

Abstract

Advances in Advanced Driver Assistance Systems (ADAS) and autonomous vehicles provide safety-critical control functions such as pedestrian detection, collision warnings, and lane tracking through an array of sensors mounted throughout a vehicle. To support the constant stream of sensor data, real-time systems are needed to periodically process the sensor data to ensure correct functionality of assistive and autonomous control functionality. In this paper, I discuss the use of CUDA, a programmable GPU platform for parallelizing data processing, within the context of ADAS and autonomous vehicles. I also discuss the development of a platform on which to run CUDA applications under Earliest Deadline First (EDF) scheduling, a real-time scheduler, and I evaluate the feasibility of running CUDA applications under EDF scheduling to process sensor data.

I. Introduction

Recent advances in automotive technology have involved outfitting vehicles with cameras and sensors designed to provide Advanced Driver Assistance Systems (ADAS) as well as partial-to-full autonomous vehicles. Such embedded systems perform pedestrian detection, lane changing assistance, blind spot monitoring, and more. Since cameras are cheaper sensors than many of the more sophisticated sensor technologies usable on vehicles, much work has been focused around processing image data as a means of performing safety-critical control functions. The goal of this research is to evaluate real-time image processing tasks on a development platform suitable for embedding into a vehicle. Specifically, I focus on the feasibility of running CUDA image processing tasks under the Earliest Deadline First (EDF) real-time scheduler. Additionally, I perform my evaluations on the Nvidia Jetson TK1 embeddable platform.

In Section II, I will discuss the motivation and current work done to develop and improve technologies that go into ADAS and autonomous vehicles. In Section III, I will delve into the details of the technologies used in my research. Specifically, I will discuss the use of CUDA in image processing, the choice of EDF as a real-time scheduler for scheduling periodic real-time tasks, and the Nvidia Jetson TK1 as a viable embeddable platform for vehicles. In Section IV, I will discuss my approach and detail my implementation for executing CUDA under EDF scheduling on the Jetson TK1. This section will be developed in two subsections: the first subsection details how to get EDF (SCHED_DEADLINE in Linux) on the TK1, and the second subsection details working with CUDA applications under SCHED_DEADLINE. Section V details my experimental setup and results. In Section VI and VII, I will conclude with potential ways forward with this research, and I will briefly mention other contributions.

II. Motivation and Current Work

Motivation

Modern Advanced Driver Assistance Systems (ADAS) and autonomous vehicles require the use of an array of sensors and cameras in order to carry out tasks such as rear-view obstruction detection, smart cruise control, parking assistance, and more. Some of the sensors used include long-range radar, LIDAR (Light Detection and Ranging), cameras, ultrasound, and orientation sensors [1]. However, many of these sensors are expensive and not suitable for embedding in consumer vehicles. For instance, at the time of writing, Google's self driving car uses a LIDAR sensor from Velodyne that costs around \$75,000, though research is being done to reduce this cost [2]. Instead of using expensive

sensors on commodity cars, researchers have been looking into using cheap cameras as a means to allow vehicles to see and parse their environment through the use of image processing and computer vision algorithms [9,10,16]. In image processing applications, GPU-based solutions are markedly more powerful than their CPU counterparts. CUDA, a GPU-based software platform for developing parallelized applications [4], is currently one of the leading platforms for developing parallelized image processing tasks used in implementing ADAS functionality.

These image processing tasks must also be done in real-time, where tasks must complete by a certain deadline. A real-time system is characterized by its ability to schedule application tasks in order to meet precise timing constraints. In the strictest sense, a real-time system must release some computed result within a defined window of time. A real-time system runs a set of *tasks*, which are units of work that run periodically. Each recurrence of a task is called a *job*, and a job is said to be *released* when the system scheduler chooses a task to run. When the scheduler prepares a job to be released, it assigns a *deadline* to the job, which is the time constraint in which the job must complete. The execution time, or *runtime*, of a job is a measure of how long the job takes to run until completion. If the runtime of a job exceeds its deadline, then the job is said to have exhibited a *deadline miss*. A set of tasks is said to be *schedulable* if a real-time system can always schedule the tasks such that no deadline misses occur. In the case of camera data, a real-time image processing job must be able to finish processing one frame of camera data before the next frame becomes available. This execution requirement is imposed on the image job as its deadline. There are two notions of real-time systems: hard real-time, where a task is guaranteed to complete execution before its deadline, and soft real-time, where a task is allowed to occasionally exceed its deadline without losing critical functionality. For the purposes of my research, I will be working with soft real-time systems given that CUDA does not lend itself easily to hard real-time systems.

My research is intended to contribute to the examination of CUDA as a viable platform for implementing ADAS functionality. Specifically, I examine the feasibility of leveraging CUDA functionality under the EDF real-time scheduler in Linux. CUDA provides the image processing, and EDF provides a mechanism to run CUDA tasks periodically and ensures that tasks that exceed their allotted runtime do not negatively impact other critical tasks in the system. This is all run on the Nvidia Jetson TK1, an embeddable board capable of supporting both CUDA and EDF and has a low cost suitable for embedding in consumer vehicles.

Current Work

Glenn Elliot, a PhD student from the University of North Carolina at Chapel Hill, developed a real-time multi-GPU framework called GPUSync as part of his dissertation [5]. This framework allows GPUs to be used in a real-time system, specifically LITMUS^{RT}, a real-time variant of Linux developed by UNC Computer Science. Specifically, his work presents a configurable multi-GPU scheduling framework capable of managing CUDA tasks under a fully real-time system.

In industry, Mobileye is a company specializing in developing vehicle-integrated ADAS systems using monoscopic cameras [6]. Their technologies perform tasks such as lane tracking, forward vehicle collision warning, adaptive cruise control, pedestrian and forward object detection, and traffic sign detection. Currently, Mobileye utilizes a proprietary specialized vision system on a chip (VSoC) design to provide the speed necessary to consume and process camera data. This is unlike using GPUs for computation as Mobileye's VSoC is a dedicated piece of hardware that is targeted at only running vision tasks.

Google for the past several years at the time of writing have been working on developing fully autonomous vehicles [7]. Their vehicles utilize more expensive components such as LIDAR and other

more sophisticated sensors beyond cameras. While self-driving cars are not directly related to ADAS, much of the research going into fully autonomous vehicles shares much overlap with ADAS functionality.

Subaru has developed their own version of driver assistance technology called Eyesight [8]. It is an integrative system that provides pre-collision detection, adaptive cruise control, pre-collision braking, and lane departure warning.

Much research that has gone into computer vision is directly applicable to ADAS. This includes pedestrian detection, vehicle detection, headlight control, and general feature extraction and detection of vehicular environment with the intention of modifying control over the vehicle to assist the driver [9,10,16].

III. Approach

For this research, I chose to examine the application of CUDA applications under the real-time Earliest Deadline First scheduler. I begin with an overview of CUDA as an application platform, Earliest Deadline First as a real-time scheduler, and the Nvidia Jetson TK1 as a viable embedded platform for hosting ADAS-related image processing tasks.

CUDA

CUDA is a Nvidia-developed platform for programming GPUs in order to leverage their parallelism for general computing. In the realm of image processing and computer vision, the computation model fits nicely; most image processing tasks consist of many independent operations on parts of an image, which fits the independent and parallel execution of CUDA threads. Conceptually, the CUDA programming paradigm executes a single block of code, known as a *kernel*, across multiple independently executing threads [11]. CUDA operates under the SIMD (Single Instruction, Multiple Data) architecture, meaning each thread executes the same instruction in lock-step, but each thread is free to access any location from a globally shared memory. In order to communicate with the GPU from the CPU, operations are funneled through two engines: the copy engine (CE) and the execution engine (EE). The CE handles memory copies between CPU memory and GPU memory (assuming the GPU memory is discrete from the CPU memory), while the EE handles kernels that are to be executed on the GPU.

CUDA supports execution of multiple kernels on the GPU through the use of *streams*. CUDA streams allow different kernels (and multiple memory copies if there are more than one CE) to execute simultaneously on the GPU, assuming there are available resources. If CUDA streams are not defined, all GPU operations are done on the default stream. For the purposes of my research, I have limited my work to the single CUDA default stream.

In order to communicate with the GPU, CUDA introduces a *CUDA context*. Contexts are either initialized explicitly using CUDA's driver API or implicitly using the CUDA runtime API. For this research, I am working only with the CUDA runtime API. When a CUDA context initializes, it spawns a child thread, called the *callback thread*, from the main application. This callback thread serves as an intermediary between the main application thread and the GPU. Whenever the GPU returns a signal, e.g. a kernel finished executing or an asynchronous memory copy finished, the callback thread receives the signal from the GPU and passes it on to the main application thread. The use of a callback thread allows CUDA operations to be executed asynchronously. By default, CUDA memory copies are synchronous, but the CUDA API provides asynchronous versions of the memory copy routines. On the other hand, kernel launches are by default asynchronous, which allows application developers to queue

up multiple kernels to be launched. While the GPU executes, the CPU is not blocked and is free to perform operations in parallel to GPU execution. Kernel launches do not have a synchronous variant. Instead, application developers must explicitly synchronize with the GPU, which will suspend the execution of the main application thread until all operations pending on the GPU have finished.

CUDA does not work well with real-time systems. This is due to CUDA's reliance on the GPU, which has its own internal thread scheduler and so is not directly influenced by the CPU scheduler. Furthermore, it is impossible to preempt tasks on the GPU, which is a necessary function in implementing real-time systems. In my research, I am only concerned with CUDA applications that run fast enough for experimental purposes, and formally characterizing how CUDA can run under a general real-time system is outside the scope of this research.

Earliest Deadline First Scheduling

Earliest Deadline First (EDF) scheduling is a real-time process scheduler. EDF characterizes tasks by arrival time, execution time, and deadline, where tasks with the closest deadline are selected to run next in the system. This allows EDF to guarantee that all tasks that enter the system can finish by their deadline, assuming the task-set satisfies the schedulability test for EDF. The schedulability test for EDF is as follows: let W_i be the worst case execution time of task i , let D_i be the relative deadline of task i , and let M be the number of processors on a system. In order for all of the EDF tasks to be schedulable, they must satisfy the following inequality:

$$\left(\sum_{i=1}^n \frac{W_i}{D_i} \right) \leq M$$

If the inequality is not satisfied, then EDF cannot schedule the n tasks and the system is said to be overloaded and therefore not schedulable.

EDF also provides predictability and task isolation amongst tasks running under EDF [12]. Predictability refers to the degree of control the system and user has over running tasks. Task isolation refers to how EDF schedules tasks such that each task in the system does not interfere with one another. In EDF, predictability is achieved by characterizing tasks with worst case execution times and deadlines, which allows the scheduler to select the task with the earliest deadline. Temporal isolation is guaranteed by only allowing one EDF task to run at any given time.

EDF scheduling entered the Linux kernel under the name SCHED_DEADLINE in Linux 3.14 [13]. In Linux, EDF is implemented alongside a resource reservation mechanism called the Constant Bandwidth Server (CBS) [12]. Together, SCHED_DEADLINE characterizes tasks using three parameters: the allotted runtime, a deadline, and a period [17]. Tasks are allocated a total amount of runtime that must complete by the relative deadline computed from the beginning of each period. The CBS provides both predictability and temporal isolation between EDF tasks through task throttling and task management. By characterizing tasks with runtime, deadline, and period, SCHED_DEADLINE manages tasks to ensure they do not overrun their allotted runtime via the CBS and reschedules tasks so the earliest deadline runs first via EDF. Tasks that use up all of their runtime are throttled and are rescheduled until their next period. This prevents misbehaving tasks from executing longer than the time allotted to them, and ensures temporal isolation between tasks.

Nvidia Jetson TK1

The Nvidia Jetson TK1 is an embeddable platform that runs Linux with native support for CUDA applications. It sports a Tegra K1 system on a chip (SoC) that packages a GPU with 192 CUDA cores and a quad-core Cortex-A15 ARM CPU [14]. At a cost of \$192 per board with a GPU capable of up to 326 GFLOPS[*], the Jetson TK1 is definitely a cost-viable platform for performing image tasks associated with ADAS systems. The Jetson TK1 also supports the Linux operating system, which provides the necessary environment to run the Linux EDF scheduler `SCHED_DEADLINE`. At the time of writing, the Jetson TK1 officially runs the Linux kernel version 3.10, which does not include support for EDF scheduling. This incompatibility is rectified in my implementation of CUDA running under EDF.

IV. Implementation

Implementation of CUDA under EDF is broken up into two parts: EDF scheduling on the Nvidia Jetson TK1, and CUDA under EDF/`SCHED_DEADLINE` scheduling.

SCHED_DEADLINE on the Jetson TK1

The Jetson TK1 officially supports the Linux kernel up to version 3.10. Nvidia made extensive modifications to the 3.10 kernel in order to make the kernel capable of running on the TK1 and to add support for programming the GPU. Later Linux kernels beyond version 3.10 have added support for running on the TK1 board, however they do not support the programmable GPU driver interface. In order to be able to support both CUDA and `SCHED_DEADLINE` on the Jetson TK1, either the official kernel needs to be upgraded, or necessary `SCHED_DEADLINE` functionality needs to be backported to the Nvidia 3.10 kernel.

My approach to get `SCHED_DEADLINE` running on the TK1 started off by attempting to port the Nvidia modifications from Linux 3.10 to Linux 3.18 (which supports `SCHED_DEADLINE`). Conceptually, the desired strategy was to assume that the Nvidia modifications could be represented as a patch-set that sits atop the vanilla 3.10 kernel sources. Based on that assumption, I would then be able to replay the modifications onto the 3.18 kernel, fix code that underwent API changes between 3.10 and 3.18, and have a working TK1 system that supports both EDF and CUDA. Realistically, I discovered the modifications could not be represented as a nice patch-set that was built off of a 3.10 release. Rather, a thorough examination of the Nvidia kernel git commit history reveals extensive changes and merges that date back to Linux kernel versions as far back as 3.8 [15].

Instead of searching for nice patch-sets, I identified core components of the modified 3.10 sources that were integral in the operation of the kernel on the TK1: machine-specific code for the Tegra SoC, the Nvidia drivers for video controllers, and the GK20A driver (the driver for the GPU). My strategy was to copy over the identified core components to a vanilla copy of Linux 3.18 and run multiple iterations of build, examine errors, fix, repeat. Without reading through every commit of the Nvidia kernel sources, I would have no chance at knowing what modifications were made, so I relied on the assumption that build errors were highly indicative of missing code and functionality. By selecting three kernel components that I identified as being core components in the Nvidia kernel and compiling my modified 3.18 kernel, I was able to use build errors to identify and pull in other Nvidia kernel modifications that were not directly included in the three core components. After iterating multiple times, I ended with a compiled 3.18 kernel with the Nvidia modifications that I deemed necessary for the functioning of the TK1. Executing the kernel, however, ended in failure, with the kernel refusing to proceed past the early boot phase. Relying on kernel “earlyprintk” for debugging and a serial console setup, I worked through several of the issues (clock subsystem errors, DTB errors, and

others), but still ended up with a hung system. Rather than attempt to debug runtime kernel errors, I adopted a different strategy to get SCHED_DEADLINE on the TK1.

My second strategy was to instead pull down the SCHED_DEADLINE commits since the first SCHED_DEADLINE patches in Linux 3.14 and replay them on top of the Nvidia 3.10 kernel. Linux kernel commits are often done in patch-sets, and patch-sets are identified by a text-based tag in the commit message. For SCHED_DEADLINE, the tag is “sched/deadline”. For my second strategy, I assumed that most of the SCHED_DEADLINE patches can be found by simply looking for commits with the sched/deadline tag. Any other commit that was required by the sched/deadline patch-set could be found by either figuring out why a patch failed to be applied cleanly or by observing build errors. Using the latest Linux kernel sources (version 4.4.7 at the time of writing), I pulled all commits tagged with sched/deadline and all auxiliary commits that the patch-set depended on and replayed those commits on top of the Nvidia 3.10 kernel to produce a modified Nvidia 3.10 kernel with SCHED_DEADLINE.

Unlike my first attempt, the resulting kernel from my second attempt booted properly and supported running processes under SCHED_DEADLINE through the use of the schedtool-dl utility. In order to test SCHED_DEADLINE was working properly, I used the schedtool-dl utility to run multiple CPU tasks under SCHED_DEADLINE. Figure 1 shows several CPU-based yes processes scheduled under SCHED_DEADLINE. A yes program is an infinite loop that prints out the character “y”, which is demonstrative of a computation-intensive program. The graph was obtained using the kernelshark tool. The x-axis in Figure 1 is time, and the y-axis indicates the task. Solid colored (non-white) bars indicate that a specific task is running, and vertical lines indicate kernel scheduling events (e.g. schedule wakeups and task switches.) Figure 1 demonstrates how SCHED_DEADLINE schedules tasks to have predictability and task isolation. So long as the set of tasks in the system does not change, each yes instance is run periodically every 100ms as shown. In addition, each job execution does not overlap with any other job executions, giving the system task isolation.

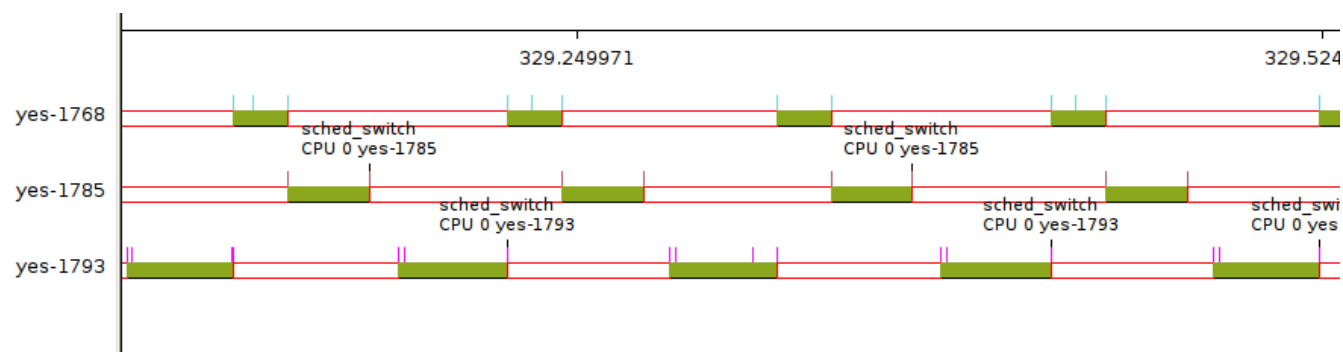


Figure 1. Three yes programs running under SCHED_DEADLINE on a single CPU. Each yes instance was given allotted runtimes 20ms, 30ms, and 40ms, respectively. Deadline and period were both 100ms.

CUDA under EDF/SCHED_DEADLINE

Running CUDA applications under SCHED_DEADLINE does not work out of the box. Specifically, CUDA context creation fails with an error that “all CUDA-capable devices are busy or unavailable”. CUDA context creation occurs at the first call to the GPU, e.g. a `cudaFree(0)` call or equivalent. On the first CUDA call, the callback thread is spawned as a child thread to the main application thread. The de-facto way to spawn threads on Linux is to use the pthreads library, which

means the callback thread must be spawned using pthreads. In order to verify this, I used LD_PRELOAD to confirm that CUDA was indeed using pthreads. LD_PRELOAD is a technique whereby a user-created shared library can be loaded before all other shared libraries upon application start up. This allows the user to control calls to other functions in libraries loaded after the user-created library. (See Section VII for a more detailed example of using LD_PRELOAD.)

Knowing CUDA spawns the callback thread with pthreads establishes why CUDA doesn't run under SCHED_DEADLINE by default. Pthreads by default inherit the scheduling class of the parent thread. Under the default fair scheduler, the callback thread would run under the fair scheduling class. Under SCHED_DEADLINE, the callback thread would run under SCHED_DEADLINE. In order to see if pthreads will actually run under SCHED_DEADLINE, I constructed a verification test to see if it was possible to spawn a thread when the entire application ran under SCHED_DEADLINE. It turns out that the pthread library is unable to properly spawn a SCHED_DEADLINE thread because the thread creation routine returns with the error EAGAIN ("try again"). Therefore, CUDA callback threads cannot run under SCHED_DEADLINE.

Instead of running the callback thread under SCHED_DEADLINE, I chose to run the callback thread under a different scheduling policy in the realtime class. The intended multithreading model has the main application thread running under SCHED_DEADLINE and the callback thread running under a realtime scheduling policy (e.g. First-In-First-Out (FIFO) or Round-Robin (RR) policies.) The purpose of running the callback thread under a realtime scheduler is to push the priority of the callback thread higher than other non-essential userspace tasks to ensure the callback thread notifies the main application thread of any GPU signals as fast as possible. Implementation-wise, all functionality related to setting the scheduling policy and priority of the callback thread is packaged within a shared library. The shared library must be preloaded using LD_PRELOAD in order for the library to hook into the pthread library and to modify thread creation to run threads under a realtime scheduling class.

V. Analysis

Analysis of CUDA running under EDF is presented by first describing the experimental setup, and then the actual experimental results.

Experimental Setup

As a simple benchmark program, I used a CUDA matrix multiplication program, modified to be periodic and to run under SCHED_DEADLINE. The following pseudocode details how the benchmark program is structured:

```
init_callback_thread()
switch_to_sched_deadline()
while (TRUE) {
    run_cuda_matrix_multiplication()
    cuda_device_synchronize()
    sched_yield()
}
```

The callback thread is spawned with the SCHED_FIFO scheduling policy and a priority of 99, the highest realtime FIFO priority. The EDF scheduling parameters vary and will be detailed in the experimental results. Each call to the CUDA matrix multiplication routine is considered a single iteration. In order to count the finished kernel computation as one iteration, I call

`cuda_device_synchronize()` which will cause the application to wait for the GPU to finish. The `sched_yield()` function tells the SCHED_DEADLINE scheduler to stop execution of the current thread and to reschedule the current thread's execution for the next EDF period.

When explicitly synchronizing the CPU with the GPU, CUDA provides three options for how the CPU interacts with the scheduler while waiting for the GPU: wait on a spinlock, yield to scheduler, or blocking on a synchronization primitive. Waiting on a spinlock simply uses CPU cycles by continuously checking a conditional variable until the scheduler stops the thread or the callback thread notifies the application thread of GPU results by updating the conditional variable. Yielding to the scheduler tells the scheduler to reschedule the current thread. Blocking synchronization will block on a synchronization primitive until the callback thread notifies the main process of GPU results. My experiments predominantly use spinlocks, but I also demonstrate the effects of using blocking synchronization. Yielding to the scheduler is not discussed because it has the same overall effects as blocking synchronization.

Results and Analysis

Each experiment is introduced with the parameters used in the experiment: the SCHED_DEADLINE reservation, the CUDA-scheduler interaction policy, and approximate worst-case execution time of a single matrix multiplication iteration with and without SCHED_DEADLINE.

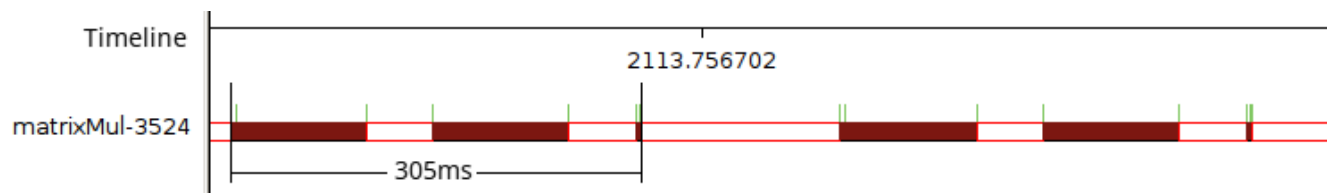


Figure 2.

Figure 2 is CUDA matrix multiplication with an allotted runtime of 100ms with a deadline/period of 150ms. Without SCHED_DEADLINE, the runtime of each iteration is approximately 37ms in the worst case, while with SCHED_DEADLINE each iteration takes approximately 305ms. In this experiment, CUDA is configured to spinlock wait for a GPU result. From Figure 2, it is apparent that the results are undesirable. A single task that normally takes 37ms under fair scheduling now runs in 305ms under SCHED_DEADLINE. The graph above reveals that matrixMul is going through 3 periods per 1 iteration, and the bulk of the time is wasted in the first two periods. The last period sees an incredibly small slice of execution time, which may be an artifact of using `sched_yield` to always start an iteration at the beginning of a new period.

A possible explanation for this behavior relies on how SCHED_DEADLINE operates in Linux. SCHED_DEADLINE has the highest priority of any userspace thread in a Linux system, which means no other userspace process can preempt a SCHED_DEADLINE task. Only the CBS/EDF systems are allowed to stop a SCHED_DEADLINE task in order to switch it out for another task. Following the execution of matrixMul, the first period sees memory allocations and memory copies as well as the kernel launch. Post-kernel launch, the application spin waits on the GPU result, but since the callback thread cannot run, the application uses CPU cycles, spinning until the end of its period. In the results above, this behavior of spin waiting until the end of the period occurs twice, which implies that the callback thread somehow never gets the chance to execute until sometime after the second period.

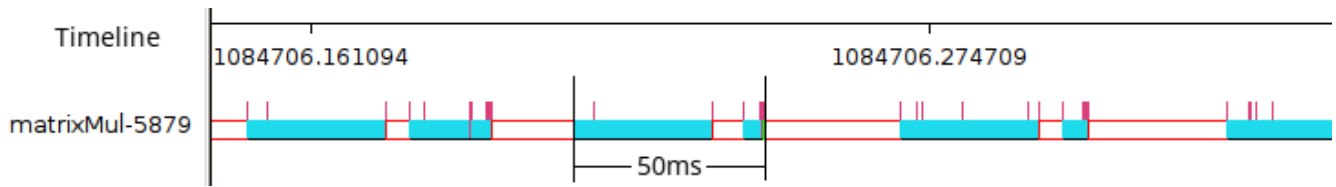


Figure 3.

Figure 3 is CUDA matrix multiplication with an allotted runtime of 25ms with a deadline/period of 30ms. Without SCHED_DEADLINE, the runtime of each iteration is approximately 37ms in the worst case, while with SCHED_DEADLINE each iteration takes approximately 50ms. In this experiment, CUDA is configured to spinlock wait for a GPU result. A single iteration is distinctively partitioned into a long execution block with runtime 25ms and a smaller execution block with runtime of approximately (37-25)ms. This is expected behavior, as the first block overruns the allotted runtime (37ms > 25ms), the CBS throttles the thread, and EDF reschedules the thread to run at the next period.

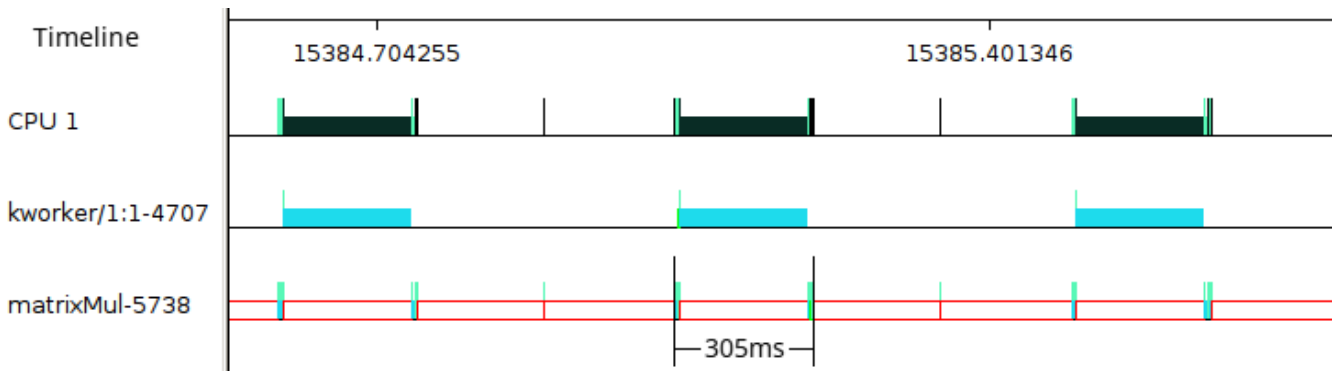


Figure 4.

Figure 4 is CUDA matrix multiplication with an allotted runtime of 100ms with a deadline/period of 150ms. Without SCHED_DEADLINE, the runtime of each iteration is approximately 37ms in the worst case, while with SCHED_DEADLINE each iteration takes approximately 305ms. In this experiment, CUDA is configured to use blocking synchronization to wait for a GPU result. An interesting result is observed with the addition of a kernel kworker thread that serves as a dual to behavior shown in Figure 2. When matrixMul blocks on a synchronization primitive, matrixMul suspends execution and the kworker thread begins to use a spinlock, most likely waiting for an interrupt. Compared to the experiment outlined in Figure 2, I observed that spinlock is simply waiting in userspace while blocking synchronization is the spinlock equivalent in kernelspace. This provides insight into why blocking synchronization does not provide any benefit over spinlock waiting because by deferring the waiting to the kworker thread, matrixMul is rescheduled by SCHED_DEADLINE to run at the next period. An unresolved component of this result is that the runtime for one multiplication iteration is approximately two times the deadline/period, yet matrixMul was never woken up by EDF at the 150ms mark inside the 305ms runtime.

A side observation from the experiments is how performance of the CUDA matrix multiplication application varied from processor core to processor core. If the CUDA application ran on the 0th indexed processor, then the application would run slowly. If the CUDA application ran on either the 1st, 2nd, or 3rd indexed processor, then the application would run faster than on the 0th indexed

processor. This phenomenon can be attributed to the majority of interrupts being bound to CPU 0, notably timer interrupts. Therefore, the 0th indexed processor is already experiencing a lot of interrupts, and a running CUDA application on CPU 0 would be preempted many times by kernel interrupts bound on CPU 0.

VI. Concluding Remarks and Future Work

While this research has developed and demonstrated a proof-of-concept with CUDA running under EDF scheduling, much more work is required to fully realize CUDA under EDF as a viable means of implementing ADAS vision systems. Despite not achieving a satisfactory result where the CUDA application behaves the same both under fair scheduling and EDF scheduling with a generous deadline, this research marks an early step in analyzing CUDA under EDF. Furthermore, at the time of writing I could not find any body of literature that has attempted to execute and analyze CUDA applications under EDF scheduling.

More work can be expanded to explore different ways to execute CUDA applications under EDF. Despite the inflated runtimes of CUDA applications under EDF versus under fair scheduling, there are different avenues that can be investigated further. The first avenue is an approach to rectify the inflated CUDA application runtimes under EDF. In the experiment associated with Figure 2, I noted that the inflated runtime may have come from the main application thread not allowing the callback thread to execute due to the main application thread having the highest priority in the system due to SCHED_DEADLINE policy. One possible approach to fixing this issue is to explicitly bind the main application thread to one CPU and the callback thread to a different CPU. If the EDF task is really inhibiting the callback thread's ability to execute, then binding the callback thread to a different CPU than the one that runs the EDF task will theoretically allow the callback thread to run independently of the EDF task. This should allow the callback thread to update the main application thread whenever the GPU result is completed rather than whenever the main application thread reaches the end of its allocated runtime. Therefore binding the EDF task and the callback thread to separate CPUs should fix the issue of the callback thread being unable to execute while the main application thread is executing.

The second approach is to use task splitting. Instead of requiring a single task completion within one period, the concept of task splitting allows a single task to complete over two or more periods. For example, suppose that a camera outputs a frame every 50ms, and a CUDA application task under EDF is split in two halves, both with a period of 25ms. The CUDA application can then spend the first period performing memory copies, and finishes with a kernel launch. Assuming the kernel finishes before the second period, the CUDA application would start on the second period, copy the results back from the GPU, and perform any other operations based on the GPU results. This approach exists as a fallback in case CUDA under EDF refuses to run as a single task.

Beyond fixing issues between CUDA and EDF discovered in my research, further experiments are needed before CUDA under EDF on the Jetson TK1 becomes a viable approach to ADAS systems. My research was concerned with a simple matrix multiplication benchmark which is not quite representative of an actual image processing algorithm. Future experiments can incorporate actual image processing algorithms, e.g. fastHOG [16]. It is also necessary to quantify deadline misses with CUDA applications. Since CUDA does not fit well under a real-time system due to its reliance on a GPU, future experiments should examine what happens when a CUDA kernel exceeds the period of the CPU-bound EDF task. Finally, it is useful to experiment with multiple CUDA applications running under EDF. Because the GPU is non-preemptible, the dynamic nature of EDF scheduling may introduce conditions where multiple CUDA tasks clash for the GPU. An extension to this would be to introduce the use of CUDA streams to run multiple distinct CUDA kernels simultaneously on the GPU.

VII. Other Contributions

In collaborative work with Vance Miller, we developed a resource locking library for CUDA. The library is built on the GPUSync [5] framework by Glenn Elliot. Specifically, the library provides two implementations of GPU locking: kernel-space locking and user-space locking. The locking library treats the GPU's copy and execute engines as lockable resources; so long as a process holds either a copy or execute lock, other processes cannot access the locked resources until the lock is released. The purpose of the locking library is to provide a means of introducing priority on GPU operations. Tasks sent to the GPU are put on a blackbox queue, which means actual order of execution is not known. Furthermore, priority inversions are possible when running tasks on the GPU, as a lower priority process may get their kernel run before a higher priority process. The locking library alleviates this issue by wrapping CUDA calls that send tasks to the GPU and issuing locks to processes that request the lock first, which will typically be higher priority processes.

The locking library works as an "interception" library using LD_PRELOAD. Our interception library interposes between the CUDA runtime library and the CUDA application binary, intercepting calls to CUDA routines that send tasks to the GPU. The main classes of intercepted routines are memory copy routines and the singular kernel launch routine. Our implementation wraps these routines with logic to check if a calling process is able to obtain a resource lock on the copy or execute engine. If so, the calling process is granted the lock and our library calls the appropriate CUDA runtime library routine on behalf of the calling process, otherwise the process is suspended until it can obtain the lock. In order to initialize the interception library, a call to `cudaFree(0)` is needed at the beginning of the running CUDA application. Note that this approach works with the regular CUDA paradigm of calling `cudaFree(0)` in order to initialize the CUDA context early in the application.

References

- [1] Wong. "The Internet of Things is Here to Stay." *Electronic Design*. 23 Jan 2015. Web. Accessed 4 April 2016.
- [2] McFarland. "The \$75,000 problem for self-driving cars is going away." *The Washington Post*. 4 Dec 2015. Web. Accessed 4 Apr 2016.
- [4] Nickolls, Buck, Garland, Skadron. 2008. Scalable Parallel Programming with CUDA. Queue 6, 2 (March 2008), 40-53. DOI=<http://dx.doi.org/10.1145/1365490.1365500>
- [5] Elliott. (2015). Real-Time Scheduling for GPUs with Applications in Advanced Automotive Systems. UNC-Chapel Hill Electronic Theses and Dissertations.
- [6] Gat et al. (2004). A Monocular Vision Advance Warning System for the Automotive Aftermarket.
- [7] Google. Self driving cars, <https://www.google.com/selfdrivingcar/>. Web. Accessed 20 April 2016.
- [8] Buchholz. "Safety and driver assist via Subaru's EyeSight." SAE International. 26 Sept 2012. Web. Accessed 20 April 2016.
- [9] Bertozzi, Massimo, et al. "Stereo vision-based vehicle detection." IEEE Intelligent Vehicles Symposium. 2000.
- [10] Yue Wang, Eam Khwang Teoh, Dinggang Shen, Lane detection and tracking using B-Snake, Image and Vision Computing, Volume 22, Issue 4, 1 April 2004, Pages 269-280, ISSN 0262-8856, <http://dx.doi.org/10.1016/j.imavis.2003.10.003>.
- [11] Nvidia. "CUDA C Programming Guide." 1 Sept 2015. Web. Accessed 15 Feb 2016.

- [12] Lelli. "SCHED_DEADLINE: How to use it." Retis Lab. 26 June 2014. Web. Accessed 10 March 2016.
- [13] Faggioli et al. "sched/deadline: Add SCHED_DEADLINE structures & implementation." Linux kernel gitweb, commit id aab03e05e8f7e26f51dee792beddcb5cca9215a5. 13 Jan 2014. Web. Accessed 21 February 2016.
- [14] Jetson TK1 Wiki. Web. Accessed 14 April 2016.
- [15] Nvidia. Tegra L4T kernel source gitweb. Web. Accessed 18 February 2016.
- [16] Prisacariu and Reid. (2009). fastHOG - a real-time GPU implementation of HOG. Department of Engineering Science, Oxford University.
- [17] Linux kernel documentation. "sched-deadline.txt". Web. Accessed 21 April 2016.