Making OpenVX Really “Real Time”[*]

Ming Yang[1], Tanya Amert[1], Kecheng Yang[1,2], Nathan Otterness[1], James H. Anderson[1], F. Donelson Smith[1], and Shige Wang[3]
[1]University of North Carolina at Chapel Hill [2]Texas State University [3]General Motors Research

Abstract

OpenVX is a recently ratified standard that was expressly proposed to facilitate the design of computer-vision (CV) applications used in real-time embedded systems. Despite its real-time focus, OpenVX presents several challenges when validating real-time constraints. Many of these challenges are rooted in the fact that OpenVX only implicitly defines any notion of a schedulable entity. Under OpenVX, CV applications are specified in the form of processing graphs that are inherently considered to execute monolithically end-to-end. This monolithic execution hinders parallelism and can lead to significant processing-capacity loss. Prior work partially addressed this problem by treating graph nodes as schedulable entities, but under OpenVX, these nodes represent rather coarse-grained CV functions, so the available parallelism that can be obtained in this way is quite limited. In this paper, a much more fine-grained approach for scheduling OpenVX graphs is proposed. This approach was designed to enable additional parallelism and to eliminate schedulability-related processing-capacity loss that arises when programs execute on both CPUs and graphics processing units (GPUs). Response-time analysis for this new approach is presented and its efficacy is evaluated via a case study involving an actual CV application.

1 Introduction

The push towards deploying autonomous-driving capabilities in vehicles is happening at breakneck speed. Semi-autonomous features are becoming increasingly common, and fully autonomous vehicles at mass-market scales are on the horizon. In realizing these features, computer-vision (CV) techniques have loomed large. Looking forward, such techniques will continue to be of importance as cameras are both cost-effective sensors (an important concern in mass-market vehicles) and a rich source of environmental perception.

To facilitate the development of CV techniques, the Kronos Group has put forth a ratified standard called OpenVX [42]. Although initially released only four years ago, OpenVX has quickly emerged as the CV API of choice for real-time embedded systems, which are the standard’s intended focus. Under OpenVX, CV computations are represented as directed graphs, where graph nodes represent high-level CV functions and graph edges represent precedence and data dependencies across functions. OpenVX can be applied across a diversity of hardware platforms. In this paper, we consider its use on platforms where graphics processing units (GPUs) are used to accelerate CV processing.

Unfortunately, OpenVX’s alleged real-time focus reveals a disconnect between CV researchers and the needs of the real-time applications where their work would be applied. In particular, OpenVX lacks concepts relevant to real-time analysis such as priorities and graph invocation rates, so it is debatable as to whether it really does target real-time systems. More troublingly, OpenVX implicitly treats entire graphs as monolithic schedulable entities. This inhibits parallelism[1] and can result in significant processing-capacity loss in settings (like autonomous vehicles) where many computations must be multiplexed onto a common hardware platform.

In prior work, our research group partially addressed these issues by proposing a new OpenVX variant in which individual graph nodes are treated as schedulable entities [23, 51]. This variant allows greater parallelism and enables end-to-end graph response-time bounds to be computed. However, graph nodes remain as high-level CV functions, which is problematic for (at least) two reasons. First, these high-level nodes still execute sequentially, so some parallelism is still potentially inhibited. Second, such a node will typically involve executing on both a CPU and a GPU. When a node accesses a GPU, it suspends from its assigned CPU. Suspensions are notoriously difficult to handle in schedulability analysis without inducing significant capacity loss.

Contributions. In this paper, we show that these problems can be addressed through more fine-grained scheduling of OpenVX graphs. Our specific contributions are threefold.

First, we show how to transform the *coarse-grained* OpenVX graphs proposed in our group’s prior work [23, 51] to *fine-grained* variants in which each node accesses either a CPU or a GPU (but not both). Such transformations eliminate suspension-related analysis difficulties at the expense of (minor) overheads caused by the need to manage data sharing. Additionally, our transformation process exposes new potential parallelism at many levels. For example, because we decompose a coarse-grained OpenVX node into finer-grained schedulable entities, portions of such a node can now execute in parallel. Also, we allow not only successive invocations of the same graph to execute in parallel but even successive invocations of the same (fine-grained) *node*.

Second, we explain how prior work on scheduling processing graphs and determining end-to-end graph response-time bounds can be adapted to apply to our fine-grained

[*]Work supported by NSF grants CNS 1409175, CPS 1446631, CNS 1563845, and CNS 1717589, ARO grant W911NF-17-1-0294, and funding from General Motors.

[1]As discussed in Sec. 3, a recently proposed extension [18] enables more parallelism, but this extension is directed at throughput, not real-time predictability, and is not available in any current OpenVX implementation.

OpenVX graphs. The required adaptation requires new analysis for determining response-time bounds for GPU computations. We show how to compute such bounds for recent NVIDIA GPUs by leveraging recent work by our group on the functioning of these GPUs [1]. Our analysis shows that allowing invocations of the same graph node to execute in parallel is *crucial* in avoiding extreme capacity loss.

Third, we present the results of case-study experiments conducted to assess the efficacy of our fine-grained graph-scheduling approach. In these experiments, we considered six instances of an OpenVX-implemented CV application called HOG (histogram of oriented gradients), which is used in pedestrian detection, as scheduled on a multicore+GPU platform. These instances reflect a scenario where multiple camera feeds must be supported. We compared both analytical response-time bounds and observed response times for HOG under coarse- vs. fine-grained graph scheduling. We found that bounded response times could be guaranteed for all six camera feeds only under fine-grained scheduling. In fact, under coarse-grained scheduling, just one camera could (barely) be supported. We also found that observed response times were substantially lower under fine-grained scheduling. Additionally, we found that the overhead introduced by converting from a coarse-grained graph to a fine-grained one had modest impact. These results demonstrate the importance of enabling fine-grained scheduling in OpenVX if real time is *really* a first-class concern.

Organization. In the rest of the paper, we provided needed background (Sec. 2), describe our new fine-grained scheduling approach (Sec. 3), present the above-mentioned GPU response-time analysis (Sec. 4) and case study (Sec. 5), discuss related work (Sec. 6), and conclude (Sec. 7).

2 Background

In this section, we review prior relevant work on the real-time scheduling of DAGs and explain how this work was applied previously for coarse-grained OpenVX graph scheduling [23, 51]. The prior scheduling work of relevance takes considerable space to cover, so for the time being, we focus on generic (perhaps non-OpenVX) DAGs. Our review of this prior work draws heavily from a previous paper by three of the authors [52]. We specifically consider a system $G = \{G^1, G^2, \ldots, G^N\}$ comprised of N DAGs. The DAG G^i consists of n^i nodes, which correspond to n^i sequential tasks $\tau_1^i, \tau_2^i, \ldots, \tau_{n^i}^i$. Each task τ_v^i releases a (potentially infinite) sequence of *jobs* $\tau_{v,1}^i, \tau_{v,2}^i, \ldots$. The edges in G^i reflect precedence relationships. A particular task τ_v^i's *predecessors* are those tasks with outgoing edges directed to τ_v^i, and its *successors* are those with incoming edges directed from τ_v^i.

The j^{th} job of task τ_v^i, $\tau_{v,j}^i$, cannot commence execution until the j^{th} jobs of all of its predecessors finish. Such dependencies only exist for the *same invocation* of a DAG, not across invocations. That is, while jobs are sequential, *intra-task parallelism* (*i.e.*, parallel node invocation) is possible: *successive jobs of a task are allowed to execute in parallel.*

Ex. 1. Consider DAG G^1 in Fig. 1. Task τ_4^1's predecessors are tasks τ_2^1 and τ_3^1, *i.e.*, for any j, job $\tau_{4,j}^1$ waits for jobs $\tau_{2,j}^1$ and $\tau_{3,j}^1$ to finish. If intra-task parallelism is allowed, then $\tau_{4,j}^1$ and $\tau_{4,j+1}^1$ could execute in parallel. $\Diamond$

For simplicity, we assume that each DAG G^i has exactly one *source task* τ_1^i, with only outgoing edges, and one *sink task* $\tau_{n^i}^i$, with only incoming edges. Multi-source/multi-sink DAGs can be supported with the addition of singular "virtual" sources and sinks that connect multiple sources and sinks, respectively. Virtual sources and sinks have a worst-case execution time (WCET) of zero.

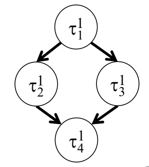


Figure 1: DAG G^1.

Source tasks are released sporadically, *i.e.*, for the DAG G^i, the job releases of τ_1^i have a minimum separation time, or *period*, denoted T^i. A non-source task τ_v^i $(v > 1)$ releases its j^{th} job $\tau_{v,j}^i$ after the j^{th} jobs of all its predecessors in G^i have completed. That is, letting $r_{v,j}^i$ and $f_{v,j}^i$ denote the release and finish times of $\tau_{v,j}^i$, respectively, $r_{v,j}^i \geq \max\{f_{w,j}^i \mid \tau_w^i \text{ is a predecessor of } \tau_v^i\}$. The *response time* of job $\tau_{v,j}^i$ is defined as $f_{v,j}^i - r_{v,j}^i$, and the *end-to-end response time* of the j^{th} invocation of the DAG G^i as $f_{n^i,j}^i - r_{1,j}^i$.

Deriving response-time bounds. An end-to-end response-time bound can be computed inductively for a DAG G^i by scheduling its nodes in a way that allows them to be viewed as sporadic tasks and by then leveraging response-time bounds applicable to such tasks. When viewing nodes as sporadic tasks, precedence constraints must be respected. This can be ensured by assigning an *offset* Φ_v^i to each task τ_v^i based on the response-time bounds applicable to "up-stream" tasks in G^i, and by requiring the j^{th} job of τ_v^i to be released exactly Φ_v^i time units after the release time of the j^{th} job of the source task τ_1^i, *i.e.*, $r_{v,j}^i = r_{1,j}^i + \Phi_v^i$, where $\Phi_1^i = 0$. With offsets so defined, every task τ_v^i in G^i (not just the source) has a period of T_i. Also, letting C_v^i denote the WCET of τ_v^i, its *utilization* can be defined as $u_v^i = C_v^i / T^i$.

Ex. 1 (cont'd). Fig. 2 depicts an example schedule for the DAG G^1 in Fig. 1. The first (resp., second) job of each task has a lighter (resp., darker) shading to make them easier to distinguish. Assume that the tasks have deadlines as shown, and response-time bounds of $R_1^1 = 9$, $R_2^1 = 5$, $R_3^1 = 7$, and $R_4^1 = 9$, respectively. Based on these bounds, we define corresponding offsets $\Phi_1^1 = 0$, $\Phi_2^1 = 9$, $\Phi_3^1 = 9$, and $\Phi_4^1 = 16$, respectively. With these response-time bounds, the end-to-end response-time bound that can be guaranteed is determined by R_1^1, R_3^1, and R_4^1 and is given by $R^1 = 25$. The task response-time bounds used here depend on the scheduler employed. For example, if all tasks are scheduled via the global earliest-deadline-first (G-EDF) scheduler, then per-task response-time bounds can be determined from tardiness analysis for G-EDF [20, 24]. In fact, this statement applies to any G-EDF-like (GEL) scheduler [25].[2] Such schedulers

[2]Under such a scheduler, each job has a priority point within a constant distance of its release; an earliest-priority-point-first order is assumed.

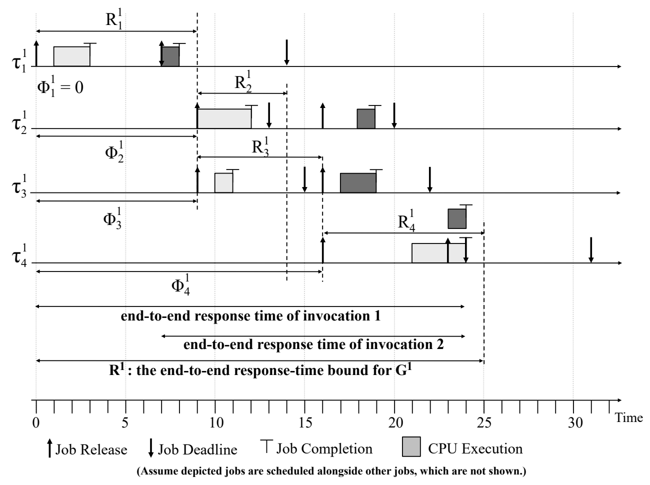


Figure 2: Example schedule of the tasks in G^1 in Fig. 1.

will be our focus. Recall that, according to the DAG-based task model introduced here, successive jobs of the same task might execute in parallel. We see this with jobs $\tau^1_{4,1}$ and $\tau^1_{4,2}$ in the interval $[23, 24)$. Such jobs could even finish out of release-time order due to execution-time variations. $\Diamond$

Early releasing. Using offsets may cause non-work-conserving behavior: a given job may be unable to execute even though all of its predecessor jobs have completed. Under any GEL scheduler, work-conserving behavior can be restored in such a case without altering response-time bounds [20, 24, 25] via a technique called *early releasing* [2], which allows a job to execute "early," before its "actual" release time.

Schedulability. For DAGs as considered here, schedulability conditions for ensuring bounded responses times hinge on conditions for ensuring bounded tardiness under GEL scheduling. Assuming a CPU-only platform with M processors, if intra-task parallelism is forbidden, then the required conditions are $u^i_v \leq 1$ for each v and $\sum u^i_v \leq M$ [20, 25]. On the other hand, if arbitrary intra-task parallelism is allowed, then only $\sum u^i_v \leq M$ is required and *per-task utilizations can exceed* 1.0 [24, 25]. These conditions remain unaltered if arbitrary early releasing is allowed.

Coarse-grained OpenVX graphs. In two prior papers by our group [23, 51], the techniques described above, but *without* intra-task parallelism, are proposed for scheduling acyclic[3] OpenVX graphs using G-EDF,[4] with graph nodes implicitly defined by high-level OpenVX CV functions. We call OpenVX graphs so scheduled *coarse-grained* graphs.

Given the nature of high-level CV functions, the nodes of a coarse-grained graph will typically involve executing both CPU code and GPU code. Executing GPU code can introduce *task suspensions*, and under G-EDF schedulability analysis, suspensions are typically dealt with using *suspension-oblivious analysis* [15]. This entails analytically viewing suspension time as CPU computation time and can result in significant processing-capacity loss.

3 Fine-Grained OpenVX Graph Scheduling

In this section, we propose a fine-grained scheduling approach for OpenVX graphs obtained by applying four techniques. First, to eliminate suspension-based capacity loss, we treat CPU code and GPU code as separate graph nodes. Second, to reduce response-time bounds, we allow intra-task parallelism. Third, to avoid non-work-conserving behavior and enable better observed response times, we allow early releasing. Finally, we use a scheduler (namely, G-FL—see below) that offers advantages over G-EDF. We elaborate on these techniques in turn below after first providing a brief introduction to GPU programming using CUDA.

CUDA basics. The general structure of a CUDA program is as follows: **(i)** allocate necessary memory on the GPU; **(ii)** copy input data from the CPU to the GPU; **(iii)** execute a GPU program called a *kernel*[5] to process the data; **(iv)** copy the results from the GPU back to the CPU; **(v)** free unneeded memory. To handle data dependencies, CUDA provides a set of synchronization functions. For example, such a function would be invoked between steps (iii) and (iv). These functions are configured on a per-device basis to wait via spinning or suspending. In this paper, we consider only waiting by suspending because the kernel executions in the workloads of interest are too long for spinning to be viable.

DAG nodes as CPU or GPU nodes. In our fine-grained scheduling approach, we avoid suspension-related capacity loss due to kernel executions by more finely decomposing an OpenVX graph so that each of its nodes is either a CPU node or a GPU node that executes a kernel. Additionally, we distinguish between regular CPU nodes and the necessary CPU work to launch a GPU kernel and await its results. In this paper, we assume that copy operations are included in CPU nodes. In the workloads of interest to us, copies are short, so any resulting suspension-based capacity loss is minor. More lengthy copies could instead be handled as separate nodes, similarly to how we handle kernels.

In the rest of this section, we use a continuing example to illustrate various nuances of our fine-grained approach.

Ex. 2. Fig. 3(a) depicts a simple coarse-grained graph comprised of two tasks, τ^2_x and τ^2_y. Fig. 3(b) shows a fine-grained representation of this same graph. Task τ^2_x is a simple CPU-only CV function and is represented by one fine-grained CPU task, τ^2_1. τ^2_y is more complex, and its fine-grained representation consists of six tasks, $\tau^2_2, \cdots, \tau^2_7$, where τ^2_5 is a GPU task, and τ^2_4 and τ^2_6 are CPU tasks that launch the GPU kernel and await its completion, respectively.[6] $\Diamond$

[3]As described in these papers, cycles can be dealt with by relaxing graph constraints or by combining certain nodes into "super-nodes." Adapting these techniques to our context is beyond the scope of this paper.

[4]While G-EDF was the focus of [51], in experiments presented in [23], real-time work was limited to execute on one socket of a multi-socket machine and thus was only globally scheduled within a socket.

[5]Unfortunate terminology, not to be confused with an OS kernel.

[6]The synchronization call to await results may be launched before the GPU kernel has completed, but this overhead is extremely short.

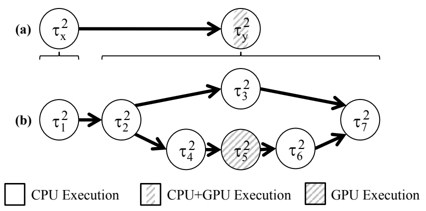


Figure 3: **(a)** Coarse- and **(b)** fine-grained representations of the same DAG G^2. τ_x^2 is simple sequential CPU code, so it is represented by one fine-grained task. τ_y^2 is more complex and consists of both CPU and GPU parts, some of which can execute in parallel.

An end-to-end response-time bound for a fine-grained graph can be obtained from per-node bounds as discussed in Sec. 2, with the copy operations in CPU nodes dealt with using suspension-oblivious analysis. For GPU nodes, new analysis is needed, which we provide for NVIDIA GPUs in Sec. 4.[7] Note that, in work on the prior coarse-grained approach [23, 51], locking protocols were used to preclude concurrent GPU access, obviating the need for such analysis.

Ex. 2 (cont'd). Possible schedules for the graphs in Fig. 3 are depicted in Fig. 4. As before, successive jobs of the same task are shaded differently to make them easier to distinguish. Recall from Sec. 2 that all tasks in a graph share the same period; in these schedules, all periods are 5 time units, shown as the time between successive job release times (up arrows).

Fig. 4(a) depicts the graph's schedule as a single monolithic entity, as implied by the OpenVX standard. OpenVX lacks any notion of real-time deadlines or phases, so these are excluded here, as is a response-time bound. The depicted schedule is a bit optimistic because the competing workload does not prevent the graph from being scheduled continuously. Under monolithic scheduling, the entire graph must complete before a new invocation can begin. As a result, the second invocation does not finish until just before time 28.

Fig. 4(b) depicts coarse-grained scheduling as proposed in prior work [23, 51], where graph nodes correspond to high-level CV functions, as in Fig. 3(a). Nodes can execute in parallel. For example, $\tau_{y,1}^2$ and $\tau_{x,2}^2$ do so in the interval $[5, 6)$. However, intra-task parallelism is not allowed: $\tau_{y,2}^2$ cannot begin until $\tau_{y,1}^2$ completes, even though its predecessor ($\tau_{x,2}^2$) is finished. Note that, under coarse-grained scheduling, GPU execution time is also analytically viewed is CPU execution time using suspension-oblivious analysis. This analytical impact is not represented in the figure.

Fig. 4(c) depicts a fine-grained schedule for the graph in Fig. 3(b), but without intra-task parallelism. In comparing insets (b) and (c), the difference is that nodes are now more fine-grained, enabling greater concurrency. As a result, $\tau_{7,2}^2$ completes earlier, at time 25. The detriments of suspension-oblivious analysis for GPU kernels are also now avoided. ◊

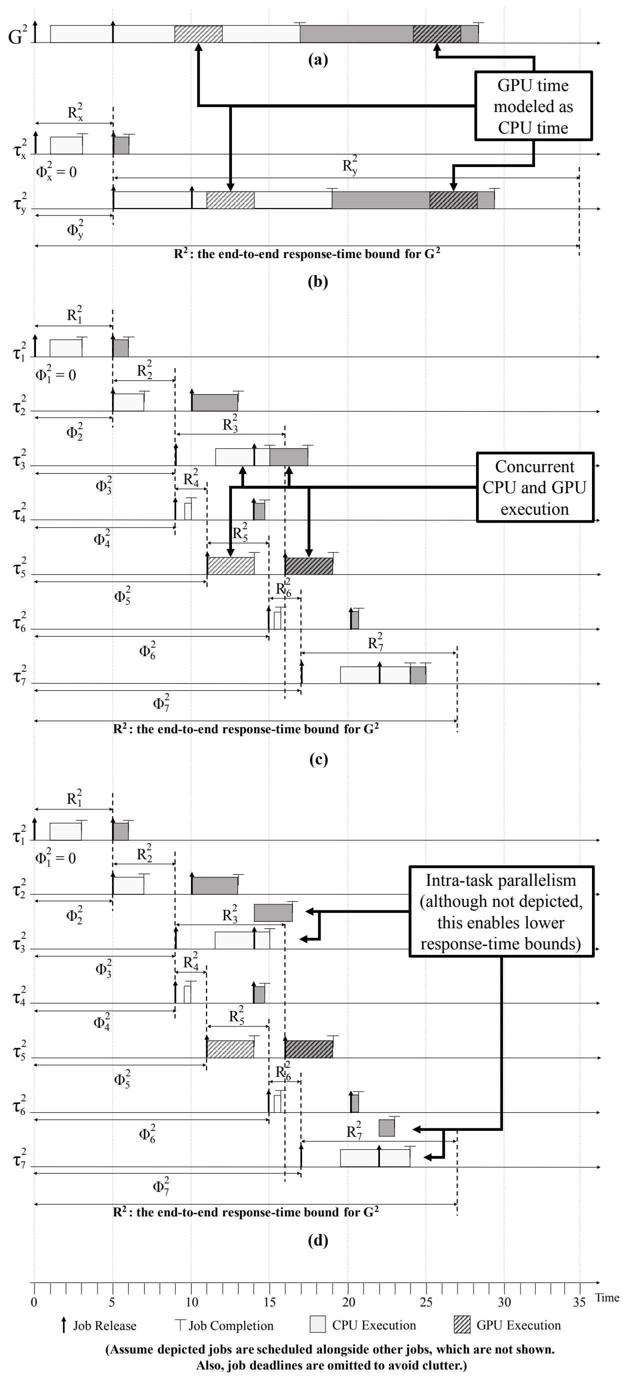


Figure 4: Example schedules of the tasks corresponding to the DAG-based tasks in G^2 in Fig. 3. **(a)** Monolithic scheduling. **(b)** Coarse-grained scheduling as in prior work. Fine-grained scheduling as proposed here **(c)** without and **(d)** with intra-task parallelism.

Intra-task parallelism. Our notion of fine-grained graph scheduling allows intra-task parallelism, *i.e.*, consecutive jobs of the same task may execute in parallel. Such parallelism can cause successive invocations of the same graph to complete out of order. This can be rectified via buffering.[8]

[7]Response times for copies, if handled as separate nodes, are trivial to bound because they are FIFO-scheduled.

[8]The buffer size can be determined based on the calculated response-time bounds.

Ex. 2 (cont'd). Lower response times are enabled by intra-task parallelism, as depicted in Fig. 4(d). In this schedule, $\tau_{3,1}^2$ and $\tau_{7,1}^2$ are able to execute in parallel with their predecessors $\tau_{3,2}^2$ and $\tau_{7,2}^2$, respectively, reducing the completion time of $\tau_{7,2}^2$ to time 23. Observe that $\tau_{7,2}^2$ completes before $\tau_{7,1}^2$, so some output buffering would be needed here. $\Diamond$

Allowing intra-task parallelism has an even greater impact on *analytically derived* response-time bounds [24], and as noted earlier, enables task utilizations to exceed 1.0.

Early releasing. Although omitted in Fig. 4 for clarity, early releasing can decrease *observed* response times without affecting analytical bounds.

Ex. 2 (cont'd). Allowing $\tau_{7,1}^2$ to be early released once $\tau_{6,1}^2$ completes in Fig. 4(d) reduces the overall graph's completion time to just under 23 time units. $\Diamond$

G-FL scheduling. The approach in Sec. 2 applies to any GEL scheduler. As shown in [25], the global fair-lateness (G-FL) scheduler is the "best" GEL scheduler with respect to tardiness bounds. We therefore perform CPU scheduling using G-FL instead of G-EDF.[9]

Periods. An additional benefit of fine-grained scheduling is that it allows for shorter periods.

Ex. 2 (cont'd). The period used in Fig. 4 seems reasonable in insets (c) and (d): notice that each job finishes before or close to its task's next job release. In contrast, in insets (a) and (b), response times could easily be unbounded. $\Diamond$

Recently proposed OpenVX extensions. The Kronos Group recently released the *OpenVX Graph Pipelining, Streaming, and Batch Processing Extension* [43], which enables greater parallelism in OpenVX graph executions. However, this extension is not available in any current OpenVX implementation and still lacks concepts necessary for ensuring real-time schedulability. While we have not specifically targeted this extension, an ancillary contribution of our work is to provide these needed concepts. In particular, the parallelism enabled by this extension's pipelining feature is actually subsumed by that allowed in our fine-grained graphs. Furthermore, the batching feature allows a node to process multiple frames instead of just one, potentially increasing computation cost; this could increase the node's utilization, possibly even exceeding 1.0. Introducing intra-task parallelism as we have done enables such nodes to be supported while still ensuring schedulability.

Rest of the paper. Despite the potential benefits of fine-grained scheduling described above, additional issues remain. First, as noted earlier, response-time bounds for GPU nodes are needed in order to compute end-to-end response-time bounds. We derive such bounds for NVIDIA GPUs in Sec. 4. Second, decomposing a coarse-grained graph node into a set of fine-grained ones can introduce additional overhead due to data sharing. We examine this issue via a case study in Sec. 5. In this study, we also compare both analytical response-time bounds and observed response times under coarse- vs. fine-grained scheduling.

[9]As explained in Sec. 5, we actually consider two variants of G-FL, a "clustered" variant in which G-FL is applied on a per-socket basis on our test platform, and a fully global variant that is applied across all sockets.

4 GPU Response-Time Bound

In this section, we derive a response-time bound for tasks executing on NVIDIA GPUs. To facilitate this, we first introduce additional background relating to NVIDIA GPUs.

4.1 NVIDIA GPU Details

The compute units of NVIDIA GPUs are *streaming multiproccessors* (*SMs*), typically comprised of 64 or 128 physical GPU cores. The SMs together can be logically viewed as an *execution engine* (EE). Execution on these GPUs is constrained by the number of available GPU threads, which we call *G-threads* to distinguish from CPU threads; on current NVIDIA GPUs, there are 2,048 G-threads per SM.

CUDA programs submit work to a GPU as kernels. A kernel is run on the GPU as a series of thread blocks. These thread blocks, or simply *blocks*, are each comprised of a number of G-threads. The number of blocks and G-threads per block (*i.e.*, the *block size*) are set at runtime when a kernel is launched. The GPU scheduler uses these values to assign work to the GPU's SMs. *Blocks are the schedulable entities on the GPU.* All G-threads in a block are always scheduled on the same SM, and execute non-preemptively.

In prior work, we documented scheduling rules used by NVIDIA GPUs when either all GPU work is submitted from the same address space or NVIDIA's multi-process service (MPS) is used, which we assume [1]. For simplicity, we restate here only the rules needed for our purposes. These rules govern how kernels are enqueued on and dequeued from a FIFO *EE queue*, as depicted in Fig. 5. CUDA also provides a concept called a *CUDA stream* that adds an additional layer of queueing in the form of *stream queues* prior to the EE queue. However, as explained later, we assume streams are used in a way that effectively obviates these additional queues. (Our statement of Rule G2 has been simplified from the original to reflect this assumption.) The following terminology is used in the rules below. A block is *assigned* when it is scheduled for execution on an SM. A kernel is *dispatched* when one or more of its blocks are assigned. A kernel is *fully dispatched* when its last block is assigned.

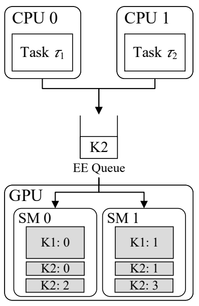
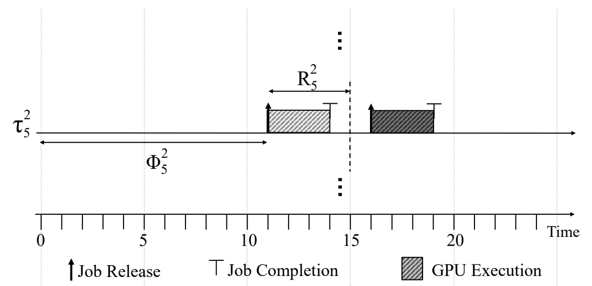


Figure 5: GPU scheduling; kernel Kk's bth block is Kk:b.

G2 A kernel is enqueued on the EE queue when launched.

G3 A kernel at the head of the EE queue is dequeued from that queue once it becomes fully dispatched.

X1 Only blocks of the kernel at the head of the EE queue are eligible to be assigned.



(Assume depicted jobs are scheduled alongside other jobs, which are not shown.)

Figure 6: GPU-relevant portion of the schedule in Fig. 4.

R1 A block of the kernel at the head of the EE queue is eligible to be assigned if its resource constraints are met.

Constrained resources include shared memory, registers, and (of course) G-threads. We assume that an NVIDIA-provided CUDA compiler option limiting register usage is applied to obviate blocking for registers. We consider techniques for handling shared-memory-induced blocking later.

Ex. 3. In the simple case depicted in Fig. 5, the GPU is comprised of two SMs. Two tasks submit one kernel each, and these are immediately enqueued on the EE queue upon launch (Rule G2). Kernel K1 is comprised of two blocks of 1,024 G-threads each; K2 is comprised of six blocks of 512 G-threads each. K1 is fully dispatched, so it has been dequeued from the EE queue (Rule G3). The remaining two blocks of $K2$ do not fit on either SM, and thus are not yet assigned (Rule R1); K2 is not fully dispatched, so it is still at the head of the EE queue. Any new kernel K3 would be behind K2 in the EE queue, so its blocks would be ineligible to be assigned until K2 is fully dispatched (Rule X1). ◊

4.2 System Model

One of the virtues of the DAG scheduling approach we proposed in Sec. 3 is that concerns pertaining to CPU and GPU work can be considered separately. Fig. 6 shows the subset of Fig. 4(d) involving GPU work; using our approach, GPU kernels are just sporadic tasks that can be analyzed independently from CPU tasks. In deriving a response-time bound, we therefore restrict our attention to a set τ of n independent sporadic GPU tasks $\{\tau_1, \tau_2, \cdots, \tau_n\}$, which are scheduled via the rules in Sec. 4.1 on a single GPU with multiple SMs. Each task τ_i has a period T_i, defined as before.

With our focus on GPU-using tasks, additional notation (to be illustrated shortly) is needed to express execution requirements. Each job of task τ_i consists of B_i *blocks*, each of which is executed in parallel by *exactly*[10] H_i G-threads. H_i is called the *block size*[11] of τ_i, and $H_{max} = \max_i\{H_i\}$ denotes the maximum block size in the system. We denote by C_i the per-block worst-case execution *workload* of a block of τ_i, where one unit of execution workload is defined by the work completed by one G-thread in one time unit. In summary, a GPU task τ_i is specified as $\tau_i = (C_i, T_i, B_i, H_i)$.

Note that C_i corresponds to an amount of execution *workload* instead of execution *time*. As each block of task τ_i requires H_i threads concurrently executing in parallel, the worst-case execution time of a block of τ_i is given by C_i/H_i.

Def. 1. (block length) For each task τ_i, its *maximum block length* is defined as $L_i = C_i/H_i$. The *maximum block length* of tasks in τ is defined as $L_{max} = \max_i\{L_i\}$.

The *utilization* of task τ_i is given by $u_i = C_i \cdot B_i/T_i$, and the *total system utilization* by $U_{sum} = \sum_{\tau_i \in \tau} u_i$.

Let $\tau_{i,j}$ denote the $j^{th} (j \geq 1)$ job of τ_i. The *release time*[12] of job $\tau_{i,j}$ is denoted by $r_{i,j}$, its *(absolute) deadline* by $d_{i,j} = r_{i,j} + T_i$, and its *completion time* by $f_{i,j}$; its *response time* is defined as $f_{i,j} - r_{i,j}$. A task's response time is the maximum response time of any its jobs.

SM constraints. We consider a single GPU platform consisting of g identical SMs, each of which consists of m G-threads (for NVIDIA GPUs, $m = 2048$). A single block of τ_i must execute on H_i G-threads that belong to the *same* SM. That is, as long as there are fewer than H_i available G-threads on each SM, a block of $\tau_{i,j}$ cannot commence execution even if the total number of available G-threads (from multiple SMs) in the GPU exceeds H_i. On the other hand, different blocks may be assigned to different SMs for execution even if these blocks are from the same job.

Similar to G-thread limitations, there are per-SM and per-block limits on shared-memory usage on NVIDIA GPUs. In experimental work involving CV workloads on NVIDIA GPUs spanning several years, *we have never observed blocking due to shared-memory limits on any platform for any workload*. Thus, in deriving our response-time bound in Sec. 4.4, we assume that such blocking does not occur. After deriving the bound, we discuss ways in which shared-memory blocking can be addressed if it becomes an issue.

Ex. 4. Our GPU task model is illustrated in Fig. 7. There are two SMs, and $B_1 = 2$ and $B_2 = 6$. The height of a rectangle denoting a block is given by H_i and its length, which denotes its runtime duration, by L_i; the area is bounded by C_i. ◊

Intra-task parallelism. We assume that intra-task parallelism is allowed: consecutive jobs of the same task can execute in parallel if both are pending and sufficient G-threads are available. This is often the case in CV processing pipelines where each video frame is processed independently. Additionally, Thm. 1 below shows that severe schedulability-related consequences exist if intra-task parallelism is forbidden. Practically speaking, intra-task parallelism can be enabled by assuming per-*job* streams. A stream is a FIFO queue of operations, so two kernels submitted to a single

[10]Blocks are executed in units of 32 G-threads called *warps*. *Warp schedulers* switch between warps to hide memory latency. This can create interference effects that must be factored into the timing analysis applied to blocks, which we assume is done in a measurement-based way. With warp-related interference incorporated into timing analysis, the G-threads in a block can be treated as executing simultaneously.

[11]Current NVIDIA GPUs require block sizes to be multiples of 32, and the CUDA runtime rounds up accordingly. Additionally, the maximum possible block size is 1,024. A task's block size is determined offline.

[12]For the time being, we assume that jobs of GPU tasks are not early released, but we will revisit this issue at the end of Sec. 4.4.

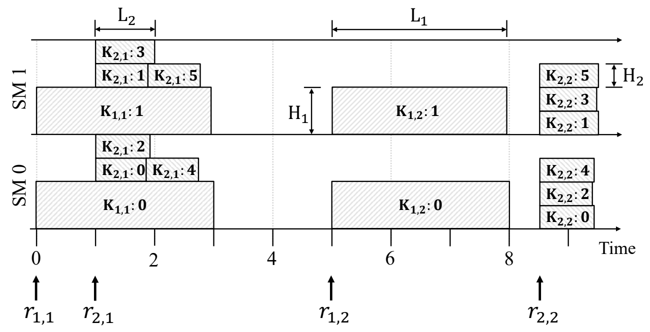


Figure 7: A possible schedule corresponding to Fig. 5, where $m = 2048$, $\tau_1 = (3072, 5, 2, 1024)$, and $\tau_2 = (512, 8, 6, 512)$; rectangle K$_{i,j}$:$b$ corresponds to the bth block of job $\tau_{i,j}$.

stream cannot execute concurrently. Thus, the alternative of using per-*task* streams would preclude intra-task parallelism. Note that, with each job issuing one kernel at a time, any actual stream queueing is obviated.

4.3 Total Utilization Constraint

According to the rules in Sec. 4.1, idle G-threads can exist while the kernel at the head of the EE queue has unassigned blocks. In particular, this can happen when the number of idle threads on any one SM is insufficient for scheduling such a block. Such scenarios imply that some *capacity loss* is fundamental when seeking to ensure response-time bounds for GPUs. We express such loss by providing a *total utilization bound* and proving that any system with U_{sum} at most that bound has bounded response times. The utilization bound we present relies on the following definition.

Def. 2. (unit block size) The *unit block size*, denoted by h, is defined by the greatest common divisor (gcd) of all tasks' block sizes and m, *i.e.*, $h = \gcd(\{H_i\}_{i=1}^{n} \cup \{m\})$.

The theorem below shows that capacity loss can be extreme if intra-task parallelism is forbidden.

Theorem 1. *With per-task streams, for any given g, m, H_{max}, and h, there exists a task system τ with U_{sum} greater than but arbitrarily close to h such that the response time of some task may increase without bound in the worst case.*

Proof. For any m and H_{max}, $m = Z \cdot h$ for some integer $Z \geq 1$ and $H_{max} = K \cdot h$ for some integer $K \geq 1$, because h is a common divisor of m and H_{max}. Recall that there are g SMs. Consider the following task system:

τ_i	C_i	T_i	B_i	H_i	L_i
τ_1	h	1	1	h	1
τ_2	$2\varepsilon \cdot H_{max}$	$1+\varepsilon$	1	H_{max}	2ε
τ_3	$2\varepsilon \cdot h$	$1+\varepsilon$	$g \cdot Z - K$	h	2ε

For this task system, $u_1 = h$, $u_2 = \frac{2H_{max}}{1+\varepsilon}\varepsilon$, and $u_3 = \frac{2h\cdot(g\cdot Z-K)}{1+\varepsilon}\varepsilon$, so $U_{sum} \to h^+$ as $\varepsilon \to 0^+$.

Now consider the following job execution pattern, which is illustrated in Fig. 8 for $g = 2$: τ_1 releases its first job at time 0, τ_2 and τ_3 release their first jobs at time $1 - \varepsilon$, all three tasks continue to release jobs as early as possible, and

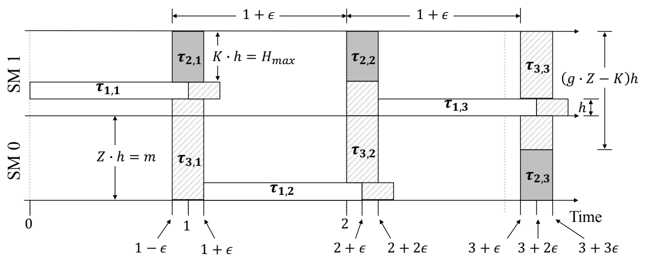


Figure 8: Unbounded response time using per-task streams.

every block executes for its worst-case execution workload. Assume that, every time when τ_2 and τ_3 simultaneously release a job, the job of τ_2 is enqueued on the EE queue first. Note that in Fig. 8, block boundaries for τ_3 are omitted when possible for clarity.

At time 0, $\tau_{1,1}$ is the only job in the EE queue and is therefore scheduled. Then, at time $1 - \varepsilon$, $\tau_{2,1}$ and $(g \cdot Z - K - 1)$ blocks of $\tau_{3,1}$ are scheduled for the interval $[1-\varepsilon, 1+\varepsilon)$. As a result, all available G-threads are then occupied. Therefore, the remaining one block of $\tau_{3,1}$ must wait until time 1 when $\tau_{1,1}$ finishes. Note that, with per-task streams, a job cannot enter the EE queue until the prior job of the same task completes. $\tau_{1,2}$ enters the EE queue at time 1 after $\tau_{3,1}$, which entered at time $1 - \varepsilon$. Thus, $\tau_{1,2}$ must wait to begin execution until after the last block of $\tau_{3,1}$ is assigned and once sufficient G-threads become available at time $1 + \varepsilon$.

This pattern repeats, with task τ_1 releasing a job every time unit but finishing a job every $1 + \varepsilon$ time units. Thus, its response time increases without bound. □

For example, on the test platform considered in Sec. 5, $g = 80$, $m = 2048$, and h can be as small as 32. Thus, close to 99.98% of the hardware capacity may be wasted!

In contrast, as we show in Sec. 4.4, if intra-task parallelism is allowed, then any task set with $U_{sum} \leq g \cdot (m - H_{max} + h)$ has bounded response times. Furthermore, the following theorem shows that this utilization bound is tight (under our analysis assumptions).

Theorem 2. *With per-job streams, for any given g, m, H_{max}, and h, there exists a task system τ with U_{sum} greater than but arbitrarily close to $g \cdot (m - H_{max} + h)$ such that the response time of some task may increase without bound in the worst case.*

Proof. For any m and H_{max}, integers P and Q exist such that $m = P \cdot H_{max} + Q$, where $P \geq 1$ and $0 \leq Q < H_{max}$. Furthermore, by the definition of h, $H_{max} = K \cdot h$ for some integer $K \geq 1$, and $m = Z \cdot h$ for some integer $Z \geq 1$. Thus, $m = P \cdot H_{max} + Q = P \cdot K \cdot h + Q = Z \cdot h$. Consider the following task set (if $Q = 0$, then τ_2 need not exist):

τ_i	C_i	T_i	B_i	H_i	L_i
τ_1	$\varepsilon \cdot H_{max}$	1	$g \cdot P$	H_{max}	ε
τ_2	$\varepsilon \cdot Q$	1	g	Q	ε
τ_3	h	1	$g \cdot (Z - K + 1)$	h	1

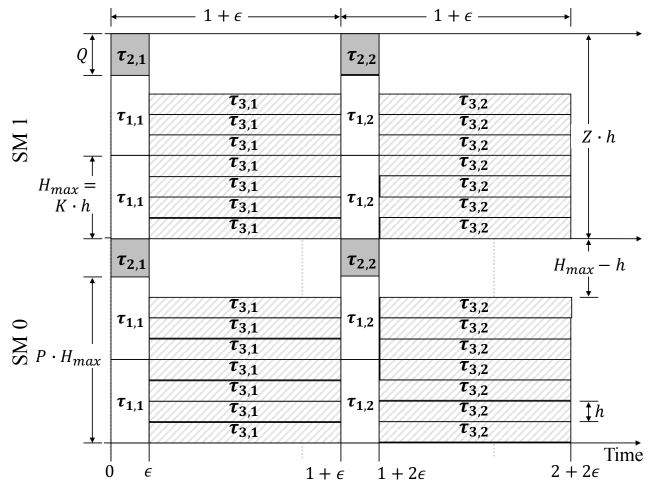


Figure 9: Unbounded response time using per-job streams.

For this task system, $u_1 = (H_{max} \cdot g \cdot P)\varepsilon$, $u_2 = (Q \cdot g)\varepsilon$, and $u_3 = h \cdot g \cdot (Z - K + 1) = g \cdot (m - H_{max} + h)$, so $U_{sum} \to g \cdot (m - H_{max} + h)^+$ as $\varepsilon \to 0^+$.

Now consider the following job execution pattern, which is illustrated in Fig. 9 for $g = 2$: all three tasks release jobs as soon as possible, *i.e.*, at time instants $0, 1, 2, \ldots$, the EE enqueue order is always τ_1, τ_2, and then τ_3, and every block executes for its worst-case execution workload.

At time 0, the $g \cdot P$ blocks of τ_1 are scheduled first, leaving Q available G-threads in each SM. Next, the g blocks of τ_2 are scheduled using the remaining Q G-threads on each SM. Thus, all G-threads are fully occupied in the time interval $[0, \varepsilon)$. As we often see in experiments, the $g \cdot (Z - K + 1)$ blocks of τ_3 are distributed to the g SMs evenly, and are scheduled for the time interval $[\varepsilon, 1 + \varepsilon)$. Note that, although we currently do not have sufficient evidence to guarantee this even distribution, it at least represents a potential *worst case*.

Notice that there are only $m - (Z - K + 1) \cdot h = (H_{max} - h)$ G-threads available on each of the g SMs during the interval $[\varepsilon, 1 + \varepsilon)$. Therefore, none of the blocks of $\tau_{1,2}$, which has a block size of H_{max}, will be scheduled before time $1 + \varepsilon$. As a result, no blocks of $\tau_{2,2}$ or $\tau_{3,2}$ will be scheduled before time $1 + \varepsilon$ either, because they are enqueued after $\tau_{1,2}$ on the FIFO EE queue.

This pattern repeats, with each of the three tasks releasing a job every time unit but finishing a job every $1 + \varepsilon$ time units, so the response time of each task increases without bound. □

4.4 Response-Time Bound

In this section, we derive a response-time bound assuming per-job streams are used (*i.e.*, intra-task parallelism is allowed) and the following holds.

$$U_{sum} \leq g \cdot (m - H_{max} + h) \tag{1}$$

Our derivation is based on the following key definition.

Def. 3. (busy and non-busy) A time instant is called *busy* if and only if at most $(H_{max} - h)$ G-threads are idle in *each* of the g SMs; a time instant is called *non-busy* if and only if at least H_{max} G-threads are idle in *some* of the g SMs. A time interval is called *busy* if and only if every time instant in that interval is busy.

By Def. 2, h is the minimum amount by which the number of idle G-threads can change, so "more than $(H_{max} - h)$ G-threads are idle" is equivalent to "at least H_{max} G-threads are idle." Thus, busy and non-busy time instants are well-defined, *i.e.*, a time instant is either busy or non-busy.

To derive response-time bounds for all tasks in the system, we bound the response time of an arbitrary job $\tau_{k,j}$. The following two lemmas bound the unfinished workload at certain time instants. In the first lemma, t_0 denotes the latest non-busy time instant at or before $\tau_{k,j}$'s release time $r_{k,j}$, *i.e.*, $t_0 = r_{k,j}$ or $(t_0, r_{k,j}]$ is a busy interval.

Lemma 1. *At time t_0, the total unfinished workload from jobs released at or before t_0, denoted by $W(t_0)$, satisfies $W(t_0) \leq L_{max} \cdot (g \cdot m - H_{max})$.*

Proof. Suppose there are b blocks, $\beta_1, \beta_2, \ldots, \beta_b$, that have been released but are unfinished at time t_0. For each block β_i, let $H(\beta_i)$ denote its block size and let $C(\beta_i)$ denote its worst-case execution workload. By definition, t_0 is a non-busy time instant, so by Def. 3, at least H_{max} G-threads are idle in some SM at time t_0. Because this SM has enough available G-threads to schedule any of the b blocks, they all must be scheduled at time t_0. These facts imply

$$\sum_{i=1}^{b} H(\beta_i) \leq g \cdot m - H_{max}. \tag{2}$$

Therefore,
$$\begin{aligned}
W(t_o) &= \sum_{i=1}^{b} C(\beta_i) \\
&= \{\text{by the definition of } L_i \text{ in Def.1}\} \\
&\quad \sum_{i=1}^{b} (L(\beta_i) \cdot H(\beta_i)) \\
&\leq \{\text{by the definition of } L_{max} \text{ in Def.1}\} \\
&\quad \sum_{i=1}^{b} (L_{max} \cdot H(\beta_i)) \\
&= \{\text{rearranging}\} \\
&\quad L_{max} \cdot \sum_{i=1}^{b} H(\beta_i) \\
&\leq \{\text{by (2)}\} \\
&\quad L_{max} \cdot (g \cdot m - H_{max}).
\end{aligned}$$

The lemma follows. □

Lemma 2. *At time $r_{k,j}$, the total unfinished workload from jobs released at or before $r_{k,l}$, denoted by $W(r_{k,j})$, satisfies $W(r_{k,j}) < L_{max} \cdot (g \cdot m - H_{max}) + \sum_{i=1}^{n}(B_i \cdot C_i)$.*

Proof. Let $\mathsf{new}(t_0, r_{k,j})$ denote the workload released during the time interval $(t_0, r_{k,j}]$, and let $\mathsf{done}(t_0, r_{k,j})$ denote the workload completed during the time interval $(t_0, r_{k,j}]$. Then,

$$W(r_{k,j}) = W(t_0) + \mathsf{new}(t_0, r_{k,j}) - \mathsf{done}(t_0, r_{k,j}). \quad (3)$$

As each task τ_i releases consecutive jobs with a minimum separation of T_i, $\mathsf{new}(t_0, r_{k,j})$ can be upper bounded by

$$\begin{aligned}\mathsf{new}(t_0, r_{k,j}) &\le \sum_{i=1}^{n} \left(\left\lceil \frac{r_{k,j} - t_0}{T_i} \right\rceil \cdot B_i \cdot C_i \right) \\ &< \{\text{because } \lceil a \rceil < a + 1\} \\ &\quad \sum_{i=1}^{n} \left(\left(\frac{r_{k,j} - t_0}{T_i} + 1 \right) \cdot B_i \cdot C_i \right) \\ &= \{\text{rearranging}\} \\ &\quad (r_{k,j} - t_0) \sum_{i=1}^{n} \frac{B_i \cdot C_i}{T_i} + \sum_{i=1}^{n} (B_i \cdot C_i) \\ &= \{\text{by the definitions of } u_i \text{ and } U_{sum}\} \\ &\quad (r_{k,j} - t_0)\, U_{sum} + \sum_{i=1}^{n} (B_i \cdot C_i). \quad (4)\end{aligned}$$

By Def. 3, $(t_0, r_{k,j}]$ being a busy time interval implies that at most $(H_{max} - h)$ G-threads in each of the g SMs are idle at any time instant in this time interval. That is, at least $g \cdot (m - H_{max} + h)$ G-threads are occupied executing work at any time instant in $(t_0, r_{k,j}]$. Therefore,

$$\mathsf{done}(t_0, r_{k,j}) \ge (r_{k,j} - t_0) \cdot g \cdot (m - H_{max} + h). \quad (5)$$

By (3), (4), and (5),

$$\begin{aligned}W(r_{k,j}) &< W(t_0) + (r_{k,j} - t_0)\, U_{sum} + \sum_{i=1}^{n} (B_i \cdot C_i) \\ &\quad - (r_{k,j} - t_0) \cdot g \cdot (m - H_{max} + h) \\ &= \{\text{rearranging}\} \\ &\quad (r_{k,j} - t_0) \cdot (U_{sum} - g \cdot (m - H_{max} + h)) \\ &\quad + W(t_0) + \sum_{i=1}^{n} (B_i \cdot C_i) \\ &\le \{\text{by (1) and } t_0 \le r_{k,j}\} \\ &\quad W(t_0) + \sum_{i=1}^{n} (B_i \cdot C_i) \\ &\le \{\text{by Lemma 1}\} \\ &\quad L_{max} \cdot (g \cdot m - H_{max}) + \sum_{i=1}^{n} (B_i \cdot C_i).\end{aligned}$$

The lemma follows. □

The following theorem provides our response-time bound.

Theorem 3. *$\tau_{k,j}$ finishes the execution of all of its blocks by time $r_{k,j} + R_k$, where*

$$R_k = \frac{L_{max} \cdot (g \cdot m - H_{max}) + \sum_{i=1}^{n} (B_i \cdot C_i) - C_k}{g \cdot (m - H_{max} + h)} + L_k. \quad (6)$$

Proof. Since the EE queue is FIFO, we omit all jobs released after $r_{k,j}$ in the analysis. Thus, any workload executed at or after $r_{k,j}$ is from $W(r_{k,j})$. We also assume each block of $\tau_{k,j}$ executes for its worst-case workload C_k (if any of its blocks executes for less, $\tau_{k,j}$'s completion is not delayed).[13]

Let β^* denote the *last-finished* block of $\tau_{k,j}$. Then, the workload from other blocks or jobs at $r_{k,j}$ is $W(r_{k,j}) - C_k$. Let t^* denote the time instant at which β^* starts execution. Then, $[r_{k,j}, t^*)$ is a busy interval (else β^* would have executed before time t^*). Let $\mathsf{done}(r_{k,j}, t^*)$ denote the workload completed during the time interval $[r_{k,j}, t^*)$. Then, by Def. 3,

$$\mathsf{done}(r_{k,j}, t^*) \ge (t^* - r_{k,j}) \cdot g \cdot (m - H_{max} + h). \quad (7)$$

The workload C_k from β^* executes beyond time t^*, so $\mathsf{done}(r_{k,j}, t^*) \le W(r_{k,j}) - C_k$. By (7), this implies $t^* \le r_{k,j} + \frac{W(r_{k,j}) - C_k}{g \cdot (m - H_{max} + h)}$. At time t^*, β^* executes continuously for L_k time units. Thus, β^* finishes by time $r_{k,j} + \frac{W(r_{k,j}) - C_k}{g \cdot (m - H_{max} + h)} + L_k$. By Lemma 2, the theorem follows. □

Discussion. As noted in Sec. 4.2, the absence of shared-memory-induced blocking is assumed in the above analysis. This limitation could be eased by introducing blocking terms, but we leave this for future work. Alternatively, through offline profiling, one could restrict the per-SM G-thread count of m to some value m' such that, if only m' G-threads are used per SM, no shared-memory-induced blocking ever occurs. The analysis above could then be applied with m replaced by m'. While one might expect this analysis to be sustainable in the sense that m per-SM G-threads could really be used at runtime, we found a counterexample where increasing m' to m causes response times to increase. Thus, the restricted G-thread count of m' would actually have to be enforced. This could potentially be done by creating a never-ending kernel per SM that monopolizes $m - m'$ G-threads.

Early releasing (see Secs. 2 and 3) must be restricted for GPU tasks. Because the FIFO EE queue effectively prioritizes work by *actual* enqueueing times, uncontrolled early releasing can change priorities. Also, this can lead to a violation of the sporadic task model if consecutive jobs of the same task τ_i have enqueueing times less than T_i time units apart. Thus, the early releasing of GPU tasks must be guarded to ensure that this minimum separation is maintained.

5 Case Study

In this section, we detail the case-study experiments we performed, and compare our fine-grained DAG scheduling approach to monolithic and coarse-grained DAG scheduling.

[13] Other jobs' blocks may or may not execute for their worst case.

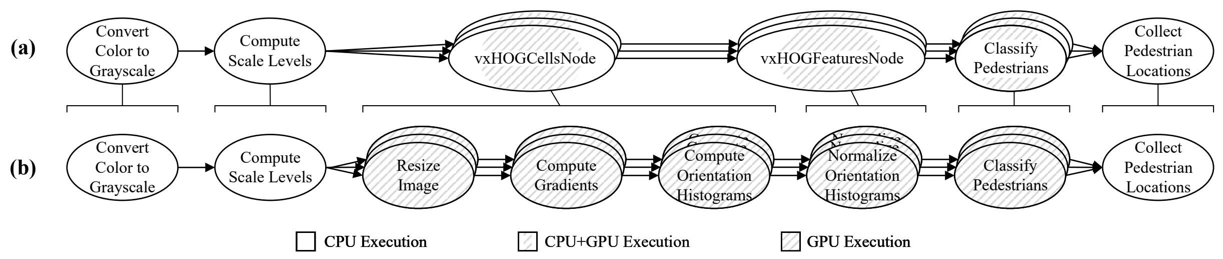


Figure 10: **(a)** Monolithic/coarse-grained and **(b)** fine-grained HOG DAGs. Our experiments used 13 scale levels; three are shown here. In (b), there are also two CPU nodes per kernel that launch that kernel and await its results. These nodes have short execution times and are omitted from much of our discussion for simplicity. However, they were fully considered in our evaluation.

5.1 Experimental Evaluation

All of our case-study experiments focused on the *Histogram of Oriented Gradients* (HOG) algorithm, a well-known CV technique for recognizing pedestrians in input images [19].

Why HOG? The HOG algorithm is required by the OpenVX 1.2 specification,[14] ensuring its relevance in real-world graph-based CV. HOG is inherently a multi-stage technique: it calculates a directional *gradient* for each pixel, sorts gradients into histograms, normalizes lighting and contrast, and then performs classification. These computations are performed at multiple image-scale levels, using successively smaller versions of the original image. These steps require both CPU and GPU computations, meaning that HOG fits naturally into a DAG-based model.

HOG implementation and DAG variations. At the time of writing, no open-source implementations of version 1.2 of OpenVX exist, so we based our case-study experiments on the HOG implementation available in the open-source CV library *OpenCV*.[15] OpenCV provides CV functions, but does not structure computations as DAGs. To create DAGs, we split the OpenCV HOG code into separate functions, and designated each function as a DAG node. We compared the response times of successively finer-grained notions of DAG scheduling, corresponding to monolithic, coarse-grained, and fine-grained HOG DAGs.[16]

The monolithic version of HOG corresponds to the type of DAG that one might specify using OpenVX, and consists of a single DAG of six types of nodes (three are replicated per scale level), as shown in Fig. 10(a). Implementing this DAG required the fewest changes to OpenCV code, as monolithic execution requires only a single thread to sequentially execute the six nodes' functions. Coarse-grained HOG uses the same DAG as monolithic HOG, but, as discussed in Sec. 2, each of the six nodes is a schedulable entity, with scheduling via G-EDF with early releasing. We also used G-EDF, but without early releasing, as a monolithic DAG scheduler.

In fine-grained HOG, several of the coarse-grained nodes are refined, as shown in Fig. 10(b). This DAG reflects our new fine-grained approach, where nodes are treated as tasks and the techniques (early releasing, intra-task parallelism, and G-FL scheduling) in Sec. 3 are applied.

Fine-grained DAG implementation. Implementing fine-grained HOG introduced a series of challenges. For example, intra-task parallelism required multiple instances of each DAG to ensure each node can have multiple jobs executing in parallel. Other challenges included priority points that varied (for launching GPU kernels and awaiting results), handling inter-process communication (IPC) between CPU and GPU nodes, enforcing guards on early releasing for GPU nodes, and computing task offsets from response-time bounds in order to run the experiments.

As in prior work [23], we used PGM$^{\mathrm{RT}}$ [21] to handle IPC in the coarse- and fine-grained HOG variants. PGM$^{\mathrm{RT}}$ introduces producer/consumer buffers and mechanisms that enable producer nodes to write output data and consumer nodes to suspend until data is available on all inbound edges.

Test platform. Our evaluation platform was selected to over-approximate current NVIDIA embedded offerings for automotive systems, such as the Drive PX2. This platform features a single NVIDIA Titan V GPU, two eight-core Intel CPUs, and 32 GB of DRAM. Each core features a 32-KB L1 data cache, a 32-KB L1 instruction cache, and a 1-MB L2 cache, and all eight cores on a socket share an 11-MB L3 cache. The system was configured to run Ubuntu 16.04 as an OS, using version 2017.1 of the LITMUS$^{\mathrm{RT}}$ kernel [39], with hardware multi-threading disabled.

Overall computational workload. One would expect contention for hardware resources in many real-world use cases, such as an autonomous vehicle that processes data from multiple camera feeds. We approximated a "contentious" workload by executing six HOG processes on our hardware platform—the limit based on our platform's DRAM, CPU, and GPU capacity. This number of HOG processes makes ensuring bounded response times difficult without careful consideration of resource allocation. This scenario also reflects the very real possibility of executing at high platform utilization, as is commonly done in the automotive industry. To

[14]`https://www.khronos.org/registry/OpenVX/specs/1.2/OpenVX_Specification_1_2.pdf`, Sec. 3.53.1.

[15]`https://docs.opencv.org/3.4.1/d5/d33/structcv_1_1HOGDescriptor.html`.

[16]Our source code is available online at `https://github.com/Yougmark/opencv/tree/rtss18`.

	Monolithic G-EDF	Monolithic C-EDF	Coarse-Grained G-EDF	Coarse-Grained C-EDF	Fine-Grained G-FL	Fine-Grained C-FL
Analytical Bound (ms)	N/A	N/A	N/A	N/A	542.39	477.25
Observed Maximum Response Time (ms)	170091.06	243745.21	427.07	428.50	125.66	131.43
Observed Average Response Time (ms)	84669.47	121748.05	136.57	121.52	65.99	66.06

Table 1: Analytical and observed response times. A bound of N/A indicates unschedulability, so no bound could be computed.

ensure consistency with the GPU scheduling rules in Sec. 4.1, we conducted all of our experiments using NVIDIA's MPS.

Video-frame-processing can potentially experience cache-affinity-loss issues under global scheduling. We therefore considered two variants of both G-EDF and G-FL: a truly global variant where any of the six DAGs can be scheduled on any of our platform's 16 CPUs, and a "clustered" variant where the six DAGs are partitioned between the machine's two sockets, with scheduling being "global" only within a socket. We refer to the latter variants as C-EDF and C-FL, respectively, where the "C" prefix stands for "clustered."

5.2 Results

Our experiments were designed to examine analytical response-time bounds and observed response times under the considered scheduling approaches. We also sought to examine the overhead required to support fine-grained scheduling.

Analytical bounds. To compute analytical response-time bounds, we first computed CPU WCETs and worst-case GPU workloads via a measurement process. All worst-case values were calculated as the 99th percentile of 30,000 samples obtained with all six DAGs executing together to cause contention. For each GPU task τ_i, we used NVIDIA's profiling tool `nvprof` to measure B_i and H_i, and instrumented the CUDA kernels to measure L_i *on the GPU* using the `globaltimer` performance-counter register. For HOG, $H_{max} = 256$ and $h = 64$. We measured CPU WCETs using Feather-Trace [14]. For all DAGs, $T_i = 33ms$.

We computed fine-grained response-time bounds by using Thm. 3 in Sec. 4.4 for GPU nodes and Thm. 2 from [53] for CPU nodes and by then applying the techniques in Sec. 3 to obtain an overall end-to-end bound. We tried computing analytical bounds for the coarse-grained (resp., monolithic) C-EDF and G-EDF variants using prior work [51] (resp., [20]), but found these variants to be unschedulable.[17] These results are summarized in the first row of Tbl. 1.

Obs. 1. With respect to schedulablity, the monolithic and coarse-grained variants could not even come close to supporting all six cameras (*i.e.*, DAGs).

With respect to schedulability, the monolithic variants could not even support one camera, because the overall execution time of a single monolithic DAG far exceeds its period. The coarse-grained variants were only slightly better, being able to support just one camera (in which case the choice of variant, C-EDF vs. G-EDF, is of little relevance). In this

[17]In the original coarse-grained work [23, 51], locking protocols were used to preclude concurrent GPU accesses. We instead allowed concurrent accesses and used the analysis in Sec. 4, but the coarse-grained variants were still unschedulable.

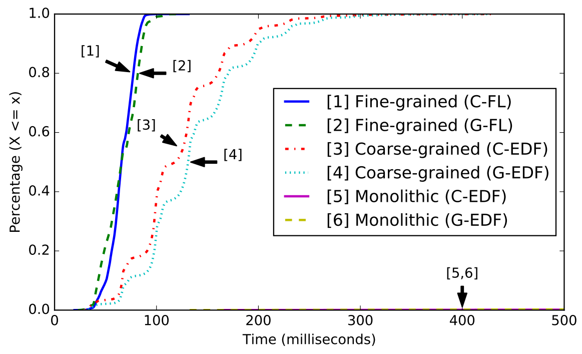


Figure 11: CDFs of response times for varying DAG granularity.

case, adding a second HOG DAG increased GPU responses to the point of causing a CPU utilization-constraint violation. Note that the increase in CPU utilization is due to using suspension-oblivious analysis.

Obs. 2. With respect to schedulability, both fine-grained variants were able to support all six cameras.

The better schedulability of the fine-grained variants largely resulted from scheduling shorter tasks with intra-task parallelism, though switching to a fair-lateness-based scheduler also proved beneficial. In particular, we found that the scheduler change reduced node response times by 0.1–9.9%. While this reduction is modest, it is still useful. Nonetheless, these reductions suggest that most of the schedulability improvements stemmed from increasing parallelism.

Observed response times. Fig. 11 plots DAG response-time distributions, which we computed for all tested variants from measurement data. Corresponding worst- and average-case times are also reported in Tbl. 1.

Obs. 3. The average (resp., worst-case) observed response time under the fine-grained variants was around 66 ms (resp., 130 ms), which is substantially lower than the non-fine-grained variants. (For reference, the response time of an alert driver is reported to be around 700 ms [27].)

This observation is supported by Fig. 11 and Tbl. 1. Note that the difference between clustered and global scheduling was not substantial. This is because the aggregate memory footprint of all frames concurrently being accessed under both variants tended to far exceed the size of the L3 cache.

Obs. 4. The analytical fine-grained response-time bounds upper bounded the observed worst-case times.

This observation is supported by Fig. 11 and Tbl. 1. While the listed bounds of 477.25 ms and 542.39 ms in Tbl. 1 may seem high, note that they are based on considering worst-case scenarios that may be unlikely to occur in practice (and they are still within the limit mentioned in [27]). Moreover, the monolithic and coarse-grained variants were unable to guarantee *any* bounds when scheduling all six DAGs.

Obs. 5. Observed response times exhibited lower variation under fine-grained scheduling.

This observation is supported by Fig. 11. The fine-grained variances in this plot are several orders of magnitude less than the variances for the other variants.

Obs. 6. Early releasing improved observed response times.

To verify this observation, we conducted additional experiments in which we disabled early releasing for the fine-grained G-FL variant. In these experiments, we found that early releasing reduced observed response times by 49%.

Overhead of DAG conversion. We estimated the overhead of converting from a coarse-grained DAG to a fine-grained one by comparing the computed WCET of every coarse-grained node with the sum of the computed WCETs of the fine-grained nodes that replace it. The total percentage increase across all nodes was deemed to be *overhead*.

Obs. 7. The additional overhead introduced to support fine-grained scheduling had modest impact.

From our collected data, the total overhead was 14.15%.

6 Related Work

This paper and the prior work it builds upon [23, 51] focus specifically on supporting GPU-using real-time CV workloads. The only other work on this topic known to us is a recent paper by Zhou *et al.* [56] that proposes a technique based on reordering and batching (see Sec. 3) kernels to speed deep neural networks. However, they provided no schedulability analysis. More broadly, a large body of work of a general nature exists pertaining to GPU-enabled real-time systems. Much of this work focuses on either treating GPUs as non-shared devices to enable highly predictable GPU usage [22, 32, 33, 46, 47, 48, 50] or seeking to improve schedulability by simulating preemptive execution [7, 32, 34, 57]. Other work has focused on timing analysis for GPU workloads [8, 9, 10, 11, 12], techniques for remedying performance bottlenecks [28], direct I/O communication [3], energy management [45], and techniques for managing or evaluating GPU hardware resources, notably cache and DRAM [16, 17, 26, 29, 35, 40, 49].

The scheduling rules discussed in Sec. 4.1 resulted from an effort by our group to develop a *model* of GPU execution, particularly for NVIDIA GPUs. This effort has delved into a number of aspects of NVIDIA GPUs marketed for embedded systems [1, 41, 54]. Much of this work is rooted in the observation that GPU sharing will become essential for effectively utilizing less-capable embedded GPUs. GPU sharing has also been explored by others in the context of throughput-oriented systems [55].

There has been much prior work on scheduling real-time DAG-based multiprocessor task systems; representative papers include [4, 5, 6, 13, 30, 31, 36, 37, 38, 44]. However, this work is largely directed at verifying hard-real-time schedulability instead of merely deriving response-time bounds.

7 Conclusions

In this paper, we proposed a fine-grained approach for decomposing and scheduling acyclic OpenVX graphs. We also explained how to leverage prior work to compute end-to-end response-time bounds for these graphs. For GPU-using workloads, end-to-end bounds require response-time bounds for GPU tasks. We presented the first ever such bounds for NVIDIA GPUs, and showed that these bounds are tight under certain assumptions. To illustrate the efficacy of our proposed fine-grained approach, we presented an experimental case study. We saw in this study that our fine-grained approach enabled response-time bounds to be guaranteed and observed response times to be reduced. A notable aspect of our fine-grained approach is its crucial reliance on allowing intra-task parallelism, a feature forbidden in most conventional real-time task models.

This paper opens up many avenues for future work. First, methods for dealing with cycles in OpenVX graphs explored previously [23, 51] need to be incorporated into our fine-grained approach. Second, although shared-memory-induced GPU blocking is exceedingly rare in our experience, our GPU response-time analysis needs to be extended to fully deal with its effects. Third, tools that automate the resource-allocation options considered in our case study would be useful. Fourth, it would be desirable to augment our case study with a schedulability study that examines general trends. Finally, while we have made a case herein for introducing real-time concepts and fine-grained scheduling into OpenVX, an actual OpenVX implementation that incorporates these elements has yet to be produced.

References

[1] T. Amert, N. Otterness, M. Yang, J. Anderson, and F. D. Smith, “GPU scheduling on the NVIDIA TX2: Hidden details revealed,” in *RTSS ’17*.

[2] J. Anderson and A. Srinivasan, “Early-release fair scheduling,” in *ECRTS ’00*.

[3] J. Aumiller, S. Brandt, S. Kato, and N. Rath, “Supporting low-latency CPS using GPUs and direct I/O schemes,” in *RTCSA ’12*.

[4] S. Baruah, “Federated scheduling of sporadic DAG task systems,” in *IPDPS ’15*.

[5] ——, “Improved multiprocessor global schedulability analysis of sporadic DAG task systems,” in *ECRTS ’14*.

[6] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, “A generalized parallel task model for recurrent real-time processes,” in *RTSS ’12*.

[7] C. Basaran and K. Kang, “Supporting preemptive task executions and memory copies in GPGPUs,” in *ECRTS ’12*.

[8] K. Berezovskyi, K. Bletsas, and B. Andersson, “Makespan computation for GPU threads running on a single streaming multiprocessor,” in *ECRTS ’12*.

[9] K. Berezovskyi, K. Bletsas, and S. Petters, “Faster makespan estimation for GPU threads on a single streaming multiprocessor,” in *ETFA ’13*.

[10] K. Berezovskyi, F. Guet, L. Santinelli, K. Bletsas, and E. Tovar, “Measurement-based probabilistic timing analysis for graphics processor units,” in *ARCS ’16*.

[11] K. Berezovskyi, L. Santinelli, K. Bletsas, and E. Tovar, “WCET measurement-based and extreme value theory characterisation of CUDA kernels,” in *RTNS ’14*.

[12] A. Betts and A. Donaldson, “Estimating the WCET of GPU-accelerated applications using hybrid analysis,” in *ECRTS ’13*.

[13] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese, “Feasibility analysis in the sporadic DAG task model,” in *ECRTS ’13*.

[14] B. Brandenburg and J. Anderson, “Feather-trace: A lightweight event tracing toolkit,” in *OSPERT ’07*.

[15] ——, “Optimality results for multiprocessor real-time locking,” in *RTSS ’10*.

[16] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna, “SiGAMMA: Server based integrated GPU arbitration mechanism for memory accesses,” in *RTNS ’17*.

[17] R. Cavicchioli, N. Capodieci, and M. Bertogna, “Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms,” in *RTNS ’17*.

[18] K. Chitnis, J. Villareal, R. Giduthuri, T. Schwartz, F. Brill, and T. Lepley, “OpenVX graph pipelining extension,” Online at https://www.khronos.org/registry/OpenVX/extensions/vx_khr_pipelining/html/index.html, 2018.

[19] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” in *CVPR ’05*.

[20] U. Devi and J. Anderson, “Tardiness bounds under global EDF scheduling on a multiprocessor,” *Real-Time Systems*, vol. 38, no. 2, pp. 133–189, 2008.

[21] G. Elliott, N. Kim, J. Erickson, C. Liu, and J. Anderson, “Minimizing response times of automotive dataflows on multicore,” in *RTCSA ’14*.

[22] G. Elliott, B. Ward, and J. Anderson, “GPUSync: A framework for real-time GPU management,” in *RTSS ’13*.

[23] G. Elliott, K. Yang, and J. Anderson, “Supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms,” in *RTSS ’15*.

[24] J. Erickson and J. Anderson, “Response time bounds for G-EDF without intra-task precedence constraints,” in *OPODIS ’11*.

[25] J. Erickson, B. Ward, and J. Anderson, “Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling,” *Real-Time Systems*, vol. 50, no. 1, pp. 5–47, 2014.

[26] B. Forsberg, A. Marongiu, and L. Benini, “GPUGuard: Towards supporting a predictable execution model for heterogeneous SoC,” in *DATE ’17*.

[27] M. Green, ““How long does it take to stop?” Methodological analysis of driver perception-brake times,” *Transportation Human Factors*, vol. 2, no. 3, 2000.

[28] A. Horga, S. Chattopadhyay, 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.

[29] P. Houdek, M. Sojka, and Z. Hanzálek, “Towards predictable execution model on ARM-based heterogeneous platforms,” in *ISIE ’17*.

[30] X. Jiang, N. Guan, X. Long, and W. Yi, “Semi-federated scheduling of parallel real-time tasks on multiprocessors,” in *RTSS ’17*.

[31] X. Jiang, X. Long, N. Guan, and H. Wan, “On the decomposition-based global edf scheduling of parallel real-time tasks,” in *RTSS ’16*.

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

[33] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, “TimeGraph: GPU scheduling for real-time multi-tasking environments,” in *USENIX ATC ’11*.

[34] H. Lee and M. 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.

[35] A. Li, G. van den Braak, A. Kumar, and H. Corporaal, “Adaptive and transparent cache bypassing for GPUs,” in *SIGHPC ’15*.

[36] J. Li, K. Agrawal, C. Lu, and C. Gill, “Analysis of global EDF for parallel tasks,” in *ECRTS ’13*.

[37] J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill, and C. Lu, “Mixed-criticality federated scheduling for parallel real-time tasks,” *Real-Time Systems*, vol. 53, no. 5, pp. 760–811, 2017.

[38] J. Li, A. Saifullah, K. Agrawal, C. Gill, and C. Lu, “Analysis of federated and global scheduling for parallel real-time tasks,” in *ECRTS ’14*.

[39] LITMUS$^{\mathrm{RT}}$ Project, http://www.litmus-rt.org/.

[40] 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.

[41] N. Otterness, M. Yang, S. Rust, E. Park, J. Anderson, F. Smith, A. Berg, and S. Wang, “An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads,” in *RTAS ’17*.

[42] The Khronos Group, “OpenVX: Portable, Power Efficient Vision Processing,” Online at https://www.khronos.org/openvx/.

[43] ——, “The OpenVX Graph Pipelining, Streaming, and Batch Processing Extension to OpenVX 1.1 and 1.2,” Online at https://www.khronos.org/registry/OpenVX/extensions/vx_khr_pipelining/OpenVX_Graph_Pipelining_Streaming_and_Batch_Processing_Extension_1_0.pdf.

[44] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, “Multicore real-time scheduling for generalized parallel task models,” in *RTSS ’11*.

[45] M. Santriaji and H. Hoffmann, “MERLOT: Architectural support for energy-efficient real-time processing in GPUs,” in *RTAS ’18*.

[46] U. Verner, A. Mendelson, and A. Schuster, “Batch method for efficient resource sharing in real-time multi-GPU systems,” in *ICDCN ’14*.

[47] ——, “Scheduling periodic real-time communication in multi-GPU systems,” in *ICCCN ’14*.

[48] ——, “Scheduling processing of real-time data streams on heterogeneous multi-GPU systems,” in *SYSTOR ’12*.

[49] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, “Demystifying GPU microarchitecture through microbenchmarking,” in *ISPASS ’10*.

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

[51] K. Yang, G. Elliott, and J. Anderson, “Analysis for supporting real-time computer vision workloads using OpenVX on multicore+GPU platforms,” in *RTNS ’15*.

[52] K. Yang, M. Yang, and J. Anderson, “Reducing response-time bounds for DAG-based task systems on heterogeneous multicore platforms,” in *RTNS ’16*.

[53] K. Yang and J. Anderson, “Optimal GEDF-based schedulers that allow intra-task parallelism on heterogeneous multiprocessors,” in *ESTIMedia ’14*.

[54] 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 *ECRTS ’18*.

[55] 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, pp. 1522–1532, 2014.

[56] H. Zhou, S. Bateni, and C. Liu, “S^{3}DNN: Supervised streaming and scheduling for GPU-accelerated real-time DNN workloads,” in *RTAS ’18*.

[57] H. Zhou, G. Tong, and C. Liu, “GPES: A preemptive execution system for GPGPU computing,” in *RTAS ’15*.