An Evaluation of the NVIDIA TX1 for Supporting Real-time Computer-Vision Workloads*

Nathan Otterness[1], Ming Yang[1], Sarah Rust[1], Eunbyung Park[1],
James H. Anderson[1], F. Donelson Smith[1], Alex Berg[1], and Shige Wang[2]
[1]Department of Computer Science, University of North Carolina at Chapel Hill
[2]General Motors Research

Abstract

Autonomous vehicles are an exemplar for forward-looking safety-critical real-time systems where significant computing capacity must be provided within strict size, weight, and power (SWaP) limits. A promising way forward in meeting these needs is to leverage multicore platforms augmented with energy-efficient graphics processing units (GPUs) as accelerators. Such an approach is being strongly advocated by NVIDIA, whose Jetson TX1 board is currently a leading multicore+GPU solution marketed for autonomous systems. Unfortunately, no study has ever been published that expressly evaluates the effectiveness of the TX1, or any other comparable platform, in hosting safety-critical real-time workloads. In this paper, such a study is presented. Specifically, the TX1 is evaluated via benchmarking efforts, black-box evaluations of GPU behavior, and case-study evaluations involving computer-vision workloads inspired by autonomous-driving use cases.

1 Introduction

Safety-critical embedded systems are undergoing an evolution towards greater autonomy. This evolution is perhaps best exemplified by current trends in the automotive industry. In the coming years, vehicles are envisioned that will have increasingly sophisticated decision-making capabilities. Such capabilities are expected to continually evolve, ultimately resulting in fully autonomous vehicles that operate themselves in all traffic conditions with no human intervention.

The push towards full autonomy will not be easy because it will require resolving two conflicting realities. On the one hand, increased autonomy will require *significant computing capacity*. On the other hand, the need for efficient designs will give rise to strict limits on both *monetary cost* and *size, weight, and power* (*SWaP*). The latter reality seriously impinges upon the former.

Given this dilemma, the choice of hardware platform to utilize in autonomous vehicles is not straightforward. One choice that is receiving considerable attention today is the usage of energy-efficient multicore platforms equipped with *graphics processing units* (*GPUs*) that can speed mathematical computations inherent to signal processing, image processing, motion planning, *etc.* Indeed, various GPU manufacturers have taken note of the emerging market with respect to autonomous vehicles and have begun to offer GPU-augmented multicore platforms specifically catered to embedded use cases. Perhaps the most prominent such platform today is NVIDIA's Jetson TX1, which is marketed as *"The embedded platform for autonomous everything"* [20].

Despite this marketing slogan, no published study exists that expressly evaluates the effectiveness of the TX1, or any other comparable energy-efficient embedded GPU platform, in hosting safety-critical real-time workloads. This is a serious concern, because safety-critical systems require *certified* system designs. *How can certification be possible if the characteristics of the underlying hardware platform that impact predictable execution are not well understood?*

Motivated by this question, we present in this paper an evaluation of the TX1 that seeks to determine its suitability for hosting safety-critical workloads arising in autonomous-driving use cases. Our evaluation of the TX1 is a first step in a broader project, the goal of which is to evaluate the efficacy of various proposed hardware platforms for enabling autonomous driving. We chose to consider the TX1 first because it is arguably the most highly touted GPU-augmented multicore platform today for autonomous-driving use cases.

Basic TX1 overview. Before describing the nature of our evaluation, and some of our conclusions, we first present a brief overview of the TX1 (more details are provided later).

As seen in Fig. 1, the Jetson TX1 is a single-board computer containing a quad-core 64-bit ARM CPU, 4 GB of DRAM memory, and an integrated GPU. As explained in more detail in Sec. 2.1, *integrated* GPUs share DRAM memory with the host CPU platform, in contrast to *discrete* GPUs, which have private DRAM memory. Integrated GPUs are the *de facto* choice in embedded applications where SWaP is a concern. The TX1's monetary cost is quite modest, approximately 600 USD [21].

Contributions. Our ultimate conclusions regarding the TX1's usefulness in autonomous-driving use cases are based

*Work supported by NSF grants CPS 1239135, CNS 1409175, CPS 1446631, and CNS 1563845, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and funding from General Motors.

on case-study experiments in which achievable frame rates were measured for computer-vision workloads motivated by such use cases. Our focus on computer vision is justified by the fact that sensing through cameras is commonly used to realize semi-autonomous features in production vehicles today. Furthermore, similar computations must be supported under other sensing methods, such as LIDAR.

In order for these case studies to be meaningful, it is important for the TX1's GPU to be managed in a way that is reasonable for a safety-critical application. Thus, before presenting our case-study experiments, we first present results obtained by running benchmarks that seek to resolve basic GPU management options.

One such option that we consider is a feature inherent in integrated GPUs called *zero-copy memory*. Zero-copy memory eliminates the need for copying data to and from DRAM associated with the GPU, instead allowing both CPU and GPU components of GPU-using programs to share memory space. Conventional GPU-using programs, which are written using NVIDIA's CUDA API, function as if the GPU is discrete and the zero-copy option is not available, but this may not be the best choice for an integrated platform like the TX1. While it ostensibly would be preferable to utilize zero-copy memory, our benchmark experiments reveal that the situation regarding zero-copy is more nuanced.

Another option that we consider is whether *GPU co-scheduling* should be allowed, *i.e.*, whether different tasks should be able to access the GPU concurrently. In most prior work on real-time GPU management, co-scheduling is forbidden due to concerns that concurrent accesses may adversely interfere with each other. However, on a single-board platform with a single, less-capable GPU, co-scheduling may be desirable if GPU computations commonly use only a fraction of its computational capacity, leaving it consistently underutilized. To elucidate this issue, we conducted additional benchmark experiments where different GPU computations were concurrently launched by different tasks. These experiments show that some co-scheduling might be preferable but is also unlikely to provide benefits beyond what existing real-time GPU management systems already offer.

Informed by these benchmarks, we conducted case-study experiments in which different computer-vision computations had to be supported simultaneously on the TX1. We are interested in supporting multiple computations because any realistic autonomous-driving framework would have to multiplex the processing from different sensor feeds at the same time. The specific computations that we considered were the CaffeNet image-classification system and a road sign recognition application. Across all of our experiments, these computations typically were able to sustain frame rates in the range of $[20, 30]$ frames per second. While impressive for an embedded device, these rates still may not be high enough for autonomous-driving use cases, especially when worst-case provisioning must be used.

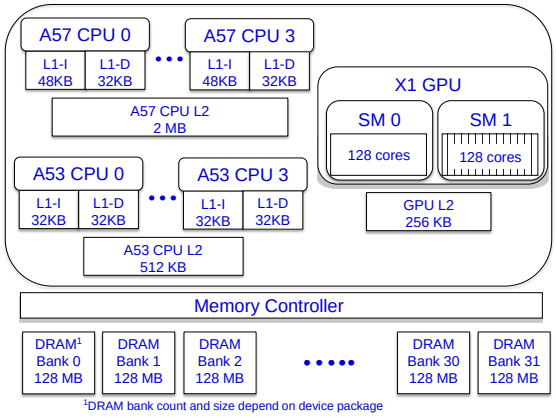

Figure 1: Jetson TX1 architecture.

Organization. In rest of the paper, we discuss relevant background information (Sec. 2), describe our benchmarks (Sec. 3), present the results of our benchmark experiments (Secs. 4 and 5), discuss our case study experiments (Sec. 6), and conclude (Sec. 7).

2 Background

In this section, we provide needed background on the NVIDIA Jetson TX1 and GPU programming fundamentals using NVIDIA's CUDA API. We also discuss prior related work on real-time GPU management frameworks.

2.1 The NVIDIA Jetson TX1

As illustrated in Fig. 1, the TX1 employs an SOC (system-on-chip) design that incorporates a quad-core 1.91 GHz 64-bit ARM A57 processor and an integrated Maxwell GM20B GPU. The CPUs share a 2-MB L2 cache. The GPU has 256 cores configured on two streaming-multiprocessor (SM) controllers, has a 256-KB L2 cache, and provides up to 512 32-bit GFLOPS. The TX1 is a "big-little" platform in which an additional lower power, lower performance quad-core ARM A53 is provided on chip, but is not directly accessible to software and is only activated in low-power modes. The ARM CPUs and the GPU share 4 GB of 1600 MHz DRAM memory partitioned into 32 banks.

The TX1 features an integrated GPU. Such a GPU tightly shares DRAM memory with CPU cores, typically draws between 5 and 15 watts, and requires minimal cooling and additional space. The alternative to an integrated GPU is a discrete GPU. Discrete GPUs are packaged on adapter cards that plug into a host computer bus, have their own local DRAM memory that is completely independent from that used by CPU cores, typically draw between 150 and 250 watts, need active cooling, and occupy substantial space.

2.2 CUDA Programming Fundamentals

The following is a high-level description of CUDA, the API for GPU programming provided by NVIDIA.

A GPU is fundamentally a co-processor that performs

operations requested by CPU programs. CUDA programs use a set of C or C++ library routines to request GPU operations that are implemented by a combination of hardware and device-driver software. The typical structure of a CUDA program is as follows: **(i)** allocate GPU-local (device) memory for data; **(ii)** use the GPU to copy data from host memory to GPU device memory; **(iii)** launch a program—called a *kernel*—to run on the GPU cores to compute some function on the data; **(iv)** use the GPU to copy output data from the device memory back to the host memory; **(v)** free the device memory. When invoking a CUDA kernel, the programmer specifies the number of GPU threads to use during the kernel's execution and how the threads are organized into groups called *thread blocks*. Having multiple threads executing the kernel enables the significant parallelism afforded by GPUs. Kernel launches are always asynchronous, requiring the CPU process to explicitly wait for them to complete.

On integrated GPUs, CUDA provides a *zero-copy* option where programs can simply pass a pointer to shared memory where data used by a kernel is located—that is, explicit copying from CPU-local memory to GPU-local memory is avoided. CUDA also supports a different memory-access mechanism called *unified memory* on both discrete and integrated GPUs. Unified memory is similar to zero-copy, as a single memory pointer can be used in both CPU and GPU code. The difference between unified and zero-copy appears during kernel execution, where, in the case of unified memory, the GPU driver transparently transfers data on-demand between CPU-local memory and GPU-local memory.

CUDA operations pertaining to a given GPU are ordered by associating them with a *stream*. By default, there is a single stream for all programs that share a GPU, but multiple streams can be optionally created. Operations in a given stream are executed in FIFO order, but the order of execution across different streams is determined by the GPU scheduling in the device driver. They may execute concurrently (or out of request order with respect to other streams).

Programmers can think of a GPU as being abstractly composed of one or more *copy engines* (*CEs*) that implement transfers of data between host memory and device memory, and an *execution engine* (*EE*) (consisting of many parallel processors) that executes GPU kernels. The TX1 has two CEs, one for each direction of transfer. EEs and CEs operate concurrently. When there are multiple streams, kernels and copy operations from different streams can also operate concurrently depending on the GPU hardware. To the best of our knowledge, complete details of kernel attributes and policies used by NVIDIA to schedule kernels and copy operations is not available.

2.3 Related Work

The black-box nature of GPU programming has limited both the scheduling and analysis techniques available for real-time GPU usage. As a result, much prior work treats a single GPU as an atomic entity—a real-time task locks an entire GPU, or individual EEs or CEs, for the duration of any GPU computation. Such an approach is taken in TimeGraph [12], RGEM [11], GPUSync [8], and several other frameworks [25, 26, 27, 29]. The viewpoint taken in all of this work is that *GPU co-scheduling must be avoided because concurrently executing kernels might adversely interfere with each other*. However, we are aware of no work directed at real-time systems in which such interference is actually demonstrated or its effects quantified.

In a precursor to this paper, our group conducted an investigation of the high-level effects of uncontrolled co-scheduling on the execution times of a variety of image-processing benchmarks [23]. We conducted this work using both the NVIDIA TX1 and TK1 (a similar, but weaker, single-board computer). This work found that unmanaged co-scheduling can lead to improved average-case performance. However, we did not examine in depth how this benefit was achieved or what the limitations of it were.

Work has also been directed at splitting GPU tasks into smaller sub-tasks to approximate preemptive execution or improve utilization [2, 11, 16, 31]. A framework called Kernelet [30] falls into this category, but is of particular interest to us due to the fact that GPU co-scheduling is considered in order to improve utilization. Kernelet, however, requires heavy instrumentation and does not consider co-scheduling unmodified workloads. Additionally, the developers of Kernelet do not provide an in-depth investigation into the GPU's actual behavior or interference effects during co-scheduling, which, in fairness, was not one of their main objectives. Others have published further techniques for managing or evaluating GPU hardware resources, including the cache [17, 18, 28] and direct I/O communication [1].

In addition to GPU management techniques, a significant body of work has been directed at timing analysis of GPU workloads. These have included statistical techniques applied to measurements obtained on hardware [5, 6, 7], techniques for estimating worst-case performance using formal models [3, 4], and one recent work that used simulation to identify and remedy performance bottlenecks [10].

3 Benchmark Programs

In our experiments, we made use of a variety of benchmark workloads, which are summarized in this section. During all of our experiments, each instance of a benchmark was pinned to a single CPU core, performed memory copies asynchronously, and was configured to block while waiting to synchronize with the GPU. For benchmarks requiring image inputs, we pre-loaded the input data into memory before beginning measurements. Unless otherwise noted, all of our benchmarks used CUDA version 8.0.

Synthetic benchmarks. We used three synthetic benchmarks for the purpose of measuring GPU performance. We

have made the source code for these benchmarks available online.[1]

- **Random Memory Walk**: Creates 64 CUDA threads, grouped into two blocks of 32 threads each, that access 256 MB of GPU memory in a random pattern. Each thread begins at a different offset in the random walk. This benchmark can be configured to place the array in zero-copy, unified, or traditional GPU memory. It was created to measure memory performance with as little cache benefit as possible.
- **In-Order Memory Walk**: Identical to the random memory walk, but accesses the array of memory in order, with the different threads spaced equally across the array. In contrast to the random memory walk, this benchmark was created to compare performance with full caching benefits.
- **Convbench (CONV)**: Executes convolutional neural-network layers as used in image recognition. The input is a 227×227 color image; the output is a matrix of neural-network parameters.

Image-processing benchmarks. We considered three benchmarks that carry out image-processing tasks using the GPU. Stereo Disparity was adapted from the CUDA samples distributed by NVIDIA [19]. CaffeNet is part of the Caffe framework, an open-source tool available online.[2]

- **Stereo Disparity (SD)**: Extracts 3D depth information from 2D images taken with a stereo camera. The input consists of left and right 640×533 color images; the output is a 640×533 grayscale image.
- **CaffeNet**: A neural network trained to classify images based on AlexNet [15], and running under the Caffe deep learning framework [9]. The input to this benchmark is one or more images, and the output is a list of classification probabilities.
- **Road-Sign Recognition**: This is an application intended for use in semi-autonomous cars. It identifies road signs in streams of images. (This is proprietary code made available by an industry partner.)

4 Memory Considerations

GPU applications for autonomous vehicles need to quickly retrieve data from sensors or storage, and also quickly transfer data to or from GPU memory. In conventional CUDA programs, memory is explicitly copied from the CPU to the GPU portions of DRAM and vice versa. On integrated GPUs like the TX1, the CPU and GPU have unified DRAM memory, giving CUDA programs a wider range of mechanisms to retrieve data from the GPU, or to transfer data to it. Before describing our experimental methods, we first give an overview of the different memory-transfer methods available to CUDA programs.

Traditional memory. Conventional CUDA programs use traditional memory to store and access memory, where data must be explicitly copied from CPU to GPU portions of DRAM. The GPU may be able to perform additional optimizations using traditional memory, but the time to copy large amounts of the same data between the CPU and GPU memory, in addition to greater programming complexity, can be a drawback.

Zero-copy memory. With zero-copy memory, the CPU and the GPU can access the same memory area, avoiding GPU memory allocations and data copies between CPU and GPU memory. In practice this advantage may be offset by claims that zero-copy memory accessed by the GPU bypasses all caching. Strangely enough, there seems to be no mention of this in official CUDA documentation, but older presentations and forum posts by NVIDIA staff members have indicated this is the case due to their consistency model and concerns about maintaining cache coherence [13, 24].

Unified memory. As mentioned in Sec. 2.2, unified memory is similar to zero-copy since they both use the same memory pointer between the CPU and GPU, but in reality it presents somewhat of a hybrid between the simplified programming of zero-copy memory and the caching benefits of traditional memory. This is achieved by the GPU driver transparently transferring memory on-demand between the CPU and GPU, whenever buffers are accessed [14].

Evaluation of memory transfer mechanisms. Our first set of experiments used the random and in-order memory walk benchmarks described in Sec. 3. We designed these synthetic benchmarks to facilitate switching between memory transfer mechanisms, which would have required significant code modifications in our image-recognition programs. Additionally, using memory-focused benchmarks allowed us to better isolate memory activity by eliminating other non-memory-related computations.

Execution times were recorded on the GPU for each block of 32 threads. Each thread performs a constant number of steps (approximately 256,000) in the random or in-order walk within a single measurement. Recording time on the GPU itself avoids potential inaccuracies present when timing entire kernels, which is the granularity provided by NVIDIA's standard debugging tools. Times were measured by reading the GPU's `globaltimer` register, which maintains a 64-bit count of nanoseconds.

Differences between CUDA versions. We initially conducted these experiments using CUDA version 7.0 on Linux

[1] SD and CONV are available at `https://github.com/yalue/PeriodicTaskReleaser`. The memory walk benchmarks are available at `https://github.com/yalue/MicrobenchmarksCUDA`.

[2] `https://github.com/BVLC/caffe`.

	CUDA 7.0	CUDA 8.0
Traditional memory	242.5	679.34
Zero-copy memory	672.6	680.2
Unified memory	674.3	680.1

Table 1: Average-case times (in ms) for the random walk benchmark for CUDA 7.0 and 8.0.

	CUDA 7.0	CUDA 8.0
Traditional memory	4.4	3.1
Zero-copy memory	76.1	3.2
Unified memory	4.4	3.2

Table 2: Average-case times (in ms) for the in-order walk benchmark for CUDA 7.0 and 8.0.

for Tegra (L4T)[3] version 24.1. Upon updating our system to CUDA 8.0 and L4T 24.2, we observed significant changes in all of our measurements. Tbls. 1 and 2 show the results of our memory benchmarks.

Obs. 1. Random traditional memory accesses are slower under CUDA 8.0 compared to CUDA 7.0.

This observation is supported by Tbl. 1. The most surprising change due to transitioning from CUDA 7.0 to 8.0 was that the newer version of CUDA caused traditional memory to become around three times *slower* in the random walk.

Obs. 2. Under CUDA 8.0, unified and zero-copy memory perform nearly identically, which was not the case under 7.0.

This observation is supported by Tbl. 2 and by Figs. 2 and 3, which provide cumulative distribution functions (CDFs) based on recorded kernel times for the walk through the array. Under CUDA 7.0, unified memory was slower than zero-copy during the random walk, but as fast as traditional memory during the in-order walk. Under CUDA 8.0, in-order zero-copy memory accesses see a significant improvement, even though zero-copy memory was earlier claimed to bypass the cache. We speculate that the changes in memory performance from CUDA 7.0 to CUDA 8.0 are the result of optimizing for a use case that emphasizes sequential (streaming) memory accesses at the price of incurring some performance degradation for arbitrary access patterns.

Data transfer times. Applications using zero-copy memory do not need to explicitly transfer data to the GPU. This leads to a potential trade-off between memory access times under zero-copy memory and increased data transfer times under traditional memory because of copy times. We conducted a short experiment to measure the CPU-GPU data transfer rate on the TX1 to obtain an estimate of when this trade-off may favor using one method over the other. Our experimental setup was very simple: we copied data ranging from 1 KB to 1GB from the CPU to the GPU. The data transfer times were obtained from NVIDIA's `nvprof` tool.

The result of this experiment was that it is possible to transfer about 8 megabytes of data per millisecond using

[3]Linux for Tegra is NVIDIA's official build of the Linux kernel and root file system for the TX1.

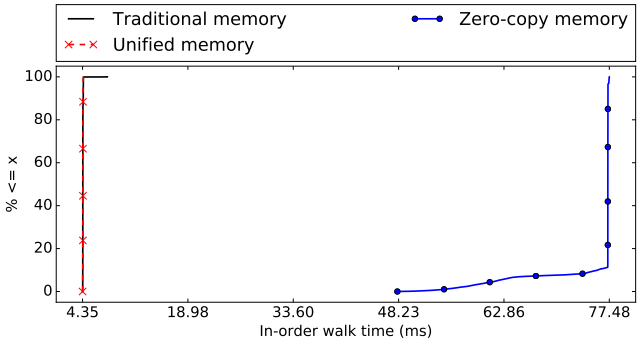

Figure 2: CDFs of memory access times during the in-order walk benchmark under CUDA 7.0. Note: traditional and unified memory overlap.

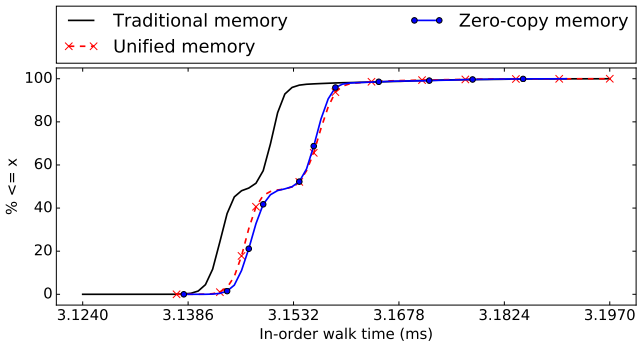

Figure 3: CDFs of memory access times during the in-order walk benchmark under CUDA 8.0. Note: zero-copy and unified memory overlap.

traditional GPU memory on the TX1. Most of our benchmarks have rather small input sizes: the combined size of SD's 640×533 color image inputs is 2.6MB, and CaffeNet's inputs, 256×256 images, will be even smaller. Therefore, we believe that data transfer times are unlikely to play a significant role in the overall time of our subsequent experiments, which are usually between 15-30 milliseconds.

Conclusions regarding memory-management best practices. Based on our results, using traditional memory under CUDA 8.0 will not adversely affect performance when compared to unified or zero-copy memory, unless data transfer times dominate memory access times. More importantly, our experiments revealed that CUDA version changes can alter behavior as fundamental as memory access. For safety-critical systems, it is therefore necessary that all analysis and development be carried out under a single version of CUDA.

5 GPU Scheduling

As discussed in Sec. 2, the overarching trend in real-time GPU management has been to provide exclusive GPU access to only one CPU program at a time. This approach has been motivated by the closed-source and undocumented nature of GPU hardware and drivers, meaning the only mechanism sure to guarantee non-interference between GPU workloads was to eliminate the possibility entirely. While this may

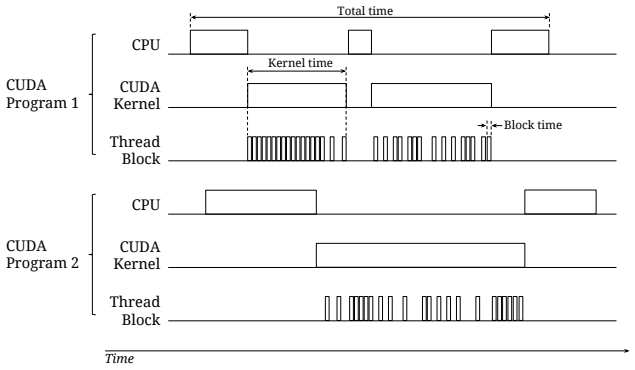

Figure 4: Diagram illustrating the relation between CUDA programs, kernels, and thread blocks for two co-scheduled programs.

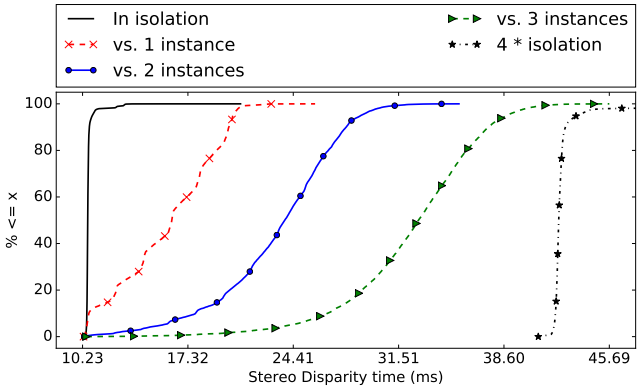

Figure 5: CDFs of total time for SD with up to four co-scheduled SD instances. The WCET for the "4 * isolation" curve is 110 ms.

be reasonable on a multi-GPU system, more justification is required for any utilization loss on the single-GPU TX1. Towards this end, we performed a black-box examination of GPU scheduling on the TX1.

Our examination involved co-scheduling image-processing benchmarks and recording timing information. The results shown below primarily involve experiments conducted using the SD benchmark, but CONV was also considered in one experiment. These benchmarks were described earlier in Sec. 3.

GPU scheduling hierarchy. Fig. 4 depicts how GPU work is subdivided and scheduled, and is labeled with specific time ranges. This figure shows which portions of two co-scheduled CUDA programs may be active at a given time. At the top level of the hierarchy are CUDA programs, containing both CPU and GPU portions. For our benchmarks, we use *total time* to denote the time required for an entire CUDA program, starting from the beginning of the initial CPU computation and ending with the completion of the final CPU computation. By definition, total time includes any time spent executing or waiting for GPU work. CPU and GPU execution may overlap within a single CUDA program, even though this is not represented in Fig. 4. In our benchmarks, total time refers to a single invocation of the benchmark code (*e.g.*, the processing of one image) rather than the entire lifetime of the CUDA program.

The second level of the hierarchy in Fig. 4 is the CUDA kernel. One or more kernels may be invoked during the course of a CUDA program. *Kernel time* starts when a single kernel is submitted to some level of the black-box GPU kernel-scheduling queue and ends when all threads have completed and the driver unblocks the CPU waiting to synchronize with the GPU. Kernel times can be measured using NVIDIA's `nvprof` tool. In this work, we use "kernel time" to specifically refer to the time required by a *single* CUDA kernel, as denoted in Fig. 4. We call the sum of all kernel times *total kernel time* instead.

The smallest unit of scheduling depicted in Fig. 4 and considered herein is the *thread block*. Recall from Sec. 2.2 that the number of GPU threads for each CUDA kernel and their grouping into blocks is specified at runtime and is user-controlled. In our image-processing benchmarks, thread blocks are used to logically subdivide threads among different chunks of input data. In GPU scheduling, however, thread blocks are not merely a means for the programmer to organize threads into groups. As shown in Fig. 4, thread blocks are treated as units of work to be scheduled. We call the runtime of a single thread block *block time*. In our experiments, block times were measured on the GPU by recording the value of the `globaltimer` register at the start and end of each thread block.

Performance impact of uncontrolled co-scheduling. Our first experiment was devised to evaluate how much of a benefit GPU co-scheduling can produce on the TX1. In this experiment, we ran up to four concurrent instances of the SD benchmark. Each instance was pinned to a separate CPU core, had all of its pages locked into physical memory, and ran for as many iterations as possible for 10 minutes. In this experiment, we measured total time (as defined above), in order to capture all possible benefits due to co-scheduling.

Obs. 3. GPU co-scheduling can lead to reduced total time, when compared to sequential execution.

This observation is supported by Fig. 5, which provides CDFs based on recorded total times for a single instance of SD executing alongside a varying number of SD competitors. The "In isolation" curve corresponds to the case in which no competitors exist, and the subsequent curves correspond to cases where one, two, or three competitors exist. The "4 * isolation" curve is not a real measurement, but was obtained by scaling up the isolation curve by a factor of four. It is included to provide a rough estimate of the time necessary to complete four instances if all four benchmarks were forced to run sequentially. In the co-scheduling case, the total time necessary for four instances to complete is represented by the "vs. 3 instances" curve, which represents the time a single instance takes to complete when running concurrently with three competitors. The difference in median times between these two curves shows that we can save 10 milliseconds on

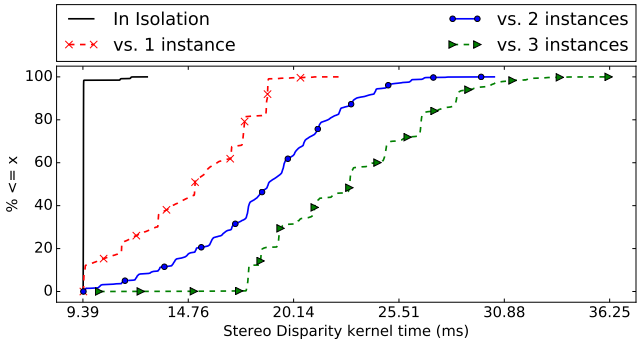

Figure 6: CDFs of kernel time with up to four co-scheduled SD instances.

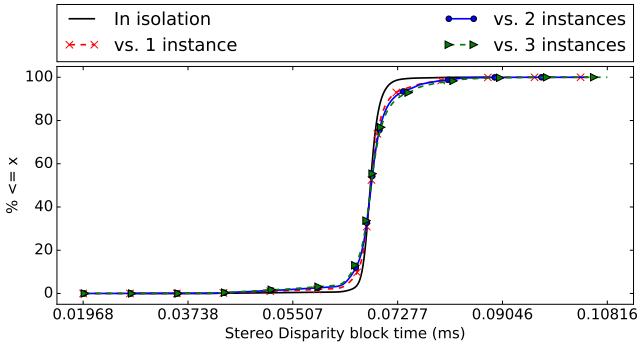

Figure 7: CDFs of block time with up to four co-scheduled SD instances.

average. The measured worst-case execution time (WCET) shows an even greater benefit due to co-scheduling: approximately 70 milliseconds.

In prior work, we carried out a larger variety of similar experiments, and the results shown in Fig. 5 agree with our previous findings [23]. Given these results, one may ask why co-scheduling should be avoided? In response, we first stress that our experiments only covered a small subset of infinitely many possible workloads, and we refer to our prior work for specific examples where co-scheduling may be undesirable [23]. Secondly, a deeper inspection of co-scheduling reveals internal policies that limit the potential benefits of GPU co-scheduling on performance.

The impact of co-scheduling on kernel times. The results in prior paragraphs established that co-scheduling can be beneficial in some cases, but gave no indication of the root cause for the increasing total times during co-scheduling. The next step in our examination of the scheduling hierarchy in Fig. 4 involved measuring kernel times in the presence of co-scheduling.

Obs. 4. Co-scheduling affects kernel times similarly to total times.

This observation is supported by the kernel-time CDFs shown in Fig. 6. Like total times, kernel times still expanded in the presence of co-scheduling, but not to the point where any benefit of co-scheduling no longer exists. The kernel-time CDFs are less distinct and more noisy than the total-time CDFs, but we have no way to control where individual kernels actually overlapped on the GPU. It is likely that some kernels, even with concurrent benchmarks, were able to use the GPU with little to no competition, leading to the tighter grouping of curves in Fig. 6.

The impact of co-scheduling on block times. Since total times and kernel times were both inflated by co-scheduling, we decided to investigate the next lower level in the scheduling hierarchy: block times. This experiment followed an identical setup to the total-time and kernel-time experiments, with the sole difference that the CPU portion of the program logged the block times recorded on the GPU.[4]

Obs. 5. Block times are minimally affected by co-scheduling.

This observation is supported by the block-time CDFs given in Fig. 7. This figure shows that block times are virtually unaffected by co-scheduling. Especially striking is the fact that the median block time, approximately 70 microseconds, was indistinguishable among all scenarios. We already observed that kernel times differ substantially when co-scheduled (see Fig. 6). Nominally, kernel time should be the sum of all block times, but if block times do not change under co-scheduling, how can kernel time change? We can only conclude that the majority of the slowdown due to co-scheduling is due to inter-block scheduling delays. We examined this in detail in our next experiment.

Block scheduling behavior. To shed light on the behavior of the GPU scheduler, we conducted an additional experiment in which actual scheduling timelines were recorded. Rather than recording block durations, we instead recorded the start and end time of each block. We present only results where two instances of SD and two instances of CONV were co-scheduled, but we observed similar effects regardless of the collection of benchmarks. After obtaining measurements, we looked for an interval of time where kernel times from the four programs overlapped, and then generated a timeline of active blocks in this particular interval.

Fig. 8 contains one such timeline of the active block count. The horizontal axis represents time measured in GPU clock ticks (recorded from the GPU's `clock64` register). The vertical axis is subdivided into four sections, each of which contains a line representing the number of currently running blocks for each of the co-scheduled programs on one of the two SMs of the TX1's GPU. This line ranges between zero to eight currently running blocks for SD, and zero to two blocks for CONV. We used the `nvprof` tool to confirm that Fig. 8 contains a *single* kernel for each of the programs, so any activity in this figure is due to inter-block scheduling, and not kernels starting or ending.

Obs. 6. Co-scheduled GPU programs from different pro-

[4]The SD kernel was instrumented to record block times for all experiments but they were not written to logs when measuring total time to avoid additional overheads.

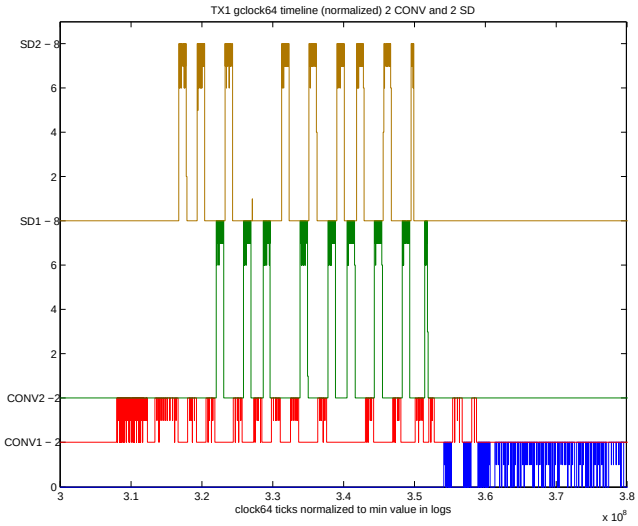

Figure 8: Timeline of block assignments to a single SM.

cesses are not truly concurrent, but are multiprogrammed instead.

This observation is supported by Fig. 8, which shows that even though multiple blocks from a single kernel are scheduled concurrently, *blocks from different kernels never overlap*. CONV uses blocks containing 1,024 threads, and SD uses blocks containing 256 threads, so up to 2,048 threads were in some phase of execution at any given time.[5]

We conducted other experiments with different combinations of CUDA programs, but we always observed identical block-scheduling behavior to that shown in Fig. 8 (the alignment of blocks as depicted in Fig. 4 also reflects this). If we assume this is always the case, then we can conclude that co-scheduled GPU workloads will never result in improved GPU kernel execution times. Put another way, GPU co-scheduling only improves times by overlapping GPU tasks with CPU computations. This is likely still beneficial to overall utilization, but existing real-time GPU management systems already feature the ability to overlap CPU and GPU execution [8] while offering more control than the version of co-scheduling discussed here.

Conclusions about GPU co-scheduling. It may be worthwhile to consider co-scheduling specific sets of CUDA programs on the GPU. While this may lead to slower or less predictable total times in individual programs, we observed no cases where the co-scheduled WCET exceeded the sum of the WCETs in isolation. On the other hand, our observations show that co-scheduling never caused true concurrent execution on the GPU. This implies that any benefit from co-scheduling is due solely to overlapping CPU-GPU computations or reduced queueing time in the GPU driver's kernel-scheduling queue.

[5] 2,048 is the maximum number of threads concurrently schedulable on one SM in the TX1. In future work, we may investigate even smaller units of scheduling, called *warps*, in order to gain insight about how individual threads within blocks are scheduled.

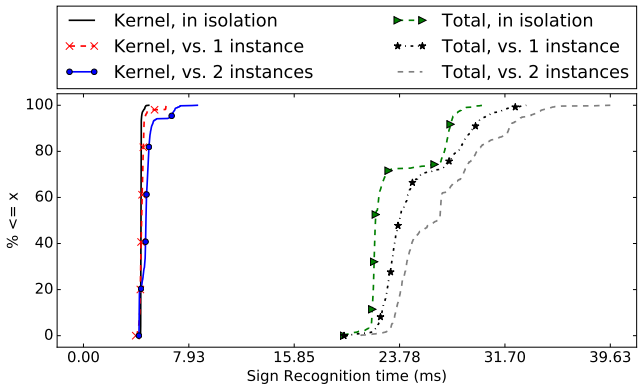

Figure 9: CDFs of both kernel and total times for up to three co-scheduled sign-recognition instances.

6 Case Study: Image-Processing Tasks

Our earlier experiments revealed that traditional GPU memory should still be used on the TX1, and that co-scheduling can be considered to improve utilization, but these results are meaningless if the TX1 is incapable of predictably sustaining an autonomous-vehicle workload in the first place. We addressed this final point by running workloads performing computer-vision tasks, which are fundamental to camera-based autonomous vehicle navigation. Our benchmarks for these experiments included both CaffeNet and the sign-recognition workload described in Sec. 3.

In our experiments, we focused on achieving 30 frames per second under the assumption that this rate will be easily provided by common cameras. We intended this merely as an easy point of reference for our benchmarks rather than as a requirement for autonomous driving. It is possible that real-life autonomous or semi-autonomous driving systems require either lower or higher frame rates, and specifics probably differ between vehicle manufacturers.

Sign-recognition performance. In this experiment, we ran up to three co-scheduled sign recognition instances. We recorded both total times and kernel times experienced by a single instance in each scenario.

Obs. 7. Up to two instances of the sign recognition task can reliably process 30 frames per second.

Fig. 9 supports this observation. We justify our claim of reliability based on the observed WCET with a single competitor being under 33 milliseconds. The median time per frame is sufficient for 30 frames per second even with three co-scheduled instances, but the WCET is higher in this case. The good co-scheduling results for the sign-recognition benchmark are probably due to its relatively short kernel times which, unlike our other benchmarks, account for less than half of the total time and therefore are more likely to overlap with CPU time from other co-scheduled instances.

Overall, sign recognition performed fairly well on the

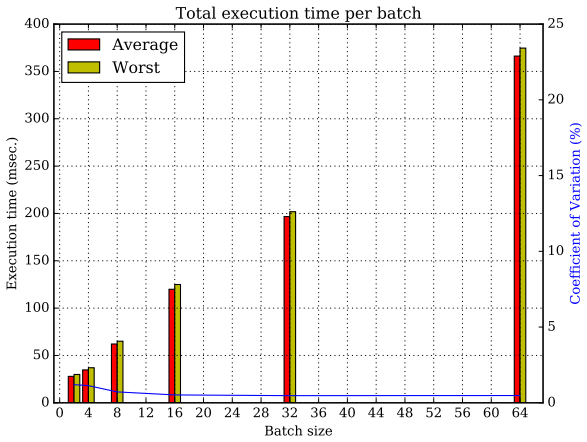

Figure 10: Total execution time per batch (response time per image) with different batch sizes of CaffeNet, along with the coefficient of variation.

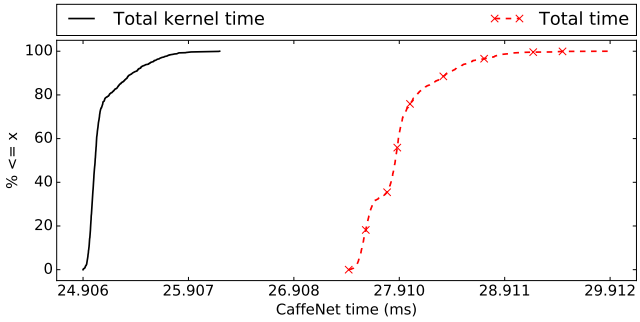

Figure 11: CDFs of total kernel time and total time for CaffeNet with a batch size of two.

TX1, even in the presence of some competition. This, however, is only one of several image-processing tasks that must be carried out in an autonomous driving scenario. In order to consider more general-purpose image classification (such as pedestrian detection), we also evaluated the CaffeNet benchmark's performance.

CaffeNet performance. CaffeNet has a much higher memory requirement than sign recognition, preventing us from co-scheduling multiple instances of it. However, CaffeNet has a different mechanism for controlling parallelism: it can be configured to classify more than one image at once. The parameter controlling this is called *batch size*: with a batch size of k, k images are processed at once. Like co-scheduling, parallelism can be increased to improve throughput at the expense of latency by increasing batch size.

In an automotive use case, an increased batch size will only be useful if there are multiple image inputs arriving simultaneously. Were only a single input stream available, one would need to wait for multiple frames to fill up an entire "batch" before any could be processed—clearly an unacceptable latency penalty. However, an autonomous vehicle is likely to have at least front and rear cameras, and possibly even more. We therefore conducted experiments to quantify the latency/throughput tradeoff in CaffeNet on the TX1. These experiments cycled through a dataset of 2,400 images until 1,200 iterations of several batch sizes were completed.

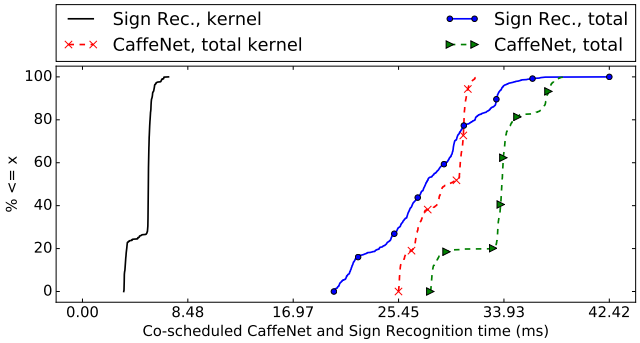

Figure 12: CDFs of the kernel times and total times running CaffeNet and sign-recognition tasks.

Obs. 8. CaffeNet on the TX1 can reliably support 30 frames per second if the batch size is at most two.

This observation is supported by the results presented in Fig. 10, and shown in more detail for a batch size of two in Fig. 11. In both of these plots, the WCET for a batch time of two is shown to be 29.9 ms. Unlike our other benchmarks, CaffeNet invokes many different CUDA kernels per iteration, so this figure includes *total kernel time*, the sum of all kernel times, in addition to *total time*, the per-iteration time including CPU and GPU operations.

As a final experiment, we tested the performance of both CaffeNet and sign recognition while co-scheduled.

Co-scheduling mixed image-processing tasks. In this final experiment, we concurrently executed both CaffeNet, processing a batch size of two images, and a single instance of the sign recognition benchmark.

Obs. 9. Neither sign recognition nor CaffeNet is able to maintain 30 frames per second while co-scheduled with the other.

Fig. 12 supports this observation. Both benchmarks have WCETs of approximately 40 ms, giving a worst-case frame rate of 24 frames per second. CaffeNet is also no longer even able to support 30 frames per second with its median performance. Given that the benchmarks were able to sustain little more than 30 frames per second in isolation, our analysis in Sec. 5 makes it is unsurprising that this is no longer achievable for either benchmark while co-scheduled, especially in the case of CaffeNet, which has little potential for co-scheduling due to its large proportion of kernel time.

Case-study conclusions. The TX1 has a limited ability to support reasonable frame rates for the image-processing tasks considered here. Both sign recognition and CaffeNet were capable of predictable 30 frame-per-second performance in isolation, but even running the two together reduced the frame rate to about 24 per second.

7 Conclusion

We evaluated the NVIDIA Jetson TX1's ability to host safety-critical applications. We began by examining one of the most distinctive aspects of an integrated GPU: its shared DRAM. Next, we evaluated co-scheduling as a mechanism to improve GPU utilization, and concluded with case studies examining image-processing benchmarks representative of autonomous-driving workloads.

The outcomes of our memory experiments not only confirmed that utilizing zero-copy memory is unlikely to provide a performance benefit, but also that changing CUDA versions can have a major impact on measurement-based timing analysis. We found that co-scheduling appears to be a viable option for improving utilization as long as it is carefully evaluated for each individual use case. Finally, we found that both CaffeNet and sign recognition can run at approximately 30 frames per second on the TX1 in isolation. When applying our findings related to co-scheduling, we can conclude that these tasks should be able to continue running predictably at approximately 24 frames per second while co-scheduled.

The TX1's feasibility for autonomous driving depends on whether the estimate of approximately 24 frames per second is sufficient to ensure safety. In the future, we hope to continue our in-depth investigation into GPU scheduling and specific embedded platforms. One platform in particular we hope to examine is NVIDIA's Drive PX2 [22], an upcoming single-board computer with over double the CPU, GPU, and memory capacity of the TX1. Finally, we hope that the increasing developer attention paid to self-driving cars and other autonomous systems will lead to greater openness in both hardware and software, so that the safety of consumers does not need to rely on black-box experimentation.

Acknowledgment: We are grateful to Lars Nyland of NVIDIA for helpful discussions.

References

[1] J. Aumiller, S. Brandt, S. Kato, and N. Rath. Supporting low-latency CPS using GPUs and direct I/O schemes. In *RTCSA '12*.

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

[3] K. Berezovskyi, K. Bletsas, and B. Andersson. Makespan computation for GPU threads running on a single streaming multiprocessor. In *ECRTS '12*.

[4] K. Berezovskyi, K. Bletsas, and S. Petters. Faster makespan estimation for GPU threads on a single streaming multiprocessor. In *ETFA '13*.

[5] K. Berezovskyi, F. Guet, L. Santinelli, K. Bletsas, and E. Tovar. Measurement-based probabilistic timing analysis for graphics processor units. In *ARCS '16*.

[6] K. Berezovskyi, L. Santinelli, K. Bletsas, and E. Tovar. WCET measurement-based and extreme value theory characterisation of CUDA kernels. In *RTNS '14*.

[7] A. Betts and A. Donaldson. Estimating the WCET of GPU-accelerated applications using hybrid analysis. In *ECRTS '13*.

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

[9] R. Girshic, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR '14*.

[10] A. Horga, S. Chattopadhyayb, P. Elesa, and Z. Peng. Systematic detection of memory related performance bottlenecks in GPGPU programs. In *JSA '16*.

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

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

[13] kayccc. Regarding usage of zero copy on tx1 to improve performance. NVIDIA Forums. Online at `https://devtalk.nvidia.com/default/topic/922626/jetson-tx1/regarding-usage-of-zero-copy-on-tx1-to-improve-performance/`.

[14] Y. Kini. CUDA on mobile. Online at `http://on-demand.gputechconf.com/gtc/2016/presentation/s6384-yogesh-kini-nvidia-cuda.pdf`.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

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

[17] A. Li, G. van den Braak, A. Kumar, and H. Corporaal. Adaptive and transparent cache bypassing for GPUs. In *SIGHPC '15*.

[18] X. Mei and X. Chu. Dissecting GPU memory hierarchy through microbenchmarking. In *TPDS '16*.

[19] NVIDIA. Cuda sample programs. Online at `http://docs.nvidia.com/cuda/cuda-samples`.

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

[21] NVIDIA. Jetson TX1 embedded developer kit. Online at `http://www.nvidia.com/object/jetson-tx1-dev-kit.html`.

[22] NVIDIA. Self-driving vehicles development platform. Online at `http://www.nvidia.com/object/drive-px.html`.

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

[24] A. Rao. Compute with tegra K1. Online at `http://on-demand.gputechconf.com/gtc/2014/video/S4906-mobile-compute-tegra-K1.mp4`.

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

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

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

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

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

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

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