CUPiD^{RT}: Detecting Improper GPU Usage in Real-Time Applications*

Tanya Amert and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

***Abstract*—Computer-vision applications typically rely on graphics processing units (GPUs) to accelerate computations. However, prior work has shown that care must be taken when using GPUs in real-time systems subject to strict timing constraints; without such care, GPU use can easily lead to unexpected delays not only on the GPU device but also on the host CPU. In this paper, a software library is presented that can detect the improper use of GPUs for safety-critical computer-vision applications. This library was used to analyze several GPU-using sample applications available as part of OpenCV, a popular computer-vision library, revealing the presence of issues in all ten applications considered. Additionally, a case study is presented, detailing the response-time improvements to one of the applications when such issues are corrected.**

***Index Terms*—Graphics Processors, Hardware/Software Interfaces, Image Processing, Computer Vision, Real-Time Systems**

I. Introduction

Safety-critical systems increasingly rely on computer-vision applications to perceive their environments. For example, autonomous vehicles utilize object-detection algorithms to identify pedestrians and other vehicles in their vicinity, and optical-flow algorithms to localize the vehicle within the scene over time. Graphics processing units (GPUs) were designed for massively parallel processing of graphics algorithms, and thus are well-suited to such computer-vision applications.

NVIDIA has positioned itself as a market leader in GPU acceleration; NVIDIA GPUs are supported by many common vision and machine-learning frameworks, such as PyTorch [39], TensorFlow [1], and OpenCV [10], and NVIDIA GPUs were used for navigation in the original Tesla Model X [42]. Unfortunately, NVIDIA designs its GPUs for throughput rather than predictability. Thus, great care is needed when using them in safety-critical systems subject to strict timing constraints.

The problem. Prior work by Yang *et al.* [50] detailed several pitfalls that can arise when using NVIDIA's CUDA API to utilize NVIDIA GPUs in real-time systems, leading to unexpected delays not only on the GPU, but also on the host CPU. One such scenario is shown in Fig. 1, which depicts an experiment in which four CPU threads submit commands to individual CUDA streams.[1] The CUDA API command (`cudaFree`) submitted by CPU Thread 3 at time (a) results in another *unrelated* CPU thread being blocked until all prior GPU work has completed (indicated by the arrow from time (b) to time (c)). Such blocking must factor into response-time analysis if it may occur in safety-critical applications, and thus can lead to system capacity loss. To our knowledge, no prior work has explored whether existing applications (*e.g.*, those utilizing OpenCV) are, in fact, subject to these pitfalls.

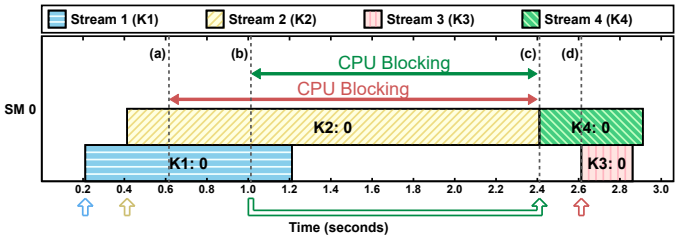

Fig. 1. The results of an implicit synchronization experiment from Yang *et al.* [50] depicting CPU blocking due to freeing GPU memory.

Our solution. In this paper, we present CUPiD^{RT} (CUDA Pitfall Detector for Real-Time Systems), a software library designed to detect the improper use of NVIDIA GPUs for real-time applications. In particular, we detected issues based on two of the most egregious pitfalls listed by Yang *et al.*, and used CUPiD^{RT} to analyze ten GPU-using sample applications provided as part of OpenCV. In addition, we performed a case study to explore the response-time changes when such issues are mitigated. Before describing our contributions in more detail, we briefly discuss prior work on using GPUs in safety-critical systems.

Prior work. Prior work has traditionally focused on how to use existing applications in real-time systems. Rather than taking advantage of possible concurrency between programs, many approaches have managed the GPU as a single mutual-exclusion shared resource [16], [23], [24], [43]–[45], [48]. Other work has sought to understand existing GPU scheduling policies [2], [33], [36]–[38] or to change GPU scheduling policies by changing the driver code itself [12]. Beyond the scheduling of GPU computations, multiple approaches have also sought to evaluate and manage the use of specific GPU hardware resources (*e.g.*, cache and DRAM) [3], [13], [14], [21], [27], [29], [47].

More closely related to our work, Horga *et al.* sought to detect performance bottlenecks due to memory [20]. However, to our knowledge, only Yang *et al.* [50] provided guidelines for properly interfacing with an NVIDIA GPU from a software perspective, and no prior work has tackled the general question

*This was supported by NSF grants CNS 1563845, CNS 1717589, CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

[1]CUDA streams are FIFO queues of GPU operations.

of whether a specific implementation is well-suited to the timing requirements of a real-time system.

Contributions. In this paper, we address the problem of detecting improper GPU use through three contributions. First, we present CUPiD$^{\mathrm{RT}}$, a library we developed to intercept operations submitted to NVIDIA GPUs and report issues based on the most severe pitfalls identified by Yang *et al.* [50]. CUPiD$^{\mathrm{RT}}$ can be used with any applications that utilize NVIDIA GPUs, and although CUPiD$^{\mathrm{RT}}$ was designed with real-time systems in mind, throughput-oriented applications can also benefit from detection and remediation of these issues.

Second, we detail the results produced by using CUPiD$^{\mathrm{RT}}$ to analyze ten GPU-using OpenCV sample applications. Based on two of the most problematic pitfalls identified by Yang *et al.*, we configured CUPiD$^{\mathrm{RT}}$ to detect issues related to specific API calls (*e.g.*, allocating and freeing GPU memory and the submission of work to synchronous CUDA streams). Of the ten applications, we found that each was subject to at least one of the issues we sought to detect.

Finally, we present a case study demonstrating the benefits of using CUPiD$^{\mathrm{RT}}$ to verify that issues have been resolved. For one of the applications we analyzed, we modified the source code to remedy the discovered issues, and compared the end-to-end execution times and GPU-computation submission times of the application with and without modifications. In the case study, we found that our modifications led to a 74.1% reduction in execution times and a 74.9% reduction in computation submission times.

Organization. The remainder of this paper is organized as follows. In Sec. II, we provide needed background on NVIDIA GPUs and the pitfalls that can arise when they are used in real-time systems. We give an overview of CUPiD$^{\mathrm{RT}}$ in Sec. III, and discuss the results of our OpenCV application analysis and case study in Sec. IV. We overview related work on designing real-time applications for CPU+GPU platforms in Sec. V before concluding in Sec. VI.

II. Background

We focus our attention on NVIDIA GPUs, as NVIDIA has been the market leader in GPUs. This is due in large part to the CUDA API, which enables general-purpose hardware-accelerated parallelism in C/C++ programs. In this section, we give an overview of NVIDIA GPUs and the CUDA API, and summarize the pitfalls that can arise when using NVIDIA GPUs in safety-critical systems.

A. Using NVIDIA GPUs

NVIDIA provides both discrete GPUs and system-on-chip solutions with integrated GPUs. An NVIDIA GPU is comprised of an *Execution Engine* (EE), used to perform computations, and one or more *Copy Engines* (CEs), which copy data between the host (CPU) and the device (GPU), or from one GPU memory location to another. All GPU operations are submitted to *CUDA streams*, which are first-in first-out queues; by default, the *NULL stream* is used, serializing all operations.

Listing 1 Vector Addition Routine using CUDA.

```
    /* Performs single addition: C[i] = A[i] + B[i] */
 1: static __global__ void vecAdd(int* A, int* B, int* C) {
 2:     // Calculate index using built-in thread, block info
 3:     int i = blockDim.x * blockIdx.x + threadIdx.x;
 4:     C[i] = A[i] + B[i];
 5: }

    /* Element-wise addition: C = A + B, returns pointer to C
       (assumes n is a multiple of 32) */
 6: int * vectorAdd(int* A, int* B, int n) {
 7:     // Allocate CPU (host) memory for result array C
 8:     size_t bytes = n * sizeof(int);
 9:     int* C = (int *) malloc(bytes);
10:
11:     // Allocate GPU (device) memory for arrays A, B, and C
12:     int *d_A, *d_B, *d_C;
13:     cudaMalloc(&d_A, bytes);
14:     cudaMalloc(&d_B, bytes);
15:     cudaMalloc(&d_C, bytes);
16:
17:     // Copy arrays A and B from CPU to GPU memory
18:     cudaMemcpy(d_A, A, bytes, cudaMemcpyHostToDevice);
19:     cudaMemcpy(d_B, B, bytes, cudaMemcpyHostToDevice);
20:
21:     // Launch the kernel
22:     int nt = 32;              // threads per block
23:     int nb = n / 32;          // blocks per grid
24:     int sm = 0;               // no shared memory used
25:     cudaStream_t stream; // user-defined stream
26:     cudaStreamCreate(&stream);
27:     vecAdd<<<nb, nt, sm, stream>>>(d_A, d_B, d_C);
28:     cudaStreamSynchronize(stream);
29:     cudaStreamDestroy(stream);
30:
31:     // Copy results from GPU to CPU
32:     cudaMemcpy(C, d_C, bytes, cudaMemcpyDeviceToHost);
33:
34:     // Free GPU memory for arrays A, B, and C
35:     cudaFree(d_A);
36:     cudaFree(d_B);
37:     cudaFree(d_C);
38:
39:     return C;
40: }
```

To maximize concurrency, asynchronous *user-defined streams* can be used, enabling multiple computations to be performed simultaneously on the EE.

An NVIDIA GPU is accessed via the CUDA API, which is a C/C++ extension that enables general-purpose GPU computing. An example CUDA-using program is shown in Listing 1.

Functions that execute on the GPU are called *kernels*; an example kernel is shown in lines 1–5. Kernels are executed in single-instruction-multiple-data (SIMD) fashion; the data to be used can be determined by CUDA-provided variables (line 3).

When *launching* a kernel (line 27), the programmer specifies the distribution of GPU threads into *thread blocks* (line 22) and thread blocks into a *grid* (line 23). Additional optional arguments include the per-block shared-memory requirement (line 24), and the stream into which to launch the kernel (line 25). By default, no shared memory is used, and all kernels

are launched into the NULL stream. Note that kernel launches are typically asynchronous with respect to the CPU,[2] so a stream synchronization (line 28) is used to ensure the kernel has completed before copying the results back to the CPU.

Additionally, this example demonstrates common CUDA API functions, such as `cudaMalloc` (lines 13–15), `cudaMemcpy` (lines 18 and 19), and `cudaFree` (lines 35–37) used to allocate, copy to and from, and free GPU device memory, respectively. Unlike `cudaMalloc` and `cudaFree`, `cudaMemcpy` has an asynchronous variant, `cudaMemcpyAsync`, that can be used with user-defined streams for additional concurrency.

B. Pitfalls in using NVIDIA GPUs

As the CUDA API was designed for throughput rather than predictability, care must be taken when using NVIDIA GPUs in real-time systems. Yang *et al.* [50] detailed various pitfalls that can arise with NVIDIA GPUs. They focused on sources of explicit and implicit synchronization, which can delay GPU operations (intentionally or not). They also listed inconsistencies in the NVIDIA documentation, leading to further difficulty in developing real-time GPU-using applications.

We now discuss the pitfalls listed by Yang *et al.* Note that we retain their original pitfall numbering, but group them differently for the purpose of our discussion.

Synchronization-related pitfalls. Explicit synchronization can be performed via CUDA operations such as `cudaStreamSynchronize`. Implicit synchronization, on the other hand, occurs as a side effect of a non-synchronization-related operation, such as `cudaFree` or a kernel launched in the NULL stream. The synchronization-related pitfalls Yang *et al.* identified are as follows:

- **P1** Explicit synchronization does not block future commands issued by other tasks.[3]
- **P4** Some CUDA API functions will block future, unrelated, CUDA tasks on the CPU.

As they observed, while one CPU thread waits due to an explicit synchronization command, GPU work issued by other CPU threads may execute for the entire duration of the synchronization. Thus, commands such as `cudaDeviceSynchronize` do not serve as barriers between CPU threads. Implicit synchronization proves more nefarious: in their explorations, Yang *et al.* found that some API functions can block other CUDA-using threads *on the CPU*, rather than simply blocking future GPU work; such behavior is depicted in Fig. 1 as a result of a call to `cudaFree`.

Best-practices-related pitfalls. Beyond synchronization-related pitfalls, Yang *et al.* also identified two pitfalls related to best practices for CUDA-using software:

- **P5** The suggestion from NVIDIA's documentation to exploit concurrency through user-defined streams may be of limited use for improving performance in thread-based tasks.
- **P6** Async CUDA functions use the GPU-synchronous NULL stream by default.

[2]See Sec. 3.2.6.1 of the CUDA Programming Guide [32] for more details.

[3]Yang *et al.* referred to CPU threads as *tasks* to avoid confusion with GPU threads.

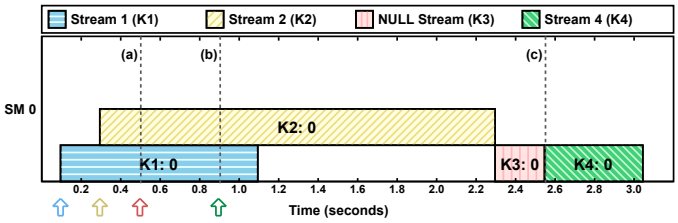

Fig. 2. The results of an experiment from Yang *et al.* [50] depicting implicit synchronization due to a kernel launched in the NULL stream.

In their experiments, Yang *et al.* found that when using user-defined streams, concurrency could be thwarted by the presence of unexpected implicit synchronization. In addition, due to the use of default parameters in CUDA API functions, simple mistakes can lead to unexpected use of the NULL stream, further limiting concurrency.

An example of the implicit synchronization introduced by the NULL stream is depicted in Fig. 2. In this example, Kernel K3 is launched into the NULL stream at time (a), preventing K3 or later-launched kernels (K4) from executing concurrently with earlier-launched kernels (K1 and K2), even though available resources exist. Thus, Kernel K4 is blocked on the GPU for 1.65 seconds, from time (b) to time (c).

Documentation-related pitfalls. The NVIDIA documentation is at best sparse, and at worst incorrect and self contradicting. Yang *et al.* listed the following documentation-related pitfalls:

- **P2** Documented sources of implicit synchronization may not occur.
- **P3** The CUDA documentation neglects to list some functions that cause implicit synchronization.
- **P7** Observed CUDA behavior often diverges from what the documentation states or implies.
- **P8** CUDA documentation can be contradictory.
- **P9** What we learn about current black-box GPUs may not apply in the future.

In particular, CUDA API functions related to allocating, copying, setting, and freeing memory were shown to behave contradictory to the documented behavior. For example, `cudaFree` variants were not documented to cause implicit synchronization. Conversely, the NVIDIA documentation states that `cudaMalloc` does cause implicit synchronization, but Yang *et al.* were unable to observe such behavior.

Although their experiments considered multiple generations of NVIDIA GPUs (Maxwell, Pascal, and Volta) and CUDA versions (8.0 and 9.0), Pitfall P9 indicates that their findings may not hold in the future. However, we note that we were able to reproduce their synchronization-related experiments on a Titan V (Volta architecture) with CUDA 10.2.

III. CUPiD^{RT}

In this section, we introduce CUPiD^{RT}. To do so, we first list the issues we seek to detect in GPU-using programs, and relate them to the most egregious of the pitfalls listed by Yang *et al.* [50]. Then, we describe how we designed CUPiD^{RT} to detect these issues at runtime.

A. Issues We Detect

Yang *et al.* identified pitfalls that focus primarily on ways in which GPU-using programs may behave unexpectedly. However, it is not clear how to rectify or even detect the issues that arise due to these pitfalls. In fact, the majority of the pitfalls described in Sec. II-B center around errors and ambiguities in NVIDIA's CUDA documentation.

In the design of CUPiD^{RT}, we have focused on the most egregious issues that arise when attempting to perform timing analysis of GPU-using programs. In particular, we are concerned with unexpected cross-CPU-thread blocking (Pitfall P4) and NULL-stream GPU operations (Pitfall P6).

Issue: Unexpected CPU blocking. Of the pitfalls described in Sec. II-B, Pitfall P4 is perhaps the most problematic, as it relates to an unexpected source of blocking on the CPU. Such potential blocking must be taken into account during real-time schedulability analysis, and corresponds to lost system capacity. Thus, we seek to detect any sources of unexpected CPU blocking due to GPU use.

The two sources identified by Yang *et al.* are the variants of the CUDA API functions `cudaMalloc` and `cudaFree`, which serve to allocate and free GPU memory, respectively. These function calls are necessary if GPU memory is to be used, and are only problematic while other operations (*e.g.*, kernel launches) are occurring. Thus, we exclude the setup and cleanup regions of programs from our analysis, as described in Sec. III-B. We call the remaining region the *analyzed region*.

Issue: NULL-stream GPU operations. As detailed in Pitfall P6, asynchronous CUDA functions (and kernel launches) default to using the NULL stream if no stream parameter is provided. For asynchronous functions in particular, this can cause further unexpected delays as kernels and other operations must then execute sequentially (see Amert *et al.* [2] for an overview of NVIDIA's GPU scheduling rules).

We thus also configured CUPiD^{RT} to detect the use of the NULL stream, both in kernel launches and in other CUDA API functions (*e.g.*, `cudaMemcpyAsync`). As these operations are typically part of the actual work of a GPU-using program, we do not distinguish between code regions, and have configured CUPiD^{RT} to report any detections of NULL stream use, including those during setup and cleanup.

B. Analyzing GPU-Using Programs

Program analysis can be performed in one of two ways: (1) at compile time, using a static code analyzer, or (2) at runtime, based on tracing of CUDA API calls. Although static analysis guarantees all issue occurrences are detected regardless of program inputs and settings, dynamic analysis can be used if the code path of interest is known. Additionally, as the improper usage we seek to detect is based on individual CUDA operations rather than a more complex sequence of commands, we chose dynamic analysis for CUPiD^{RT}. We leave a static-analysis implementation to future work.

CUPiD^{RT} design. CUPiD^{RT} is comprised of a series of scripts to trace the behavior of a CUDA-using program at runtime, logging all CUDA calls (*e.g.*, `cudaMalloc` or kernel launches), and then parse the log file to detect the issues described in Sec. III-A. For the tracing stage, we utilize the `nvprof` tracing tool provided by NVIDIA. When analyzing the trace output, we detect issues corresponding to the following operations:

- **`cudaMalloc`**: Any invocation of `cudaMalloc` (or a variant) inside the analyzed region.
- **`cudaFree`**: Any invocation of `cudaFree` (or a variant) inside the analyzed region.
- **NULL stream kernels**: Any kernel submitted to the NULL stream, regardless of location in the program.
- **Asynchronous CUDA functions**: Any asynchronous function (*e.g.*, `cudaMemcpyAsync`) called with a stream parameter of 0 (the NULL stream), regardless of location in the program.
- **Other CUDA functions**: Any other NULL-stream-using function (*e.g.*, `cudaMemset`), regardless of location within the program.

Using CUPiD^{RT} to analyze a program. There are two steps to use CUPiD^{RT} to analyze a GPU-using program. First, the analyzed region must be annotated in the program source code. Then, CUPiD^{RT} can be used to run the GPU-using program; CUPiD^{RT} scripts analyze the tracing output and report any issues detected. Our implementation is based on CUDA 10.2, and is provided open source online.[4]

The only source-code modifications necessary to use CUPiD^{RT} are analyzed-region demarcations, for which we utilize the NVIDIA Tools Extension (NVTX) API. The statements `nvtxMarkA("TRACE_START")` and `nvtxMarkA("TRACE_END")` must be added at the start and end, respectively, of each analyzed region; these functions add messages to the trace output without otherwise affecting program functionality.

IV. Evaluation

In this section, we present the results of using CUPiD^{RT} to detect issues in ten GPU-using sample applications provided with OpenCV. Then, we demonstrate the value of remedying these issues via a case study involving one of the applications.

A. OpenCV Sample Applications

We evaluated the benefits of CUPiD^{RT} by analyzing GPU-using applications provided as part of OpenCV [10], a popular computer-vision framework. We considered ten sample applications provided with OpenCV 3.4, as listed in Table I. These

[4] Available online at https://github.com/tkortz/cupid-rt.

TABLE I
ISSUE OCCURRENCES FOR TEN GPU-USING OPENCV SAMPLE APPLICATIONS.

OpenCV Sample	Description	`cudaMalloc`/ `cudaFree` calls	NULL stream kernels	NULL stream async functions	Other NULL stream ops
bgfg_segm	Segmentation	0	3	0	5
farneback_optical_flow	Optical Flow	10	2	0	14
generalized_hough	Feature Extraction	8	22	8	26
hog	Object Detection	64	64	143	157
houghlines	Feature Extraction	4	3	0	8
morphology (Erode/Dilate)	Morphology	2	2	0	2
morphology (Open/Close)		5	4	0	3
optical_flow (Brox)	Optical Flow	7	3993	40	43
optical_flow (TV-L1)		123	2937	0	262
pyrlk_optical_flow	Optical Flow	23	21	13	26
stereo_match	Stereo Matching	0	2	0	7
super_resolution	Super Resolution	26	168	0	74

sample applications cover a broad range of different computer-vision algorithms:

- *Background/Foreground Segmentation*: A video sequence is processed to classify pixels in each video frame as being in the background or foreground of the image.
- *Optical Flow*: Using two images, the movement of each pixel from one image to the other is determined. Optical-flow algorithms include the Brox *et al.* algorithm [11], the Farnebäck algorithm [17], the iterative Lucas-Kanade method with image pyramids [28], and the TV-L1 method [40], [52]; these algorithms are demonstrated in three different OpenCV sample applications.
- *Feature Extraction*: Features of certain types (*e.g.*, lines or circles) are located in an individual image. The Generalized Hough Transform extends the original Hough transform to arbitrary shapes; these algorithms are demonstrated in two sample applications.
- *Object Detection*: Although similar to feature extraction, object detection entails detecting high-level objects, such as pedestrians. The Histogram of Oriented Gradients (HOG) algorithm [15], [41] detects pedestrians by identifying features to be classified by a Support Vector Machine (SVM); in the `hog` sample application, detections are performed for each frame of a video.
- *Morphology*: Morphological operations modify binary images, *e.g.*, by "eroding" the foreground (shrinking any white regions) or by "dilating" the foreground (expanding any white regions). More complex operations, such as "opening" and "closing," can be performed by combining the basic erosion and dilation operations.
- *Stereo Matching*: Given a pair of stereo images taken from different positions, corresponding points in the two images can be used to compute a depth map of the distance to any point in the scene.
- *Super Resolution*: Given an input video, the resolution of each frame in the video can be increased by leveraging optical-flow techniques [18], [30].

B. Experimental setup

We performed our experiments on a machine with a single Titan V NVIDIA GPU, two eight-core 2.10-GHz Intel CPUs, and 32-GB of DRAM. Each core includes a 32-KB L1 data cache, a 32-KB L1 instruction cache, and a 1-MB L2 cache; all eight cores on a CPU socket share an 11-MB L3 cache. We used Ubuntu 16.04, CUDA 10.2, the NVIDIA 440.33 driver, and all experiments were run using native Linux scheduling.

C. Evaluation of Issue Detection via CUPiD$^{\mathrm{RT}}$

We list the issues detected by CUPiD$^{\mathrm{RT}}$ for each of the ten applications in Table I. When analyzing the applications, we used default parameters when possible. For applications that act on videos, we inserted the `nvtxMarkA` statements at the beginning and end of the processing loop, as discussed in Sec. III-B. For applications that process just one or two images, we added a processing loop and the `nvtxMarkA` statements. Note that results for two `optical_flow` sample algorithms are omitted as they duplicate other samples.

`cudaMalloc/cudaFree` calls. We found that eight of the ten applications included calls to `cudaMalloc` and `cudaFree` within the analyzed regions. This is due, in part, to the structure of OpenCV. Sample applications showcase functionality provided by various "modules," *e.g.*, the `hog` sample application utilizes the `cudaobjdetect` module to perform GPU-based object detection. Unfortunately, inspection of the code reveals that the vast majority of the GPU-memory allocations and frees that we observed are within these modules themselves.

Obs. 1: OpenCV modules, as implemented, are prone to unexpected CPU blocking.

Several of the applications we considered allocated and later freed over twenty regions of GPU memory when processing an individual video frame or pair of images. In fact, `hog` allocated

64 memory regions per video frame, and the TV-L1 algorithm of `optical_flow` allocated over 100 GPU-memory regions.

We expect that the execution times of these applications would be greatly improved by mediating these issues (recall the discussion of CPU blocking in Sec. III-A); we explore the benefits of such fixes for the `hog` application in Sec. IV-D.

Unfortunately, for other applications with high GPU-memory allocation and free counts, these operations occur deep within the module implementations (*e.g.*, involving dynamic pools of GPU memory), and thus would be extremely challenging to extricate into setup and cleanup code regions.

NULL-stream GPU operations. The number of NULL-stream operations (both kernels and other operations) are given in the rightmost three columns in Table I. Unlike the frequent GPU-memory allocations and frees throughout the OpenCV modules source code, the modules do take a CUDA stream as a parameter. Thus, the presence of issues with NULL-stream operations is primarily due to how the modules are used, rather than their implementations. However, taking a stream as a parameter does not guarantee that module-internal functionality always uses it.

Obs. 2: Even modules that are implemented to take a stream parameter may still internally use the NULL stream.

Using CUPiD$^{\mathrm{RT}}$, we found that all ten of the applications submitted at least some kernels to the NULL stream and performed other synchronous NULL-stream operations, and four of the ten submitted asynchronous operations (*e.g.*, `cudaMemcpyAsync`) to the NULL stream.

Only two of the ten applications utilized user-defined streams for kernel execution: `farneback_optical_flow` submitted only two of its 158 kernels to the NULL stream, and for `super_resolution`, only 168 of its 884 kernels were submitted to the NULL stream. However, these remaining NULL-stream kernels are the result of inner functionality that does not use the provided stream parameter.

The `hog` application performed 143 asynchronous and 157 synchronous copy operations using the NULL stream; these NULL-stream operations were hard-coded in the module implementation. In the next section, we explore the benefits of fixing these issues for the `hog` application.

D. Case Study: Impact of Issue Remediation

To explore the benefits of remedying the issues detected by CUPiD$^{\mathrm{RT}}$, we performed a case-study experiment using the `hog` OpenCV sample application. We modified the application to perform all `cudaMalloc`s before processing the video and all `cudaFree`s after processing was complete. Additionally, we modified `hog` to perform all per-frame kernel executions and other GPU operations in user-defined streams rather than the NULL stream. After making these changes, we used CUPiD$^{\mathrm{RT}}$ to ensure that no further issues remained.

Before we discuss the results of our case-study experiments, we describe the structure of the HOG algorithm in more detail. The algorithm can be represented as a graph, as shown in Fig. 3. Each video frame is processed at a configurable number

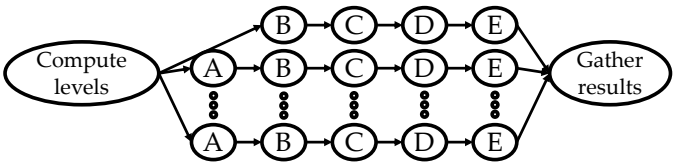

Fig. 3. Structure of the HOG algorithm.

of levels of detail (the default in the `hog` application is 13). For each level, five GPU kernel operations are performed in sequence: the image is resized (A), gradients are computed (B), and then histograms of the gradients are computed (C), normalized (D), and classified using an SVM (E). Finally, the potential detections are aggregated. The first level processes the original image, so the resize operation occurs 12 times; the other four kernels occur 13 times each. Thus, a total of 64 kernels are submitted to the GPU per video frame.

Impact on execution times. For the first part of our case study, we sought to measure the impact of issue remediation on the overall execution time of the `hog` application. We modified the application to run within a configurable number of threads, such that one, two, three, or four threads were performing the HOG algorithm simultaneously; this is meant to simulate a vehicle in which multiple cameras might be processing different video streams simultaneously.

All experiments represent the results of processing 5,000 video frames. For each frame, we used two calls to `clock_gettime` to measure the elapsed time from just before uploading the image to the GPU (thus, after reading in the frame from the video file) until just after downloading the resulting detections from the GPU and aggregating them into detected locations. The program was configured to process video frames as quickly as possible, rather than at a set frequency, thus maximizing potential cross-thread interference.

The per-frame execution times are depicted as cumulative distribution functions (CDFs) in Fig. 4. The corresponding worst- and average-case execution times are also reported in Table II, along with the 90th- and 99th-percentile values.

Obs. 3: Per-frame execution times increase when multiple threads perform the HOG algorithm in parallel.

This is expected (Otterness *et al.* [36] observed similar behavior for other applications) and occurs for both the original `hog` implementation and our modified version with issues fixed. For example, the 90th-percentile of the original HOG execution times increased from 10.29 milliseconds for one thread to 39.77 milliseconds with four threads.

Obs. 4: Per-frame execution times significantly improve after all detected issues are fixed.

This can be observed when comparing curves between the original implementation and our modified implementation. For instance, the 90th-percentile measurement for the modified implementation with four threads was 10.29 milliseconds, a 74.1% decrease compared to the original implementation.

It is worth noting that our modified implementation with four threads had a 99th-percentile execution-time measurement

TABLE II
PER-FRAME EXECUTION TIMES (IN MILLISECONDS) OF HOG WITH AND WITHOUT FIXING ISSUES.

Configuration	avg	90th	99th	max
Original x1	10.18	10.29	11.17	12.64
Original x2	17.12	18.71	21.16	24.30
Original x3	24.39	29.32	33.48	45.43
Original x4	32.83	39.77	47.61	63.36
Modified x1	2.84	2.90	2.99	5.93
Modified x2	4.93	5.67	6.16	8.77
Modified x3	5.29	7.68	9.04	14.06
Modified x4	7.36	10.29	13.06	24.27

In addition to the average and maximum per-frame execution times, the 90th- and 99th-percentile results are also listed.

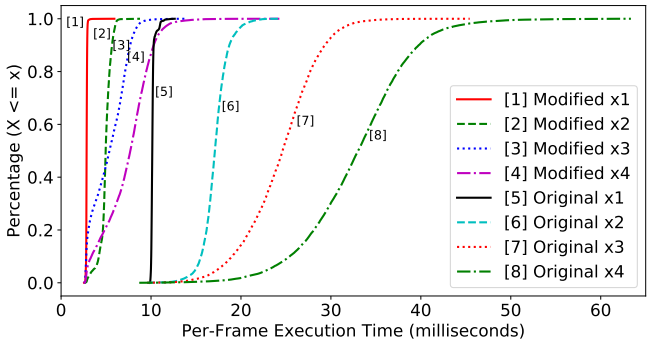

Fig. 4. CDF of HOG per-frame execution times with and without fixing issues.

of 13.06 milliseconds, which is not much higher than the worst-case measurement of 12.64 milliseconds for *one thread* in the original implementation. This, combined with the improvement of the single-threaded configurations, suggests that CUPiD^{RT} can be useful for performance tuning in addition to ensuring predictable execution for real-time applications; we explore this in more detail later.

Impact on CPU blocking. We can use the time required to launch[5] a kernel as a proxy for the CPU blocking time. If CPU blocking occurs, such as that depicted in Fig. 1, we expect to observe multiple peaks in a histogram of kernel launch times. For example, most launches should take a small amount of time, but some launches would take longer due to overlapping with CPU-blocking operations, such as `cudaFree`. We measured kernel launch times via calls to `clock_gettime` immediately before and after each kernel-launch CUDA command.

Histograms of kernel launch times for the original `hog` implementation and our fixed version are depicted as probability density functions (PDFs) in Fig. 5 for each of the five kernels in HOG. These curves are normalized, such that the area under the curve sums to one. Note that the x-axis is truncated; the worst-case values range up to 8.5 milliseconds. Kernel launch times are also reported in Table III.

[5]Note that this does not include the time to complete execution.

TABLE III
KERNEL LAUNCH TIMES (IN MICROSECONDS) OF HOG WITH AND WITHOUT FIXING ISSUES, WITH PERCENTAGE IMPROVEMENTS IN BOLD.

Kernel	avg	90th	95th	99th	max
A (orig.)	44.02	81.62	169.40	346.75	4134.34
A (fixed)	25.22	42.12	54.39	97.91	4352.95
% Decr.	**42.71**	**48.40**	**67.89**	**71.76**	**-5.29**
B (orig.)	56.96	178.18	242.89	407.58	8530.10
B (fixed)	17.18	37.71	56.11	109.28	5157.46
% Decr.	**69.84**	**78.84**	**76.90**	**73.19**	**39.54**
C (orig.)	91.12	224.56	297.26	462.55	5127.76
C (fixed)	26.91	59.37	84.81	153.05	5128.24
% Decr.	**70.47**	**73.56**	**71.47**	**66.91**	**-0.01**
D (orig.)	90.35	229.09	297.46	474.76	5232.71
D (fixed)	26.54	58.92	84.84	150.43	4911.89
% Decr.	**70.63**	**74.28**	**71.48**	**68.31**	**6.13**
E (orig.)	88.98	228.64	309.67	490.57	6280.39
E (fixed)	21.94	49.88	69.27	123.16	4592.75
% Decr.	**75.34**	**78.18**	**77.63**	**74.89**	**26.87**

In addition to average and maximum kernel launch times, the 90th-, 95th-, and 99th-percentile results are also listed, as are the percentage decreases after fixing detected issues.

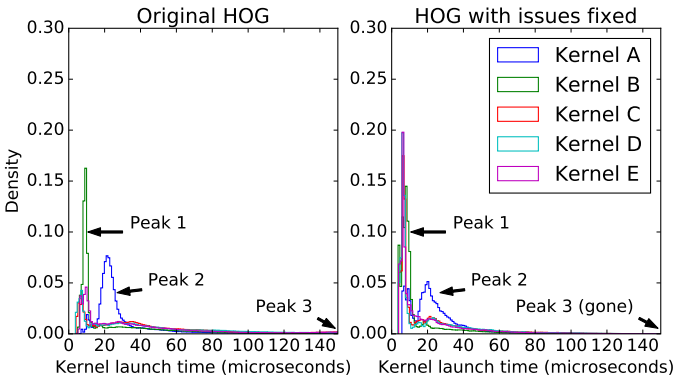

Fig. 5. Distribution of HOG kernel launch times before and after fixing issues, with one, two, or four HOG threads executing simultaneously.

Obs. 5: Fixing detected issues results in many more kernel launches taking the minimum observed time.

This behavior can be observed by comparing the height of the first peak (occurring around 8 microseconds) for each kernel between the two plots in Fig. 5. The increase in the peak height for our modified implementation indicates that the average-case kernel launch duration is significantly reduced; this is also evident in Table III.

The increase in the first peak height also suggests that our modifications led to reductions in later peaks. To observe this, we look at a different region of the PDFs, depicted in Figs. 6 and 7. These plots depict the first three peaks at around 8, 21, and 165 microseconds, respectively.

Obs. 6: Fixing issues results in much more predictable kernel launch times.

This can be observed when comparing Figs. 6 and 7. In

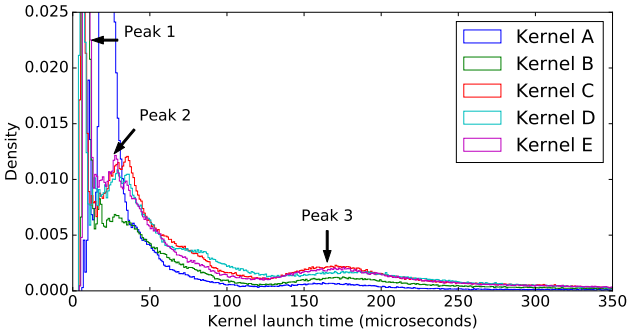

Fig. 6. Zoomed-in distribution of the original HOG kernel launch times.

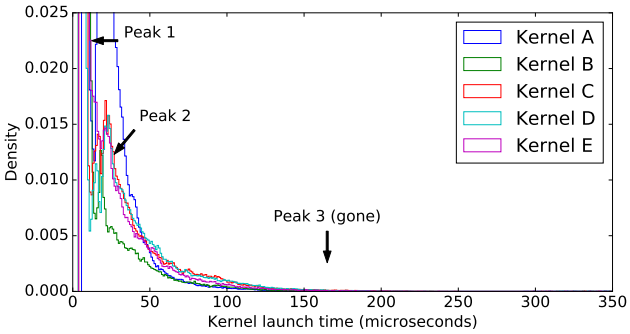

Fig. 7. Zoomed-in distribution of HOG kernel launch times after fixing issues.

particular, fixing issues completely removed the third peak at around 165 microseconds, as shown in Fig. 7. Furthermore, our modifications reduced the 99th-percentile launch times for all kernels by up to 74.89%, as shown in Table III.

Unfortunately, our modifications did not always improve the worst-case observed kernel launch times.

Obs. 7: Worst-case kernel launch times of some kernels are higher after fixing issues.

This suggests that some other unexpected behaviors may be occurring. However, the 99th-percentile results shown in Table III indicate that these are likely outliers. These outliers do, however, impact the worst-case per-frame execution times, as worst-case values for the four-thread modified implementation listed in Table II are significantly higher than the 99th-percentile execution times.

Summary. We have observed both per-frame execution times and the distribution of kernel launch times between the original `hog` implementation and our modified implementation with all detected issues fixed. We now make one remaining observation regarding the case study as a whole.

Obs. 8: Using CUPiD$^{\mathrm{RT}}$ to ensure all detected issues are resolved improves throughput in addition to predictability.

Our fixes greatly reduced each of the average, 90th-, 95th-, and 99th-percentile kernel launch times, as listed in Table III. Furthermore, our modifications resulted in more predictable memory usage; our modified implementation used 129 MB of GPU memory throughout its runtime, compared to the original implementation, for which memory usage ranged from 139-147 MB. Combined with the improved per-frame execution times listed in Table II, we have shown that the issues we have designed CUPiD$^{\mathrm{RT}}$ to detect can greatly hinder both average and worst-case performance and thus predictability.

Discussion. CUPiD$^{\mathrm{RT}}$ can be a valuable tool for the development of systems which require optimizing for either worst-case or average-case performance. However, despite the benefits we have shown of resolving the issues detected by CUPiD$^{\mathrm{RT}}$, such efforts can require comprehensive application refactoring.

Obs. 9: Not all applications are well suited to remedying the issues detected by CUPiD$^{\mathrm{RT}}$.

Fixing issues was fairly straightforward for the HOG application. However, we had limited success using the same techniques with other OpenCV sample applications that used dynamically allocated GPU memory buffers. For these applications, remedying `cudaMalloc` and `cudaFree` uses was prohibitively challenging, as it would have required massively refactoring the underlying OpenCV modules to avoid using GPU memory-buffer pools. Further work is necessary to determine how best to fix issues in these cases.

Additionally, our fixes to the HOG application reduced the runtime flexibility of the program; *e.g.*, the choice of grayscale versus full-color image processing cannot be changed after program initialization (as color scheme impacts GPU memory allocated). However, we consider this to be a reasonable trade-off for real-time systems, in which inputs and program behavior can reasonably be assumed stable after design time.

V. Related Work

We now summarize the existing work on GPU use in real-time applications. We focus on three areas: how access to the GPU is managed, how timing analysis is performed for GPU-using applications, and how GPU-using applications may need to be modified to be amenable to real-time analysis.

Arbitrating GPU access. Managing GPU use in a real-time system can be done in one of two ways. Synchronization mechanisms can be used to arbitrate GPU-access order. Alternatively, scheduling approaches can either use built-in scheduling policies or modify them through driver changes.

Synchronization mechanisms typically treat GPUs as requiring mutually exclusive access [16], [23], [24], [43]–[45], [48]. For multi-GPU systems, this can be extended to a k-exclusion lock [31]. Simultaneous access to multiple GPUs can be managed via a nested locking protocol [46]. As an alternative to locking-protocol-based synchronization approaches, Kim *et al.* [25] introduced a GPU server task to reorder and submit all GPU operations on behalf of other GPU-using tasks.

Scheduling approaches, on the other hand, utilize knowledge of existing scheduling rules or modify those rules. Scheduling rules have been deduced via micro-benchmarking experiments for NVIDIA [2], [33], [36] and AMD [37], [38] GPUs. Capodieci *et al.* [12] implemented a preemptive earliest-deadline first (EDF) scheduler for NVIDIA GPUs, but details are unavailable due to non-disclosure agreements. Kato *et al.* [24] modified the open-source Nouveau driver for NVIDIA GPUs to implement a non-preemptive fixed-priority scheduler.

However, the Nouveau driver does not support CUDA, precluding its use for general-purpose real-time computing.

Timing analysis of GPU-using workloads. Schedulability analysis of GPU-enabled real-time applications requires GPU timing analysis. Some prior work has studied this from a holistic perspective, considering high-level impacts of GPU sharing [34], [35] or by developing response-time analysis based on NVIDIA's scheduling rules [49]. Further work has explored WCET analysis for GPU-using programs and individual kernels [5]–[9], [19]. Other performance bottlenecks that can impact GPU timing analysis have been explored by Horga *et al.* [20] and Yang *et al.* [50].

The accuracy of timing analysis can be greatly improved given knowledge of the hardware platform. Jain *et al.* [22] performed micro-benchmarking experiments to determine the NVIDIA GPU memory hierarchy. Other work has explored the evaluation and management of specific GPU hardware resources [3], [13], [14], [21], [27], [29], [47].

Application changes for GPU use in real-time systems. Given GPU-access arbitration and a means of performing timing analysis, a GPU-enabled real-time workload may still require changes to the applications themselves to ensure schedulability. The use of CUPiD$^{\mathrm{RT}}$, presented in this paper, can help identify some necessary application changes.

Other work has explored algorithm-level changes. For example, object detection via a neural network can be modified to use lower-resolution images or even a batch of images grouped into a single input; such techniques have been shown to improve both utilization and throughput [51], [54]. Additionally, Heo *et al.* [19] explored modifying deep neural networks (DNNs) to choose between pre-determined network configurations at runtime based on time remaining until deadlines. Other approaches have explored splitting GPU computations into smaller sub-computations to be scheduled independently [4], [23], [26], [49], [53].

VI. Conclusion

In this paper, we have presented CUPiD$^{\mathrm{RT}}$, a software tool designed to detect issues in GPU-enabled code. We used CUPiD$^{\mathrm{RT}}$ to analyze ten GPU-using OpenCV applications, and showed that all were subject to at least one of the issues we sought to detect. Although we designed CUPiD$^{\mathrm{RT}}$ with a focus on real-time workloads, our case study experiment demonstrated that fixing all issues detected by CUPiD$^{\mathrm{RT}}$ can result in significant improvements in throughput, in addition to a reduction in spurious GPU-operation launch times.

In the future, we plan to modify CUPiD$^{\mathrm{RT}}$ to use static analysis rather than relying on NVIDIA's tracing tools. Static analysis will enable us to report which lines of code represent issues, as well as ensure a more comprehensive set of detections that does not rely on runtime behavior (*e.g.*, a program may conditionally allocate or free GPU memory, and runtime behavior may not demonstrate this). We will then target automatically fixing issues, utilizing CUPiD$^{\mathrm{RT}}$ to ensure none remain. This will be a significant effort, and will require static analysis and compiler techniques.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[2] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU scheduling on the NVIDIA TX2: Hidden details revealed," in *Proceedings of the 38th IEEE Real-Time Systems Symposium*, 2017, pp. 104–115.

[3] L. B. B. Forsberg, A. Marongiu, "GPUguard: Towards supporting a predictable execution model for heterogeneous SoC," in *Proceedings of the 20th Conference on Design Automation and Test in Europe*, 2017, pp. 318–321.

[4] C. Basaran and K. Kang, "Supporting preemptive task executions and memory copies in GPGPUs," in *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, 2012, pp. 287–296.

[5] K. Berezovskyi, K. Bletsas, and B. Andersson, "Makespan computation for GPU threads running on a single streaming multiprocessor," in *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, 2012, pp. 277–286.

[6] K. Berezovskyi, K. Bletsas, and S. Petters, "Faster makespan estimation for GPU threads on a single streaming multiprocessor," in *Proceedings of the 18th IEEE Conference on Emerging Technologies and Factory Automation*, 2013, pp. 1–8.

[7] K. Berezovskyi, F. Guet, L. Santinelli, K. Bletsas, and E. Tovar, "Measurement-based probabilistic timing analysis for graphics processor units," in *Proceedings of the 29th International Conference on Architecture of Computing Systems*, 2016, pp. 223–236.

[8] K. Berezovskyi, L. Santinelli, K. Bletsas, and E. Tovar, "WCET measurement-based and extreme value theory characterisation of CUDA kernels," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, 2014, pp. 279–288.

[9] A. Betts and A. Donaldson, "Estimating the WCET of GPU-accelerated applications using hybrid analysis," in *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, 2013, pp. 193–202.

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

[11] T. Brox, A. Bruhn, N. Papenberg, and J. Weickert, "High accuracy optical flow estimation based on a theory for warping," in *Proceedings of the 8th European Conference on Computer Vision*, 2004, pp. 25–36.

[12] N. Capodieci, R. Cavicchioli, M. Bertogna, and A. Paramakuru, "Deadline-based scheduling for GPU with preemption support," in *Proceedings of the 39th IEEE Real-Time Systems Symposium*, 2018, pp. 119–130.

[13] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna, "SiGAMMA: Server based integrated GPU arbitration mechanism for memory accesses," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, 2017, pp. 48–57.

[14] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms," in *Proceedings of the 22nd IEEE International Conference on Emerging Technologies and Factory Automation*, 2017, pp. 1–10.

[15] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 1, 2005, pp. 886–893.

[16] G. Elliott, B. Ward, and J. Anderson, "GPUSync: A framework for real-time GPU management," in *Proceedings of the 34th IEEE Real-Time Systems Symposium*, 2013, pp. 33–44.

[17] G. Farnebäck, "Two-frame motion estimation based on polynomial expansion," in *Proceedings of the 13th Scandinavian Conference on Image Analysis*, 2003, pp. 363–370.

[18] S. Farsiu, M. D. Robinson, M. Elad, and P. Milanfar, "Fast and robust multiframe super resolution," *IEEE transactions on image processing*, vol. 13, no. 10, pp. 1327–1344, 2004.

[19] S. Heo, S. Cho, Y. Kim, and H. Kim, "Real-time object detection system with multi-path neural networks," in *Proceedings of the 26th Real-Time*

and Embedded Technology and Applications Symposium, 2020, pp. 174–187.

[20] A. Horga, S. Chattopadhyayb, P. Elesa, and Z. Peng, "Systematic detection of memory related performance bottlenecks in GPGPU programs," *Journal of Systems Architecture*, vol. 71, pp. 73–87, 2016.

[21] P. Houdek, M. Sojka, and Z. Hanzálek, "Towards predictable execution model on ARM-based heterogeneous platforms," in *Proceedings of the 26th IEEE International Symposium on Industrial Electronics*, 2017, pp. 1297–1302.

[22] S. Jain, I. Baek, S. Wang, and R. Rajkumar, "Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs," in *Proceedings of the 25th Real-Time and Embedded Technology and Applications Symposium*, 2019, pp. 29–41.

[23] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "RGEM: A responsive GPGPU execution model for runtime engines," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, 2011, pp. 57–66.

[24] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *Proceedings of the USENIX Annual Technical Conference*, 2011, pp. 17–30.

[25] H. Kim, P. Patel, S. Wang, and R. Rajkumar, "A server-based approach for predictable GPU access control," in *Proceedings of the 23rd International Conference on Embedded and Real-Time Computing Systems and Applications*, 2017, pp. 1–10.

[26] H. Lee and M. A. A. Faruque, "Run-time scheduling framework for event-driven applications on a GPU-based embedded system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1956–1967, 2016.

[27] A. Li, G. van den Braak, A. Kumar, and H. Corporaal, "Adaptive and transparent cache bypassing for GPUs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 17:1–17:12.

[28] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, 1981, pp. 674–679.

[29] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2016.

[30] D. Mitzel, T. Pock, T. Schoenemann, and D. Cremers, "Video super resolution using duality based TV-L1 optical flow," in *Proceedings of the 31st Symposium of the German Association for Pattern Recognition*, 2009, pp. 432–441.

[31] C. Nemitz, K. Yang, M. Yang, P. Ekberg, and J. Anderson, "Multiprocessor real-time locking protocols for replicated resources," in *Proceedings of the 28th Euromicro Conference on Real-Time Systems*, 2016, pp. 50–60.

[32] NVIDIA, "CUDA C++ programming guide v10.2.89," Online at https://docs.nvidia.com/cuda/archive/10.2/cuda-c-programming-guide/index.html, 2019.

[33] I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna, "Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective," in *Proceedings of the 26th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2020, pp. 213–225.

[34] N. Otterness, V. Miller, M. Yang, J. Anderson, and F. D. Smith, "GPU sharing for image processing in embedded real-time systems," in *Proceedings of the 12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2016, pp. 23–39.

[35] N. Otterness, M. Yang, T. Amert, J. Anderson, and F. D. Smith, "Inferring the scheduling policies of an embedded CUDA GPU," in *Proceedings of the 12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2017, pp. 47–52.

[36] N. Otterness, M. Yang, S. Rust, E. Park, J. Anderson, F. D. Smith, A. Berg, and S. Wang, "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads," in *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium*, 2017, pp. 353–363.

[37] N. Otterness and J. H. Anderson, "AMD GPUs as an alternative to NVIDIA for supporting real-time workloads," in *Proceedings of the 32nd Euromicro Conference on Real-Time Systems*, 2020, pp. 10:1–10:23.

[38] ——, "Exploring AMD GPU scheduling details by experimenting with "worst practices"," in *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, 2021.

[39] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[40] J. S. Pérez, E. Meinhardt-Llopis, and G. Facciolo, "TV-L1 optical flow estimation," *Image Processing On Line*, vol. 2013, pp. 137–150, 2013.

[41] V. Prisacariu and I. Reid, "fastHOG–a real-time GPU implementation of HOG," Department of Engineering Science, Oxford University, Tech. Rep., 2009.

[42] D. Shapiro, "Wings come standard: Tesla Motors Model X rolls out with Tegra onboard," Online at https://blogs.nvidia.com/blog/2015/09/30/tesla-motors-model-x-nvidia/.

[43] U. Verner, A. Mendelson, and A. Schuster, "Batch method for efficient resource sharing in real-time multi-GPU systems," in *Proceedings of the 15th International Conference on Distributed Computing and Networking*, 2014, pp. 347–362.

[44] ——, "Scheduling periodic real-time communication in multi-GPU systems," in *Proceedings of the 23rd International Conference on Computer Communication and Networks*, 2014, pp. 1–8.

[45] U. Verner, A. Schuster, M. Silberstein, and A. Mendelson, "Scheduling processing of real-time data streams on heterogeneous multi-GPU systems," in *Proceedings of the 5th Annual International Systems and Storage Conference*, 2012, pp. 8:1–8:12.

[46] B. Ward and J. Anderson, "Supporting nested locking in multiprocessor real-time systems," in *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, 2012, pp. 223–232.

[47] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2010, pp. 235–246.

[48] 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 *Proceedings of the 30th International Conference on Supercomputing*, 2016, pp. 30:1–30:13.

[49] M. Yang, T. Amert, K. Yang, N. Otterness, J. Anderson, F. D. Smith, and S. Wang, "Making OpenVX really 'real time'," in *Proceedings of the 39th Real-Time Systems Symposium*, 2018, pp. 80–93.

[50] M. Yang, N. Otterness, T. Amert, J. Bakita, J. Anderson, and F. D. Smith, "Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems," in *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, 2018, pp. 20:1–20:21.

[51] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, J. Anderson, and J.-M. Frahm, "Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge," in *Proceedings of the 25th Real-Time and Embedded Technology and Applications Symposium*, 2019, pp. 305–317.

[52] C. Zach, T. Pock, and H. Bischof, "A duality based approach for realtime TV-L1 optical flow," in *Proceedings of the 29th Symposium of the German Association for Pattern Recognition*, 2007, pp. 214–223.

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

[54] H. Zhou, S. Bateni, and C. Liu, "S^3DNN: Supervised streaming and scheduling for GPU-accelerated real-time DNN workloads," in *Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2018, pp. 190–201.