Hardware Compute Partitioning on NVIDIA GPUs for Composable Systems

Joshua Bakita
University of North Carolina at Chapel Hill, USA

James H. Anderson
University of North Carolina at Chapel Hill, USA

Abstract

As GPU-using tasks become more common in embedded, safety-critical systems, efficiency demands necessitate sharing a single GPU among multiple tasks. Unfortunately, existing ways to schedule multiple tasks onto a GPU often either result in a loss of ability to meet deadlines, or a loss of efficiency. In this work, we develop a system-level spatial compute partitioning mechanism for NVIDIA GPUs and demonstrate that it can be used to execute tasks efficiently without compromising timing predictability. Our mechanism supports composable systems by not requiring task, driver, or hardware modifications. In our evaluation, we demonstrate sub-1-μs overheads, stronger partition enforcement, and finer-granularity partitioning when using our mechanism instead of NVIDIA's Multi-Process Service (MPS) or Multi-instance GPU (MiG) features.

2012 ACM Subject Classification Computer systems organization → Heterogeneous (hybrid) systems; Computer systems organization → Real-time systems; Software and its engineering → Scheduling; Software and its engineering → Concurrency control; Computing methodologies → Graphics processors; Computing methodologies → Concurrent computing methodologies

Keywords and phrases Real-time systems, composable systems, graphics processing units, CUDA

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2025.18

Funding Work supported by NSF grants CPS 2038960, CPS 2038855, CNS 2151829, CPS 2333120, and ONR contract N0001424C1127.

1 Introduction

Rapid developments in artificial intelligence (AI)—especially deep neural networks (DNNs) running on GPUs [18]—have led to new cyber-physical systems, from intelligent assistants to self-driving cars. Real-world safety or usability concerns impose practical response-time deadlines on these systems, which may also need to run multiple AI tasks—such as one DNN for a conversational interface alongside others for object detection or planning in a self-driving car. However, this raises a problem–how to schedule GPU-using tasks onto a GPU efficiently while reliably meeting deadlines? When scheduling a GPU, generally either competitive sharing [42, 27, 45]—tasks run concurrently and fight for resources—or mutual exclusion [12, 11, 3]—one task runs at a time—are recommended.

Unfortunately, competitive sharing increases efficiency at the cost of timing predictability [11, 35, 2, 41, 7, 40], whereas mutual exclusion gives up efficiency for predictability. Without timing predictability, one cannot guarantee met deadlines. This tension puts embedded system designers in a difficult position. The easy option—trading off predictability for efficiency—is dangerous for safety-critical systems like self-driving cars.

This problem is exacerbated by the issue of composability. Each GPU-using task may be developed by different groups, may not be modifiable by the scheduler, and may change out-of-step with other tasks during a device's lifetime. This puts further burden on the scheduling system, as it must guarantee efficient and predictable execution for each task, even as tasks change opaquely.

Table 1 Comparison of spatial GPU compute partitioning mechanisms.

Mechanism	Portable	Logical Isol.	Transparent	Low-overhead	Hardware Enf.	Dynamic	Granular
Software [13, 39, 15, 37, 43]	✓	×	×	×	×	✓	✓
`libsmctrl` [4]	✓	×	×	✓	✓	✓	✓
NVIDIA MiG [29]	×	✓	✓	✓	✓	×	×
NVIDIA MPS [27]	✓	∼	∼	✓	∼	×	✓
`nvsplit` (ours)	✓	∼	✓	✓	✓	✓	✓

In this work, we demonstrate that spatial partitioning of GPU compute cores is an effective path to resolving this problem. To show this, we uncover and repurpose hardware capabilities in all NVIDIA GPUs to build a new system-level spatial partitioning mechanism. *Spatial partitioning*—a way to run tasks concurrently on mutually exclusive sets of cores—allows concurrent task execution for efficiency, while minimizing shared-resource interference between tasks to protect timing predictability. Our work builds on two key insights: GPUs are architecturally well-suited to spatial partitioning, and all NVIDIA GPUs contain hardware capabilities that can be leveraged to implement spatial partitioning.

While our work is motivated by the computational needs of DNNs, it is *not* constrained to DNNs—we consider arbitrary *unmodified* CUDA-using GPU tasks. We focus on NVIDIA GPUs for their market-leading technology and adoption, and on CUDA-using tasks because of an intermediate-software limitation—we believe that the hardware capabilities we unveil and leverage could be applied to spatially partition any NVIDIA GPU workload in the future.

Prior work. Prior work on the spatial partitioning of NVIDIA GPU cores is limited. Table 1 (returned to with rigorous definitions in Sec. 2.5 and Sec. 3) classifies key works. Most prior work ("Software" in Table 1) modifies tasks to cooperatively yield unallocated compute cores [39, 15, 37, 43, 13, 46]—these approaches suffer the inability to enforce partitions on misbehaving tasks. `libsmctrl` by Bakita and Anderson [4] addresses this enforcement problem, but requires task modification and a shared address space among all tasks. Multi-instance GPU (MiG) from NVIDIA [29] can hardware-enforce partitions without these compromises, but its partitions are static, less fine-grained, and only available on data-center GPUs. The only NVIDIA-provided option for all their GPUs, the Execution Resource Provisioning feature of the Multi-Process Service (MPS), does not enforce robust partition boundaries. We revisit these mechanisms at length in Sec. 3.

Contributions. In this work, we:

1. Reverse-engineer NVIDIA MPS (current state-of-the-art), unveiling previously-unknown hardware capabilities and quantifying its real-time safety.
2. Discover and mitigate efficiency and predictability pitfalls of NVIDIA MPS, including how it may assign two partitions of 50% to the *same* 50% of the GPU.
3. Develop a new system-level spatial-partitioning mechanism for NVIDIA GPUs—`nvsplit`—built on principles from `libsmctrl` and NVIDIA MPS.
4. Show that `nvsplit` supports unmodified tasks without measurable overheads or portability limitations.

5. Demonstrate that `nvsplit` has more efficient and granular partition enforcement than NVIDIA MiG and MPS, and equivalent or better timing predictability.
6. Reveal that NVIDIA MiG has a serious and undocumented cost to compute performance.

Organization. We introduce our system model, overview GPU architecture, and discuss spatial partitioning in Sec. 2. We review prior work in Sec. 3. In Sec. 4, we elucidate the implementation and pitfalls of NVIDIA MPS, and then in Sec. 5 apply the hardware capabilities used by MPS to develop `nvsplit`. We evaluate the overheads, partition enforcement, and granularity of `nvsplit` in Sec. 6, and conclude in Sec. 7.

2 Background

In this section, we summarize necessary background on the GPU (from [4, 5]), and discuss why spatial partitioning enables efficiency and timing predictability.

2.1 System Model

We assume an `x86_64` or `aarch64` platform containing at least one embedded or discrete NVIDIA GPU of the Volta (2018) generation or newer.

Tasks are assumed to be closed-source, unmodifiable, CUDA-using Linux processes. Non-CUDA GPU-using tasks may coexist, but may not use spatial partitioning.

We focus on embedded, real-time systems, where resources are limited but execution-time deadlines must be met.

2.2 GPU Usage Model

Each task executing on a GPU has an associated GPU *context*. This context includes per-GPU-task state, such as an on-GPU virtual address space. Unless explicitly stated otherwise, we assume a one-to-one mapping of GPU-context to CPU-task.

Work is dispatched into a context via one or more compute *kernels*. A kernel is launched by passing a GPU-executable binary, and a number of GPU *threads*, to a GPU-usage library such as CUDA or OpenCL. The library will then compose and pass a Task Metadata Descriptor (TMD) to the GPU for execution. Each GPU thread concurrently executes the same instructions, but operates at a different data offset. For example, a kernel for an element-wise array addition could replace a for loop over every index value with a GPU thread per value. The threads would then execute the loop body over all index values in parallel, rather than having to iterate.

Threads are organized into groups known as *blocks*, and all threads within a single block execute on the same SM (Streaming Multiprocessor; a group of compute cores). A series of kernel executions may be serialized via first-in-first-out command queues known as *streams* in CUDA. Only kernel executions in the same stream are serialized; kernels in separate streams are permitted to execute concurrently.

2.3 GPU Architecture

In Fig. 1 we illustrate the typical architecture of an NVIDIA GPU chip, using the Ada-generation AD102 as an exemplar (used in the RTX 4090 and RTX 6000 Ada GPU models). NVIDIA GPUs are subdivided into several independent *engines*, with the most significant being the Compute/Graphics Engine. Smaller engines handle special functions such as bulk

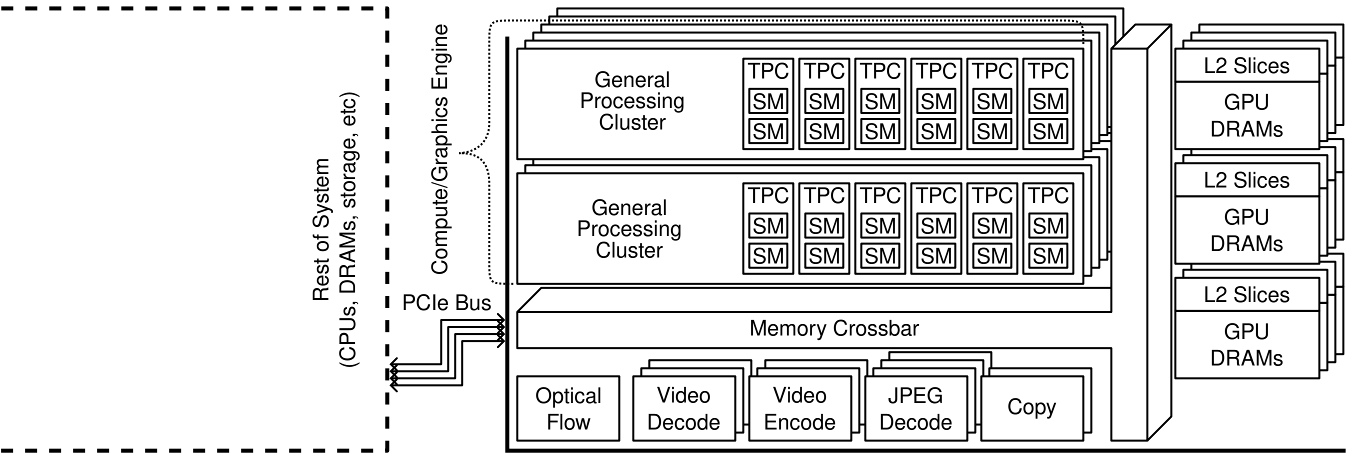


Figure 1 NVIDIA Ada Lovelace discrete GPU (AD102).

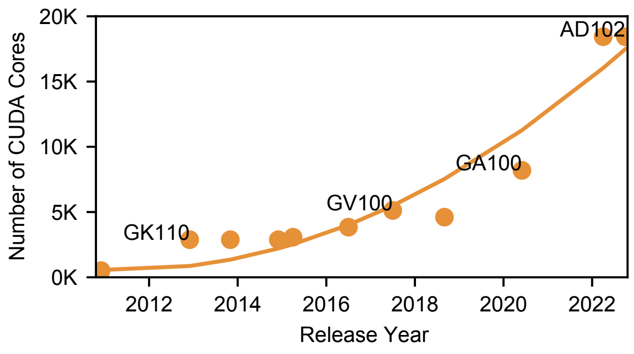


Figure 2 Number of CUDA cores in NVIDIA's top-end GPUs; 2011 to 2023. (Select points annotated with chip IDs.)

data movement (the Copy Engines) and video processing. Each engine may have a different context active on it at a time [5].

The Compute/Graphics Engine is subdivided into *General Processing Clusters* (GPCs). Each GPC is independently connected to each DRAM controller and the co-located last-level, level-two (L2) cache slices. The GPCs subdivide into *Thread Processing Clusters* (TPCs) of two *Streaming Multiprocessors* (SMs) each. Each SM contains dozens of compute cores (128 each on the AD102) and a level-one (L1) cache. The AD102 contains 18,432 compute cores total; Fig. 2 illustrates how core counts have increased in recent years.

We next discuss how GPU contexts are scheduled onto hardware engines.

2.4 GPU Scheduling

We present the scheduling pipeline from CPU task to GPU compute cores in Fig. 3. CPU tasks insert kernel launch commands (represented as TMDs) into CUDA streams, and those CUDA streams are mapped onto a smaller or equivalent number of GPU *channels*.[1] Which GPU channel(s) may access the Compute/Graphics Engine at a time is dictated by the *runlist*, making the runlist the central arbitrator.

GPU runlists (typically one for each GPU engine [5]) are composed of *time-slice groups*

[1] Strictly, channels contain a queue called a *pushbuffer*. The channel count is important—using more streams than channels causes blocking [5].

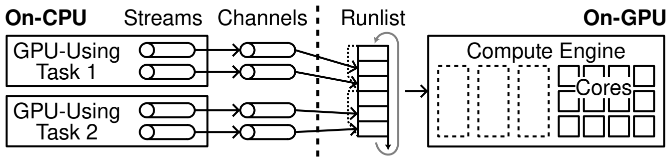


Figure 3 The scheduling pipeline for two tasks using the GPU Compute/Graphics Engine—the runlist arbitrates which task uses the engine at a time. (Dashed boxes at right represent intra-engine scheduling stages we skip—see [4].)

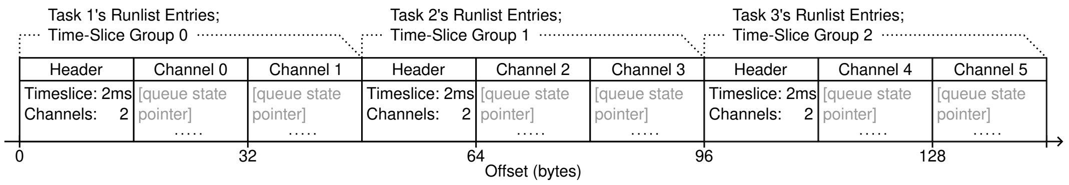

Task 1's Runlist Entries; Time-Slice Group 0
Task 2's Runlist Entries; Time-Slice Group 1
Task 3's Runlist Entries; Time-Slice Group 2
Header
Channel 0
Channel 1
Header
Channel 2
Channel 3
Header
Channel 4
Channel 5
Timeslice: 2ms Channels: 2
[queue state pointer]
0
32
64
96
128
Offset (bytes)

Figure 4 Example Compute/Graphics Engine runlist of three tasks.

(TSGs) of channels. TSGs are executed in a work-conserving, preemptive round-robin order by default. While a TSG is active, commands from all its contained channels are pulled and executed on the runlist-associated engine. Each TSG has an associated time-slice; upon expiration of this timeslice the runlist scheduler preempts any in-progress commands and switches to the next TSG. Such a switch is also triggered by the exhaustion of commands from all the TSG's channels. Note that all a TSG's channels must be from the same context.[2]

We show a runlist for three tasks in Fig. 4, following the in-memory layout. In this example, TSG 0 would be executed for 2 ms, and during that time, commands from channels 0 and 1 would be received and executed by the Compute/Graphics Engine. After 2 ms, any active commands from channels 0 and 1 would be preempted, and the GPU would move on to commands from the channels of TSG 1 for 2 ms. This process repeats for TSG 2, then loops back to TSG 0. The time-slice can be set differently for each TSG.

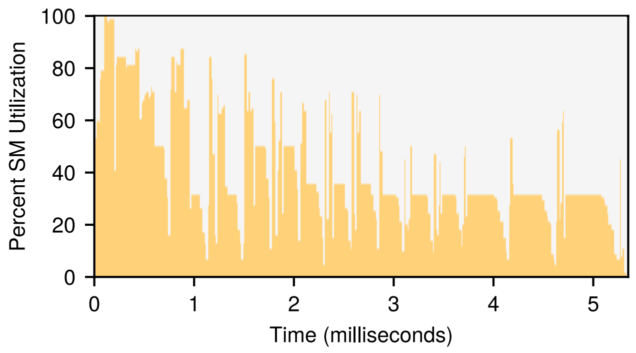


Figure 5 GPU utilization over one inference of the YoloV2 DNN in Darknet on the 4,352-core NVIDIA RTX 2080 Ti.

This GPU scheduling algorithm ensures that the Compute/Graphics Engine has only one context active at a time. This can lead to significant under-utilization, as many GPU-using tasks are unable to saturate all GPU compute cores on their own [42, 27, 45]. We demonstrate this for one inference of the YOLOv2 image-detection network in Fig. 5. At no point is the network able to utilize the entire GPU—it utilizes only 40% of the GPU's SMs on average.

[2] See line 293 in `manuals/ampere/ga100/dev_ram.ref.txt` [30].

Increasing GPU core counts (Fig. 2) have only worsened this problem.

To efficiently use the GPU, idle compute capacity must be reclaimed for other tasks. Unfortunately, concurrently running multiple tasks on the GPU (e.g., by sharing a single context) leads to a new set of problems.

2.5 Interference and Spatial Partitioning

When multiple tasks execute concurrently on a GPU, *shared-resource interference* can occur. This is where contention for shared hardware resources such as caches or compute cores creates slowdowns between tasks. Such interference creates unpredictability, as knowing when and how tasks will interfere on what shared resource is an unresolved area of study. Thus, to ensure timing predictability, few hardware resources should be shared.

Prior work has avoided such interference on the GPU by only allowing one task to access the GPU at a time [12, 11, 3]. This can lead to underutilization—concurrently running multiple tasks would allow for more efficient use of GPU hardware.

Spatial partitioning allows for concurrency without interference. It prevents hardware resources from being shared between tasks by partitioning them into mutually exclusive subsets, and assigning each subset to a concurrently running task. Without shared resources, interference is prevented, and execution times remain predictable.

In designing a spatial partitioning mechanism, it should satisfy the following properties:

Portable: The mechanism should work on a wide set of GPU models.

Logically Isolated: The mechanism should preserve logical isolation between tasks, *e.g.*, virtual address space isolation and independent exception handling.

Transparent: The mechanism should not require changes to tasks, *e.g.*, recompilation.

Low-overhead: The mechanism should add negligible overhead to critical-path operations, *e.g.*, a kernel launch.

Hardware-enforced: Partitions should be enforced by hardware to protect from malicious or misbehaving tasks.

Dynamic: Partitions should be reconfigurable without restarting a task.

Granular: Partitions should be defined in granular units, *e.g.*, a TPC for compute partitioning.

The first four properties are desirable for any software system, but the last three are partitioning-specific. We now discuss prior work in light of these requirements.

3 Related Work

Efficiently using a GPU in a real-time system requires concurrently running tasks on the GPU Compute/Graphics Engine, with partitioning of shared hardware resources. Prior work has identified the GPU DRAMs (with co-located L2 cache slices) and SMs (with co-located L1 caches) as needing to be partitioned [40, 4, 15]. This section covers prior work on and towards such partitioning.

3.1 DRAM Partitioning

The history of DRAM partitioning is extensive ([19, 20, 44] are exemplars), but we are aware of only one work that has extended CPU-centric approaches to the GPU: Fractional GPUs by Jain *et al.* [15]. This work uses a memory-organization approach known as *page-coloring* [19] to force each task onto prescribed GPU DRAM and L2 units. Such memory reorganization requires difficult-to-obtain model-specific hashing functions, but the principle is generally applicable to any GPU. NVIDIA has since developed a proprietary alternative, but this is not available on most GPUs and cannot be enabled independently of MiG [29, 9].

The DRAM partitioning technique of Fractional GPUs satisfies all desired spatial isolation properties stated in Sec. 2.5, so we assume the application of such an approach and focus on the remaining problem: compute partitioning.

3.2 Compute Partitioning

Prior work on spatial partitioning for GPU compute cores can be divided into academic-provided and NVIDIA-provided solutions. We summarize these works in Table 1.

Academic-provided solutions. Third-party solutions for spatial partitioning on NVIDIA GPUs have been limited to cooperation-based software mechanisms until recently. One system [39], which has been recently improved [15, 37], is commonly used in papers that claim to partition NVIDIA GPUs. This approach depends on kernels launching blocks on all SMs, and on each block aborting if it finds itself executing on an SM outside of its partition. This approach is vulnerable to a full loss of partitioning if any block misbehaves and does not cooperatively yield an unassigned SM. Another cooperation-based variant known as persistent threads [43, 13], still used in recent work [46], is similarly vulnerable to misbehaving tasks. Given the inability of these works to *enforce* partitioning, they are vulnerable to a loss of isolation. We group these works under the “Software” heading in Table 1, as they implement partitioning via task modification rather than via hardware features.

`libsmctrl` by Bakita and Anderson [4] addresses the enforcement problem by leveraging undocumented hardware capabilities to enforce partitions of TPCs. Unfortunately, like earlier mechanisms, `libsmctrl` requires merging tasks into the same context to concurrently execute them, compromising logical isolation and transparency.

NVIDIA-provided solutions. Partitioning solutions from NVIDIA are able to enforce partitioning at a hardware level, but were not designed with the needs of an embedded real-time system in mind. The two principal options provided by NVIDIA are the Multi-instance GPU (MiG) feature, and the Multi-Process Service (MPS).

NVIDIA MiG [29] allows multiple contexts to concurrently run on the GPU by splitting the GPU into static, fixed-size partitions. Each partition may then concurrently run a different task. Partition options are highly limited, with at best four possible partition sizes, and the smallest partition size is 14 SMs. MiG is implemented by duplicating every part of the hardware scheduling pipeline for every GPC [9], and is only available on NVIDIA’s highest-end datacenter GPUs. This approach cannot provide fine-granularity partitioning, and requires hardware modifications that NVIDIA has shown no intent to make widely available. This leads us to classify MiG as non-granular and non-portable in Table 1.

Conversely, NVIDIA MPS [27] is available on any recent discrete NVIDIA GPU. MPS (since the Volta architecture),[3] enables the Compute/Graphics Engine to concurrently execute multiple tasks, but does not control which SMs each task is assigned to. This may result in two tasks sharing an SM [27], which can cause as much as a 100x performance degradation [40]. Due to this limitation, we classify MPS in Table 1 as only partially providing granular,

[3] Pre-Volta-generation MPS works differently, providing little-to-no isolation between co-running tasks. When we refer to MPS in this paper, we refer to the Volta-generation-and-newer version.

hardware-enforced partitioning. Other properties of MPS are undocumented, and we derive those later in this work.

3.3 Barriers to Better Compute Partitioning

Unfortunately, devising better solutions for spatial partitioning of GPU compute has been stymied by the scarce details released about NVIDIA GPU architecture and capabilities—even instruction encodings are secret [14, 17, 16]. Many years of investigation have begun to contravene this limitation.

Otterness *et al.* [34] and Amert *et al.* [2] elucidated the scheduling of CUDA kernels via black-box experiments and introduced the `cuda_scheduling_examiner` tool, which we use. Olmedo *et al.* [32] and Bakita and Anderson [4] used these results and other sources to construct a model of how the underlying GPU kernel dispatch hardware works. Parallel work by Capodieci *et al.* [6], Spliet and Mullins [38], and Bakita and Anderson [5] clarified the preemption and high-level scheduling capabilities of NVIDIA GPUs. This last work [5] also introduced two tools we use: the `nvdebug` tool for directly extracting GPU state, and the `gpu-microbench` suite for examining scheduling behavior. Outside of the academic community, the Nouveau [24] and Mesa [23] reverse-engineered NVIDIA GPU driver projects have documented GPU hardware capabilities and CPU-to-GPU interfaces. We lean heavily on all these prior works throughout our paper.

4 How Does MPS Work?

In search for a better spatial partitioning mechanism for real-time embedded systems, we begin by investigating the only portable mechanism for co-running unmodified tasks on NVIDIA GPUs—MPS. What hardware capabilities does MPS leverage, and how MPS fall short of the properties we desire in a spatial partitioning mechanism? Given the absence of prior work, we investigate these questions experimentally. This is relevant both to our work, and to the safety of other works which propose using MPS in embedded systems.

4.1 Methodology

To determine how MPS interacts with the GPU scheduling pipeline, we applied the `nvdebug` tool [5], `gpu-microbench` suite [5], and `cuda_scheduling_examiner` toolkit [34] to test and observe GPU state and behavior with and without MPS. Adding MPS changed more aspects of GPU state than `nvdebug` was able to display, so we extended it on Ada (2022) and older GPUs, drawing layout information from open-source NVIDIA code [30, 26, 28] and the `nouveau` driver [24]. Our improved version of `nvdebug` is available.[4]

To understand the semantic meaning of this newly accessible GPU state, we leveraged context and definitions from NVIDIA patents touching on the runlist scheduler [10], kernel scheduling pipeline [36, 1], MPS [8] and MiG [9]. Unfortunately, patents may describe infeasible or impossible devices, and so we only tenuously relied on them, verifying described behavior with experiments.

[4] Available at `http://rtsrv.cs.unc.edu/cgit/cgit.cgi/nvdebug.git` and within our artifact.

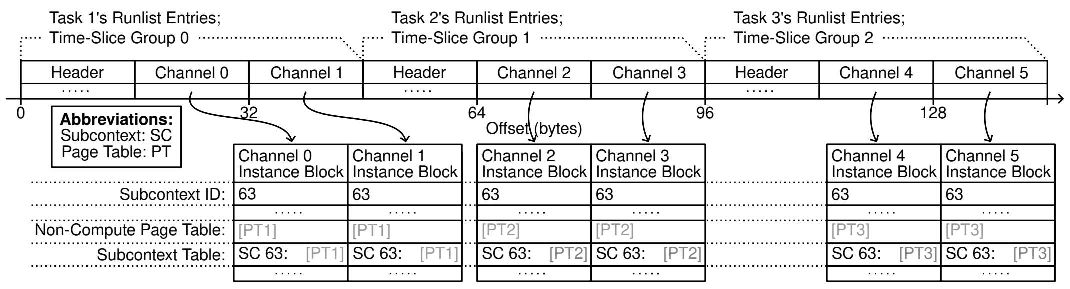


Figure 6 Compute/Graphics Engine runlist of three tasks; reillustrated from Fig. 4 with detail.

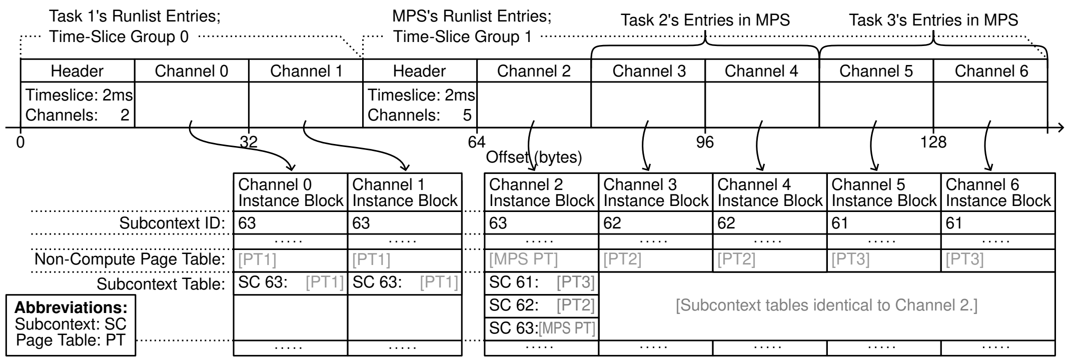


Figure 7 Compute/Graphics Engine runlist of three tasks, with Task 2 and 3 run as MPS clients.

4.2 MPS Terminology

MPS uses a client-server paradigm, where each CUDA-using task is a client of at most one MPS server task. The MPS server acts as an intermediary to the GPU for its clients, and allows clients to run concurrently with one another. Relative to the GPU, a system of two MPS clients and one MPS server would appear as a single task, since clients only access the GPU through the MPS server. Multiple MPS servers may exist, each with different clients.[5] In this case, each MPS server would appear to be a separate task to the GPU.

4.3 How MPS Modifies Runlist Scheduling

We now investigate how adding MPS modifies arbitration between GPU-using tasks, *i.e.*, how it modifies the runlist and associated data structures. To enable our subsequent analysis of MPS's pitfalls, this section is unusually detailed.

We begin by reillustrating Fig. 4, with detail from our extensions to `nvdebug`, in Fig. 6. This figure retains the same runlist as Fig. 4, but expands on the configuration of each channel. The channel-configuration data structure is known as the channel's *instance block*, and—besides describing the command queue (not shown for space)—specifies the virtual address space to be used for commands from the channel. Virtual address spaces are configured in a peculiar way; each channel includes an indexed list of page tables, and the page table to use is selected by specifying an index into that list. The list of page tables is called the "Subcontext

[5] Support for multiple MPS servers is only implicitly documented. The environment variable `CUDA_MPS_PIPE_DIRECTORY` can be set to control where an MPS server advertises itself, and where CUDA searches for the MPS server. Two MPS servers cannot advertise to the same path, and so this variable must be set uniquely for each to allow multiple servers to exist.

Table," and the channel's index into it is called the channel's "Subcontext ID."[6] Adding to the confusion, this mechanism only selects the page table for the Compute/Graphics Engine; the instance block includes a separate page table configuration field for other engines, such as Copy.[7] In our experiments, we always observed the same page table configuration for all engines; we mirror that in Fig. 6. Note that all channels in a TSG have an identical subcontext table—this is a requirement for channels sharing a context.[8] All of these details are important in order to discuss how MPS effects address-space isolation.

In Fig. 7, we illustrate how runlist and channel configurations change when MPS is enabled. This is still a system of the same three GPU-using tasks, but an MPS server has been started with Task 2 and 3 as clients (Task 1 remains independent of MPS). Both the runlist location and virtual address space configuration have changed for the tasks now running as MPS clients. We discuss each change in turn.

With MPS enabled, the two tasks now running as MPS clients no longer have independent TSGs in the runlist. The runlist only contains two TSGs: one for Task 1 (TSG 0), and one for the MPS server (TSG 1). While Tasks 2 and 3 do not retain independent TSGs, they do retain independent channels within the MPS server's TSG (Channels 3–4 for Task 2 and 5–6 for Task 3). (Note that the MPS server has taken Channel 2 for its own use, and so Tasks 2 and 3 are forced to use channels with higher IDs.) When this runlist is scheduled, Task 1 will, as before, execute for 2 ms before its budget expires and it is preempted. The GPU will then switch to the next TSG in the runlist, the one for the MPS server. This will likewise be run for 2 ms before the GPU loops back to Task 1. While the MPS server's TSG is active, commands from all its channels are sent to the Compute/Graphics Engine concurrently. The effect is such that all the MPS client tasks's CUDA streams concurrently execute as though they were in the same program. Note that the two MPS clients share a single 2 ms time-slice with a 4 ms period, whereas before they each got a single 2 ms time-slice at a 6 ms period. This is a reduction in GPU time available jointly to the two tasks, from two-thirds to one-half. This is a side-effect of enabling MPS to be aware of: if any other tasks in the system continue to run without MPS, the total GPU time available to all MPS clients will be less than the total time those tasks would have collectively received if run apart.[9]

We now consider how virtual address space configurations change—or stay the same—with the addition of MPS. Visible in Fig. 7, at the bottom of the Channel 2–6 Instance Blocks, the Subcontext Table for every channel of the MPS server TSG is triple the size of the tables in Fig. 6. This allows the table to include separate page tables for MPS, Task 2, and Task 3; the appropriate one is selected by the Subcontext ID on each channel. Each channel also has a non-compute page table identical to the one selected by its Subcontext ID. In this manner, Tasks 2 and 3 retain distinct virtual address spaces, just as without MPS.[10]

4.4 Evaluating Spatial Partitioning in MPS

What does this mean for MPS's suitability to real-time embedded systems? We answer by considering MPS under each of our desired properties (Sec. 2.5). Without lack of generality,

[6] Subcontext ID is also "Virtual Engine ID" (VEID) in some sources.
[7] See line 297 in `manuals/ampere/ga100/dev_ram.ref.txt` [30].
[8] See line 293 in `manuals/ampere/ga100/dev_ram.ref.txt` [30].
[9] The time-slice length of the MPS server could be extended to ensure tasks retain access to an equivalent fraction of GPU time, but this is unsupported by the NVIDIA driver.
[10] Why the convoluted TSG-wide Subcontext Table, rather than a per-channel page table? This may speed up context switches by allowing a TSG's page tables to be read all at once, rather than requiring a scan of all a TSG's channels.

we assume that all MPS clients are associated with a single server in this subsection.

4.4.1 Portability

The version of MPS we study is supported on all of NVIDIA's GPUs since Volta (2018), including their embedded "Jetson" GPUs (as of CUDA 12.5).

4.4.2 Logical Isolation

Without MPS, contexts are isolated from one another in their GPU addresses spaces, hardware scheduling decisions, and exception handling. MPS preserves isolation in only the first area.

The MPS documentation states that MPS clients have fully isolated virtual address spaces [27, Sec. 1.1.2]. Our findings support this—each MPS client exclusively uses its own page table. This follows from the per-subcontext page tables discussed in Sec. 4.3. Specifically, the unique-per-MPS-client subcontext ID is passed along with commands to the Compute/Graphics Engine,[11] and this ID is used to access and maintain per-subcontext page table state throughout the scheduling and execution pipeline [8]. This isolation also covers non-compute engines, as their page table[6] is always configured to match the per-subcontext page table. Only the kernel-level driver can change a channel's page table or subcontext ID,[12] ensuring that no client may reconfigure itself to access another client's pages.

Other areas lack isolation, bringing us to our first pitfall:

▶ **Pitfall 1.** *MPS clients share a per-server limit on the number of concurrently executing kernels.*

The use of subcontexts does not isolate MPS clients from one another in the GPU's hardware scheduling pipeline. Prior work on this pipeline [4] found that two tasks co-running in a single context may conflict due to a hardware limit on the number of concurrent kernels.[13] While an NVIDIA patent [8] suggests this limit is maintained per subcontext, we do not observe this, even on the most-recent Hopper- and Ada-generation GPUs. We tested by launching several hundred kernels, finding that the number of kernels a task can concurrently execute is reduced by the number of kernels concurrently executing in other MPS clients.

▶ **Pitfall 2.** *A crash in any MPS client may crash all MPS clients.*

The MPS documentation warns that MPS clients are not isolated from "fatal GPU faults" in other MPS clients [27, Sec. 2.2.3]. Such faults include errors such as out-of-bounds memory accesses in kernels. We test and find this lack of isolation persists on NVIDIA's latest Hopper- and Ada-generation GPUs (at least for out-of-bounds memory accesses).

NVIDIA's drivers report exceptions on a per-subcontext basis,[14] and other documentation hints that exceptions do not necessarily halt the entire TSG.[15] We urge NVIDIA to leverage these capabilities to enhance MPS's fault isolation.

In summary, MPS preserves address space isolation, but does not fully isolate the scheduling pipeline nor prevent on-GPU exceptions from crashing other MPS clients. This partial isolation is better than `libsmctrl` and software partitioning, but worse than MiG.

[11] See line 2525 in `manuals/ampere/ga100/dev_pbdma.ref.txt` [30].
[12] See line 2545 in `manuals/ampere/ga100/dev_pbdma.ref.txt` and line 525 in `dev_ram.ref.txt` [30].
[13] Specifically, "task slot exhaustion" in Bakita and Anderson [4].
[14] See line 1115 in `src/nvidia/src/kernel/gpu/mmu/arch/volta/ kern_gmmu_gv100.c` [26].
[15] See line 666 in `manuals/ampere/ga100/dev_runlist.ref.txt`

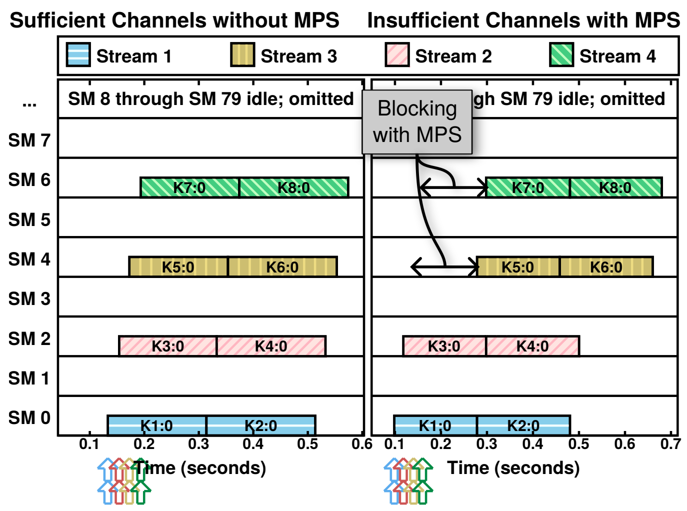


Figure 8 Four streams, each with two one-block kernels launched in them on an otherwise-idle NVIDIA Titan V GPU. Left is without MPS, right is with MPS defaults.

4.4.3 Transparency

We define a transparent partitioning mechanism as one that does not require any changes to tasks (Sec. 2.5). This property encompasses more than no binary modification; tasks should not need to be modified to account for a different set of available features. While MPS does not require modifying task binaries, it changes the set of supported features and the scheduling behavior in a semi-transparency-compromising way.

▶ **Pitfall 3.** *MPS clients cannot launch kernels from on the GPU.*

CUDA Dynamic Parallelism (CDP) [25, Sec. 9][16] allows for one CUDA kernel to launch another without involving the CPU. MPS omits support for this feature [27, Sec. 2.3.2]. We verified that this restriction persists even on NVIDIA's latest Ada-generation GPUs; any MPS client attempting to use this feature will get an error during initialization. We could not identify a conclusive reason why this feature is unsupported with MPS.

▶ **Pitfall 4.** *MPS clients support fewer concurrent CUDA streams.*

The MPS server also changes at least one scheduling-related property for its clients; the number of channels available to a task. Each MPS client only has access to two channels by default, whereas each non-MPS task has access to eight by default. An insufficient number of channels can result in implicit synchronization [5], so this changed default can significantly impact scheduling behavior, as shown in Fig. 8. Each subfigure shows how eight single-block kernels in four streams execute over time (x-axis) on the GPU's SMs (y-axis). Kernel launch times are indicated with arrows at bottom, and the stream of each kernel is color-and-pattern coded. On the left, we show the system running without MPS; on the right, we show it running with MPS. No other work is running in this system. With MPS (right), we see that kernel launches are blocked due to channel exhaustion (as in [5]).

▶ **Pitfall 5.** *MPS clients receive fake SM IDs.*

[16] CDP is also called CUDA Native Parallelism (CNP) or "GWC" [22, 26].

In GPU kernels, the special register `%smid` "returns the SM identifier on which a particular thread is executing" [31, Sec. 10.8]. Unfortunately, we found this register returns inconsistent values across MPS clients. For example, each MPS client's kernel's blocks start on `%smid` zero, and subsequent blocks start on sequentially-increasing SM IDs—even if another client claimed to be executing on those SMs. This indicates that the `%smid` register is emulated for each MPS client, as suggested in an NVIDIA patent [8]. We tested and found this behavior on Volta-, Turing-, Ampere-, and Ada-generation GPUs. This pitfall primarily hinders GPU study by obfuscating the SM assignment algorithm.

In summary, MPS affects available CUDA features, scheduling concurrency, and hardware behavior, making it only partially transparent.

4.4.4 Overheads

GPU commands, such as kernel launches, are sent to the GPU via the queue encapsulated within a channel. MPS clients have direct access to their channel queues, and so no overheads are added to the kernel-launch critical path. We verified this on Volta-, Turing-, and Ada-generation GPUs via the `measure_launch_oh` benchmark we added to `gpu-microbench`.

Task startup overheads, such as the time for library loading, are affected by MPS. The MPS server is lazily initialized, meaning the first MPS client pays an extra startup overhead. As this can be avoided by using a dummy task to pull forward server initialization, we do not consider it a pitfall. We show other startup overheads to be negligible in Sec. 6.

4.4.5 Partitioning Capability

As noted in Sec. 3, MPS only supports a static, per-task limit on what fraction of a GPU's TPCs a task may use. This limit, set via an environment variable,[17] is called "Execution Resource Provisioning."

▶ **Pitfall 6.** *The partition size for an MPS client is static.*

The partition size is a specified at context creation, and NVIDIA provides no API to change the partition size for an already-created context.

▶ **Pitfall 7.** *MPS partitions are not bound to specific SMs.*

No API is provided to specify a specific set of SMs, TPCs, or GPCs onto which an MPS client is partitioned, and we find that MPS also does not select a set internally.

The MPS documentation does not clarify how partitioning is enforced. However, we find that when MPS is running, a GPU register[18] is set to enable "dynamic partitioning" in the Work Distribution Unit (WDU). The WDU is the GPU hardware unit responsible for dispatching blocks of pending kernels to TPCs [4]. A patent filed by NVIDIA when execution resource provisioning was added to MPS appears to describe this feature [8].

In the patent, NVIDIA describes their dynamic execution resource partitioning system as associating each subcontext (MPS client) with a configurable number of credits. Every time a block of a kernel is dispatched to a TPC not previously occupied by its subcontext, the number of credits is decremented. Once a subcontext reaches zero credits, the WDU will only dispatch blocks from that subcontext to TPCs already in-use by that subcontext. When

[17] `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE`
[18] `NV_PGRAPH_PRI_CWD_PARTITION_CTL` [30]; CWD is a synonym for the WDU [4].

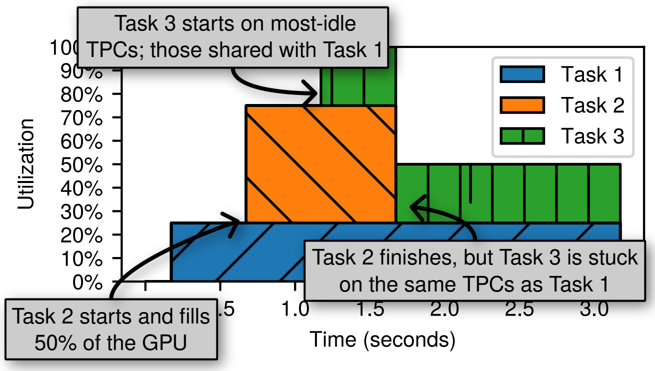


Figure 9 Three tasks run with MPS, released at time 0, 0.5, and 1.0 respectively on an NVIDIA Titan V GPU. MPS's execution resource provisioning limits each task to 50% of the GPU. MPS quirks force Task 1 and 3 to share TPCs, leaving half idle.

a TPC is vacated by all work from the subcontext, the number of credits is incremented. In effect, this caps the number of TPCs that any subcontext may concurrently have kernels executing on, but makes no guarantees about which TPCs are assigned to each subcontext.

We find this behavior holds when partitioning with MPS. This is a problem, as GPU partitions mis-aligned with hardware can lead to cache, bus, TLB, and other interference [4, 40]. We give an experimental example in the following pitfall.

▶ **Pitfall 8.** *The hardware implementation of MPS's partitioning feature is prone to assigning two tasks to the same set of SMs, leaving other SMs idle.*

As MPS makes no guarantees about which TPCs are assigned to which MPS client, two clients with partitions of 50% may each be assigned to the same set of TPCs. We demonstrate this with an experimental result in Fig. 9 for a system of three tasks running as MPS clients, each with a 50% GPU partition. For this example, it is important to know that the WDU tries to spread work out to as many TPCs as possible, and assigns blocks of large kernels to less-occupied TPCs first [32]. Task 1 started first. Task 1 was a light workload and only required about a quarter of the GPU, but the WDU spread that work across as many TPCs as possible, causing Task 1 to partially occupy 50% of the TPCs. Task 2 was a heavy workload, and upon joining, it fully occupied the 50% of the TPCs unoccupied by Task 1. At this point, around 0.75 seconds, all TPCs were busy with either Task 1 or Task 2, but the TPCs containing Task 1 had capacity for more work. As Task 3 joined, it occupied the remaining capacity on the TPCs in-use by Task 1. Later, when Task 2 terminated, the 50% of TPCs it was using became idle. Unfortunately, Task 3 remained on the same TPCs as Task 1, as it had already maxed out its active-on-50%-of-TPCs limit. This left 50% of the GPU unused, despite substantial pending work.

Such partition settings, where the sum of allocated compute exceeds 100%, is NVIDIA-recommended [27, Sec. 2.3.5].

This pitfall could be worse. The WDU will normally use an alternate assignment algorithm for small kernels, packing kernels unto SMs before spreading them onto idle SMs [32]. If this behavior persisted between kernels of different MPS clients, a system of only two MPS clients could be bound to to the same set of TPCs. Fortunately, after repeating a variant of the experiments from prior work [32], we did not find a similar behavior between MPS

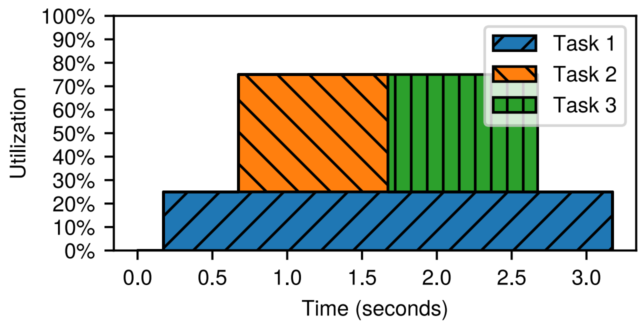


Figure 10 Same experiment as in Fig. 9, but with `libsmctrl` used to partition Task 1 onto different TPCs than Tasks 2 and 3.

clients.[19] This unexpected behavior appears triggered by the aforementioned GPU register[13] for dynamic partitioning—zeroing this register restores the normal algorithm.

In summary, MPS's partition sizes cannot be changed dynamically, its partition boundaries are weak, and its partitions may unexpectedly overlap. This completes our classification of MPS in Table 1, showing that no prior work is adequate for spatial partitioning GPU compute cores in an embedded, real-time system.

5 Compute Partitioning with `nvsplit`

Among prior work (Table 1), `libsmctrl` and MPS have complementary strengths. MPS supports at least some logical isolation and transparency, whereas `libsmctrl` supports every desired partitioning property. We combine these two system—and fix bugs in both—to build `nvsplit` with the union of MPS and `libsmctrl`'s best attributes.

Specifically, `nvsplit` takes `libsmctrl` and turns it into an automatically-loaded library that can apply partitions to any CUDA-using task (without modification), and adds a control tool called `nvsplitctrl` to support setting or dynamically changing the partition for any running task. This is only useful if tasks can run concurrently, so `nvsplit` requires starting MPS first to provide this capability. `nvsplit` then provides some optional mitigations to MPS's transparency-compromising pitfalls.

5.1 MPS + `libsmctrl`

MPS provides no rigorous or dynamic means for partitioning compute cores, but `libsmctrl` [4] is able to provide both of these properties. Combining the two is simple.

`libsmctrl` implements partitioning by modifying the TMD for each kernel immediately before it is uploaded to the GPU. The TMD contains a field that specifies which TPCs the hardware may run the kernel on, and `libsmctrl` modifies this field to effect partitioning.

Tasks dispatch launches the same, with as without MPS—by inserting launch commands in their channel queues. MPS does not interdict any portion of this process, meaning that TMDs are untouched by MPS. Thus, including `libsmctrl` in an MPS client and setting a global TPC mask is sufficient to limit all kernels of that client to the specified set of TPCs.

[19] To workaround Pitfall 5, we observed the change in block distribution by limiting each task to one TPC, and chaining a large kernel after the small one that would be assigned to the same TPC without MPS. The large kernels do not limit the utilization of each other, indicating that the preceding small kernel of each client ran on a different TPC.

We adjusted the experiment of Fig. 9 to use `libsmctrl`, assigned Task 1 the lower 50% of TPCs, and assigned Task 2 and Task 3 the upper 50%. In this configuration, Task 3 completed a quarter-second faster, as it did not get stuck on the same TPCs as Task 1. In this way, `libsmctrl` avoids Pitfalls 7 and 8 of MPS.

Unfortunately, `libsmctrl` requires minor task modification and recompilation to use, compromising its transparency. We address this limitation in `nvsplit`.

5.2 Supporting Unmodified Tasks

`libsmctrl` effects partitioning through interactions with the CUDA library; tasks only need a single API call to enable a partition. `nvsplit` eliminates the need for modification by effectively making the loader perform this API call.

Specifically, we make `nvsplit` fully transparent to a task by instructing the loader to pre-load `nvsplit.so` before calling into the task. (This can be done on Linux by setting the `LD_PRELOAD` environment variable to `nvsplit.so`.) As part of pre-loading, the loader will automatically call library constructor functions—we write one into `nvsplit` that loads CUDA, reads our new `NVSPLIT_MASK` environment variable, sets up the task for partitioning control via `nvsplitctrl`, and registers the TMD interception callback with CUDA to enable partitioning. The loader than resumes as normal, and whenever the task executes a kernel launch, our pre-registered callback will be triggered inside CUDA and the TPC mask will be applied—no task modification required.

5.3 Dynamic Partition Changes

`nvsplit` supports dynamic partition changes by exposing a shared-memory region from each partitioned task that contains the current partition setting. Changes to this setting are automatically detected and applied to all subsequent kernel launches. We provide a tool called `nvsplitctrl` that can dynamically change a task's TPC partition given the target task's PID. However, as with `libsmctrl`, already-launched kernels cannot be re-partitioned; the new partition setting will only apply to kernels launched after the change.

5.4 Minimizing MPS's Pitfalls

`nvsplit` uses MPS to allow tasks to run concurrently, and thus inherits some of its pitfalls. Fortunately, Pitfalls 6–8 do not apply, and Pitfalls 4–5 can be mitigated.

Mitigating Pitfall 4 The number of channels provided to each client by the MPS server can be configured via an environment variable, and MPS clients mirror the behavior of non-MPS-tasks once the number of channels per task is configured to its non-MPS default (8 channels).[20] We recommend that NVIDIA adapt this configuration as the MPS default.

This mitigation comes with a small cost. TSGs are limited to a maximum of 128 channels [30], meaning that an MPS server can only service 15 clients of 8 channels each (after subtracting 8 MPS-server-internal channels). We verified this 15-client limit on both Volta- and Ada-generation GPUs.

[20] Set the environment variable `CUDA_DEVICE_MAX_CONNECTIONS` to 8 before launching the MPS server.

Mitigating Pitfall 5 We found that the %*smid* register can be restored to its consistent, non-MPS behavior by toggling off the WDU's dynamic partitioning register.[13]

However, this pitfall should be mitigated only when necessary. To better support tasks which ignore the CUDA programming model and attempt to execute work on self-selected SMs (such as persistent threads [43, 13]), we can "hide" the existence of unallocated SMs with the WDU's dynamic partitioning capability. To do this, leave on fake SM IDs, and set MPS's partition limit to match the number of SMs allocated by `nvsplit`. This ensures that tasks only see allocated SMs, with seemingly-contiguous SM IDs.

5.5 Additional Enhancements

In the course of investigating Pitfall 5, we identified an error in how `libsmctrl` determined mappings from SMs to GPCs. `libsmctrl` assumed that SM IDs are assigned linearly to on-chip GPCs, such that the first n SM IDs would be in GPC 0, the next n in GPC 1, and so on. We uncover that SM ID to GPC mappings are actually arbitrary, and configured by the NVIDIA driver in a striping-like configuration by default. Knowing these mappings is critical to align partitions on GPC boundaries, and so we include a fixed implementation of the API for determining SM-to-GPC mappings in `nvsplit`. We repeated the "Partitioning Strategy" experiments from the `libsmctrl` paper [4] on `nvsplit`, but found no significant differences. We recommend the SE-packed strategy from prior work [4].

5.6 Limitations

`nvsplit` is still prone to Pitfalls 1–3 of MPS, does not support co-running more than 15 tasks, and is only compatible with CUDA-using tasks.

However, it is portable to any recent discrete NVIDIA GPU, does not require task modifications, preserves address space isolation, and is available as open-source software.[21]

6 Evaluation

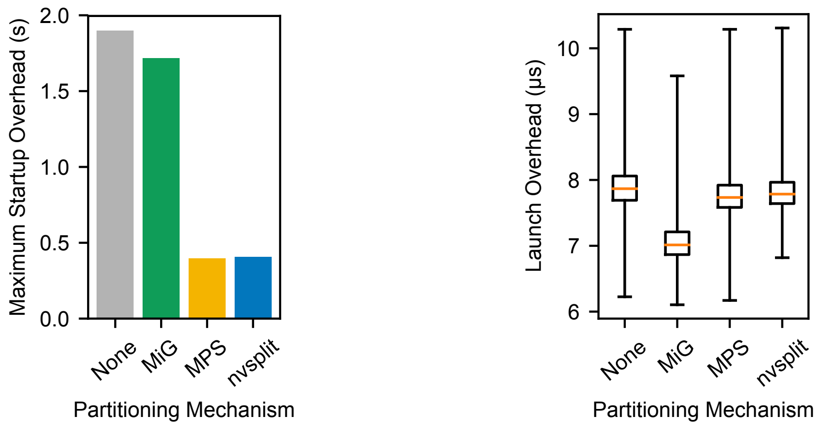
We evaluate `nvsplit` against MiG, MPS, and (where applicable) against no partitioning. We compare overhead impact, strength of partition enforcement, and granularity of partitioning. Our other target properties—portability, logical isolation, transparency, and dynamic reconfigurability—are largely binary properties, and have already been discussed.

Experiments in this section were run on an NVIDIA A100 40GB GPU running CUDA 12.4 on a 6-core, Linux 5.4 machine with the NVIDIA 550.78 GPU driver. At time of writing, the A100 is the only GPU that supports all of MiG, MPS, and `nvsplit`. We use the SE-packed partitioning strategy [4, 33] to assign SMs to `nvsplit` partitions. As MiG cannot perform a 50/50 split of the A100, we use a 57/43 split for all mechanisms.[22]

6.1 Evaluating Partitioning Overheads

We measure startup overhead and launch overheads for all mechanisms via benchmarks run in the 57% partition. Startup overhead is the time from `exec()` to first kernel running on the GPU of a minimal program, and launch overhead is the time from `cuLaunch()` to the

[21] Available at `http://rtsrv.cs.unc.edu/cgit/cgit.cgi/libsmctrl.git` and within our artifact.
[22] MiG can only partition on GPC boundaries, and there are seven GPCs in the A100, meaning that a 57/43 split is as close as MiG can get to 50/50.



(a) Max time to start a CUDA-using task and launch the first kernel (of 100 samples).

(b) 0, 25, 50, 75, and 99th percentile time to launch a CUDA kernel (1 million samples).

Figure 11 Overheads of each partitioning mechanism.

kernel running on the GPU. We added these benchmarks to the `gpu-microbench` suite [5],[23] and ran each experiment once to prime caches before gathering the data displayed in Fig. 11.

▶ **Observation 1.** *No partitioning mechanism adds startup or launch overheads.*

As shown in Fig. 11a and Fig. 11b, no partitioning mechanism worsens observed worst- or average-case startup or launch times.[24]

▶ **Observation 2.** *Only MiG reduces launch overheads.*

Uniquely, MiG statically binds a hardware scheduling pipeline to each partition's TPCs [9] such that only a subset of TPCs have to be set up, and considered as part of a kernel launch. We suspect this is what lowers launch and startup overheads with MiG.

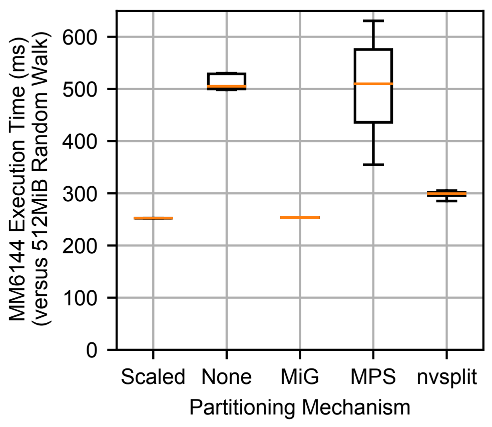
▶ **Observation 3.** *MPS and `nvsplit` have the lowest startup overheads.*

With MPS, newly launched tasks (MPS clients) only initialize channels and subcontexts, with the MPS server providing the parent context and TSG. This appears to significantly reduce startup overheads for MPS and MPS-based `nvsplit`, more than compensating for the added client-server communication cost of MPS.

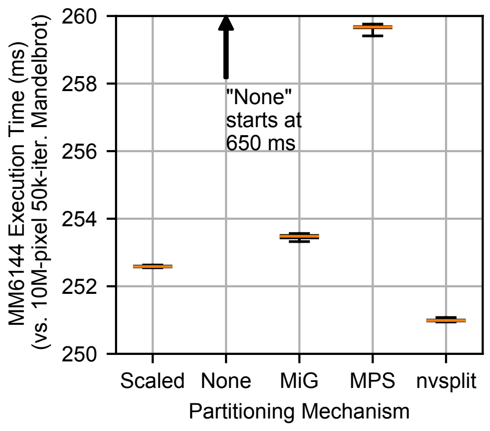
6.2 Evaluating Partition Enforcement

To evaluate how strongly partitions are enforced, we measured the execution time of a 6144×6144 matrix multiply (yielding 36×2^{10} blocks of 1024 threads) ("MM6144") in a 57% partition while interfering tasks executed on the remainder of the GPU. We used the `mandelbrot` and `random_walk` tasks from the `cuda_scheduling_examiner` toolkit as compute- and memory-heavy interfering tasks respectively. Interfering tasks executed continuously, out of sync with each other and the matrix multiply. We carefully configured this experiment to sidestep the

[23] Available at `http://rtsrv.cs.unc.edu/cgit/cgit.cgi/gpu-microbench.git/` and within our artifact.
[24] Fig. 11b omits 100th percentile (max) times, as use of Linux's `isolcpus`, `sched_yield()` and `nohz=full` options were insufficient to eliminate all noise, potentially due to SMIs [21] or interrupts.



(a) Partition enforcement versus memory-heavy competition (random walks of GPU DRAM).



(b) Partition enforcement versus compute-heavy competition (Mandelbrot set generation). Axis cropped.

Figure 12 Partition enforcement vs. a memory-heavy or compute-heavy competitor. Specifically, 0, 25, 50, 75, and 100th percentile time to execute many large matrix multiplies in a 57% partition for each partitioning mechanism while competing tasks run in the remaining 43% partition (100 samples each).

pitfalls of MPS discussed in Sec. 4.4, focusing exclusively on how well a partition is enforced in an otherwise-ideal scenario. Good partition enforcement means that the execution time distribution of our MM6144 task does not shift in the presence of interfering tasks.

The results against memory- and compute-heavy interference are shown in Fig. 12a and Fig. 12b respectively. Each plot includes a baseline "Scaled" time for comparison—this is *not* a measured value, but is a scaled-up value derived from the execution time of the benchmark without any partitioning or interference. For example, if MM6144 took 244 ms when run alone on the GPU, the "Scaled" value for a 57% partition would be $144/0.57 = 253$ ms. This value is the minimum execution time possible for the MM6144 task in a 57% partition before accounting for sympathetic caching effects.

We specifically choose this matrix multiply task as it was most sensitive to interference among several other synthetic tasks we tested, and it could be more-precisely timed than a full neural network. Since we use the same partitioning mechanism as `libsmctrl`, the enforcement results previously shown for real tasks [4] continue to apply.

▶ **Observation 4.** *MPS's partitioning mechanism can worsen predictability and efficiency, even when used correctly.*

When competing work is memory-bound, as in Fig. 12a, MPS does nothing to prevent memory contention, only limiting the compute available for MM6144. This results in average (line in box) and worst-case (top of whisker) execution times for our MM6144 task *larger than with no partitioning at all.*

▶ **Observation 5.** *`nvsplit` can provide partition enforcement approaching MIG without requiring hardware modifications.*

For memory-bound competing work (Fig. 12a), MiG beats `nvsplit`, likely because MiG also partitions DRAM. Interestingly, even though `nvsplit` includes *no explicit memory partitioning*, its implicit partitioning of the L0, L1, and TLB caches by aligning partitions to GPCs appears to be enough to beat MPS and approach MiG's performance.

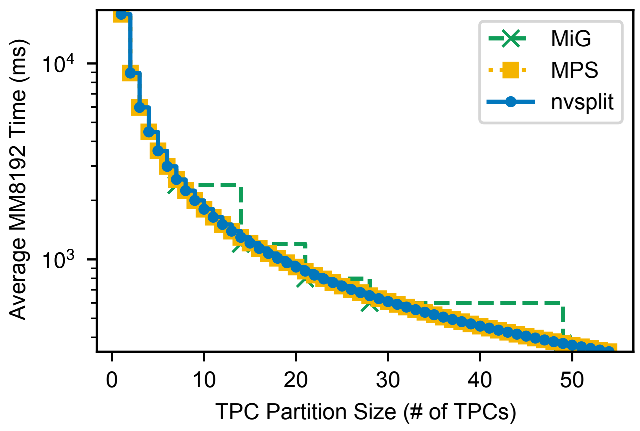


Figure 13 Partitioning granularity comparison. Specifically, mean time to execute a matrix multiply at each partition size, for each partitioning mechanism (10 samples).

Surprisingly, for compute-bound competing work (Fig. 12b), `nvsplit` *beats MiG* (and every other approach) on average- and worst-case execution times—without the hardware modifications of MiG. Upon further investigation, we found that MiG compromises compute speed to serve other design goals. This unexpected behavior is *not documented*, and causes MiG to fail in other ways, as we will explore under Obs. 7.

In all, `nvsplit` enforces partitioning much better than MPS, and even better than MiG in some cases—all without requiring task, driver, or hardware modification.

6.3 Evaluating Partition Granularity

To evaluate partitioning granularity, we recorded the total execution time of a 8192×8192 matrix multiply (yielding 64×2^{10} blocks of 1024 threads) ("MM8192") at every possible partition size for each partitioning method and plot the results as points in Fig. 13. The lines in Fig. 13 represent the closest available configuration which allocates at most the specified number of TPCs. For example, there is no MiG configuration for 10 TPCs, so we plot time for the closest available allocation of no more than 10 TPCs—7 TPCs.

▶ **Observation 6.** *`nvsplit` is the most granular partitioning mechanism.*

Both MPS and `nvsplit` can specify partition sizes down to the per-TPC level, visible as the different MM8196 execution times for each setting in Fig. 13. However, `nvsplit` can assign specific TPCs to a partition, in contrast to MPS's generic percentage value. For this 54-TPC GPU, that means `nvsplit` supports a total of 2^{54} different partition settings per-task, MPS supports 54 per-task, and MiG only supports 5 per-task.

▶ **Observation 7.** *MiG cannot access 9% of the A100 GPU cores (5 TPCs).*

Visible in Fig. 13, the largest-available MiG partition contains only 49 TPCs, whereas MPS or `nvsplit` are able to access up to 54 TPCs. Upon further investigation, we find that no configuration of MiG partitions on the A100 can access more than 49 TPCs total. Every partition size is an even multiple of 7 TPCs, and 7 does not divide 54 evenly, wasting the remainder—5 TPCs. This surprising issue is *not documented*, and means that enabling MiG immediately and inherently disables 9% of the A100 GPU. Concerningly, we found that this issue is even worse on NVIDIA's newer GPUs. On the H100 GPU (SXM-80GB version tested), we found a loss of 6–15% (depending on the MiG partition size chosen).[25]

7 Conclusion

In this work, we developed a system-level spatial partitioning mechanism for NVIDIA GPU compute cores, `nvsplit`. Our mechanism allows for GPU-using tasks to run *both* efficiently and time-predictably by running concurrently on disjoint sets of GPU cores.

We demonstrated that our mechanism is portable, transparent, and low-overhead, and has the ability to provide granular, dynamic, logically-isolated, and hardware-enforced partitions. As part of this work, we exposed critical pitfalls of NVIDIA's MPS-based partitioning mechanism, and revealed previously-undocumented capacity loss issues inherent to NVIDIA MiG. In future work, we aim to extend `nvsplit` to support partitioning non-CUDA workloads, and to build GPU schedulers on `nvsplit` that efficiently and predictably schedule tasks across both time and space.

References

1 Karim M Abdalla, Lacky V Shah, Jerome F Duluk Jr, Timothy John Purcell, Tanmoy Mandal, and Gentaro Hirota. Scheduling and execution of compute tasks, Jun 2015. U.S. Patent 9,069,609.

2 Tanya Amert, Nathan Otterness, Ming Yang, James H. Anderson, and F. Donelson Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *Proceedings of the 38th IEEE Real-Time Systems Symposium*, pages 104–115, Dec 2017. `doi:10.1109/RTSS.2017.00017`.

3 Tanya Amert, Zelin Tong, Sergey Voronov, Joshua Bakita, F Donelson Smith, and James H Anderson. TimeWall: Enabling time partitioning for real-time multicore+accelerator platforms. In *Proceedings of the 42nd IEEE Real-Time Systems Symposium*, pages 455–468, Dec 2021. `doi:10.1109/RTSS52674.2021.00048`.

4 Joshua Bakita and James H Anderson. Hardware compute partitioning on NVIDIA GPUs. In *Proceedings of the 29th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 54–66, May 2023. `doi:10.1109/RTAS58335.2023.00012`.

5 Joshua Bakita and James H Anderson. Demystifying NVIDIA GPU internals to enable reliable GPU management. In *Proceedings of the 30th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 294–305, May 2024. `doi:10.1109/RTAS61025.2024.00031`.

6 Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for GPU with preemption support. In *Proceedings of the 39th IEEE Real-Time Systems Symposium*, pages 119–130, Dec 2018. `doi:10.1109/RTSS.2018.00021`.

7 Nicola Capodieci, Roberto Cavicchioli, Ignacio Sañudo Olmedo, Marco Solieri, and Marko Bertogna. Contending memory in heterogeneous SoCs: Evolution in NVIDIA Tegra embedded platforms. In *Proceedings of the 26th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, Aug 2020. `doi:10.1109/RTCSA50079.2020.9203722`.

8 Jerome F Duluk Jr, Luke Durant, Ramon Matas Navarro, Alan Menezes, Jeffrey Tuckey, Gentaro Hirota, and Brian Pharris. Dynamic partitioning of execution resources, Apr 2022. U.S. Patent 11,307,903.

9 Jerome F Duluk Jr, Gregory Scott Palmer, Jonathon Stuart Ramsey Evans, Shailendra Singh, Samuel H Duncan, Wishwesh Anil Gandhi, Lacky V Shah, Eric Rock, Feiqi Su, James Leroy

[25] We suspect the capacity loss stems from a decision to make unit-size MiG slices appear identical, despite differences in the underlying GPCs. Due to floorsweeping, some GPCs will have more working TPCs than others—those TPCs must be disabled to emulate identical GPCs when using MiG.

Deming, et al. Techniques for configuring a processor to function as multiple, separate processors, Feb 2022. U.S. Patent 11,249,905.

10 Samuel H Duncan, Lacky V Shah, Sean J Treichler, Daniel Elliot Wexler, Jerome F Duluk Jr, Philip Browning Johnson, and Jonathon Stuart Ramsay Evans. Concurrent execution of independent streams in multi-channel time slice groups, Sep 2016. U.S. Patent 9,442,759.

11 Glenn A Elliott. *Real-time scheduling for GPUs with applications in advanced automotive systems.* PhD thesis, The University of North Carolina at Chapel Hill, 2015. `doi:10.17615/gk2m-0503`.

12 Glenn A Elliott, Bryan C Ward, and James H Anderson. GPUSync: A framework for real-time GPU management. In *Proceedings of the 34th Real-Time Systems Symposium*, pages 33–44, Dec 2013. `doi:10.1109/RTSS.2013.12`.

13 Kshitij Gupta, Jeff A. Stuart, and John D. Owens. A study of persistent threads style GPU programming for GPGPU workloads. In *Proceedings of the 2012 IEEE Innovative Parallel Computing Conference*, pages 1–14, May 2012. `doi:10.1109/InPar.2012.6339596`.

14 Ari B Hayes, Fei Hua, Jin Huang, Yanhao Chen, and Eddy Z Zhang. Decoding CUDA binary. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 229–241, Feb 2019. `doi:10.1109/CGO.2019.8661186`.

15 Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan 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*, pages 29–41, Apr 2019. `doi:10.1109/RTAS.2019.00011`.

16 Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the NVIDIA Turing T4 GPU via microbenchmarking, Mar 2019. `doi:10.48550/arXiv.1903.07486`.

17 Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking, Apr 2018. `doi:10.48550/arXiv.1804.06826`.

18 Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25:1097–1105, Dec 2012.

19 Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, pages 213–224, Jun 1997. `doi:10.1109/RTTAS.1997.601360`.

20 Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the 21st International Conference on Parallel Architecture and Compilation Techniques*, pages 367–376, Sept 2012. `doi:10.1145/2370816.2370869`.

21 Sizhe Liu, Rohan Wagle, James H Anderson, Ming Yang, Chi Zhang, and Yunhua Li. Autonomy today: Many delay-prone black boxes. In *Proceedings of the 36th Euromicro Conference on Real-Time Systems*, pages 12:1–12:27, July 2024. `doi:10.4230/LIPIcs.ECRTS.2024.12`.

22 Albert Meixner. System and method for launching data parallel and task parallel application threads and graphics processing unit incorporating the same, Mar 2016. U.S. Patent 9,286,114 B2.

23 Mesa Project Authors. The Mesa 3D graphics library, 2022. URL: `https://www.mesa3d.org/`.

24 Nouveau Project Authors. Nouveau: Accelerated open source driver for nVidia cards, 2022. URL: `https://nouveau.freedesktop.org/`.

25 NVIDIA. CUDA C++ programming guide, 2022. Version PG-02829-001_v11.8.

26 NVIDIA. Linux open GPU kernel module source, 2024. URL: `https://github.com/NVIDIA/open-gpu-kernel-modules`.

27 NVIDIA. Multi-process service, 2024. Version R555.

28 NVIDIA. nvgpu git repository, 2024. URL: `git://nv-tegra.nvidia.com/linux-nvgpu.git`.

29 NVIDIA. NVIDIA multi-instance GPU user guide, 2024. Version RN-08625-v2.0.

30 NVIDIA. Open GPU documentation, 2024. URL: `https://github.com/NVIDIA/open-gpu-doc`.

31 NVIDIA. Parallel thread execution ISA, 2024. Version 8.5.

32 Ignacio Sañudo Olmedo, Nicola Capodieci, Jorge Luis Martinez, Andrea Marongiu, and Marko 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*, pages 213–225, Apr 2020. `doi:10.1109/RTAS48715.2020.000-5`.

33 Nathan Otterness and James 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*, pages 24–34, Apr 2021. `doi:10.1145/3453417.3453432`.

34 Nathan Otterness, Ming Yang, Tanya Amert, James Anderson, and F Donelson 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*, pages 47–52, Jul 2017.

35 Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H. Anderson, F. Donelson Smith, Alex Berg, and Shige Wang. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 353–364, Apr 2017. `doi:10.1109/RTAS.2017.3`.

36 Timothy John Purcell, Lacky V Shah, and Jerome F Duluk Jr. Scheduling and management of compute tasks with different execution priority levels. U.S. Patent Application 13/236,473.

37 Sujan Kumar Saha, Yecheng Xiang, and Hyoseung Kim. STGM: Spatio-temporal GPU management for real-time tasks. In *Proceedings of the 25th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–6, Aug 2019. `doi:10.1109/RTCSA.2019.8864564`.

38 Roy Spliet and Robert 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*, pages 43–48, Jul 2018.

39 Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey 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*, ICS, pages 119–130, Jun 2015. `doi:10.1145/2751205.2751213`.

40 Tyler Yandrofski, Leo Chen, Nathan Otterness, James H Anderson, and F Donelson Smith. Making powerful enemies on NVIDIA GPUs. In *Proceedings of the 43rd IEEE Real-Time Systems Symposium*, pages 383–395, dec 2022. `doi:10.1109/RTSS55097.2022.00040`.

41 Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H Anderson, and F Donelson 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*, pages 20:1–20:21, Jul 2018. `doi:10.4230/LIPIcs.ECRTS.2018.20`.

42 Ming Yang, Shige Wang, Joshua Bakita, Thanh Vu, F Donelson Smith, James H Anderson, and Jan-Michael Frahm. Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge. In *Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 305–317, Apr 2019. `doi:10.1109/RTAS.2019.00033`.

43 Chao Yu, Yuebin Bai, Hailong Yang, Kun Cheng, Yuhao Gu, Zhongzhi Luan, and Depei Qian. SMGuard: A flexible and fine-grained resource management framework for GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 29(12):2849–2862, Jun 2018. `doi:10.1109/TPDS.2018.2848621`.

44 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 155–166, Apr 2014. `doi:10.1109/RTAS.2014.6925999`.

45 Husheng Zhou, Soroush Bateni, and Cong Liu. S^3DNN: Supervised streaming and scheduling for GPU-accelerated real-time DNN workloads. In *Proceedings of the 24th IEEE Real-Time* *and Embedded Technology and Applications Symposium*, pages 190–201, Apr 2018. doi:10.1109/RTAS.2018.00028.

46 An Zou, Jing Li, Christopher D Gill, and Xuan Zhang. RTGPU: Real-time GPU scheduling of hard deadline parallel tasks with fine-grain utilization. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1450–1465, May 2023. doi:10.1109/TPDS.2023.3235439.