Hardware Compute Partitioning on NVIDIA GPUs*

Joshua Bakita and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill
Email: {jbakita, anderson}@cs.unc.edu

Abstract—**Embedded and autonomous systems are increasingly integrating AI/ML features, often enabled by a hardware accelerator such as a GPU. As these workloads become increasingly demanding, but size, weight, power, and cost constraints remain unyielding, ways to increase GPU capacity are an urgent need. In this work, we provide a means by which to *spatially partition* the computing units of NVIDIA GPUs transparently, allowing oft-idled capacity to be reclaimed via safe and efficient GPU sharing. Our approach works on any NVIDIA GPU since 2013, and can be applied via our easy-to-use, user-space library titled `libsmctrl`. We back the design of our system with deep investigations into the hardware scheduling pipeline of NVIDIA GPUs. We provide guidelines for the use of our system, and demonstrate it via an object detection case study using YOLOv2.**

I. Introduction

To counteract frequency-scaling limitations, processor manufacturers have increasingly turned to multiprocessing to meet ever-increasing computational demands. While well-studied for CPUs, this surge in parallelism has also affected computational accelerators such as GPUs. As such accelerators see increasing adaptation in embedded and real-time systems, effective utilization of this parallelism is crucial to meet size, weight, power, and cost (SWaP-C) constraints.

Unfortunately, current GPU-management and scheduling approaches often treat the GPU as a single, monolithic device and enforce mutually-exclusive access control. This can result in severe capacity loss, and is akin to scheduling only one task at a time on a multicore CPU. Fig. 1(a) shows such losses on a five-compute-unit GPU.

Innovation in GPU scheduling has been held back by two issues. First, increasing GPU parallelism has not been partnered with a manufacturer-provided means to *spatially partition* compute units—the ability to run multiple tasks at once, with each task assigned to a mutually-exclusive subset of processing cores. Second, GPU hardware architecture and scheduling mechanisms are largely undocumented, preventing third parties from building efficient GPU-partitioning systems.

In this work, we present a novel mechanism for high-granularity spatial partitioning of GPU compute units on all NVIDIA GPUs from 2013 to the present day. This enables more efficient scheduling, as in Fig. 1(b). (We further explain this figure in Sec. II.) In this work, we further justify compute partitioning use by uncovering hardware parallelism and subdivisions of the memory and scheduling units in NVIDIA GPUs. We focus on NVIDIA GPUs due to their industry-leading architectures and adoption rates.

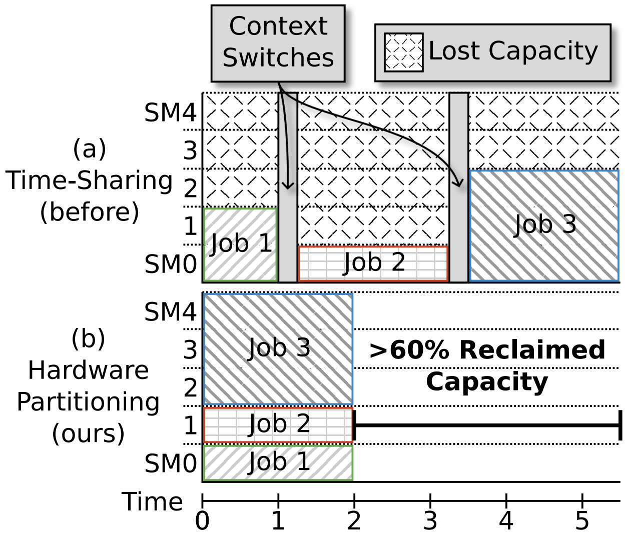

Fig. 1. With time-sharing, only one task can use the GPU at a time. This results in capacity loss when individual tasks cannot saturate the GPU compute cores (SMs) as in (a). Our work enables spatial partitioning, allowing the GPU to be subdivided among multiple tasks, reclaiming idle capacity as in (b).

Prior work. Existing efforts to partition NVIDIA GPUs have been hamstrung by elusive documentation, and thus suffer unacceptable fragility, overheads, or programming-model constraints. One approach, used to implement a concept called "fractional GPUs" [1], [2], requires each GPU computation to be modified such that it cooperatively yields unallocated computing resources. Another approach, used in some recent GPU interference-analysis work [3], [4], injects custom "blocking kernels" to force subsequent work to execute on a subset of computing resources. Both of these approaches inherit many of the problems of cooperative multitasking, including a lack of isolation from misbehaving tasks. These approaches also curtail the permissible types of kernel launches, cause unavoidable instruction-cache interference, and, among other pitfalls, require extensive expert program modification.

Contributions. In this work, we:

1) Reveal a hardware mechanism for spatial partitioning of compute units in all NVIDIA GPUs since 2013.
2) Build and demonstrate a simple, effective, and portable GPU spatial partitioning API.
3) Provide heretofore unpublished details on the hardware scheduling pipeline of NVIDIA GPUs.
4) Detail heretofore unpublished architectural patterns used in NVIDIA GPUs, including the layout of and interconnects between compute and memory units.
5) Evaluate the limits of, and develop guidelines for, the

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

effective use of spatial partitioning on NVIDIA GPUs.

6) Demonstrate via a case study how spatial partitioning can be applied to increase GPU utilization and reduce latency for a convolutional neural network (CNN).

Organization. We define key GPU terms in Sec. II before more thoroughly discussing relevant prior work in Sec. III. We detail our partitioning method and API in Sec. IV, and support the usefulness of our approach via an elucidation of GPU hardware in Sec. V. We evaluate our method and provide guidelines for its use in Sec. VI, demonstrate effectiveness via a case study in Sec. VII, and conclude in Sec. VIII.

II. Background

To provide initial terms and context, we overview recent NVIDIA GPU architecture and the CUDA programming framework often used by GPGPU (General-Purpose GPU) tasks.

GPU architecture overview. GPUs are highly parallel accelerators, typically built of several discrete functional units that each have the internal capability for parallelism. Fig. 2 shows the internal functional units of one such recent NVIDIA GPU. The eight GPCs (General Processing Clusters),[1] each consisting of sixteen SMs (Streaming Multiprocessors), collectively compose the GPU's compute/graphics engine.[2] SMs are arranged in groups of two per TPC (Thread Processing Cluster).[3] Each SM contains 64 CUDA cores. Other GPU engines include five asynchronous copy engines, three video encoding engines, one video decoding engine, and one JPEG decoding engine.[4] Prior work has shown that these engines can operate with some degree of independence from the compute/graphics engine [6].

CUDA overview. In order to simplify programming these complex accelerators, NVIDIA developed the CUDA programming language and API. An example CUDA program, which adds two vectors `A` and `B` in parallel on the GPU, is shown in Alg. 1. Procedures executed on the GPU are called *CUDA kernels*. Comments in Alg. 1 describe each step. Note that even this simple example utilizes both the compute/graphics engine (Line 13) and a copy engine (Lines 11 and 14).

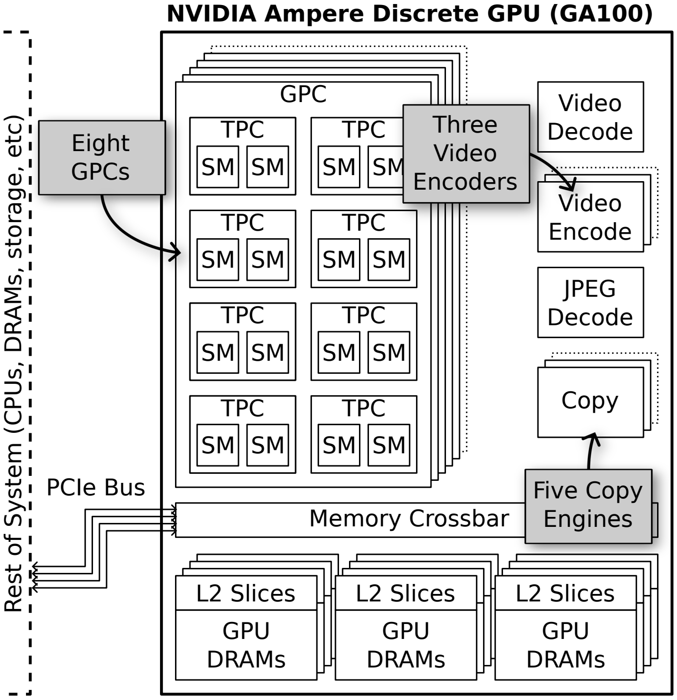

Fig. 2. Architecture of a recent NVIDIA discrete GPU.

When a kernel is launched, the number of threads is specified as a number of thread blocks,[5] and threads per block. In our example, for a 2,000-entry vector, one could set `numBlocks = 2` and `threadsPerBlock = 1000`, as a block can contain no more than 1024 threads.

All CUDA applications run in their own memory address space, called a *context*, and time-division multiplexing is used to arbitrate among active CUDA and otherwise GPU-using applications (such as display tasks) by default [7].[6] Inside

Algorithm 1 Vector Addition in CUDA.

```
 1: kernel VECADD(A: ptr to int, B: ptr to int, C: ptr to int, $len$: int)
 2:     $i$ := blockDim.x * blockIdx.x + threadIdx.x        ▷ Calculate index based on built-in thread and block information
 3:     if $i >= len$ then
 4:         return                                          ▷ Exit thread if out of vector bounds
 5:     end if
 6:     C[$i$] := A[$i$] + B[$i$]
 7: end kernel
 8: procedure MAIN
 9:     cudaMalloc(d_A, $len$)                              ▷ (i) Allocate GPU memory for arrays A, B, and C
10:     ...
11:     cudaMemcpy(d_A, h_A, $len$)                         ▷ (ii) Copy data from CPU to GPU memory for arrays A and B
12:     ...
13:     vecAdd<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, $len$)   ▷ (iii) Launch the CUDA kernel on GPU
14:     cudaMemcpy(h_C, d_C, $len$)                         ▷ (iv) Copy results from GPU to CPU array C
15:     cudaFree(d_A)                                       ▷ (v) Free GPU memory for arrays A, B, and C
16: end procedure
```

[1] Formerly known as a Graphics Processing Cluster.

[2] Due to manufacturing yields, at most seven of the eight GPCs are enabled in shipping products, with between seven and eight TPCs per GPC [5]

[3] Formerly known as a Texture Processing Cluster.

[4] Also known as `PCE`, `NVENC`, `NVDEC`, and `NVJPG` respectively.

[5] Also known as Cooperative Thread Arrays (CTAs).

[6] NVIDIA MPS or MiG can bypass this, but are only available in discrete and server GPUs respectively [8], [9]. Despite repeated calls from academia, NVIDIA has brought neither technology to its embedded chips.

a single CUDA application, multiple FIFO queues called *streams* can be used to allow work to use the GPU fully concurrently [10]. Combining all CUDA-using functionality into a single context with multiple streams is often done to avoid cases where several small applications, none of which can fully use the GPU, are exclusively timesliced [11]. A toy illustration of this is shown in Fig. 1, where combining multiple applications into a single context reduces GPU busy time by $> 60\%$—if the GPU's SMs can be hardware partitioned. Without hardware partitioning, Job 1 would attempt to use two SMs when run alongside the others, preventing Job 2 from starting. With partitioning, we can force Job 1 to use a single SM for twice as long, yielding a more efficient overall schedule.

Real-time terminology. Tasks are composed of *jobs*, with a rate of release known as a *period*. After release, each job and must complete by a subsequent *deadline*. If tasks release jobs exactly at the start of their period they are known as *periodic*. If, instead, the period only defines a minimum separation between releases, the task is known as *sporadic*. Tasks may have an associated *criticality*, which reflects their relative importance in the system. For example, in a self-driving car, a pedestrian-detection task would have high criticality, whereas in-screen display updates would have low criticality.

III. Related Work

Hardware *predictability* is key in embedded and real-time systems. Since the advent of applying GPUs towards general-purpose compute tasks, researchers have sought to develop management approaches that enforce predictable behavior when a GPU is shared among applications.

One of the most notable early works in this area is TimeGraph [12], which uses an interception layer and GPU-interrupt-driven scheduler to serially arbitrate GPU work submission. This scheduling approach theoretically still applies to NVIDIA GPUs, but requires fully open-source drivers and does not allow for parallel executions on the GPU.

Another, simpler, but more broadly applicable approach to predictably share a GPU is to treat it as a *resource* to which accesses are protected by *mutual-exclusion locking*. The GPUSync framework [13] and extensions [6] can be considered exemplars of this approach. These works further innovate by allowing for predictable *intra-GPU sharing*. Through the observation that some auxiliary GPU units such as the copy engines can operate independently of compute work, these works allow for per-GPU-engine-granularity locking. This enables, for example, one application to perform GPU computations while another performs copies.

Subsequent to these works, significant effort has gone into allowing more predictable intra-GPU sharing, particularly of the GPU's primary compute/graphics engine. The recent work of Otterness and Anderson [14], [15] provided a breakthrough in this area, by allowing multiple applications to simultaneously share commodity AMD GPUs through transparent, hardware spatial partitioning of the main compute cores. This technique has yet to be extended to NVIDIA GPUs, with recent work remaining constrained to software-emulated partitioning [1]–[4], or conceptual analysis [16].

Significant supporting work to reverse engineer GPU design has accompanied GPU management efforts. Historically, commodity GPU designs have been shrouded in secrecy. Even instruction encodings—a commonly shared piece of information for any processor—have remained secret for GPUs [17]–[19]. Notable reverse engineering works include those of Otterness *et al.*, Amert *et al.*, and Olmedo *et al.* to expose the queue structure used for compute work in NVIDIA GPUs [10], [20], [21], the work of Capodieci *et al.* and Spliet and Mullins to clarify the preemption capabilities of NVIDIA GPUs [7], [22], and the work of Jain *et al.* to elucidate the memory hierarchy in NVIDIA GPUs [1]. Outside of the academic community, the Nouveau [23] and Mesa [24] reverse-engineered NVIDIA GPU driver projects provide crucial details on GPU-to-CPU interfaces.

Despite these efforts, we are aware of no published method that *implements* transparent spatial partitioning of the compute/graphics engine for NVIDIA GPUs.

IV. Implementing Compute Partitioning

We now discuss how we enable transparent hardware partitioning of compute units in NVIDIA GPUs, and overview our easy-to-use, portable, user-space API for doing so.

A. Partitioning on TPC Boundaries

We implemented partitioning by activating two existent, but little-known and unused fields in an obscure NVIDIA data structure known as a TMD (Task MetaData) used to launch GPU compute work [25] (we return in detail to this structure in Sec. V).[7] The fields we uncovered are named `SM_DISABLE_MASK_LOWER` and `_UPPER`. These fields are publicly listed, from TMD structure V1.6 (Kepler) to V3.0 (Ampere) [26], but not used in any open-source software. As one of, if not the only, public reference to an SM mask, we decided to explore the purpose and functionality of these fields. We refer to these fields as the *SM mask* going forward.

Activating partitioning. Unfortunately, modifying a TMD created by the CUDA library is somewhat difficult—the TMD is almost immediately uploaded onto the GPU after construction. In order to modify the TMD, we discovered and utilized an undocumented CUDA debugging callback API to intercept and modify the TMD after construction, but before upload onto the GPU. Using this mechanism, we experimented with the SM mask and found, on older GPUs, it is a bitmask where an enabled bit indicates a disabled SM for the kernel described by the TMD. This can allow for spatial partitioning of SMs. Given a mutually exclusive partition configuration, partitioning can be enforced by setting the SM mask for every kernel such that every kernel in one partition is prohibited from running on any SM of another partition. We show such partitioning in Fig. 3.

[7] The TMD is called a QMD in some sources. QMD stands for Queue MetaData. This has been subsumed by the TMD, and old behavior can be accessed by flipping the `IS_QUEUE` TMD bit [25], [26]

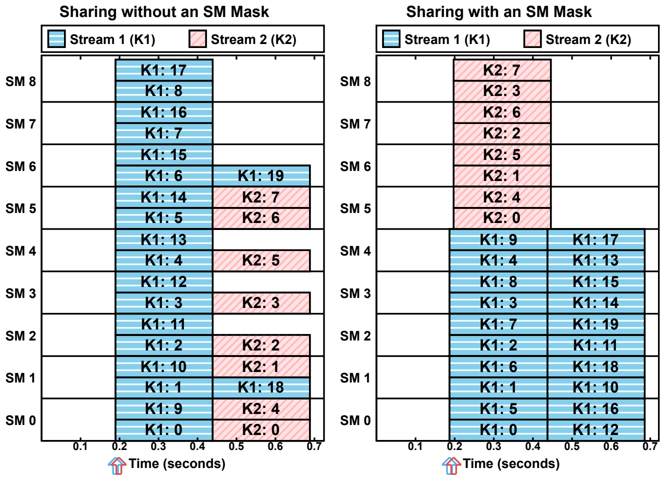

Fig. 3. Partitioning a GTX 1060 3 GB into four- and five-SM partitions.

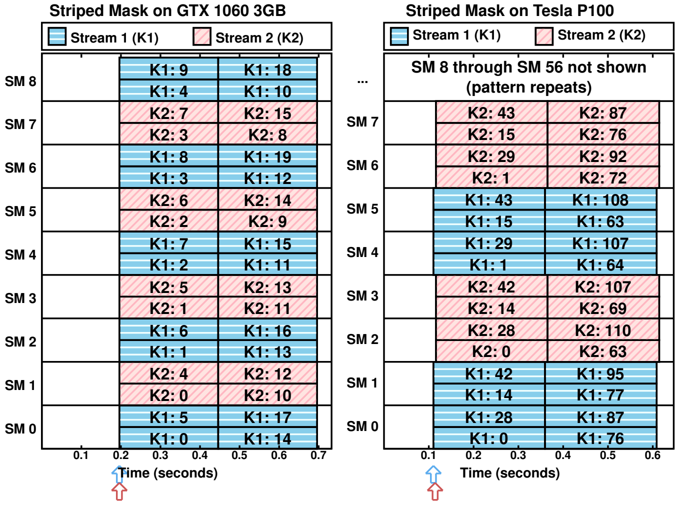

Fig. 4. An "SM mask" acting as a TPC mask on the Telsa P100 GPU.

The experiment in Fig. 3 involves two kernels, K1 and K2, which spin for a fixed amount of time. K2 is launched immediately after K1, and needs to complete about half as much work. Launch times are indicated by the colored arrows at the bottom. Each shaded square in this plot represents the execution of a block of a kernel, with height representing the number of resident threads, and width representing the length of execution. The jth block of kernel i is denoted Ki:j. We show CUDA's default behavior at left, and the behavior with an SM mask at right.

In the left plot, observe how K1's work is immediately distributed among all the SMs of the GPU upon launch, and that the blocks of K2 are prevented from starting execution until after K1's initial set of blocks completes. This pattern remains no matter the priority of K2, mirroring the findings of [10]. At right, we enable an SM mask of `0x1e0` on K1, and `0x01f` on K2. With these masks, SMs 5-8 are disabled for K1, and SMs 0-5 are disabled for K2. Observe how this partitioning also allows for K2 to begin and finish earlier, as no blocks of K1 prevent K2 from starting on SMs 5-8.

TPCs, not SMs. The promising results of Fig. 3 led us to experiment on other, more recent, NVIDIA GPUs where we discovered an interesting discrepancy. We show this in Fig. 4. We again launch two kernels, but flip every-other bit in the SM mask, and run the experiment on two GPUs: first on the GTX 1060 3GB at left, then on the Tesla P100 at right. Based on the behavior in Fig. 3, we would expect our odd/even SM mask to disable every odd SM for K1, and every even SM for K2. This expectation holds only on the left. With the P100, the pattern repeats at double the period. Why?

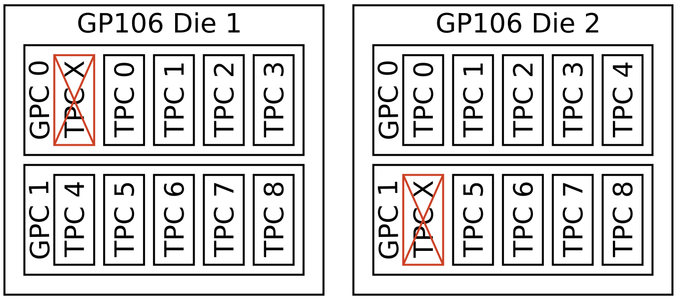

Fig. 5. Illustration of floorsweeping asymmetrically impacting GPU layout and TPC numbering for two GP106 dies.

The GTX 1060 and Tesla P100 are both based on NVIDIA's Pascal architecture, but use dies configured in a slightly different way. The P100 and newer GPUs are configured more similarly to the GA100 in Fig. 2, with two SMs per TPC, whereas the GTX 1060 and older GPUs have a one-to-one SM to TPC ratio. This key difference, and substantial supporting results, points to the "SM mask" actually serving as a TPC mask to the GPU. Presumably, the name of the field in the TMD is inherited from the days when the number of TPCs and SMs were identical, allowing for more term interchangeability.

B. Partitioning on GPC Boundaries

Many GPU resources are instantiated on a per-GPC basis (more on this in Sec. V-C), so it may be desirable to obviate contention by aligning GPU partitions to GPC boundaries. Earlier work on AMD GPUs [14] assumed a constant mapping of compute units to shader engines (a GPC's AMD equivalent). NVIDIA GPUs are different. We illustrate this for two instances of the GTX 1060 3 GB's GP106 die in Fig. 5. Disabled TPCs are crossed out, and the remaining TPCs are renumbered. Note how both dies have nine enabled TPCs, but not the *same* TPCs. The first TPC of GPC 1 is disabled at right, but not at left. Instead the first TPC of *GPC 0* is disabled. This causes TPC 4 to be in GPC 0 at right, but in GPC 1 at left. In our experiments, we find this pattern repeated across most NVIDIA GPUs from the past several years. Why would two ostensibly identical GPUs have such different internal configurations?

This difference between dies stems from *floorsweeping*, a technique increasingly employed by NVIDIA in recent years. As GPU dies have grown in size to meet ever-increasing computational demands, the likelihood of die manufacturing errors has increased. Floorsweeping enables use of these imperfect dies by fusing off ("sweeping" away) the defective parts, allowing the remaining parts to function normally. As

TABLE I
API OF OUR LIBSMCTRL LIBRARY.

Library Function	Description	Supported On
`libsmctrl_set_global_mask(uint64_t mask)`	Set TPCs disabled by default across the entire application. A set bit in the mask indicates a TPC is to be disabled globally.	CUDA 10.2 through 12.1
`libsmctrl_set_stream_mask(cudaStream_t stream, uint64_t mask)`	Set TPCs disabled for all kernels launched via `stream` (overrides global mask).	CUDA 8.0 through 12.1
`libsmctrl_set_next_mask(uint64_t mask)`	Set TPCs to be disabled for the next kernel launch from the caller's CPU thread (overrides global and per-stream masks, applies only to next launch).r	CUDA 11.0 through 12.1
`libsmctrl_get_gpc_info(uint32_t* num_gpcs, uint64_t** tpcs_for_gpc, int dev)`	Get number of GPCs for device number `dev`, and a GPC-indexed array containing masks of which TPCs are associated with each GPC.	Compute Capability 5.0 and up (`nvdebug` module required)
`libsmctrl_get_tpc_info(uint32_t* num_tpcs, int dev)`	Get total number of TPCs on device number `dev`.	Compute Capability 3.5 and up

die manufacturing errors are randomly distributed, the disabled units vary from die to die, as in Fig. 5.

Coupled with floorsweeping, dies are *binned* based on their number of defective parts. For the GP106 die shown in Fig. 5, if no TPCs are defective it is sold as a GTX 1060 6 GB, if at most one TPC is defective it is sold as a GTX 1060 3 GB, if at most two TPCs are defective it is sold as a Quadro P2000, and so on. Dies beyond a defect threshold are discarded.

When partitioning TPCs, each bit in the mask corresponds to a TPC *after* they have been renumbered, as in Fig. 5. This means that TPC to GPC mappings are necessary to partition on GPC boundaries across dies. We determined which GPU registers contain this information, and built a Linux kernel module named `nvdebug` to extract and expose this via an interface in `/proc`.[8]

C. Our API

Combining together the capability to partition TPCs and GPCs, we present our flexible user-space library and API, titled `libsmctrl`, written in C. We detail our API in Table I, with function names and parameters in the first column, function description in the next, and finally the prerequisites for use of the function. We support setting the TPC mask at global, per-stream, and per-kernel levels, where setting a mask at higher-granularity will override a lower-granularity mask. This allows for most TPCs to be idled by default, with access granted only to explicitly permitted streams or kernels. Our library supports aarch64 and x86_64 CPUs, and CUDA versions stretching as far back as 2016. Our library is fully user-space, and compiled binaries are portable to any recent Linux kernel and NVIDIA driver. When active, our library adds only a few instructions of overhead onto each kernel call. Our implementation primarily relies on an undocumented CUDA callback API, but we employ several other techniques on older CUDA versions.

Basic partitioning. An example of our API in a CUDA program is shown in Listing 1. We assume a nine-TPC GPU, but a real program should compute the mask dynamically after using `libsmctrl_get_tpc_info()` to get the number of TPCs. For most programs, we recommend disabling most TPCs by default—CUDA may implicitly launch internal kernels to support some API calls and, if no default mask is set, those calls may interfere with your partitions. In Listing 1 we set the global default on Line 3, permitting work to run only on TPC 0 unless otherwise specified. It is possible to disable all TPCs by default (with a mask of `~0ull`),[9] but we recommend against this, as it causes kernels launched with the default TPC mask to hang indefinitely (including CUDA-internal ones).

```
 1 int main() {
 2   // Allow work to only use TPC 1 by default
 3   libsmctrl_set_global_mask(~0x1ull);
 4   ...
 5   // Stream-ordering is still respected with
 6   // partitioning, so avoid the NULL stream
 7   cudaStream_t urgentStream, otherStream;
 8   cudaStreamCreate(&urgentStream);
 9   cudaStreamCreate(&otherStream);
10   // Override the global default settings
11   // Allow otherStream to use the first 5 TPCs
12   libsmctrl_set_stream_mask(otherStream,
13                             ~0x1full);
14   long_kernel<<<2048, 2048, 0, otherStream>>>();
15   // Allow urgentStream to use the last 4 TPCs
16   libsmctrl_set_stream_mask(urgentStream,
17                             ~0x1e0ull);
18   // Launch short, sporadic work as it arrives
19   bool done = 0;
20   while (!done)
21     wait_for_work_or_done(&done);
22     sporadic_work<<<32, 32, 0, urgentStream>>>();
23     cudaStreamSynchronize(urgentStream);
24   }
25 }
```

Listing 1. Example usage of partitioning API (9 SM GPU).

Continuing on in the example of Listing 1, we create CUDA streams on Lines 7–9 to allow for concurrent kernel launches (prior work [10] discusses why this is necessary), and set the TPCs these streams can use on Lines 12 and 16. We allow `otherStream` to use TPCs 0–4, and `urgentStream` to use TPCs 5–8. This allows us to launch long-running and difficult-to-preempt kernels in `otherStream`, while still being able to immediately start sporadic work via `urgentStream` as it arrives. Our API is highly flexible, and supports many other usage patterns not detailed here.

GPC partitioning. To support partitioning on GPC bound-

[8] See documentation linked at https://www.cs.unc.edu/~jbakita/rtas23-ae/.

[9] Postfix `ull` indicates a 64-bit literal, and ~ is bitwise inversion.

```
1  // Get mask of enabled TPCs for each GPC
2  uint32_t num_gpcs;
3  uint64_t *tpcs_for_gpc;
4  libsmctrl_get_gpc_info(&num_gpcs,
5                         &tpcs_for_gpc);
6  assert(num_gpcs >= 2);
7
8  // Allow the next kernel to run on any TPC in
9  // the first two GPCs
10 uint64_t tpc_mask = 0;
11 tpc_mask |= tpcs_for_gpc[0];
12 tpc_mask |= tpcs_for_gpc[1];
13
14 // The above lines created a bitmask of TPCs to
15 // enable, so invert to get the disable mask
16 libsmctrl_set_next_mask(~tpc_mask);
```

Listing 2. Example usage of GPC information API.

aries, we provide a easy-to-use wrapper function around our nvdebug kernel module, and include a brief example of this in Listing 2. This example allocates the TPCs of two GPCs to the next kernel launch. Line 4 obtains an num_gpcs-length array of bitmasks. Array index i is associated with GPC i, and if bit j is set in the mask in that entry, TPC j is associated with GPC i. A TPC may only be associated with one GPC. On Lines 10–12 we combine the bitmasks of the TPCs for GPCs 0 and 1, then apply the mask on Line 16.

D. Limitations

Our system works by intercepting TMDs after construction on the CPU, but pre-GPU-upload. As such, it will not work for kernels with TMDs constructed and executed from the GPU without the involvement of the CPU, such as when CUDA Dynamic Parallelism (CDP) or CUDA Graphs are used.

Further, we cannot partition GPUs with over 64 TPCs, as all versions of the TMD structure presently contain only 64 bits for partitioning. No shipping GPU has that many TPCs, but NVIDIA's upcoming H100 allegedly contains up to 72.

Finally, our current library does not support per-GPU global partition configuration, but could easily be extended to do so.

V. How NVIDIA GPUs Schedule Compute

The usefulness of GPU partitioning hinges on the ability of GPU hardware to supply jobs and data to each partition in parallel. In this section, we take a deep-dive into the architectural patterns of NVIDIA GPUs, with a particular focus on how they allow for parallelism at every step of a kernel's journey from user-space dispatch to completion.

We first briefly consider our sources of information before delving into the full pipeline, as numbered in Fig. 6.

A. Sourcing Architectural Details

We gleaned architectural information from cross-referenced public sources. NVIDIA's patents cover the implementation of GPU compute priorities [27], block distribution logic [28], [29], preemption design [30], and more [25], [31]–[34]. As patents can describe non-existent or infeasible inventions, we verified if the parents discuss hypothetical or actual designs by cross-referencing them with NVIDIA's open-source nvgpu driver [35], NVIDIA's public headers [26], the work of the Nouveau and Mesa projects [23], [24], and other sources.

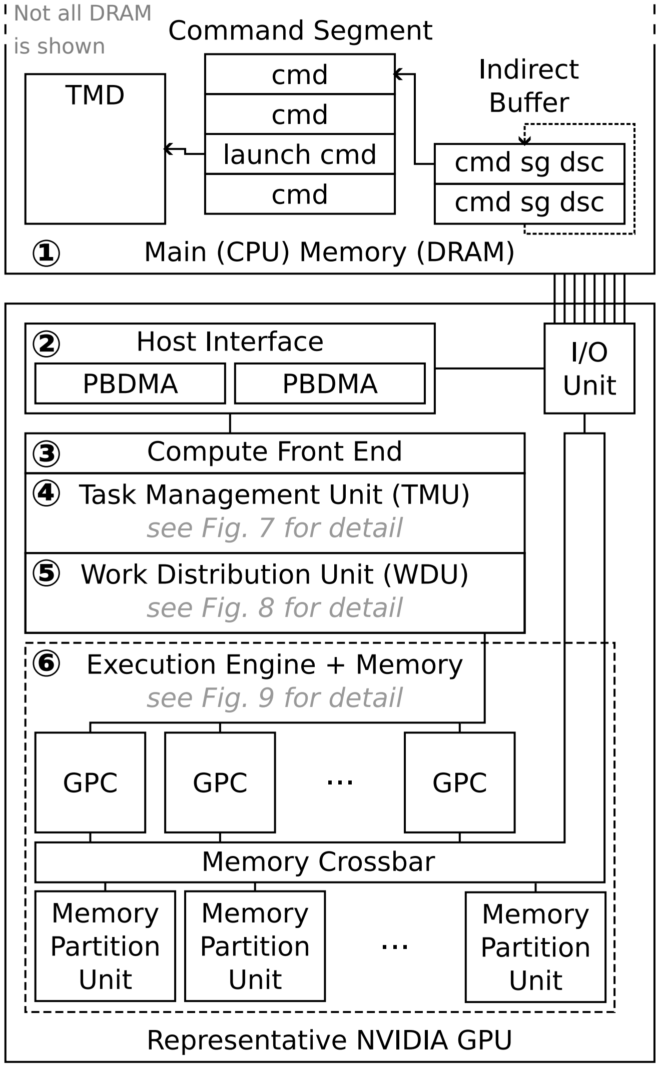

Fig. 6. High-level GPU hardware compute scheduling pipeline.

As an example of our validation process, consider NVIDIA's patent for TMD error checking [25]. Part of this patent describes, in order, the fields of the TMD structure. A publicly available header,[10] used in the Mesa project,[11] also provides the ordering and names of TMD fields. These sources match, indicating that the patent likely describes real hardware. We applied this process, or performed verification experiments, to attain the following content.

As the following information is generally a synthesis of all the above sources, we omit redundant inline citations. To the best of our knowledge, our discussion is accurate for NVIDIA GPUs from Kepler through at least Ampere.

B. The Compute Scheduling Pipeline

We now begin following the path of a single kernel through the GPU scheduling pipeline. Formally, we define a compute job on the GPU as a single kernel launch.

[10] classes/compute/clc0c0qmd.h in open-gpu-doc [26].

[11] gallium/drivers/nouveau/nvc0/clc0c0qmd.h in Mesa [24].

① **Kernel instantiation.** On NVIDIA GPUs, kernels are internally described via the aforementioned TMD structure. Beyond the already-discussed properties, it describes the thread blocks, threads per block, and shared memory resources needed for the kernel, and includes the address of the kernel's entry point (among many other fields, such as priority). After a user-space library (such as CUDA or OpenCL) constructs a TMD, it enqueues a launch command containing a pointer to the TMD into the next available command slot in a *command segment*. A command segment is a contiguous block of memory containing GPU commands.

The TMD is used as a kernel descriptor and handle throughout the scheduling pipeline, persisting until all computations of the GPU kernel are complete.

② **Host Interface.[12]** The GPU's Host Interface bridges the gap from CPU to GPU for the TMD. This unit contains one or more Pushbuffer Direct Memory Access units (PBDMAs), along with context-switching control logic, among other sub-units. The PBDMAs load commands from user space via an *indirect buffer*, which is a circular buffer of pointers to command segments.[13] These structures, shown at the top of Fig. 6, allow user-space applications to directly dispatch commands to the GPU without system call or driver overheads. The PBDMA units individually load, parse, and cache commands at a rate much faster than that of most GPU engines. While in our illustration, we show GPU command queues as located in CPU memory (as is most common), they can also be located in GPU memory. After command acquisition and parsing, the Host Interface forwards the command to the appropriate engine. In the case of a kernel launch, it passes the TMD to the Compute Front End.

③ **Compute Front End.** The Compute Front End relays TMD pointers from ②, the Host Interface to ④, the Task Management Unit. Further, we understand that this unit can orchestrate context switches in subsequent units.[14] While this unit processes TMDs at a rate decoupled from the others, it may cause work to queue if the incorrect context is active. We focus on a single context in this work, and so do not further consider said issue. Further details on this unit are elusive. [15]

④ **Task Management Unit (TMU).[16]** Illustrated in Fig. 7, the Task Management Unit queues TMDs by priority and arrival order until ⑤, the Work Distribution Unit, is ready to receive them. Due to the explicit scheduling responsibilities of the TMU, we investigate it in depth.

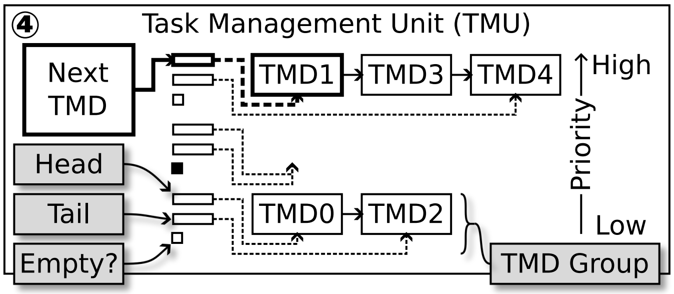

Fig. 7. A three-priority task management unit with five queued TMDs.

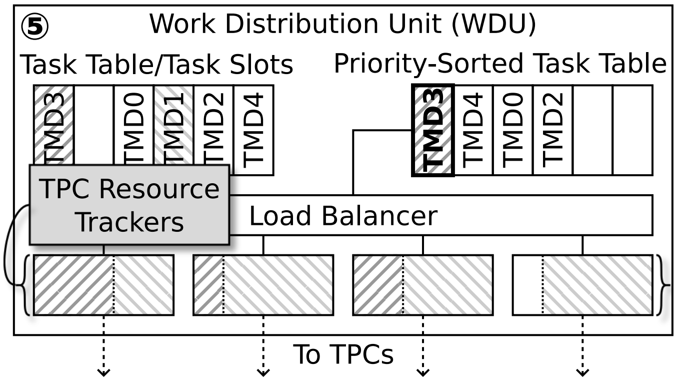

Fig. 8. A six-slot work distribution unit on a four-TPC GPU.

The TMU is built around a series of priority-level, singly-linked lists, with one linked-list-head and -tail pointer tracked for each priority level. Each list is exclusively composed of TMDs. In Fig. 7, we show a unit supporting three priority levels, with TMDs illustratively numbered by arrival order. Three TMDs are in the highest-priority-level list, none are in the medium-priority-level list, and two TMDs are in the lowest-priority-level list. Each priority-level list is formally called a *TMD Group*.

These lists allow the TMU to reorder TMDs that it receives, such that higher-priority ones skip ahead of lower-priority ones. Upon TMD arrival from the Compute Front End, or elsewhere,[17] the TMU reads the TMD's `GROUP_ID` field, and appends the TMD to the tail of the specified TMD Group.[18] For example, in Fig. 7, TMD4 is the most-recently arrived high-priority TMD, and TMD2 is the most-recently arrived low-priority TMD.

As unit ⑤ in the scheduling pipeline signals readiness for another TMD, the TMU removes and passes the head of the highest-priority non-empty list. This is illustrated with the bold outlines and "Next TMD" box in Fig. 7. Given the TMU state of Fig. 7, TMDs would exit the TMU in the following order: TMD1, TMD3, TMD4, TMD0, and then TMD2.

The TMU is the final unit which can receive TMDs at a rate decoupled from the TMD completion rate. This causes TMDs to eventually accumulate in the TMU whenever kernels are dispatched by user space at a rate faster than they can complete.

⑤ **Work Distribution Unit (WDU).[19]** Illustrated in Fig. 8 the Work Distribution Unit dispatches TMDs from *task slots*

[12] Also known as "FIFO" in Nouveau.

[13] A "pushbuffer" is a combination of these segments and pointers.

[14] Via the Front-End Context Switch (FECS) unit. See Spliet *et al.* [22].

[15] The Compute Front End (COMP FE), like the better-documented and similarly-situated Graphics Front End (GFX FE), likely also configures ⑥, the Execution Engine and Memory, but this is difficult to confirm.

[16] More recently called the "Scheduler Unit" or "SKED" in patents [33], [34], and known in marketing text as the Grid Management Unit (GMU) [36].

[17] The TMU may receive new TMDs directly from GPU compute units when features such as CUDA Dynamic Parallelism are employed.

[18] Unless field `ADD_TO_HEAD_OF_QMD_GROUP_LINKED_LIST` is true.

[19] Also known as the CUDA Work Distributor (CWD) in marketing literature [36], and as the Compute Work Distributor in a recent patent [28].

to available TPCs. The number of task slots is hardware limited,[20] forcing the WDU to make scheduling decisions with incomplete information, eg. when the number of ready kernels exceeds the number of task slots.

The WDU signals unit ④, the TMU, for a new TMD whenever a task slot becomes available. It then inserts the TMD into a task slot in the Task Table, and inserts a reference to the TMD in the Priority-Sorted Task Table. The priority-sorted table is ordered first by priority, then by arrival time. The Load Balancer dispatches thread blocks from the TMD at the head of the priority-sorted table. Once all of a TMD's blocks are launched, it is removed from the priority-sorted table, but left in the Task Table until all its blocks complete.

Fig. 8 illustrates the WDU state shortly after receiving the TMDs of Fig. 7, and after dispatching all blocks of TMD1. We reflect which kernels are executing on which TPCs via shading of the TPC Resource Trackers. These units relay states and commands between the TPCs and the WDU. Note how TMD1 is still executing (light hatching) on the TPCs, and so remains in the Task Table. TMD3, as the highest-priority and earliest-arrived TMD, is now dispatching blocks to TPCs. TMD4 is waiting for TMD3 to dispatch all its blocks before it can move to the head of the priority table and dispatch blocks.

This ordering and dispatch process can be disturbed by two things: TPC partitioning, and pending higher-priority work in ④, the TMU. When TPC partitioning is in-use, if a TPC has available capacity, but the TMD at the head of the sorted table is prohibited from executing on that TPC, the WDU *will skip forward* in the table until it finds a TMD allowed to execute on the available TPC (or reaches the end of the table). If, for example, TMD3 was prohibited from executing on TPC4, the Load Balancer could instead dispatch blocks from TMD4.

When all WDU task slots are occupied, and any task slot contains a TMD with priority lower than any pending in ④, the TMU, the lower-priority TMD will be evicted from the WDU and replaced with the higher-priority TMD. When this occurs, the evicted TMD stops dispatching blocks and completes blocks already in progress. The evicted TMD is then inserted into the *head* of the respective priority-level list in the TMU. We will further explore what this means for real workloads in Sec. VI.

This concludes our coverage of the TMD from construction to dispatch. We now consider aspects of GPU architecture that are relevant while a thread block executes.

C. Memory Parallelism

Inherited from the high-bandwidth needs of graphical-processing tasks, GPUs are massively parallel in ways far beyond the high number of CUDA cores. This subsection extends the background of Sec. II with new details on the parallelism of GPU caches, interconnects, and DRAM controllers.

⑥ **Execution engine and memory subsystem.** We illustrate the data and instruction pipelines, from L1 caches to DRAM chips, for the execution engine in Fig. 9. An "I" (resp. "D") postfix on a cache indicates an instruction (resp. data) cache. Starting at the bottom in Fig. 9, consider the Memory Partition Units. These are typically configured as one-per-DRAM chip, with each partition unit containing a DRAM controller and a subset of the L2 cache. Each partition unit independently connects to every GPC via a crossbar bus, such that partitioning Memory Partition Units will also partition the crossbar bus and L2.

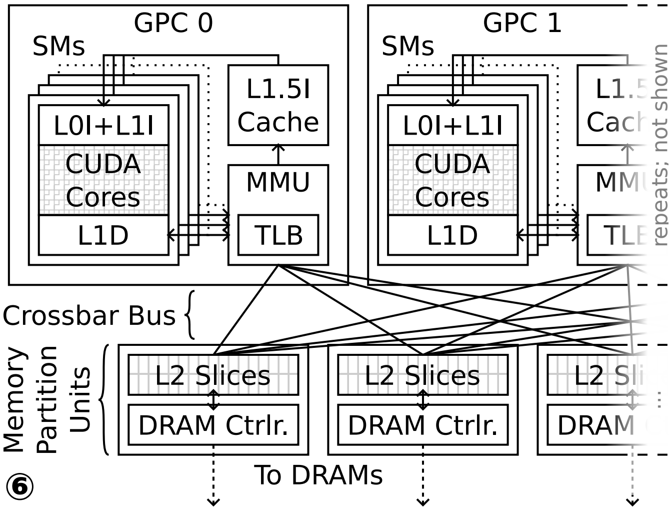

Fig. 9. Topology of GPU execution engine and memory subsystem.

The crossbar bus links inside each GPC with a Memory Management Unit (MMU, with associated Translation Lookaside Buffer, TLB) for virtual memory support.[21] This connects to L1 data caches in each SM, and to a GPC-wide L1.5I cache. The L1.5I cache respectively feeds per-SM instruction caches.

In all, each GPC can theoretically be configured to operate with an exclusive subset of the GPU's cache, bus, and DRAM resources (if Memory Partition Units are partitioned, as in [1]). Further, each SM can operate from its L1 cache without generating interference.

To summarize this section, we find GPU hardware to be more than capable of feeding jobs to many partitions simultaneously, and also find it capable of supplying partitions with uncontended cache, bus, and DRAM resources.

VI. Evaluating Compute Partitioning

We now evaluate spatial compute partitioning to assess its scalability, its intersections with hardware scheduling units, and its usefulness compared to prior work.

A. Methodology

Our experiments utilized a variant of the `cuda_scheduling_examiner` framework integrated with our `libsmctrl` library. Unless otherwise noted, all experiments ran without other processes competing for the GPU. We found minimal impact due to host platform or Linux kernel version, so we do not control for those variables in our experiments.

[20] Documented as the "Maximum number of resident grids per device" in Table 15 of the CUDA C++ Programming Guide [37].

[21] Some GPUs instead put the MMU in the Memory Partition Unit.

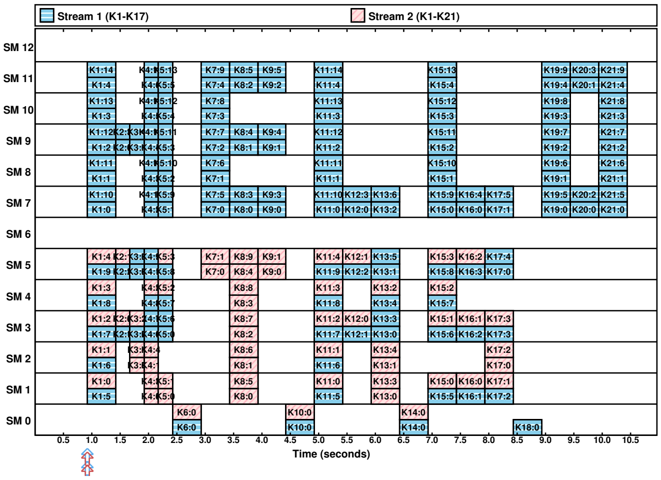

Fig. 10. Complex TPC partitioning on the GTX 970.

B. Compute Partitioning: Flexible and Portable?

At this point, we have only shown TPC partitioning in small experiments involving few kernels. How does it scale to systems of many kernels and compute partitions?

We experimented with complex arrangements of the `GPUSpin` and `multikernel` benchmarks, finding no apparent limits on the number or arrangement of TPC partitions allowed, discovering no ways TPC partitioning could compromise the semantics of CUDA, and encountering few portability limits.

We plot a representative experiment in Fig. 10. This plot shows 38 kernels, across 17 unique TPC partitions and two streams. This experiment demonstrates per-kernel partitioning, stream-ordering, and complex partitioning. We discuss these aspects before considering how they apply to GPUs beyond the GTX 970 used in Fig. 10.

Per-kernel partitions. In the experiment of Fig. 10, every kernel is assigned a different partition than its predecessor. For example, consider Stream 2 (red). In the "T" at the bottom, K7 runs two blocks in a partition including only SM 5, whereas the subsequent K8 in uses a partition including SM 1-5. K7 and K8 (along with all other kernels) are released just before the 1.0s mark, but the partition is encoded into the kernel's TMD—not as a global GPU property—so it automatically takes effect when the kernel begins.

Preservation of stream-ordering. The GPU preserves the ordering of kernels in streams, even when a subsequent kernel may require a mutually exclusive set of SMs. For example, in Stream 2 in Fig. 10, K9 only needs SM 5, and K10 only needs SM 10. However, K10 does not start until K9 completes. CUDA stream semantics require subsequent kernels in the same stream to not launch until all prior ones have been completed—we find that to be preserved, even when TPC partitioning is in use.

Complex and overlapping partitions. TPC partitions may contain holes or overlap, and have no size restrictions. (By holes, we mean non-contiguous sets of TPCs.) For example, the "E" in Stream 1 of Fig. 10 features holes in K8 and K9's partitions, which are the non-adjacent SMs 7, 9, and 11. For an example of overlap, throughout the rest of Stream 1, we allow the partitions of each kernel to fully overlap with those of Stream 2, creating the mix of kernels from different streams in the "R", "A", "S" and periods. We see no variation in the behavior as the partition size is varied, with between one and ten SMs per partition in Fig. 10.

TABLE II
GPUs TESTED IN OUR EXPERIMENTS

GPU Name	Compute Capability	Architecture
Tesla K40	3.5	Kepler
GTX 970	5.2	Maxwell
Jetson TX1	5.3	Maxwell (embedded)
Tesla P100	6.0	Pascal
GTX 1060 3 GB	6.1	Pascal
GTX 1070	6.1	Pascal
GTX 1080	6.1	Pascal
Jetson TX2	6.2	Pascal (embedded)
Titan V	7.0	Volta
Jetson Xavier	7.2	Volta (embedded)
Tesla T4	7.5	Turing
A100 40 GB	8.0	Ampere

Portability. Fig. 10 shows an experimental result from a single GPU, however, we find this behavior to hold *on every recent NVIDIA GPU*. Specifically, any GPU of Compute Capability 3.5 (2013) or greater, and CUDA 8.0 (2017) or newer supports TPC partitioning via our `libsmctrl` library. This includes embedded GPUs, such as that in the ARM64-based NVIDIA Xavier System-on-Chip. We list all specific GPUs we tested in Table II.

C. When Hardware Scheduling Breaks Down

In Sec. V, we discussed how much of the GPU, including most parts of the scheduling pipeline, is highly parallel and unlikely to hinder multiple compute partitions from running simultaneously. In this subsection, we examine the narrow cases when it *does* prevent partitions from running in parallel.

The consequences of greedy assignment. The WDU, the last step in the scheduling pipeline, dispatches blocks from kernels in global priority and then time-of-arrival order, with limited situations in which it will skip ahead. This *greedy assignment* approach can result in particularly non-optimal block assignments when overlapping partitions are in use.

We demonstrate one such case in Fig. 11. Both subfigures have identical configurations; the GPU is split into two partitions: SM 0-9 for Stream 1, and SM 0-3 for Stream 2. K1 and K2 are `GPUSpin` benchmarks launched simultaneously with identical configurations in both subfigures. In the desired outcome, at right, both kernels complete all their blocks before time 0.5s. However, this outcome only occurs approximately 50% of the time. The result at left occurs otherwise.

In the experiment of Fig. 11, K1 and K2 race to the WDU. Whichever reaches the WDU first has all its blocks dispatched first. When K1 arrives first, its blocks are dispatched across every TPC in its partition according to a previously researched assignment algorithm [21]. This saturates all the TPCs in K2's partition, forcing it to wait until K1 completes before running.

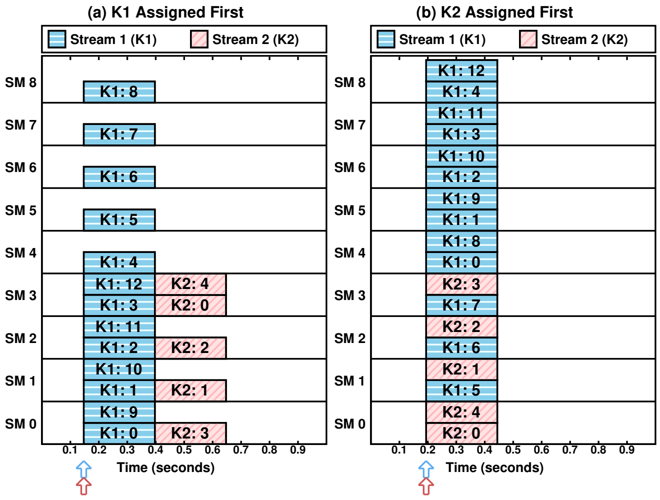

Fig. 11. Serial greedy block dispatch yielding unpredictable and bad assignments when overlapping partitions are used on GTX 1060 3 GB.

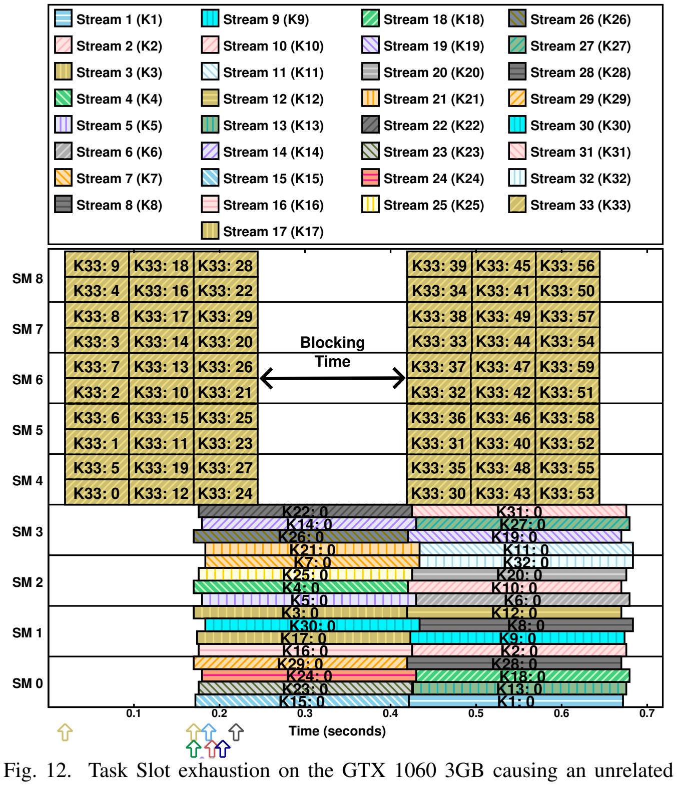

Fig. 12. Task Slot exhaustion on the GTX 1060 3GB causing an unrelated partition to block K33 from dispatching blocks.

If K2 arrives first, its blocks are dispatched across its partition, then K1's across its, resulting in a more-efficient assignment.

The consequences of task slot exhaustion. The constrained number of Task Slots are one of the few limits in the hardware scheduling pipeline in NVIDIA GPUs. Effectively, the number of Task Slots limits the number of kernels that the WDU can simultaneously consider for dispatch onto the GPU.

Task slots fill in priority order, with lower-priority tasks evicted as higher-priority ones become available in the TMU. This can cause unrelated compute partitions to block one another when many streams and stream priorities are in use. We illustrate one such case for the GTX 1060 3GB, which has 32 Task Slots, in Fig. 12.

Fig. 12 includes 33 streams, each containing a single, respectively numbered kernel. The GPU is partitioned roughly in half: Streams 1-32 on SMs 0-3, and Stream 33 on SMs 4-8. K33 consists of 60, 750ms blocks, whereas K1-K32 consist of one, 2.5s block. All kernels are derived from the `GPUSpin` benchmark. Stream 1-32 are higher priority than Stream 33.

Observe how K33 stops dispatching new blocks shortly after time 0.2s, not resuming until time 0.4s. K33 is in an independent stream and partition, but gets evicted from the WDU by the TMU after K1-K32 arrive, as they are higher priority. While these 32 kernels wait to complete execution, no Task Slots are available, and K33 must wait in the TMU, unable to dispatch blocks. Once one of the higher-priority kernels completes, as they do around time 0.4s, their Task Slot is vacated, allowing K33 to take it, and once more resume dispatching blocks.

In order to prevent this sort of blocking across unrelated partitions, we recommend using no more CUDA streams than the GPU has WDU slots, or prohibiting the use of CUDA stream priorities across all partitions. As long as the number of pending kernels does not exceed the number of WDU slots, eviction will not occur, nor will it occur if all kernels have identical priority.

D. *Evaluating Partitioning Strategy*

The principle preexisting work on spatial partitioning of GPU compute units found that the assignment of compute units to partitions is crucial on AMD GPUs, as adding compute units to partitions may *slow* the partition in some cases [14]. That work proposed and evaluated two partitioning strategies: SE-packed, and SE-distributed. SE stands for Shader Engine, and is equivalent to a GPC in our context. The SE-packed algorithm attempts to allocate all TPCs from the same GPC to the partition before expanding the partition across multiple GPCs. Alternatively, the SE-distributed algorithm attempts to distribute the TPCs of each partition across GPCs as evenly as possible. As this language has been adopted by other work [16], we continue its use here.

To evaluate if any similar slowing effects occur on NVIDIA GPUs, we port the relevant portions of the `hip_plugin_framework` used in [14] to our variant of the `cuda_scheduling_examiner` and reproduce their experiment, with the results shown in Fig. 13.

In Fig. 13, we mirror the figure layout of prior work [14]; Fig. 9, including how long a matrix multiply takes for each partition size under each assignment algorithm. Unlike on AMD GPUs, we find that adding an additional compute unit (TPC) to a partition does not slow that partition, no matter whether the unit is assigned from the same, or a different GPC. This is evident by the near-identical alignment of execution times for each algorithm. However, this finding does not obviate considerations around *inter-task* interference, and so we recommend using the SE-packed assignment strategy to minimize intra-GPC cache and bus interference.

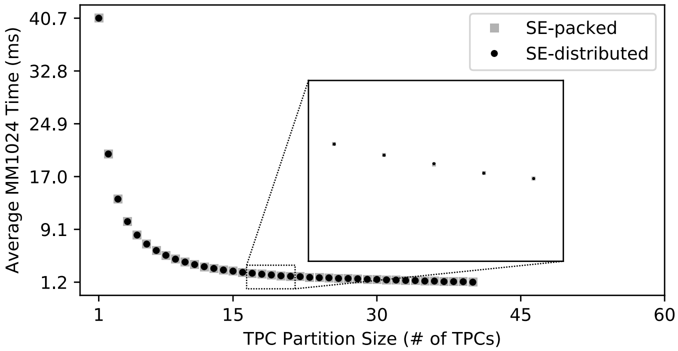

Fig. 13. SE-packed vs SE-distributed on the Titan V GPU.

VII. Case Study

In order to demonstrate the real-world usefulness of our GPU partitioning framework, we perform a case study where we apply TPC partitioning to a GPU shared by two instances of the YOLOv2 [38] convolutional neural network (CNN) trained for object detection.

Background and methodology. Object detection systems form a critical component of camera-based perception for autonomous systems. These systems take images as input, detect objects in them, and output bounding rectangles and labels for each object. Transformer-based systems [39] can reach higher accuracy than CNNs such as YOLOv2, but the comparable speed of YOLOv2-like approaches has kept them relevant in embedded computing applications. We consider YOLOv2 for these reasons, and because it is built on the DarkNet framework—a simple and easily-modifiable library.

We modify a DarkNet variant[22] to support running multiple instances of YoloV2 in parallel in different CUDA streams, to support setting TPC masks for each stream via `libsmctrl`, and to integrate with LITMUS^{RT}. LITMUS^{RT} is a patch to the Linux kernel, adding support for more formal real-time schedulers than what Linux provides by default [40]–[42]. Our host system[23] has plentiful CPU cores, so we use the partitioned fixed-priority (P-FP) LITMUS^{RT} scheduler and pin each of our instances of YOLOv2 onto a separate core. We configure YOLOv2 such that the image copy-in, detection pass, and result copy-out sequence form a single job. We run all instances on the NVIDIA Titan V GPU, with TPC partitioning configured as specified. Images are loaded from disk in parallel threads. We configure each job with a period of 420ms,[24] release jobs of all instances synchronously, and run 10,000 jobs, using a different image as input on each iteration drawn from the PascalVOC 2012 dataset [43]. Job lengths are timed using a library from prior work [44].

Protection from competitors. We first consider how our library is able to protect an instance of YOLOv2 from competing work which malfunctions. Table III contains the key data discussed throughout this section.

TABLE III
YOLOv2 Runtimes

Case	Competitor	Partition	Mean	Min	Max	St. Dev.
1	None	None	24	20	53	2.7
2	YOLOv2	None	40	35	69	2.8
3	Malfunctioning YOLOv2	None	82	73	111	2.9
4	Malfunctioning YOLOv2	**50/50**	**45**	**35**	157	22

When run alone on 8 TPCs,[25] YOLOv2 takes 24 ms on average to process a single frame (Case 1, Table III). When combined with a second YOLOv2 instance, this time nearly doubles to 40 ms (Case 2, Table III)—an unsurprising result given the halved computing resources. More surprisingly, the standard deviation remains low, and the relative min and max are hardly changed by the addition of this competing work.

However, this predictability does not persist if the competing instance malfunctions, as demonstrated by Case 3 in Table III. In this case, the competing instance spawns numerous unexpected kernels, stealing compute from the primary YOLOv2 instance. This occurs because the WDU dispatches from all runnable kernels equally, and the faulty addition of kernels from the competing instance displaces work from our primary instance. This results in a more-than-doubling of the primary instance's execution time to 82 ms—an unacceptable spillover of a malfunction that should have been contained to the competing instance.

TPC partitioning can help. When enabled and used to allocate four TPCs to each instance, the execution time distribution for the primary YOLOv2 instance nearly returns to its pre-malfunction state—about 44 ms per frame.[26] This indicates that partitioning compute units can be sufficient to protect against interference; in short:

Obs. 1. *TPC partitioning can defend a GPU-context-sharing task from losing performance to compute-intensive tasks.*

Beyond fault containment, this can be usefully applied when other tasks are unknown or have input-dependent computational sizes, but are expected to take a relatively constant amount of compute. However, to apply partitioning in this context, we must answer how partitions should be sized.

Tuning partition sizes. To investigate how the choice of TPC partitioning effects runtimes, we tested the full range of partitioning options for two instances of YOLOv2 on 8 TPCs and plot the results in Fig. 14. This plot shows, for each partitioning configuration, the mean, min, and max execution time of each instance. For example, when six TPCs are allocated to Instance A and two to B, Instance A takes 70 ms per frame, and Instance B takes 30 ms.

[22] https://github.com/AlexeyAB/darknet

[23] Our test system has a 16-core AMD 3950X with 32GiB of RAM running LITMUS^{RT} 5.4.224, NVIDIA driver 520.56.06, and CUDA 11.8.

[24] A large period provides margin for our parallel image loading from disk.

[25] Throughout our case study, we only consider up to 8 TPCs. YOLOv2 is a small network, and allocations of more than 8 TPCs (on the Titan V) yield negligible performance improvement.

[26] The maximum and standard deviation in this case are driven by approx. 7% of samples which are extreme outliers. We suspect that these are not caused by a partitioning issue—we see them in no other experiment—but rather an internal CUDA synchronization issue, such as that uncovered in recent work [45]. We intend to investigate this errata further in future work.

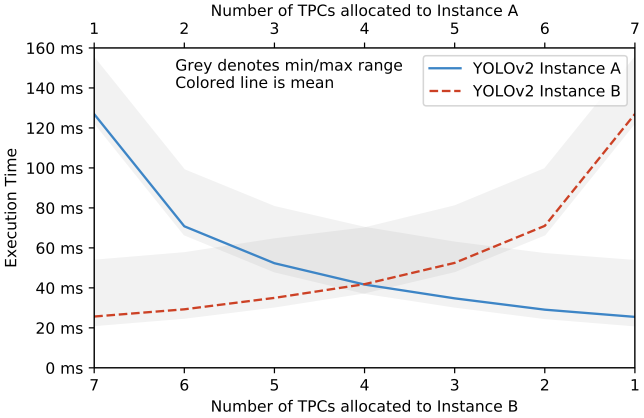

Fig. 14. Job time of two instances across various partitions of 8 TPCs.

Obs. 2. *TPC partitioning allows for smooth adjustment of task execution times.*

As we increase the number of TPCs for an instance in Fig. 14, min, mean, and max execution times all decline, if we decrease the number of TPCs, the reverse occurs. This allows for TPC allocations to be used as a proxy for priority. The larger an allocation, the sooner it will meet its deadline. However, there are further complications:

Obs. 3. *Providing additional TPCs to a task may provide a negligible performance improvement.*

Fig. 14 shows that for YOLOv2, the incremental benefit of an additional TPC declines after the addition of a second TPC. This may indicate that all layers of YOLOv2 can utilize at least two TPCs (since doubling the partition size halved execution time), but only some can utilize three or more TPCs—adding the third TPC (a 50% capacity improvement) only speeds up YOLOv2 by 14% (10 ms).

In all, our case study found TPC partitioning to be useful in real-world applications for protection, prioritization, and characterization, while imputing a minimal performance cost.

VIII. Conclusion

In this work, we uncovered a new means by which to spatially partition the computing cores on all NVIDIA GPUs since 2013. Our library, `libsmctrl`, allows for easy partitioning while exposing critical GPU details, such as the TPC to GPC mappings. We reinforce the usefulness of partitioning via a deep-dive into the capability of NVIDIA GPU hardware to support multiple parallel partitions. In our evaluation, we consider how the GPU's scheduling hardware can break partition boundaries and make non-optimal scheduling decisions in some cases. We compare to prior work, and find our approach more flexible and capable, while also easily applying to real-world workloads, as evidenced via our case study.

In future work, we hope to extend our approach such that co-running partitions need not share CUDA contexts. We further hope to investigate how the GPU enforces CUDA stream-ordering, how it schedules other engines such as copy, and how exactly the GPU Host Interface selects which queue to pull from when acquiring commands.

IX. Acknowledgements

We thank developers of the nouveau project, as well as current and former employees of NVIDIA, who informed our survey of public GPU architecture sources. We also thank Nathan Otterness for his assistance in porting his AMD Compute Unit partitioning benchmarks to CUDA.

References

[1] 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 IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2019, pp. 29–41.

[2] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, “Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS, June 2015, p. 119–130.

[3] T. Yandrofski, L. Chen, N. Otterness, J. H. Anderson, and F. D. Smith, “Making powerful enemies on NVIDIA GPUs,” in *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, Dec 2022, p. to appear.

[4] N. Feddal, H.-E. Zahaf, and G. Lipari, “Toward precise real-time scheduling on NVIDIA GPUs,” in *Proceedings of the 15th Junior Researcher Workshop for Real-Time Computing*, June 2022, pp. 20–24.

[5] NVIDIA, “NVIDIA A100 tensor core GPU architecture,” NVIDIA, Tech. Rep., 2020.

[6] G. A. Elliott, “Real-time scheduling for GPUs with applications in advanced automotive systems,” Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2015.

[7] 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*, Dec 2018, pp. 119–130.

[8] NVIDIA, “Multi-process service,” 2021, version R495.

[9] ——, “NVIDIA multi-instance GPU and NVIDIA virtual compute server,” 2020.

[10] 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*, Dec 2017, pp. 104–115.

[11] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. 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*, July 2018, pp. 20:1–20:21.

[12] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, “TimeGraph: GPU scheduling for Real-Time Multi-Tasking environments,” in *Proceedings of the 2011 USENIX Annual Technical Conference*. USENIX Association, June 2011.

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

[14] N. Otterness and J. H. Anderson, “Exploring AMD GPU scheduling details by experimenting with “worst practices”,” in *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, April 2021, pp. 24–34.

[15] N. Otterness, “Developing real-time GPU-sharing platforms for artificial-intelligence applications,” Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2022.

[16] H.-E. Zahaf, I. S. Olmedo, J. Singh, N. Capodieci, and S. Faucou, “Contention-aware GPU partitioning and task-to-partition allocation for real-time workloads,” in *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, July 2021, p. 226–236.

[17] A. B. Hayes, F. Hua, J. Huang, Y. Chen, and E. Z. Zhang, “Decoding CUDA binary,” in *2019 IEEE/ACM International Symposium on Code Generation and Optimization*, Feb 2019, pp. 229–241.

[18] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, “Dissecting the NVIDIA Volta GPU architecture via microbenchmarking,” April 2018.

[19] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, “Dissecting the NVIDIA Turing T4 GPU via microbenchmarking,” March 2019.

[20] 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 13th Annual Workshop on Operating Systems Platforms for Embedded Real Time Applications*, July 2017.

[21] 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*, Apr 2020, pp. 213–225.

[22] R. Spliet and R. Mullins, "The case for limited-preemptive scheduling in GPUs for real-time systems," in *Proceedings of 14th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, July 2018.

[23] N. P. Authors, "Nouveau: Accelerated open source driver for nVidia cards," 2022. [Online]. Available: https://nouveau.freedesktop.org/

[24] M. P. Authors, "The mesa 3d graphics library," 2022. [Online]. Available: https://www.mesa3d.org/

[25] J. F. Duluk Jr, T. J. Purcell, J. D. Hall, and P. A. Cuadra, "Error checking in out-of-order task scheduling," U.S. Patent 9,965,321, May, 2018.

[26] NVIDIA, "Open GPU documentation." [Online]. Available: https://github.com/NVIDIA/open-gpu-doc

[27] T. J. Purcell, L. V. Shah, and J. F. Duluk Jr, "Scheduling and management of compute tasks with different execution priority levels," U.S. Patent Application 13/236,473, Sep., 2011.

[28] J. F. Duluk Jr, L. Durant, R. M. Navarro, A. Menezes, J. Tuckey, G. Hirota, and B. Pharris, "Dynamic partitioning of execution resources," U.S. Patent 11,307,903, Apr., 2022.

[29] K. M. Abdalla, L. V. Shah, J. F. Duluk Jr, T. J. Purcell, T. Mandal, and G. Hirota, "Scheduling and execution of compute tasks," U.S. Patent 9,069,609, Jun., 2015.

[30] P. A. Cuadra, C. Lamb, and L. V. Shah, "Software-assisted instruction level execution preemption," U.S. Patent 10,552,201, Feb., 2020.

[31] S. H. Duncan, L. V. Shah, S. J. Treichler, D. E. Wexler, J. F. Duluk Jr, P. B. Johnson, and J. S. R. Evans, "Concurrent execution of independent streams in multi-channel time slice groups," U.S. Patent 9,442,759, Sep., 2016.

[32] M. W. Rashid, G. Ward, W.-J. R. Huang, and P. B. Johnson, "Managing copy operations in complex processor topologies," U.S. Patent 10,275,275, Apr., 2019.

[33] M. Sullivan, S. K. S. Hari, B. M. Zimmer, T. Tsai, and S. W. Keckler, "System and methods for hardware - software cooperative pipeline error detection," U.S. Patent Application 17/737,374, Aug., 2022.

[34] P. Tasinga, D. B. Yastremsky, J. Wyman, A. Ihsani, P. Nahar, and P. Bhatt, "Neural network scheduler," U.S. Patent Application 17/115,631, Jun., 2022.

[35] NVIDIA, "nvgpu git repository." [Online]. Available: git://nv-tegra.nvidia.com/linux-nvgpu.git

[36] ——, "NVIDIA's next generation CUDA compute architecture: Kepler GK110/210," NVIDIA, Tech. Rep., 2014.

[37] NVIDIA, "CUDA C++ programming guide," 2022, version PG-02829-001_v11.8.

[38] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Jul. 2017, pp. 7263–7271.

[39] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," in *European conference on computer vision*. Springer, 2020, pp. 213–229.

[40] "LITMUS$^{\mathrm{RT}}$ home page," Online at http://www.litmus-rt.org/, 2020.

[41] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2011.

[42] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. Anderson, "LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers," in *RTSS*, 2006, pp. 111–126.

[43] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results," http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[44] J. Bakita, S. Ahmed, S. H. Osborne, S. Tang, J. Chen, F. D. Smith, and J. H. Anderson, "Simultaneous multithreading in mixed-criticality real-time systems," in *Proceedings of the 27th Real-Time and Embedded Technology and Applications Symposium*, May 2021, pp. 278–291.

[45] T. Amert, Z. Tong, S. Voronov, J. Bakita, F. D. Smith, and J. H. Anderson, "TimeWall: Enabling time partitioning for real-time multicore+accelerator platforms," in *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, Dec 2021, pp. 455–468.