Title
Multitasking Real-time Embedded GPU Computing Tasks

Permalink
https://escholarship.org/uc/item/7jc3q2q9

Authors
Muyan Ozcelik, Pinar
Owens, John D

Publication Date
2016-03-01

Peer reviewed

Multitasking Real-time Embedded GPU Computing Tasks

Pınar Muyan-Özçelik
California State University, Sacramento
pmuyan@ecs.csus.edu

John D. Owens
University of California, Davis
jowens@ece.ucdavis.edu

Abstract

In this study, we consider the specific characteristics of workloads that involve multiple real-time embedded GPU computing tasks and design several schedulers that use alternative approaches. Then, we compare the performance of schedulers and determine which scheduling approach is more effective for a given workload and why. The major conclusions of this study include: (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 highlight the shortcomings of current GPU architectures with regard to running multiple real-time tasks, and recommend new features that would improve scheduling, including hardware priorities, preemption, programmable scheduling, and a common time concept and atomics across the CPU and GPU.

Keywords multitasking, real-time embedded tasks, GPU computing

1. Introduction

As Graphics Processing Units (GPUs) have become more programmable, we have been increasingly using 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 [7]. 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 good fit for running real-time embedded tasks, our ability to utilize the full potential of GPUs in embedded systems would require developing effective multitasking strategies, our focus in this paper.

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 by surveying a wide spectrum of 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. Hence, we generate these cases in a way that they would allow us to investigate the important factors that affect scheduler performance. To accomplish this goal, we differ the properties of synthetic cases from each other mainly in the workload characteristics that highlight the discrepancies among the alternate scheduling strategies and in the GPU architectures that are utilized to run the workloads. 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. In addition, we highlight the shortcomings of current GPU architectures with regard to running multiple real-time tasks and recommend new features that would allow better schedulers to be designed.

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. 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.

1.1 Motivation

Embedded systems usually involve several data-parallel real-time tasks that need to run concurrently. For instance, in the automotive computing domain, data-parallel tasks such as speed-limit-sign recognition, lane departure warning system, speech recognition, infotainment systems, etc. would need to run at the same time. To run all these tasks concurrently, cars usually dedicate a custom embedded processor to each of these tasks, which is not a cost-effective solution. If we allow the GPU to multitask between disparate embedded real-time tasks, we can consolidate all these customized digital systems into a single software-programmable and inexpensive GPU. This consolidation would in turn simplify the design and

PMAM'16, March 12-16 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4196-7/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2883404.2883408

reduce the cost of cars as well as allowing more vendors to compete for providing solutions.

Similar trends can be seen in other embedded real-time domains such as robotics or the mobile computing domain. For instance, mobile devices increasingly interact with the physical world through various data-parallel real-time applications that perform augmented reality, face recognition, fingerprint unlocking, and so on. As the number of concurrent real-time data-parallel applications that mobile devices run increases, to take the full advantage of GPUs on these platforms, we again need strategies that would allow GPUs to multitask among several real-time tasks.

1.2 Challenges

The current programming model of GPUs poses several challenges for effectively managing workloads containing multiple concurrent data-parallel real-time tasks. One important challenge stems from the fact that current 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 (e.g., asynchronous memory copy, streams, concurrent kernel execution), since they are not primarily designed for multitasking among real-time tasks, these features present several limitations for such purposes. We overcome these limitations by developing various methods that are derived from the abovementioned features (Section 4.2).

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. A latency-oriented task is characterized by its infrequent input and its expectation that the system will process its data almost instantly (e.g., a speech recognition task that processes a driver's audio commands). On the other hand, a throughput-oriented task is characterized by having streaming input. It expects the system to process a certain amount of data in a given time such that the differences between the finish times of two consecutive subtasks are uniform (e.g., a video decoding task of an infotainment system that needs to provide 30 frames per second must process one frame every 33 milliseconds). We can say that latency-oriented tasks focus more on response time, whereas throughput-oriented tasks focus more on uniformly processed data per time.

Understanding the needs of latency- and throughput-oriented tasks, and scheduling them in parallel, is a challenge. To address this challenge, we use different priority assignments and performance calculations for these two types of real-time tasks. In addition, 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. We describe how we overcome these issues in the design of our schedulers in Section 3.

2. Background

The previous section summarizes the contributions of our study at a high level by noting how we address the research challenges. In this section, we discuss the differences from previous work in order to highlight other contributions of our work. We also provide terminology and a description of the development environment used in this study.

2.1 Previous Work

Previous studies have provided techniques for scheduling parallel task systems with GPUs in the context of out-of-core data [2], irregular workloads [18], programmable rendering pipelines [17], multitasking between graphics and computation applications [4], a general purpose ray tracing engine [12], sharing the resources of a single GPU among different CPU clients [13], 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 [3] and Steinberger et al. [16] target real-time tasks. However, like most of the related prior work in this area [2, 17], these studies focus on integrating the GPU to real-time multiprocessor systems as a co-processor and offloading only parts of the work onto the GPU. On the other hand, we propose to schedule all the available data-parallel real-time work to run on a single GPU.

Kato et al. [5] and Rossbach et al. [15] also target real-time tasks. They provide support for scheduling at the operating-system and device-driver level, whereas we focus on providing this support at the application level. In addition, like the other prior studies that target real-time tasks [3, 16], these studies either do not provide plausible scenarios or perform multitasking among collaborative tasks, whereas we provide support for multitasking among non-collaborative real-time tasks in the real-world (e.g., automotive computing tasks).

Although previous research on multitasking among GPU tasks used different strategies (e.g., running the scheduler on the CPU [4, 13] or the GPU [12, 18], using persistent threads and uberkernels [12, 13, 18], interleaving data transfers with kernel execution using asynchronous memory copy [13] or zero-copy memory [16] etc.), the comparison of these techniques to each other, which is one of the main focuses of this study, is an under-researched area.

To the best to our knowledge, there is no prior study utilizing the fairly 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 features for scheduling real-time embedded tasks. In addition, we introduce several novel methods which serve as building blocks for schedulers and resolve technical difficulties that arise while implementing them.

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. Kato et al. [5] 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.

2.2 Terminology

The important technical terms used in this study include:

Task: In the context of this study, we define *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. For instance, the gradient computation performed by pedestrian detection and its related input/output frame copies constitute a task. 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, as explained in Section 4.3) and dependency information. There would be a dependency relationship between Task-A and Task-B if we need to process Task-A before processing Task-B. Task-A would be a dependee task and Task-B would be a dependent task. A *batch* of tasks is a group of tasks that the scheduler runs together at a given time. Schedulers usually run tasks in a batch together by interleaving/overlapping their copy/execute operations. Task operations consist of CUDA commands

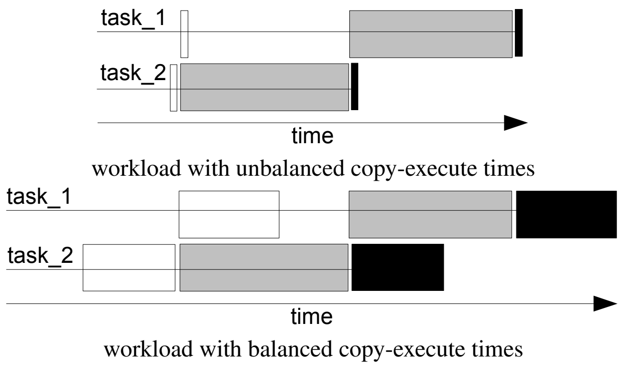
task_1
task_2
time
workload with unbalanced copy-execute times
task_1
task_2
time
workload with balanced copy-execute times

Figure 1. Examples of workloads with unbalanced and balanced copy-execute times: White, gray, and black boxes present host-to-device copy, kernel execution, and device-to-host copy operations of tasks, respectively. Assume we pursue a scheduling approach that interleaves copy/execute operations.

involving either a data transfer or a kernel execution command preceded by synchronization and/or timing commands and followed by a timing command.

Data: We run a task for each instance of data that arrives as an input. For instance, we run the gradient task for each frame of the streaming video while performing pedestrian detection. Each task has a data list. When new data arrives, it is added to the end of the list. The task's data is processed in a first-in-first-out manner.

Workload characteristics: GPU workloads that include multiple real-time tasks have several salient characteristics that are considered in the design of schedulers. These 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 transfer of one task with the kernel run 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. However, if we have unbalanced copy-execution time, the performance of running the tasks serially and the performance of interleaving copy/execute operations would be almost the same. Figure 1 illustrates examples of workloads with balanced and unbalanced copy-execute times.

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 a 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.

2.3 Development Environment

In this study, we use NVIDIA's "Fermi" [8] and newer "Kepler" GPU architectures [9] and the CUDA programming framework [10]. These architectures are preferable to the GPUs widely used today in embedded systems since Fermi and Kepler introduce key features that support multitasking among several tasks such as "concurrent kernel execution" and "dynamic parallelism". We expect these features to appear in future embedded GPUs (in fact they now appear in NVIDIA's Tegra X1[1] [11]). In addition to schedulers utilizing these fairly new features, we have also developed schedulers that do not utilize them, i.e., CpuTraditional and CpuInterleave, as explained in the next section. Hence, this study provides schedulers that can also run on today's embedded GPUs.

Exploring the advantages of new GPU features for multitasking among real-time embedded tasks before they are widely available on embedded GPUs provides insights for adding these technologies to embedded GPUs by providing recommendations such as which features are more important than others, what improvements can be done to make them more effective, etc.

3. Design of Schedulers

We explore the design space of schedulers performing multitasking among real-time GPU workloads with the characteristics mentioned in Section 2.2. We consider both techniques based on prior research and the ones we develop using the new GPU features. Hence, we survey a wide spectrum of possible scheduling strategies, which can be summarized with the taxonomy presented in Figure 2.

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 ex-

[1] Tegra X1 currently appears in one consumer device: the Android TV NVIDIA SHIELD; a future NVIDIA Jetson with Tegra X1 would be an ideal platform for further work in this area.

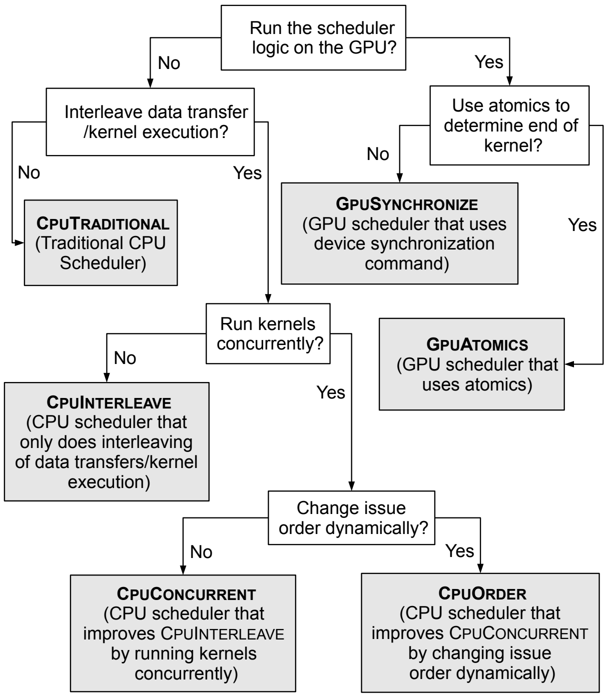
Run the scheduler logic on the GPU?
No
Yes
Interleave data transfer /kernel execution?
No
Yes
Use atomics to determine end of kernel?
No
Yes
CpuTraditional (Traditional CPU Scheduler)
GpuSynchronize (GPU scheduler that uses device synchronization command)
GpuAtomics (GPU scheduler that uses atomics)
Run kernels concurrently?
No
Yes
CpuInterleave (CPU scheduler that only does interleaving of data transfers/kernel execution)
Change issue order dynamically?
No
Yes
CpuConcurrent (CPU scheduler that improves CpuInterleave by running kernels concurrently)
CpuOrder (CPU scheduler that improves CpuConcurrent by changing issue order dynamically)

Figure 2. Tree that shows taxonomy of schedulers. White and gray boxes refer to strategy and scheduler nodes, respectively. Each scheduler uses an alternative approach which is a combination of all strategies utilized on the path from the leaf scheduler node up to the root node. We implement and compare the six different schedulers that fall out of this taxonomy.

ecution first with the execution of the other task's kernel in a batch, it issues the the device-to-host copy operation of the former task before the latter task). We provide pseudocodes of CPU schedulers in the appendix.

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.

The properties of all schedulers are summarized in Table 1. 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.

As mentioned in Section 1.2, 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. Recently introduced system-on-chip mobile GPUs that support GPU computing (e.g., the Tegra X1 [11]) would alleviate the communication cost between CPU-GPU.

Property	CpuTraditional	CpuInterleave	CpuConcurrent	CpuOrder	GpuSynchronize	GpuAtomics
Run scheduler logic on the CPU?	✓	✓	✓	✓		
Run scheduler logic on the GPU?					✓	✓
Interleave data transfer/kernel execution?		✓	✓	✓	✓	✓
Run kernels concurrently?			✓	✓	✓	✓
Change the issue order dynamically?				✓	✓	✓
Use atomic version of kernels?				✓		✓
Can be run on Fermi?	✓	✓	✓	✓		
Can be run on Kepler?	✓	✓	✓	✓	✓	✓

Table 1. Summary of properties of schedulers.

4. Implementation

We implement the scheduling strategies by constructing several methods that leverage particular features of our GPU target architectures, listed below. This section describes important methods that we have used to implement our schedulers. Muyan-Özçelik [6] provides a complete list of methods and implementation details of schedulers. This section also provides our priority and performance calculations.

4.1 Features

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 building blocks 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 [10] provides details of all the abovementioned features. We provide here brief descriptions of the subset of these features:

Stream: A stream is a sequence of commands performed on the GPU in a given issue order, which is an order of commands in the program. Different streams, on the other hand, may execute their commands out of order with respect to one another. Commands in different streams can be overlapped. For instance, kernel execution of one stream can be interleaved with data transfer of another stream, or kernel executions of different streams can run concurrently.

Hardware queue: Commands sent to the device are dispatched to the hardware queues in the given issue order. There are two types of queues: copy and execution.

Hyper-Q: The Fermi architecture has one execution queue, whereas the Hyper-Q feature introduced in the Kepler architecture provides 32 execution queues. Hence, in Fermi, execution commands of all streams are placed in the same queue, whereas in Kepler, each stream usually has its own execution queue. Likewise, in Fermi, data transfers of all streams are placed in the same copy queue, whereas in Kepler, we have multiple copy queues, and data transfers of the different streams are not placed in the same queue.

Atomic: An atomic function performs a read-modify-write atomic operation on a variable residing in global or shared memory. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is completed.

Event: Events can be recorded at any point in the program. Events recorded on a stream are completed when all commands in that stream preceding the event are completed. We can query when these events are completed and thus, we can measure the runtime of specific operations or perform synchronization between different streams.

Persistent kernel: The persistent kernel feature is the combination of persistent threads and uberkernels. The persistent threads feature advocates launching only enough GPU threads to fully occupy the GPU and keeping them alive until there is no more data left to be processed. A uberkernel allows running different operations in parallel by fusing together multiple kernels into one single kernel. In this study, we use persistent threads to bypass the hardware scheduler and continuously make scheduling decisions on the GPU until all the tasks in the workload are done. We also use an uberkernel since the persistent threads feature restricts us to use a single kernel, and the only way to perform two operations required by GPU schedulers, i.e., making decisions and launching tasks, at the same time using a single kernel is through the adoption of the uberkernel method.

Zero-copy memory: Zero-copy memory is a pinned memory mapped into the CUDA address space. GPU kernels can directly access this memory and kernel-originated data transfers utilizing zero-copy memory automatically overlap kernel executions.

Dynamic parallelism: The Kepler architecture introduces the dynamic parallelism feature that enables kernels to launch new kernels directly from the GPU. If they are launched from different streams and there are enough resources, kernels can run concurrently.

4.2 Methods

We use the abovementioned features to construct methods that implement different scheduling strategies. Most of the methods we present here are either new techniques proposed in this work or combine existing techniques in novel ways.

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 make scheduling decisions with a kernel that launches tasks into a persistent kernel (threads of the kernel that launch tasks use dynamic parallelism).

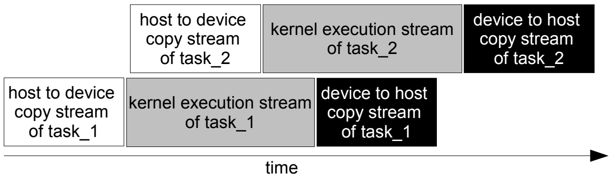
host to device copy stream of task_2
kernel execution stream of task_2
device to host copy stream of task_2
host to device copy stream of task_1
kernel execution stream of task_1
device to host copy stream of task_1
time

Figure 3. Each task operation is mapped to different streams and overlapped with operations of other tasks. Here we assume we pursue a scheduling approach that interleaves copy/execute operations and runs kernels concurrently.

Beyond serving as building blocks, these methods also resolve technical difficulties that arise while implementing our schedulers. 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 in all the scheduling strategies. Data copy and kernel execution commands are placed in copy and execution queues, relatively. As long as there are no dependencies, executions of commands in different queues can be overlapped (e.g., the copy command of one task can be interleaved with execution command of another task). However, blocked commands cause false dependencies between independent commands.

Just like any other command, events are also placed in hardware queues. However, which queue they will be placed is device/driver-dependent. In the Fermi architecture, they are by default placed in the same queue as the previous command in the same stream. Depending on in which queue they are placed, events can be blocked and cause false dependencies. We fix some of these false dependencies by enforcing events that are used to measure runtime of copy and execute operations to be placed in the copy and execution queues, respectively. It enforces these requirements by dedicating different streams to each operation of the task and initializing copy and execute streams with dummy copy and dummy execute commands, respectively. This method is a new method pioneered by this study. Resolving false dependencies by using different streams allows schedulers to overlap operations of different tasks. Figure 3 illustrates how we map two tasks to different streams and overlap operations of these tasks.

Another method that deals with false dependencies caused by the events involves not issuing a stop event between concurrent kernel launches. As indicated in the CUDA Programming Guide [10], while devices of compute capability 3.0 or lower run kernels concurrently, an operation that requires a dependency check to see if a streamed kernel launch is completed (e.g., command within the same stream as the launch being checked) blocks all later kernel launches from any stream until the kernel launch being checked is completed. That is why the stop event of the first-issued kernel inserted between the launches of kernels intended to be run concurrently causes a false dependency. The CUDA Programming Guide recommends issuing all independent operations before dependent operations to improve the potential for concurrent kernel executions. Following this recommendation, to avoid false dependencies and allow concurrent execution of kernels, we insert the stop event of the first-issued kernel after issuing concurrent kernel launches.

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, we allow this transfer to overlap with the execution of the other kernel that has been running concurrently. However, to implement the CPU scheduling strategy that changes the issue order (i.e., CpuOrder), we cannot use the "stream query" command to determine the end of concurrent kernels, since stream query blocks the concurrent kernel streams until the longest running kernel finishes. We call this difficulty the "blocking-query" problem. Hence, instead of using stream query, CpuOrder determines the end of kernels using a method that utilizes atomic operations. As we noted in Section 3, we utilize a similar method that uses atomics to implement GpuAtomics to solve the unstable performance problem caused by the device-side device synchronize command while determining the end of kernels. These methods use 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. In the CPU schedulers, the host reads this counter via zero-copy memory. Hence, this method combines different existing methods (i.e., atomics and zero-copy) in a novel way.

In addition, we develop a method that resolves a difficulty of dictating the execution order while overlapping tasks. Schedulers execute the tasks in priority order to meet the time requirements in the best way they can. Hence, it is important to set the execution order of commands. Since the Hyper-Q feature introduced in Kepler allows us to have more than one queue for each type of command (e.g., Kepler has multiple host-to-device copy queues), issue order does not always lead to execution order. For instance, since tasks use different streams, copy commands of two overlapping tasks would be placed in different copy queues, and since the hardware can execute a ready-to-execute command from any copy queue, issue order would not dictate execution order. Our method ensures execution order follows issue order by inserting a stream-wait-event command between the task operations that are supposed to execute in order. In this way, the start of the second-issued task operation would wait until the first-issued task operation is finished. Thus, this method uses the stream-wait-event command for a novel purpose (i.e., to ensure execution order follows issue order).

Finally, with the following method, we resolve a difficulty with flushing zero-copy memory. Our findings show that when the schedulers run on the Windows operating system, the host cannot see the changes the device makes on the zero-copy memory without calling host-side device synchronization. However, we cannot utilize this command since it hurts the overlap of task operations in CPU schedulers and causes a deadlock in GPU schedulers. Instead, to flush zero-copy memory, we use the stream query command on any of the streams executing concurrently with the kernel that is updating the zero-copy memory. When we use this method in our CPU schedulers, we record the stop event of the first-finished kernel on the device-to-host copy stream instead of the kernel execution stream to resolve the blocking-query problem that is caused by the use of stream query command. This method is another new method pioneered by this study.

4.3 Priority Calculations

All schedulers use the same logic for priority calculations. These calculations include determining the tasks that are ready to execute, calculating priority values for these tasks, and sorting them according to these values. A task is considered ready to execute if both of the following conditions are met: (a) it has data that has not been processed, and (b) it does not have any dependee or if it does, the related data of its dependee is already processed. Ready-to-execute tasks with precedence are assigned higher priority values.

We use different priority calculations for latency- and throughput-oriented tasks (Equation 1). These calculations are based on the time requirements of tasks, which are specified differently for latency- and throughput-oriented tasks. As a time requirement, latency tasks specify a "deadline" that indicates the time period in which the system should respond to a task's newly arrived data. On the other hand, throughput tasks specify a "separation time" that indicates the difference between the finish times of two consecutively-arriving data instances. Figure 4 illustrates the deadline and separation time. In Equation 1, "elapsed time" indicates the time since the first data in a task's data list was received, and "time since last done" is the time since the last processed data of the task is finished. To assign a priority value, schedulers also need to know the "average runtime" of tasks. Hence, before running the schedulers, we perform a profiling run to calculate the average runtime of the tasks.

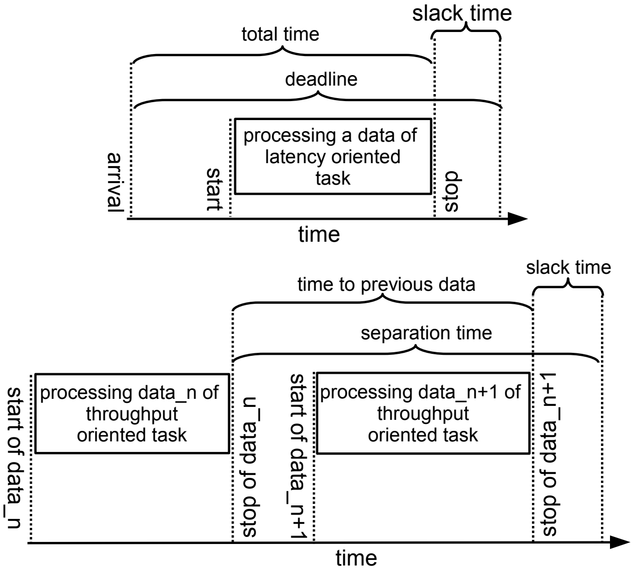

Figure 4. Time definitions for latency- and throughput-oriented tasks.

$$\begin{aligned} &\text{priority\_of\_latency\_task} = && (1)\\ &\qquad (\text{elapsed\_time} + \text{average\_runtime}) - \text{deadline} \\ &\text{priority\_of\_throughput\_task} = \\ &\qquad (\text{time\_since\_last\_done} + \text{average\_runtime}) - \text{separation\_time} \end{aligned}$$

4.4 Performance Calculations

We compute scheduler performance as a function of a value that accumulates the slack time of each data. For latency tasks, slack is calculated as the difference between the "total time" and the deadline, where total time is the difference between the arrival and finish times of data. For throughput-oriented tasks, slack time is the difference between "time to previous data" and separation time, where time to previous data is the difference between the finish times of previous and current data. Figure 4 illustrates the total time and time to previous data.

To compare the schedulers in a standard way, we scale scheduler performance to a baseline. To assign performance values, we first calculate an ideal accumulated slack time value by assuming that the system has the ability to respond to all data as soon as they are ready to execute. Then, we calculate performance values for all the schedulers by shifting their accumulated slack time values according to an ideal accumulated slack time value and taking the shifted value of CpuTraditional as the basis for the comparison. Hence, the performance value for the scheduler can be computed as indicated in Equation 2.

$$\text{performance\_of\_scheduler} = \frac{\text{accumulated\_slack\_time\_of\_CpuTraditional} - \text{ideal\_accumulated\_slack\_time}}{\text{accumulated\_slack\_time\_of\_scheduler} - \text{ideal\_accumulated\_slack\_time}} \qquad (2)$$

As can be seen from the above formula, the performance of CpuTraditional would always be 1. 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.

5. Experiments

As explained in Section 1, one of the important contributions of our study is investigating the use of various schedulers that pursue alternative approaches rather than focusing on a specific scheduler. We evaluate and compare these alternative 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 to be 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 [14] (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 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 2. We calculate the performance values using the technique introduced in Section 4.4. Comparison of performance values are only meaningful within a case and should not be compared across cases. It is because, in each case shifted accumulated slack time values of CpuTraditional is taken as a base (i.e., performance of CpuTraditional is always assigned to 1) to calculate performance of other schedulers and this value changes across cases. Hence, for instance, if one case has larger performance value than another case for a particular scheduler, it does not mean that the scheduler performs better in the former case.

	Cases					
Scheduler	**1**	**2**	**3**	**4**	**5**	**6**
CpuTraditional	1.00	1.00	1.00	1.00	1.00	1.00
CpuInterleave	1.92	2.83	3.34	**1.75**	**2.59**	**1.13**
CpuConcurrent	**3.57**	3.55	**4.37**	**1.78**	1.54	0.82
CpuOrder	**3.57**	**4.26**	**4.32**	0.66	0.11	0.29
GpuSynchronize	3.12	—	—	1.70	—	—
GpuAtomics	3.22	—	—	0.63	—	—

Case-1: small kernels, balanced operations, on Kepler, higher-priority tasks have longer kernel runtimes
Case-2: Case-1 workload 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 workload to Case-4 on Fermi
Case-6: similar to Case-5, but with unbalanced operations

Table 2. Performance of the schedulers for all cases (the best performance value for each case is indicated in **bold**).

6. Summary and Conclusions

Based on the results of the experiments, we make the following conclusions (where P[X] denotes the performance of a scheduler X):

(1) 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.

As explained in Section 2.2, large kernels cannot truly run concurrently. Hence, if large kernels and the approach that runs kernels concurrently are used, the performance on Kepler would be the same as the performance when the approach that does not run kernels concurrently is used (P[CpuConcurrent] $\approx$ P[CpuInterleave] in Case-4). When large kernels are used, the performance degrades on Fermi due to the delayed-signal phenomenon (P[CpuConcurrent] $<$ P[CpuInterleave] in Case-5). According to our observations, this phenomenon does not occur when small kernels are used on Fermi. In addition, this phenomenon does not occur on Kepler due to its support of Hyper-Q. If small/medium kernels are used, the approach that runs kernels concurrently improves performance both on Fermi and Kepler (P[CpuConcurrent] $>$ P[CpuInterleave] in Case-1, Case-2, and Case-3).

(2) 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.

Since we have unbalanced copy-execute times, the performance improvement gained from interleaving host-to-device copies with kernel executions would be minimal; moreover, it would be canceled out by the performance degradation stemming from the delayed-signal phenomenon (P[CpuConcurrent] $<$ P[CpuTraditional] in Case-6).

(3) 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.

If we have small kernels, changing the issue order approach used by a CPU scheduler (which utilizes atomics) does not improve performance on Kepler (since it has Hyper-Q). Hence, the performance of the CPU scheduler that uses atomics and changes issue order would be the same as its counterpart that does not do them (P[CpuOrder] = P[CpuConcurrent] in Case-1). When small kernels are used, the performance of the GPU scheduler that utilizes atomics would also be the same as its counterpart not utilizing atomics (P[GpuSynchronize] ≈ P[GpuAtomics] in Case-1). In CPU schedulers, by default, the higher-priority task's device-to-host copy is issued before that of the lower-priority task. Hence, if the runtime of higher-priority task is shorter than that of the lower-priority task, changing issue order would not improve performance (P[CpuOrder] ≈ P[CpuConcurrent] in Case-3). However, if the runtime of higher-priority task is longer, to get the best overlap on Fermi, we need to switch the default issue order (P[CpuOrder] > P[CpuConcurrent] in Case-2).

With respect to atomic operations, we find that they are expensive when large kernels are used; when using strategies with atomics, the runtime of kernels would increase and the performance of their CPU/GPU schedulers would decrease both on Kepler and Fermi. (P[CpuTraditional] > P[CpuOrder] ≈ P[GpuAtomics] in Case-4 and P[CpuTraditional] > P[CpuOrder] in Case-5 and Case-6).

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

GPU schedulers perform worse than their CPU counterparts. The counterpart of GpuSynchronize is CpuConcurrent since neither of them use atomics (P[GpuSynchronize] < P[CpuConcurrent] in Case-1). Likewise, the counterpart of GpuAtomics is CpuOrder since both of them use atomics (P[GpuAtomics] < P[CpuOrder] in Case-1). 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.

7. A Real-world Case

To demonstrate a plausible scenario to which this study can be applied, we now analyze 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 [1].

This real-world case includes six tasks: (1) A pedestrian detection application task, based upon the kernel that computes the gradient of a given image. This kernel is part of the HOG (Histogram of Oriented Gradients) detector used for detecting people. (2) A blind-spot tracking application task, based upon a kernel that scales a vector by a scalar. This kernel is part of the optical-flow operator used for tracking objects. (3) A road-sign detection application task, based upon the kernel that performs thresholding with hysteresis to trace edges through an image. This kernel is part of the Canny edge detector, a pre-processing step applied by the Hough Lines transform that is used for finding straight lines in an image. (4) Another road-sign detection application task, based upon the kernel that builds point lists from a given image. This kernel is part of the Hough Lines transform used for finding straight lines in an image. (5) A collision avoidance application task, based upon the kernel that calculates a disparity map from a pair of images. This kernel is part of the stereo-matching operator used for getting depth information for the objects in the scene. (6) Another collision avoidance application task, based upon the kernel that performs low textureness filtering, which is a post-processing step applied by the stereo-matching operator used for getting depth information for the objects in the scene.

We include dependency relationships between tasks belonging to the same application. Tasks belonging to the pedestrian and road-sign detection applications are throughput-oriented; the others are latency-oriented. The workload has balanced copy/execute times. In addition, unlike the synthetic cases, in this study the kernel sizes are small enough that kernels can truly run concurrently but big enough that the use of atomics would increase the runtime of kernels.

To provide a comparison of results across all 6 of our schedulers, we run this real-world workload on a Kepler GPU, which allows running GPU schedulers as well as CPU ones. Comparison of the scheduler performance is as follows (performance values, which are calculated using the technique introduced in Section 4.4, are provided in parenthesis): CpuTraditional (1.00) < CpuInterleave (1.13) < GpuAtomics (1.47) < CpuOrder (1.63) < GpuSynchronize (2.04) < CpuConcurrent (2.45).

These results are consistent with the conclusions provided in the previous section: Since kernel sizes are small enough that kernels can truly run concurrently, as indicated in conclusion-1, the approach that runs kernels concurrently is advantageous (P[CpuConcurrent] > P[CpuInterleave]). The approach that does not overlap tasks performs worst (P[CpuTraditional] is worst) because the workload has balanced copy/execute times and thus interleaving host-to-device copies with kernel executions would be advantageous (related to conclusion-2). Since kernel sizes are big enough that the use of atomics would increase runtime of kernels (unlike Case-1 which has small kernels), in accordance with conclusion-3, the schedulers utilizing atomics would be disadvantaged compared to their atomicless counterparts (P[CpuOrder] < P[CpuConcurrent] and P[GpuAtomics] < P[GpuSynchronize]). However, since the kernels are not very big, these schedulers would still perform better than the approach that does not overlap task operations (P[CpuTraditional] < P[GpuAtomics] ≈ P[CpuOrder). Finally, due to the reasons explained in conclusion-4, GPU schedulers perform worse than their CPU counterparts. We would like to stress that conclusion-4 does not mean that CPU scheduling approaches will always be better than GPU scheduling approaches. Instead it indicates that a CPU scheduler will be better than its GPU "counterpart" which use the same strategy (P[GpuSynchronize] < P[CpuConcurrent] and P[GpuAtomics] < P[CpuOrder]). Since kernel sizes are big enough that the use of atomics would increase runtime of kernels, in accordance with conclusion-3, the CPU scheduling approach that uses atomics performs worse than the GPU scheduling approach that does not use atomics (P[CpuOrder] < P[GpuSynchronize]).

8. Recommendations

Finally, we recommend the following features that address the shortcomings of current GPU architectures with respect to real-time multitasking. We believe adding these features to upcoming architectures would improve the performance of future schedulers.

Hardware priority: The current programming system only guarantees that a program will function correctly; it is unable to guarantee execution order, which is crucial for scheduling time-sensitive real-time tasks with priorities.

Preemption: Kernels run to completion and GPUs do not have access to an interrupt mechanism that would allow us to put a running kernel to sleep and wake it up again later. Having a preemption mechanism is especially important for addressing the time requirements of very low-latency real-time tasks.

Programmable GPU scheduler: The GPU's hardware scheduler manages all blocks by automatically assigning them to compute resources, hiding all scheduling decisions from the user. A programmable scheduler would better address the requirements of real-time task by allowing fine-grained management of resources and reservation of specific resources for particular blocks.

Common time concept: GPUs have no support for a common time measure on the host (CPU) and the device (GPU). Having a common time concept across the CPU and GPU would allow measuring time more precisely since we would be able to synchronize time between the host and device.

Atomics across the CPU and GPU: GPU schedulers use extra scheduling messages to cope with race conditions that might arise when both the CPU and GPU attempt to update the zero-copy memory at the same time. If atomics across the host and device were supported, we would avoid the need for such messages, and the performance of GPU schedulers would be improved.

Solutions to technical difficulties: For performance reasons, the minor issues we have identified while implementing our schedulers (e.g., avoiding false dependencies caused by events, determining the end of concurrent kernels, flushing zero-copy memory, dealing with "delayed-signal" phenomena) would be better solved at the programming-system level rather than with the various workarounds we have developed in the course of this work.

9. Future Work

We would like to extend this work in four main ways, building on the first author's dissertation [6]:

- We wish to further explore and compare to related work in this area, including application-defined schedulers, programming frameworks, and driver/operating system support, with an emphasis on how they implement important parts of scheduling systems such as work division/time slicing, work queuing, and timing.
- We would like to extend Section 3 and Section 4 by providing implementation details of schedulers (e.g., adding pseudocode for GPU schedulers) and extended descriptions of a complete list of our methods. We hope to discuss these methods in the following contexts: the technical problem each method aims to address, the high-level solution to this problem, and the implementation details of the method.
- We hope to provide further discussion on the properties and results of individual cases that are used in the experiments and on the motivations and insights of recommendations that are made for upcoming GPU architectures.
- Finally, we would like to explore the implications of system-on-chip architectures for scheduling multiple tasks. In these architectures, since CPU and GPU use a common memory, there is no need to copy data, results, or messages back and forth between the host and device. This, in turn, has several implications for the scheduling strategies used in our study.

Acknowledgments

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

A. Pseudocodes of CPU Schedulers

CPU schedulers differ mostly from each other in the way they process tasks. Algorithms 1-4 provide comparison of how CPU schedulers process tasks by listing their pseudocodes. Each algorithm highlights the part of the implementation that realizes alternative strategies and shows the methods used by the scheduler to carry out these strategies.

Performing a task operation mentioned in pseudocodes consists of data copy or kernel execution commands preceded by synchronization and/or timing commands and followed by timing commands. Synchronization and timing commands are stream wait event and event recording commands, respectively. In addition, the list of ready-to-execute tasks mentioned in pseudocodes refers to tasks that have data to be processed and have no dependee or tasks that have a dependee and related data of this dependee is already processed. Hence, this list holds tasks whose first data in their data list is ready to be executed.

```
for first two tasks, t1 and t2, in a list of ready-to-execute tasks
sorted by their priority do
    assign host to device copy and kernel execution streams
    of t2 to s2 and s4, respectively;
    perform host to device copy operation of t1;
    // make sure execution order would follow
       issue order
    make s2 wait on the stop event of host to device copy of
    t1;
    perform host to device copy operation of t2 on s2;
    perform kernel execution operation of t1;
    // do not allow concurrent run
    make s4 wait on the stop event of kernel execution of t1;
    perform kernel execution operation of t2 on s4;
    perform device to host copy operation of t1;
    perform device to host copy operation of t2;
end
```

Algorithm 1: Pseudocode for CpuInterleave.

```
for first two tasks, t1 and t2, in a list of ready-to-execute tasks
sorted by their priority do
    assign host to device stream of t2 to s2;
    perform host to device copy operation of t1;
    // make sure execution order would follow
       issue order
    make s2 wait on the stop event of host to device copy of
    t1;
    perform host to device copy operation of t2;
    record the start event of kernel execution of t1;
    launch kernel execution of t1;
    record the start event of kernel execution of t2;
    launch kernel execution of t2;
    // issue the stop event of kernel execution
       of t1 after kernel launches to avoid the
       false dependency that breaks concurrent
       run
    record the stop event of kernel execution of t1;
    record the stop event of kernel execution of t2;
    perform device to host copy operation of t1;
    perform device to host copy operation of t2;
end
```

Algorithm 2: Pseudocode for CpuConcurrent.

```
for first two tasks, t1 and t2, in a list of ready-to-execute tasks
sorted by their priority do
    assign kernel execution stream of t1 to s3 and host to
    device copy stream of t2 to s2;
    perform host to device copy operation of t1;
    // make sure execution order would follow
       issue order
    make s2 wait on the stop event of host to device copy of
    t1;
    perform host to device copy operation of t2;
    record the start event of kernel execution of t1, which
    uses atomic version of the kernel;
    launch kernel execution of t1, which uses atomic version
    of the kernel, on s3;
    record the start event of kernel execution of t2, which
    uses atomic version of the kernel;
    launch kernel execution of t2, which uses atomic version
    of the kernel;
    // see changes the device makes on
       zero-copy memory without synchronization
    cudaSteamQuery(s3);
    while true do
        // poll on zero-copy memory to determine
           which kernel has finished first
        if finish flag of t1 is set then
            1st_finished_task = t1;
            2nd_finished_task = t2;
        end
        if finish flag of t2 is set then
            1st_finished_task = t2;
            2nd_finished_task = t1;
        end
    end
    assign device to host copy stream of 1st_finished_task to
    s7;
    // issue stop event of kernel execution of
       1st_finished_task before that of
       2nd_finished_task, handle the
       blocking-query problem (caused by the
       above stream query command) by recording
       this event on device to host copy stream
       instead of kernel execution stream
    record the stop event of kernel execution of
    1st_finished_task, which uses atomic version of the
    kernel, on s7;
    record the stop event of kernel execution of
    2nd_finished_task, which uses atomic version of the
    kernel;
    // issue device to host copy of
       1st_finished_task before that of
       2nd_finished_task
    perform device to host copy operation of
    1st_finished_task on s7;
    perform device to host copy operation of
    2nd_finished_task;
end
```

Algorithm 3: Pseudocode for CPUORDER.

```
for each task, t, in a list of ready-to-execute tasks sorted by
their priority do
    perform host to device copy operation of t;
    perform kernel execution operation of t;
    perform device to host copy operation of t;
end
```

Algorithm 4: Pseudocode for CPUTRADITIONAL.

References

[1] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[2] B. Budge, T. Bernardin, J. A. Stuart, S. Sengupta, K. I. Joy, and J. D. Owens. Out-of-core data management for path tracing on hybrid resources. *Computer Graphics Forum*, 28(2):385–396, 2009.

[3] G. A. Elliott and J. H. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. In *International Conference on Real-Time and Network Systems*, pages 197–206, 2010.

[4] F. Ino, A. Ogita, K. Oita, and K. Hagihara. Cooperative multitasking for GPU-accelerated grid systems. In *International Conference on Cluster, Cloud and Grid Computing*, pages 774–779, 2010.

[5] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: GPU scheduling for real-time multi-tasking environments. In *USENIX Annual Technical Conference*, 2011.

[6] P. Muyan-Özçelik. *Running Real-time Tasks on Embedded Systems Using GPU computing*. PhD thesis, Dept. of Computer Science, University of California, Davis, 2014.

[7] P. Muyan-Özçelik, V. Glavtchev, J. M. Ota, and J. D. Owens. Real-time speed-limit-sign recognition on an embedded system using a GPU. In *GPU Computing Gems*, pages 497–516. 2011.

[8] NVIDIA. Fermi compute architecture whitepaper. `http://www.nvidia.com/object/IO_89570.html`, 2009.

[9] NVIDIA. Kepler GK110 architecture whitepaper. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012.

[10] NVIDIA. CUDA parallel computing platform. `http://www.nvidia.com/object/cuda_home_new.html`, 2015.

[11] NVIDIA. Tegra mobile processors. `http://www.nvidia.com/object/tegra.html`, 2015.

[12] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics*, 29:66:1–66:13, 2010.

[13] H. Peters, M. Koper, and N. Luttenberger. Efficiently using a CUDA-enabled GPU as shared resource. *International Conference on Computer and Information Technology*, pages 1122–1127, 2010.

[14] S. Rennich. CUDA C/C++ streams and concurrency. `http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf`, 2011.

[15] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *ACM Symposium on Operating Systems Principles*, pages 233–248, 2011.

[16] M. Steinberger, B. Kainz, B. Kerbl, S. Hauswiesner, M. Kenzel, and D. Schmalstieg. Softshell: Dynamic scheduling on GPUs. *ACM Transactions on Graphics (TOG)*, 31(6):161:1–161:11, Nov. 2012.

[17] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan. GRAMPS: A programming model for graphics pipelines. *ACM Transactions on Graphics*, 28:4:1–4:11, 2009.

[18] S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the GPU. In *High Performance Graphics*, pages 29–37, 2010.