addressing this challenge at the application level.

Focusing on soft-real-time tasks allows us to generate schedulers at the application level instead of in lower levels such as the driver or OS. Although working on driver/OS levels provides better timing guarantees, implementing scheduling techniques at these lower levels of programming stack is very complex. Since precise timing guarantees are more beneficial for hard-real-time tasks than the soft-real-time tasks that we target in this study, the benefits of using lower-level scheduling techniques would not make up for their complexity.

The details of the hardware queue structures of the Fermi and Kepler architectures are not very well documented by NVIDIA. In addition, to the best to our knowledge, no prior studies shed light on this area. Hence, another contribution of this study is that it provides a comparison of the hardware queue structures of Fermi and Kepler GPUs.

Finally, a limited number of studies have been conducted on the recently introduced NVIDIA Jetson TX1 embedded architecture and we are not aware of any prior work that has investigated the behaviours of CUDA events and hardware queues, which are provided by the supported Hyper-Q feature, on this architecture. Hence, this paper also contributes to the literature by providing insights onto this understudied area.

## 3. TERMINOLOGY AND DEVELOPMENT ENVIRONMENT

We define a *task* as the following series of operations necessary for performing certain piece of work: a host-to-device copy, a device kernel execution, and a device-to-host copy. We run a task for each instance of *data* that arrives as an input. For instance, we run the gradient task for




DOI: 10.1002/cpe

each frame of the streaming video while performing pedestrian detection. Tasks have different properties, including input data size (which determines the copy time), data arrival interval, real-time requirement type (i.e., latency or throughput), time requirement (i.e., latency tasks specify a deadline and throughput tasks specify a separation time), and dependency information. Tasks can have dependency relationships; a group of tasks that the scheduler runs together at a given time is called a *batch*. Schedulers usually run tasks in a batch together by interleaving/overlapping their copy/execute task operations. A *task operation* consists of a CUDA command involving either a data transfer or a kernel execution command preceded by synchronization and/or timing commands and followed by a timing command.

GPU workloads that include multiple real-time tasks have several salient characteristics that are considered in the design of schedulers. These *workload characteristics* are determined by the properties of individual tasks and the interplay between properties of different tasks constituting the workload. Hence, we can broadly categorize the workloads depending on the important properties of their tasks:

The copy and execution time of tasks determines a "*balanced/unbalanced copy-execute time*" categorization: Schedulers can interleave data transfers of one task with kernel runs of another task in a given batch. Hence, if the copy and execution times of tasks are close to each other, i.e., we have balanced copy-execution time, the overlap of copy and execute operations would be maximized and we will achieve a good performance improvement from interleaving tasks.

The kernel size of tasks determines the "*small/large kernel*" categorization: Kernels that are scheduled to run concurrently can execute at the same time only if there are enough resources. When small kernels are used, since they use a minimum amount of resources, they can run at the same time and we could obtain an optimal execution overlap. On the other hand, when large kernels are used, they could not truly run concurrently since execution overlap would only occur for a small period of time when the higher-priority kernel finishes and frees resources allowing the lower-priority kernel to start.

The time requirements of the task determine the "*latency/throughput oriented*" categorization: the ratio of latency and throughput tasks in the workload and the duration of their time requirements affect system performance. For instance, if the workload has many latency-oriented tasks with short deadlines, meeting time requirements would be harder.

The ratio of arithmetic operations to memory operations in task kernels determines the "*compute/memory-bound*" categorization: Naturally, workloads including a balanced combination of simultaneously running compute-bound and memory-bound tasks would lead to optimal scheduler performance.

There are quantifiable ways to measure the abovementioned task properties that determine the workload characteristics. For instance, kernel sizes of tasks can be measured by counting the number of instructions. Hence, we can determine whether a workload has small/large kernels by allowing the programmer to manually count the instructions and annotate the kernels. Alternatively, we can determine this characteristic using an explicit profiling step or a compile-time decision that can be changed by runtime statistics. Similar techniques can be used for determining whether workload has balanced/unbalanced copy-execute times.

In this study, we use NVIDIA's Fermi- and Kepler-based discrete GPUs (i.e., GeForce GT 555M and Tesla K20c, respectively), and Jetson TX1 embedded architecture along with the CUDA programming framework [26]. Working on Fermi and Kepler architectures provides important insights into the GPUs widely used today in embedded systems. It is because they introduce key features that support multitasking among several tasks (i.e., "concurrent kernel execution", "dynamic parallelism", and "Hyper-Q") and these features have begun to appear in current embedded systems such as NVIDIA Jetson TX1, a system-on-chip architecture which we also investigate in this study. In addition to schedulers utilizing these fairly new features, we have also developed schedulers that do not utilize them. Hence, this study provides schedulers that can also run on embedded GPUs that do not support concurrent kernel execution, dynamic parallelism, and Hyper-Q.



DOI: 10.1002/cpe

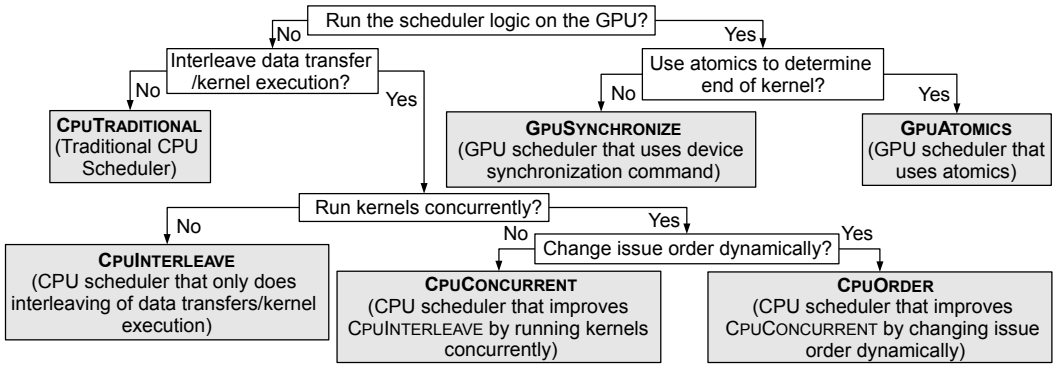



Figure 1. Tree that shows taxonomy of schedulers. White and gray boxes refer to strategy and scheduler nodes, respectively. Each scheduler uses an alternative approach that is a combination of all strategies utilized on the path from the leaf scheduler node up to the root node.

## 4. DESIGN OF SCHEDULERS

We explore the design space of schedulers performing multitasking among real-time GPU tasks by considering the salient characteristics of their GPU workloads. Based on the extensive survey we have provided in Section 2, we consider both the methods based on prior research and the ones we develop using the new GPU features, which we summarize as a taxonomy in Figure 1.

As a result of this taxonomy, we design six different schedulers: CPUTRADITIONAL, CPUINTERLEAVE, CPUCONCURRENT, CPUORDER, GPUSYNCHRONIZE, and GPUATOMICS. The first four schedulers run the scheduler logic on the CPU, whereas the last two schedulers run it on the GPU. CPUTRADITIONAL is a traditional scheduler performing one task at a time. It neither overlaps data transfer/kernel execution, nor runs kernels concurrently. CPUINTERLEAVE interleaves data transfers and kernel executions of tasks, but it does not run kernels concurrently. CPUCONCURRENT improves upon CPUINTERLEAVE and in addition to overlapping data transfers with kernel executions, also runs kernels concurrently. Likewise, CPUORDER improves upon CPUCONCURRENT and in addition to interleaving data transfers/kernel executions and running kernels concurrently, it also dynamically changes the issue order of device-to-host copy commands of tasks in a batch (i.e., to overlap the device-to-host copy of task which finishes its kernel execution first with the execution of the other task's kernel in a batch, it issues the device-to-host copy operation of the former task before the latter task).

To the best of our knowledge, no prior study utilizes concurrent kernels and dynamic parallelism for multitasking purposes. Hence, the schedulers that use concurrent kernels (CPUCONCURRENT and CPUORDER) and dynamic parallelism (GPUSYNCHRONIZE and GPUATOMICS) are contributions of this work.

Since GPU schedulers use the dynamic parallelism feature only available on the Kepler architecture, they cannot run on Fermi. Also, since they run the scheduling logic on the GPU, they inherently interleave data transfers/kernel execution, run kernels concurrently, and change the issue order dynamically. GPUSYNCHRONIZE uses the device-side device synchronize command to determine the end of kernels. However, this command has unstable performance since it is not guaranteed to wait only for the completion of work launched by synchronizing-thread's block. To provide stable performance, GPUATOMICS uses atomics instead of the device synchronize command to determine the end of kernels.

All schedulers have a while-loop structure. In this structure, the scheduler receives newly arrived data and processes these data in batches. When all data is received and processed, the scheduler is done, and it exits its loop. To pick the tasks to process in the next batch, the scheduler compares the priorities of all available tasks and selects the ones with the highest priority.



DOI: 10.1002/cpe

Our scheduler design addresses the research challenges mentioned in Section 1 in the following ways: (1) Since the GPU features that support running multiple GPU tasks (e.g., asynchronous memory copy, streams, concurrent kernel execution) are not primarily designed for multitasking among real-time tasks, these features present several limitations. We overcome these limitations by developing various methods that are derived from the abovementioned features (Section 5). (2) Understanding the needs of latency- and throughput-oriented tasks, and scheduling them in parallel, is a challenge. To address this challenge, based on the real-time requirements, which are specified differently for latency- and throughput-oriented tasks, we use different formulas for assigning priorities of these two separate types of real-time tasks. All schedulers use these priority values to sort the tasks and determine which tasks to process next. We also use different formulas for calculating scheduler performance on processing latency- and throughput-oriented tasks since these formulas are also based on differently specified real-time requirements of these separate types of tasks. Details of priority and performance calculations are provided in Muyan-Özçelik and Owens [2]. (3) GPUs lack some characteristics that are typical of real-time systems. Our scheduler design addresses these challenges in the following ways: (a) Since there is no common time concept between CPU-GPU, we measure the time on the host and device differently using events and the `clock()` function, respectively, and then perform a software synchronization. (b) Since the GPU cannot assign hardware-supported priorities to enforce the calculated task priorities, we use a certain issue order to do so (i.e., we issue the host-to-device copy and kernel execution commands of higher-priority tasks before the lower-priority ones to give the former the precedence for utilizing resources). (c) Since the GPU does not have a preemption mechanism, to reduce the response time of the system, we schedule at most two tasks at a time. This also allows us to better analyze overlap behavior of the schedulers: it is easier to track whether operations of the two tasks are overlapped and to develop techniques that allow more opportunities for overlap. However, if needed, techniques introduced in this paper can be extended to schedule more tasks at a time. (d) Since there is a slow interface between CPU-GPU, to hide the communication cost, we overlap the data transfers of one task with kernel executions of another task.

## 5. METHODS

Each scheduler uses an alternative approach that is a combination of different strategies. We develop several methods that serve as building blocks for constructing these strategies and resolve technical difficulties that arise while implementing them. One of the important contributions of this paper is that in this section, we provide in-depth descriptions of our selected methods. Muyan-Özçelik [43] provides a complete list of methods and implementation details of our schedulers. Most of the methods we present in this study are either new techniques proposed in this work or combine existing techniques in novel ways.

Although they are not specifically designed for multitasking among real-time tasks, current GPU architectures provide several software and hardware features that can be used as a basis for constructing methods that allow this multitasking. These features are related to: (1) overlapping multiple tasks (interleaving data transfers with kernel executions and concurrently running different kernels) such as asynchronous memory copy, stream, concurrent kernel execution, hardware queue, Hyper-Q, and atomic; (2) timing individual task operations such as the `clock()` device function and event; and (3) running the scheduler logic on the GPU using features such as persistent kernel, zero-copy memory, and dynamic parallelism. The CUDA documentation [26] provides details of all the abovementioned features that we use to construct methods that implement different scheduling strategies.

Some methods are used as building blocks for schedulers. For instance, to run scheduling logic on the GPU, we use methods that (1) allow communication between the CPU and GPU utilizing zero-copy memory (communication is needed to transfer newly arrived data to the GPU and to copy produced results to the CPU) and (2) fuse a kernel that makes scheduling decisions with a kernel that launches tasks into a persistent kernel (threads of the kernel that launch tasks use dynamic parallelism).



DOI: 10.1002/cpe

Beyond serving as building blocks, these methods also resolve technical difficulties that arise while implementing our schedulers. The important problems we have identified include false dependencies that occur due to the limited amount of hardware queues when events or other commands are used; not seeing the changes the device makes on the zero-copy memory from the host without calling the host-side device synchronization; and the difficulties of determining the end of kernels running concurrently.

### *5.1. Methods that resolve false dependencies*

Several methods resolve technical problems that arise due to false dependencies caused by the events that are used to measure copy and execute times of tasks. Muyan-Özçelik and Owens [2] describe in detail situations that lead to false dependencies. Several different technical problems stem from the false dependencies generated by events and we use different methods to resolve them. Due to space limitations, in this paper, we provide details for only one of these problems and its resolution.

**Problem:** Consider an example showing how events cause false dependencies. Consider that we have two tasks, t1 and t2, on two different streams, s1 and s2, on a Fermi architecture. For simplicity, assume that performing t1 consists of host-to-device copy and kernel execution operations. These operations include copy/execute commands preceded and followed by timing commands that are recordings of start and stop events, respectively. For instance, the host to device copy (HDcpy) operation of t1 consists of recording a start event on s1 (s1_start_HDcpy_t1), performing host to device copy on s1 (s1_HDcpy_t1), and recording a stop event on s1 (s1_stop_HDcpy_t1). Likewise, the kernel execution (Kexe) operation of t1 consists of performing the kernel execution on s1 (s1_Kexe_t1), preceded and followed by timing commands. In addition, for simplicity, assume that performing t2 only consists of a host-to-device copy and does not include timing commands. Commands issued to s1 and s2 for performing t1 and t2 and their issue order are provided in Table I. According to this issue order, the way these commands are issued to queues, along with the type of dependencies they have, are illustrated in Figure 2.a. There are expected dependencies between the commands issued to the same stream due to the fact that commands in a stream should be executed in order. In addition, there also exists a false dependency, which will be explained below. Figure 2.a shows that the start of kernel execution of t1 (s1_start_Kexe_t1) is placed in the copy queue instead of the execution queue since we use a Fermi architecture and the previous command on the same stream is placed in a copy queue. This is because in the Fermi architecture, events are by default placed in the same queue as the previous command in the same stream. The previous command, which is the stop of host-to-device copy of t1, is placed in the copy queue since its previous command, host-to-device copy of t1, is a copy command.

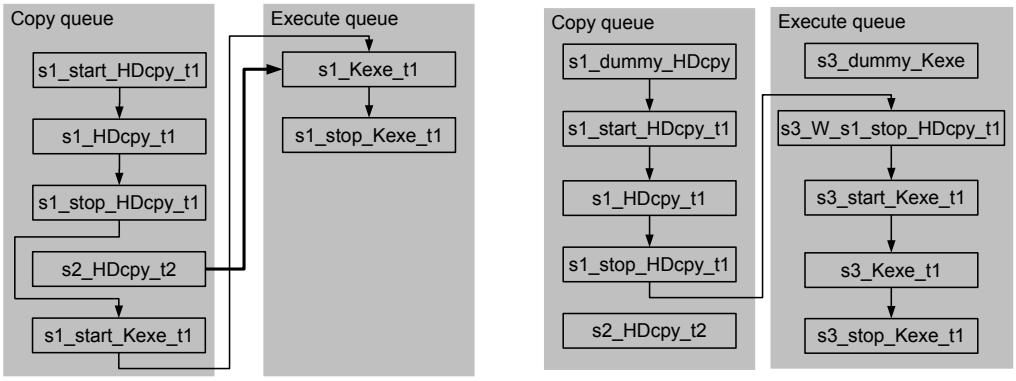
| t1 commands on s1 | t2 commands on s2 | **Issue order of all commands** |
|---|---|---|
| s1_start_HDcpy_t1 | s2_HDcpy_t2 | s1_start_HDcpy_t1 |
| s1_HDcpy_t1 | | s1_HDcpy_t1 |
| s1_stop_HDcpy_t1 | | s1_stop_HDcpy_t1 |
| s1_start_Kexe_t1 | | s2_HDcpy_t2 |
| s1_Kexe_t1 | | s1_start_Kexe_t1 |
| s1_stop_Kexe_t1 | | s1_Kexe_t1 |
| | | s1_stop_TKexe_t1 |

Table I. Commands for performing t1 and t2 and their issue order.

Since the host-to-device copy of t2 (s2_HDcpy_t2) and the kernel execution of t1 are on different streams, we expect them to overlap. However, since the start of kernel execution of t1 is placed in the copy queue after the host-to-device copy of t2, the overlap between these two commands is broken. This is because the start of kernel execution of t1 is issued before kernel execution of t1 on the same stream that the start of kernel execution of t1 should be recorded before kernel of t1 can start executing. But since the host-to-device copy of t2 is placed before the start of kernel execution of t1 on the same queue, the host-to-device copy of t2 blocks the start of t1's kernel execution, and we



DOI: 10.1002/cpe




a) the set-event-queue method is not used (a false dependency exists)

b) the set-event-queue method is used (no false dependency exists)

Figure 2. Distribution of commands given in Table I and Table II to queues are given in sub-figures a) and b), respectively. Fine lines indicate expected dependencies, whereas the bold line indicates the false dependency.

cannot record the start of t1's kernel execution until the host-to-device copy of t2 is completed. Thus, the false dependency would occur between the host-to-device copy of t2 and the kernel execution of t1, and they cannot overlap.

**Solution:** If the start of t1's kernel execution was recorded on the execution queue instead of the copy queue, it would not be blocked by the host-to-device copy of t2; hence, the overlap between the host-to-device copy of t2 and the kernel execution of t1 would have been achieved. Also, semantically, the start of kernel execution of t1 is related to kernel execution of t1; hence, it makes more sense for it to be placed in the execution queue instead of in the copy queue.

Hence, to resolve the false dependencies generated by events we develop a new method called *set-event-queue* that provides semantically coherent placement of events into queues by enforcing copy-related events to be placed in the copy queue and execution-related events to be placed in the execution queue. It enforces these requirements by performing the following: (a) dedicating different streams to each operation of the task, and (b) initializing these streams by running the relevant dummy command on them. The dummy commands make sure that events would be placed in their relevant hardware queues. On the Fermi and Kepler architectures, events would be placed in the same queue as the previous command in the same stream. Running the relevant dummy command on a stream means that we initialize a copy stream with a dummy copy command and a execution stream with a dummy kernel run. This way, since a copy stream is initialized by a dummy copy command and since after this initialization only copy commands are issued to this stream, all events issued to a copy stream would be placed in the copy queue. Likewise, all events issued to an execute stream would be placed in the execute queue.

If we consider the example provided above, for the first task, t1, the set-event-queue method uses two streams, s1 and s3, for host-to-device copy and kernel execution operations, respectively. t1's kernel execution, executed on s3 (s3_Kexe_t1), should start after t1's host-to-device copy, executed on s1 (s1_HDcpy_t1) is completed; this is because they belong to the same task. This serialization is achieved by the synchronization command making s3 wait until the stop of the host-to-device copy, issued on s1, is recorded (s3_W_s1_stop_HDcpy_t1). Hence, in addition to execution and timing commands, t1's kernel execution operation includes a synchronization command. Since s1 is a copy stream, it is initialized with a dummy host-to-device copy (s1_dummy_HDcpy). Likewise, s3 is initialized with the dummy kernel execution (s3_dummy_Kexe). Since performing the second task, t2, only includes one task operation, i.e., the host-to-device copy, the set-event-queue method still uses one stream, s2, for t2. We show the commands issued to s1, s2, and s3 for performing t1 and t2 with their issue order in Table II, and how these commands are issued to queues along with the type of dependencies they have in Figure 2.b. This figure shows that the false dependency occurring in Figure 2.a does not occur when the set-event-queue method is used.



DOI: 10.1002/cpe

| t1 commands on s1 and s3 | t2 commands on s2 | **Issue order of all commands** |
|---|---|---|
| s1 | s2 | |
| s1_dummy_HDcpy | s2_HDcpy_t2 | s1_dummy_HDcpy |
| s1_start_HDcpy_t1 | | s3_dummy_Kexe |
| s1_HDcpy_t1 | | s1_start_HDcpy_t1 |
| s1_stop_HDcpy_t1 | | s1_HDcpy_t1 |
| s3 | | s1_stop_HDcpy_t1 |
| s3_dummy_Kexe | | s2_HDcpy_t2 |
| s3_W_s1_stop_HDcpy_t1 | | s3_W_s1_stop_HDcpy_t1 |
| s3_start_Kexe_t1 | | s3_start_Kexe_t1 |
| s3_Kexe_t1 | | s3_Kexe_t1 |
| s3_stop_Kexe_t1 | | s3_stop_Kexe_t1 |

Table II. Commands for performing t1 and t2 and their issue order when the set-event-queue method is used.

### *5.2. Methods that flush zero-copy*

When implementing certain scheduling techniques, it may be necessary to see the changes the device makes on the zero-copy memory (the host memory directly accessible from GPU kernels) from the host. For instance, GPU schedulers may pass scheduling messages via zero-copy memory between the host and device. To allow the host to see the changes, we declare this memory as a volatile memory. A volatile variable located in global or shared memory is assumed to have a value that can be changed or used at any time by another thread, and therefore, any reference to this variable compiles to an actual memory read or write instruction, i.e., the compiler does not perform any optimizations and memory fence or synchronization functions are not required to guarantee that prior writes to the memory are visible by other threads.

**Problem:** Although on the Linux operating system declaring a zero-copy memory as a volatile memory was sufficient for the host to see the changes that the device makes on a zero-copy memory, our findings on the Windows operating system show that the host cannot see the changes unless it calls the device synchronization command (cudaDeviceSynchronize()) that blocks the host until the device completes all preceding commands.

However, we cannot use this command in CPU schedulers to flush the memory since it hurts the overlap of task operations, i.e., interleaving of the kernel execution and the device-to-host copy. We also cannot use it in GPU schedulers since it causes a deadlock. To progress, the persistent kernel used by the GPU schedulers needs the host to pass it some information. The host provides this information according to the changes the device makes on the zero-copy memory. However, if we call the device synchronization command after the persistent kernel is launched, in order to see the changes the host would need to wait until the persistent kernel is done; meanwhile, the persistent kernel would wait for the information that the host needs to provide. Hence, a deadlock would occur.

**Solution:** To see the changes that the device makes on the zero-copy memory from the host without using a synchronization command, we develop a novel method called *flush-zero*. The flush-zero method uses stream query command (`cudaStreamQuery()`) on any of the streams executing a kernel overlapping computation with the kernel updating the zero-copy memory. For instance, in the GPU scheduler, we call stream query on the stream that its persistent kernel runs on. We do not need to call a stream query command each time we need to read the memory; querying the stream only once before all the reads is sufficient. Hence, in the GPU schedulers, we call this query right after the persistent kernel is launched and before the host starts polling on the zero-copy memory that includes scheduling messages from the device.

### *5.3. Methods that determine the end of kernels*

We also implement methods that resolve the difficulties of determining the end of kernels running concurrently, which would allow for better overlap opportunities. Once we detect that the kernel execution of one task is finished, we can immediately start its device-to-host data transfer; hence,



DOI: 10.1002/cpe

we allow this transfer to overlap with the execution of the other kernel that has been running concurrently.

**Problem:** We can determine whether the kernel is finished by running a stream query command (`cudaStreamQuery()`) on the stream where the kernel executes. However, while kernels are running concurrently, if we call a stream query command on the stream of one of these kernels, this query may block the concurrent kernel streams until the longest-running kernel finishes. This is because finish signals of these kernels would be extended until the longest running kernel is finished. We call this difficulty the "blocking-query" problem. When the finish signal of the kernel is delayed, the execution of the upcoming commands on the stream of this kernel is delayed as well.

Assume the first-launched kernel finishes later than the second-launched kernel and the stream query of the first kernel's stream is issued before that of the second one. In this case, a query on the stream of the first kernel causes the blocking-query problem, and we cannot detect that the second kernel finishes first. We can avoid this problem by issuing a query on the stream of the shorter kernel before that of the longer one. However, since our aim is to determine which kernel finishes first, we do not know which kernel has the shorter runtime; hence, we cannot avoid this problem. Thus, a CPU scheduler that runs kernels concurrently and wishes to change the issue order (i.e., CPUORDER) cannot utilize the stream query command to determine the end of kernels. The stream query, and similar commands that can help to determine whether a kernel is finished, e.g., the event query command (`cudaEventQuery()`), are not supported on the device. Hence, GPU schedulers also cannot utilize the current API to determine the end of kernels.

**Solution:** Hence, instead of using stream query, CPUORDER determines the end of kernels using a method called *atmCnt-kernel* that utilizes atomic operations. We utilize the same method in GPUATOMICS to solve the unstable performance problem caused by the device-side device synchronize command while determining the end of kernels. The atmCnt-kernel method uses atomic-counter versions of kernels instead of regular kernels. In these versions of kernels, an atomic counter is increased when the thread is done. When this counter is equal to the number of threads that are launched, it means that the kernel is done.

Since it is an expensive operation, the addition of atomic operations introduces an overhead for the runtime of the kernels. This overhead is significant for kernels that have large number of thread (i.e., large kernels). This is because the overhead is directly proportional to the number of times atomic operations is used and as mentioned above, we perform one atomic operation per thread. Our experiments show that the use of atomic operations increases the runtime of kernels with 7.6 and 0.9 nanoseconds per thread on the Fermi and Kepler architectures, respectively. The increase is much less in Kepler due to the fact that atomics are implemented much more efficiently in the Kepler architecture [5].

In the CPU schedulers, the host reads the atomic counter via zero-copy memory to determine the end of kernels. Hence, the CPU schedulers use the atmCnt-kernel method in combination with the flush-zero method. The flush-zero method requires us to call a stream query command on any of the streams executing a kernel overlapping computation with the kernel updating the zero-copy memory. In our CPUORDER implementation, this would mean that we can call the query on any of the streams that concurrent kernels execute on. However, while kernels are running concurrently, if we call a stream query command on one of the kernel's streams, the block-query problem would occur as explained above. When the kernel finish signal is delayed, performing the upcoming commands (i.e., the stop event of kernel execution) on this kernel's stream would be delayed too.

Assume that when we call the stream query command required by the flush-memory method, we pick the stream of the first-launched kernel and this kernel happens to take longer to finish. Even if we can detect that the second-launched kernel is finished earlier by checking its counter, recording the stop event of the second-launched kernel would be delayed until the first-launched kernel is done due to the blocking-query problem.

To fix this problem, we develop another novel method called *atmCnt-CPUsched*. Instead of recording on the kernel execution stream, the atmCnt-CPUsched method records the stop event of the shorter kernel on the device-to-host copy stream of the related task that is not affected by the blocking-query problem. Thus, since we can record the stop event of the kernel execution of the task



DOI: 10.1002/cpe

possessing the shorter kernel immediately after we detect that the kernel execution of this task is finished, we can start its device-to-host copy without waiting for the longer kernel to finish. Hence, we would allow overlap between kernel execution of the task possessing the longer kernel with the device-to-host copy of the task possessing the shorter kernel.

## 6. EXPERIMENTS AND RESULTS

We first evaluate and compare our schedulers by conducting experiments on synthetic cases. A synthetic case consists of a workload that involves artificial tasks and a GPU architecture that schedulers use to process this workload. From these experiments, we conclude which strategy is more effective for a given workload and why.

Scheduler performance is affected the most by the workload characteristics that highlight the differences in the strategies used. "Balanced/unbalanced copy-execute time" and "small/large kernel" categorizations focus on the differences in interleaving copy/execution and running-kernels-concurrently strategies, respectively. On the other hand, "latency/throughput oriented" and "compute/memory bound" categorizations do not highlight any salient differences due to the fact that the schedulers use the same priority calculations and the same hardware, respectively. Hence, the synthetic cases we describe here focus on the former two categorizations.

Scheduler performance also depends on the target GPU architecture, here Fermi or Kepler. Kepler introduces dynamic parallelism and Hyper-Q features lacking in Fermi. Dynamic parallelism coupled with persistent kernel and zero-copy features allow us to run the scheduling logic on the GPU and launch kernels directly from the device. On the other hand, since the Hyper-Q feature drastically increases the number of available hardware queues, it minimizes the occurrence of false dependencies. In addition, since it allows each execution stream to have its own execution queue and prevents all copy streams from being placed in the same queue, it eliminates the need for a strategy that changes the issue order dynamically. Also, Hyper-Q improves the performance of the approach that runs kernels concurrently by preventing the occurrence of "delayed-signal" phenomena [44] (a condition that delays device-to-host copies until all concurrent kernels are done and prevents interleaving of kernel executions with device-to-host copies), which occurs when a single execution queue and large kernels are used.

If a workload consists of small kernels, schedulers that run the kernels concurrently can truly run them at the same time. In this case, if the higher-priority task has a longer kernel runtime than the lower-priority task, the later-launched, lower-priority kernel would finish first, creating a special condition that affects scheduler performance. Our synthetic cases also investigate the effects of this special condition.

We provide the list of synthetic cases and the performance of the schedulers for these cases in Table III. We calculate the performance values using the techniques explained in Muyan-Özçelik and Owens [2]. As described in our prior work, scheduler performance is a function of a value that is called "shifted accumulated slack time" which are indicated within the parentheses in Table III. To compare the schedulers in a standard way, we scale scheduler performance using a formula which takes the ratio between the "shifted accumulated slack time" of CPUTRADITIONAL and a given scheduler.

In Table III, the comparison of performance values are only meaningful within a case and should not be compared across cases. This is because, as mentioned above, in each case, the "shifted accumulated slack time" of CPUTRADITIONAL is taken as a base (i.e., the performance of CPUTRADITIONAL is always assigned to 1) to calculate performance of other schedulers and this value changes across cases as can be seen from Table III. Hence, for instance, if one case has a larger performance value than another case for a particular scheduler, it does not mean that the scheduler performs better in the former case. Within each case, schedulers that perform better than CPUTRADITIONAL would have a performance value greater than 1, e.g., if the scaled performance of the scheduler is 2, it means that the performance of the scheduler is two times better than that of CPUTRADITIONAL.



DOI: 10.1002/cpe

The dash in Table III indicates that the scheduler is not applicable to the given case. Since Fermi does not support dynamic parallelism, GPU schedulers (i.e., GPUSYNCHRONIZE and GPUATOMICS) cannot be run on this architecture. Hence, the table lists dashes for these schedulers in Case-2, Case-5, and Case-6, all of which include workloads running on Fermi.

| **Schedulers** | **Case-1: small kernels, balanced operations on Kepler, higher-priority tasks have longer kernel runtimes** | **Case-2: Case-1 work-load on Fermi** | **Case-3: same as Case-2, except higher-priority tasks have shorter kernel runtimes** | **Case-4: similar to Case-1, but with large kernels** | **Case-5: similar work-load to Case-4 on Fermi** | **Case-6: similar to Case-5, but with unbalanced operations** |
|---|---|---|---|---|---|---|
| CPUTRADITIONAL | 1.00 (122.99) | 1.00 (241.84) | 1.00 (233.17) | 1.00 (122.08) | 1.00 (187.74) | 1.00 (271.01) |
| CPUINTERLEAVE | 1.93 (63.54) | 2.83 (85.41) | 3.34 (69.76) | **1.75** (69.65) | **2.59** (72.57) | **1.13** (239.78) |
| CPUCONCURRENT | **3.55** (34.61) | 3.55 (68.19) | **4.37** (53.38) | **1.78** (68.60) | 1.54 (122.04) | 0.82 (329.30) |
| CPUORDER | **3.53** (34.81) | **4.26** (56.71) | **4.32** (53.92) | 0.66 (184.78) | 0.11 (1727.99) | 0.29 (944.57) |
| GPUSYNCHRONIZE | 3.14 (39.14) | — — | — — | 1.70 (71.97) | — — | — — |
| GPUATOMICS | 3.21 (38.36) | — — | — — | 0.63 (195.11) | — — | — — |

Table III. The performance of the schedulers for all cases (best performance value for each case is indicated as **bold**). Values in parentheses indicate the "shifted accumulated slack time values" of schedulers (in milliseconds) which are used in their performance calculation.

## 7. RUNNING THE SCHEDULERS ON NVIDIA JETSON TX1

NVIDIA's Jetson TX1 is a system-on-chip architecture with a Maxwell-based GM20B GPU. TX1's architecture couples the CPU, GPU, and memory controller onto the same chip. Since the CPU and GPU use a common memory, programmers need not copy data and results back and forth between the host and device memory. Instead, they can hold the data in zero-copy memory and access it from both the host and the device. We initially believed that on TX1, given that data transfers are not necessary, overlapping data transfers with kernel executions and using the messaging system to receive data and send results would not be a valid methodology for designing schedulers. However, we have observed that the version of our system that uses zero-copy memory performs slower compared to the version that copies the data between the host and device. This observation has also been reported by other developers and NVIDIA has responded, stating that both CPU and GPU caches are bypassed for zero-copy memory. This bypass is the reason why systems that use zero-copy memory are slower than ones that do data transfers [45]. Our implementation on TX1, then, transfers data between host and device and uses the same methodologies in its design just like schedulers running on Fermi and Kepler.

Since Maxwell is a successor to Kepler, we expected the mechanism used for scheduler timings to behave similarly on both architectures. However, we have observed that this mechanism, which is based on CUDA events, behaves differently on these architectures. This difference mainly stems from the fact that TX1 and Kepler use different policies for placing events into the hardware queues.

On Kepler, events are by default placed in the same queue as the previous command in the same stream. In addition, since Kepler has Hyper-Q, each execution/copy stream has its own



DOI: 10.1002/cpe

execution/copy queue and thus events issued to different streams are placed in different queues and false dependencies are minimized.

Although TX1 also has Hyper-Q, we have observed that the abovementioned policies are not entirely followed on TX1, especially when CUDA commands cannot be issued consecutively from the host. If the host is blocked due to the use of device/event synchronization commands or due to polling on the zero-copy memory that is updated by the device, we would see time gaps between the issuing of CUDA commands (e.g., event recordings) leading to incorrect timings on TX1.

To address the differences from previous architectures, we develop the following new methods to report correct timings for CPU schedulers running on TX1:

(1) In all CPU schedulers, processing of different batches is separated by the device synchronization command, which causes incorrect timings on TX1. We solve this problem by recording a dummy event on the default stream (stream 0) before and after processing each batch.

(2) CPUORDER uses polling on zero-copy memory to determine the kernel of which task in the current batch finishes first, and then accordingly orders the issue of kernel stop events and device-to-host copies of tasks. However, as mentioned above, polling on zero-copy memory also causes incorrect timings on TX1. To solve this problem, we avoid polling on zero-copy memory, but now cannot determine the kernel of which task finishes first. Hence, as a part of this method, we also do not change the issue order of device-to-host copies. Although we do not change this issue order, since TX1 has Hyper-Q, it will first execute the device-to-host copy of the task that first finishes its kernel execution.

(3) Although TX1 has Hyper-Q, depending on the order of the kernel stop events issued before the device-to-host copies, a false dependency may occur on CPU schedulers running on TX1. For instance, if the higher-priority task has a longer kernel runtime than a lower-priority task and the stop event of the former is issued before that of the latter, a false dependency occurs on TX1 and we cannot overlap the higher-priority task's kernel execution with the lower-priority task's device-to-host copy. This false dependency also occurs on Fermi, which lacks Hyper-Q. On Fermi, we resolve this issue in CPUORDER by making the issue order of stop events follow the kernel execution finish order of the tasks. However, CPUORDER cannot resolve this false dependency on TX1 as it can on Fermi since CPUORDER does not determine the kernel of which task finishes first in this architecture as explained above. Thus, because of the unresolved false dependency, CPUORDER timings would be reported longer than they should be on TX1 when higher-priority tasks have longer kernel runtimes. To solve this problem, we avoid issuing kernel stop events. Since we do not record kernel stop events of tasks, we can not synchronize between kernel streams and device-to-host copy streams of tasks. Thus, as a part of this method, we also perform device-to-host copies of tasks on their kernel streams.

To compare their initial performance, we run our CPU schedulers on the synthetic case, Case-1. We see that P[CPUTRADITIONAL] = 1 $<$ P[CPUINTERLEAVE] = 1.66 $<$ P[CPUCONCURRENT] = 2.22 $<$ P[CPUORDER] = 2.79, where P[X] denotes the performance of a scheduler X. As listed in Section6, P[CPUCONCURRENT] and P[CPUORDER] are the same on Kepler since it has Hyper-Q. Although TX1 also has Hyper-Q, P[CPUCONCURRENT] is lower than P[CPUORDER]. This is due to the fact that on TX1, Hyper-Q characteristics are different than Kepler when it comes to the placement of events to the execution queues. As explained above, issuing kernel stop events cause a false dependency when higher priority tasks have longer kernel runtimes as in Case-1. New methods are used in CPUORDER to resolve this false dependency on TX1, whereas they are not utilized in CPUCONCURRENT. Thus, the performance of CPUORDER is better on TX1.

Like CPUORDER, GPU schedulers involve polling on zero-copy memory. Hence, their timings are also reported incorrectly. We are in the process of developing new methods for resolving the timing problem of GPU schedulers. For this purpose, we have contacted NVIDIA engineers that work on the development of Jetson architectures to get their insights on how to resolve this problem.

## 8. CONCLUSIONS

Based on the results of the experiments, we make the following conclusions:



DOI: 10.1002/cpe

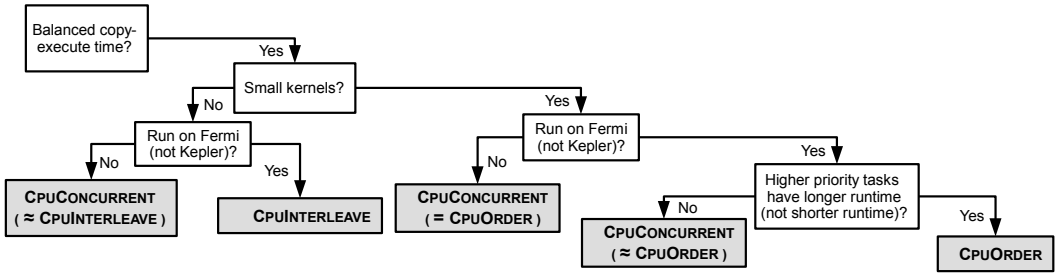



Figure 3. Recommended schedulers for given workload characteristics and GPU architectures.

(1) We observe that although TX1 has the Maxwell architecture, which is a successor to the Kepler architecture, the scheduler timing mechanism, which is based on CUDA events, behaves differently on TX1 and Kepler. This difference mainly stems from the fact that these architectures use different policies for placing events into the hardware queues, especially when CUDA commands cannot be issued consecutively from the host. Hence, scheduler timings are reported incorrectly on TX1 unless we develop new methods.

(2) Approaches that run kernels concurrently are advantageous when small kernels are used. If large kernels run concurrently, the performance does not change on Kepler and degrades on Fermi.

(3) If we have unbalanced copy-execute times in addition to large kernels, the approach that runs kernels concurrently performs even worse than the approach that does not overlap tasks on Fermi.

(4) The approach using atomics that changes the issue order only improves results on Fermi for the workloads with small kernels and when the kernel runtime of higher-priority tasks are longer. With larger kernels, using atomics is disadvantageous both on Fermi and Kepler.

(5) Approaches that perform GPU scheduling perform worse than their counterparts that perform CPU scheduling, due to the limitations of current GPU architectures.

This is perhaps the most important conclusion, and stems from the following observations: (a) running serial scheduler logic on a parallel GPU is not efficient; (b) to enforce priority between tasks, the GPU schedulers cannot use the benefit of launching kernels in parallel from the device (instead of using this benefit provided by dynamic parallelism, to give precedence of using compute resources to higher-priority tasks, just like the host-side kernel launches, we launch device-side kernels serially); (c) the GPU schedulers require transferring extra scheduling messages using zero-copy memory between the host and device, whereas the CPU schedulers do not need to spend time and resources for such transfers; and (d) since small amounts of device resources are occupied by the GPU scheduler, processing tasks takes slightly longer.

Muyan-Özçelik and Owens [2] provide further insights into the conclusions 2-5 by juxtaposing the results of individual schedulers on specific cases. Based on these conclusions, Figure 3 recommends specific schedulers for given workload characteristics and GPU architectures.

As described in Muyan-Özçelik and Owens [2], to demonstrate a plausible scenario to which this study can be applied, we have also analyzed a workload constructed from real-world tasks from four different automotive computing applications: pedestrian detection, blind-spot tracking, road-sign detection, and collision avoidance. The code for the tasks are derived from the GPU-accelerated computer vision samples of the OpenCV library [46]. The results from this real-world case are consistent with the conclusions provided in this section.

Finally, we recommend adding the following features to upcoming embedded architectures for improving the performance of future schedulers that perform real-time multitasking: hardware priority, preemption (this feature has recently been introduced in the Pascal architecture which is not available in an embedded device yet), programmable GPU scheduler, common time concept, atomics across the CPU and GPU, solutions to technical difficulties (e.g., avoiding false dependencies caused by events, flushing zero-copy memory, determining the end of concurrent kernels, dealing with delayed-signal phenomena). Muyan-Özçelik and Owens [2] provide details of these recommendations.



DOI: 10.1002/cpe

## 9. IMPLICATIONS AND FUTURE WORK

There are two important new research directions that we would like to explore in the realm of multitasking real-time embedded GPU tasks. The first one involves adding a preemption capability to schedulers. Although the preemption mechanism is one of the essential functions of traditional systems dealing with real-time task scheduling, GPUs lack this capability. However, very recently, an exciting improvement has developed in this area and NVIDIA has announced in April 2016 that their newest architecture, Pascal [47] along with the latest CUDA programming environment (CUDA 8) is introducing a new feature that enables compute preemption. When Pascal is available in an embedded device, we plan to take advantage of this newly introduced feature and add preemption capability to our system as another scheduling strategy. Although it will be supported on the hardware, we expect that adding a preemption mechanism to our system would inevitably introduce overheads. Since real-time tasks are time-sensitive, it is important to investigate in which conditions the benefits of utilizing the preemption functionality would be greater than the overhead introduced by this mechanism. Based on this investigation, we would like to determine how and when the scheduling strategy supporting preemption should be used and in what ways the performance of the compute preemption feature can be improved in the upcoming GPU architectures.

The other strategy we plan to explore is work division. Dividing tasks into subtasks would lead to finer-grained scheduling, which in turn would allow us to better address the requirements of real-time tasks. Since runtimes of individual kernels executed by the GPU are decreased when work division is used, performing work division would also reduce the need to utilize a preemption system.

ACKNOWLEDGEMENTS

This work was supported by NSF award CCF-1017399, an NVIDIA graduate fellowship, and CSUS probationary faculty development grant. We would like to thank Stephen Jones, Steve Rennich, and Barrett Williams for providing us with important insights concerning the technical aspects of our research.

REFERENCES

1. Muyan-Özçelik P, Glavtchev V, Ota JM, Owens JD. Real-time speed-limit-sign recognition on an embedded system using a GPU. *GPU Computing Gems*, vol. 1, Hwu WW (ed.). chap. 32, Morgan Kaufmann, 2011; 497–516.
2. Muyan-Özçelik P, Owens JD. Multitasking real-time embedded GPU computing tasks. *International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*, 2016; 78–87.
3. NVIDIA. Jetson TX1 module 2016. `http://www.nvidia.com/object/jetson-tx1-module.html`.
4. NVIDIA. Fermi compute architecture whitepaper 2009. `http://www.nvidia.com/object/IO_89570.html`.
5. NVIDIA. Kepler GK110 architecture whitepaper 2012. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.
6. Budge B, Bernardin T, Stuart JA, Sengupta S, Joy KI, Owens JD. Out-of-core data management for path tracing on hybrid resources. *Computer Graphics Forum* 2009; **28**(2):385–396.
7. Tzeng S, Patney A, Owens JD. Task management for irregular-parallel workloads on the GPU. *High Performance Graphics*, 2010; 29–37.
8. Sugerman J, Fatahalian K, Boulos S, Akeley K, Hanrahan P. GRAMPS: A programming model for graphics pipelines. *ACM Transactions on Graphics* 2009; **28**:4:1–4:11.
9. Ino F, Ogita A, Oita K, Hagihara K. Cooperative multitasking for GPU-accelerated grid systems. *International Conference on Cluster, Cloud and Grid Computing*, 2010; 774–779.
10. Parker SG, Bigler J, Dietrich A, Friedrich H, Hoberock J, Luebke D, McAllister D, McGuire M, Morley K, Robison A, *et al.*. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics* 2010; **29**:66:1–66:13.
11. Peters H, Koper M, Luttenberger N. Efficiently using a CUDA-enabled GPU as shared resource. *International Conference on Computer and Information Technology* 2010; :1122–1127.
12. Elliott GA, Anderson JH. Globally scheduled real-time multiprocessor systems with GPUs. *International Conference on Real-Time and Network Systems*, 2010; 197–206.
13. Steinberger M, Kainz B, Kerbl B, Hauswiesner S, Kenzel M, Schmalstieg D. Softshell: Dynamic scheduling on GPUs. *ACM Transactions on Graphics (TOG)* Nov 2012; **31**(6):161:1–161:11.
14. Aila T, Laine S. Understanding the efficiency of ray traversal on GPUs. *High Performance Graphics*, 2009; 145–149.
15. Stuart JA, Owens JD. Message passing on data-parallel architectures. *International Parallel and Distributed Processing Symposium*, 2009; 1–12.



DOI: 10.1002/cpe

16. Stuart JA, Cox M, Owens JD. GPU-to-CPU callbacks. *UnConventional High Performance Computing (in conjunction with Euro-Par)*, 2010.
17. Margiolas C, O'Boyle MFP. Portable and transparent software managed scheduling on accelerators for fair resource sharing. *International Symposium on Code Generation and Optimization (CGO)*, 2016; 82–93.
18. Intel. Thread building blocks 2016. `http://www.threadingbuildingblocks.org/`.
19. Intel. Cilk plus 2016. `http://software.intel.com/en-us/articles/intel-cilk/`.
20. Intel. Nulstein: Do-it-yourself game task scheduling 2012. `http://software.intel.com/en-us/articles/nulstein`.
21. Microsoft. ConcRT: Concurrency runtime 2016. `http://msdn.microsoft.com/en-us/library/dd504870.aspx`.
22. Microsoft. Task parallel library 2016. `http://msdn.microsoft.com/en-us/library/dd460717.aspx`.
23. Apple. Grand central dispatch 2016. `http://developer.apple.com/mac/library/documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html`.
24. Werth B. Holistic task parallelism for common game architecture patterns. *Graphics Engine Gems*, vol. 1. Jones and Bartlett, 2011; 289–296.
25. KhronosGroup. OpenCL: The open standard for parallel programming of heterogeneous systems 2016. `http://www.khronos.org/opencl`.
26. NVIDIA. CUDA parallel computing platform 2016. `http://www.nvidia.com/object/cuda_home_new.html`.
27. Boussinot F, de Simone R. The Esterel language. *IEEE*, vol. 79, 1991; 1293–1304.
28. Halbwachs N, Caspi P, Raymond P, Pilaud D. The synchronous data flow programming language LUSTRE. *IEEE*, vol. 79, 1991; 1305–1320.
29. Guernic PL, Gautier T, Borgne ML, Maire CL. Programming real-time applications with SIGNAL. *IEEE*, vol. 79, 1991; 1321–1336.
30. Benveniste A, Caspi P, Edwards S, Halbwachs N, Guernic PL, de Simone R. The synchronous languages 12 years later. *IEEE*, vol. 91, 2003; 64–83.
31. Tripakis S, Benveniste A, Caspi P, Pinello C, Sangiovanni-Vincentelli A, Sofronis C. Correct and efficient implementations of synchronous models on asynchronous execution platforms. *Workshop in Exploiting Concurrency Efficiently and Correctly*, 2009.
32. Hawkins T. Atom 2007. `http://hackage.haskell.org/package/atom/`.
33. Langley B. Windows "Longhorn" display driver model - details and requirements. *Windows Hardware Engineering Conference*, 2004.
34. Martin KE, Faith RE, Owen J, Akin A. Direct rendering infrastructure, low-level design document 1999. `http://dri.sourceforge.net/doc/design_low_level.html`.
35. Bautin M, Dwarakinath A, Chiueh T. Graphic engine resource management. *Multimedia Computing and Networking*, vol. 6818, 2008; O:1–O:12.
36. Kato S, Lakshmanan K, Rajkumar R, Ishikawa Y. Timegraph: GPU scheduling for real-time multi-tasking environments. *USENIX Annual Technical Conference*, 2011.
37. Rossbach C, Witchel E. Operating system abstractions for GPU programming 2010. `http://www.nvidia.com/content/GTC-2010/flvs/2124_GTC2010.mp4`.
38. Rossbach CJ, Currey J, Silberstein M, Ray B, Witchel E. PTask: operating system abstractions to manage GPUs as compute devices. *ACM Symposium on Operating Systems Principles*, 2011; 233–248.
39. Kato S, McThrow M, Maltzahn C, Brandt S. Gdev: First-class GPU resource management in the operating system. *USENIX Annual Technical Conference*, 2012.
40. Wang Z, Yang J, Melhem R, Childers B, Zhang Y, Guo M. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. *International Symposium on High-Performance Computer Architecture (HPCA)*, 2016; 358–369.
41. Stratton JA, Rodrigues C, Sung IJ, Obeid N, Chang LW, Anssari N, Liu GD, Hwu WM. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Technical Report IMPACT-12-01*, University of Illinois at Urbana-Champaign Center for Reliable and High-Performance Computing 2012.
42. Bakhoda A, Yuan G, Fung W, Wong H, Aamodt T. Analyzing CUDA workloads using a detailed GPU simulator. *International Symposium on Performance Analysis of Systems and Software*, 2009; 163–174.
43. Muyan-Özçelik P. Running real-time tasks on embedded systems using GPU computing. PhD Thesis, Dept. of Computer Science, University of California, Davis 2014.
44. Rennich S. CUDA C/C++ streams and concurrency 2011. `http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf`.
45. NVIDIA. Jetson TX1 and embedded systems forum: Zero-copy and managed memory on Jetson 2016. `https://devtalk.nvidia.com/default/topic/932957/zero-copy-and-managed-memory-on-jetson/?offset=2`.
46. Bradski G. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* 2000; **25**(11):122–125.
47. Mark Harris. Inside Pascal: NVIDIAs newest computing platform 2016. `https://devblogs.nvidia.com/parallelforall/inside-pascal/`.



DOI: 10.1002/cpe