Unveiling the Scheduling Mechanisms of NVIDIA RTX 4090 GPU: Insights for Enhancing Covert Attack

Yiming Li
yl5436@columbia.edu
Columbia University

Ximing Chen
xc2725@columbia.edu
Columbia University

ABSTRACT

This paper investigates the scheduling policies and block-to-SM allocation strategies of the NVIDIA RTX 4090 GPU. Through targeted experiments, we uncover the interplay of factors such as kernel queueing, resource availability, and stream priorities in GPU scheduling. Our findings reveal high prediction accuracy for thread-dominated configurations and significant unpredictability in resource-intensive scenarios. We also identify a scheduling pattern favoring even-indexed SMs over odd-indexed ones. Additionally, we explore worst-case GPU scheduling, demonstrating how improper thread layout and SM partitioning can exacerbate performance degradation. These insights have implications for optimizing co-location strategies in covert channels on GPGPUs. We propose techniques to enhance the reliability and effectiveness of covert communication and contribute to a deeper understanding of GPU scheduling mechanisms, empowering developers and security professionals to build more robust and secure GPU-based systems.

KEYWORDS

NVIDIA RTX 4090, GPU scheduling, block-to-SM allocation, covert channels, GPGPU security, performance optimization

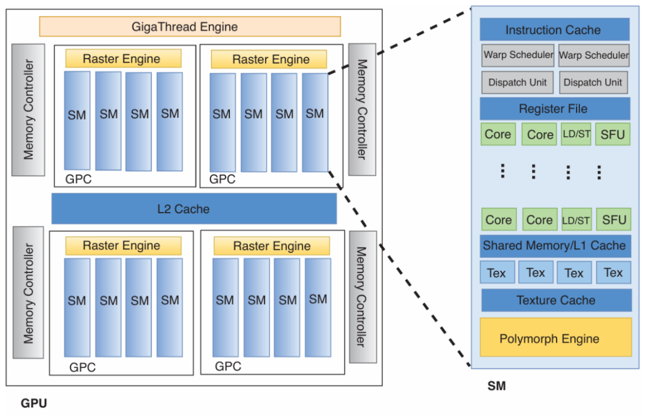

Figure 1: Architecture of NVIDIA GPGPUs.

1 INTRODUCTION

Graphics Processing Units (GPUs) have evolved from specialized components in gaming systems to central processing units in diverse computational fields such as artificial intelligence, high performance computing, and data analytics. This transformation has been largely driven by advancements in GPU programming models, notably NVIDIA's Compute Unified Device Architecture (CUDA)[14], which facilitates the exploitation of GPUs' parallel processing capabilities. The NVIDIA RTX 4090 GPU is at the forefront of this technology, boasting unprecedented computational power and sophisticated architecture designed to manage increasingly complex computations across an expanded array of Stream Multiprocessors (SMs).

A critical aspect of leveraging GPU technology effectively, particularly in security-sensitive environments, involves understanding the scheduling strategies that govern how computational tasks are allocated and executed across the GPU's SMs. These strategies determine not only performance metrics such as task latency and throughput but also the security posture of the GPU, especially against threats like covert channels. Covert channels [5, 12] on GPUs, particularly those that exploit the scheduler to achieve malicious co-location, are emerging as significant security concerns. By understanding the GPU's scheduling policies, it is possible to identify and mitigate scenarios where an attacker might manipulate scheduling behaviors to synchronize malicious tasks with target processes, thereby extracting sensitive information.

However, the mechanisms controlling task allocation within GPUs like the RTX 4090 are intricate and not well-documented, posing challenges for both developers aiming to optimize performance and security professionals needing to fortify systems against such vulnerabilities.

This paper aims to demystify the RTX 4090 GPU's scheduler policy and block-to-SM scheduling mechanisms. Through detailed black-box testing and analysis of publicly available documentation, this study reconstructs the RTX 4090's scheduling behavior, providing critical insights into how tasks are managed and distributed across the GPU's architecture. By systematically mapping out the scheduling and execution of tasks on the GPU, our research not only elucidates the scheduling mechanisms of the NVIDIA RTX 4090 GPU but also demonstrates how a thorough understanding of these processes can contribute to designing more secure systems. This knowledge is crucial for developing strategies to mitigate the risk of covert-channel attacks by preventing unwanted process co-location on GPUs, thereby enhancing the security of sensitive data processed by these powerful computing devices.

Furthermore, we explore worst-case GPU scheduling scenarios, illustrating how improper thread layout and SM partitioning can lead to significant performance degradation. By examining these worst-case conditions, we provide a comprehensive understanding of the potential pitfalls in GPU scheduling, guiding developers and security professionals to avoid these scenarios.

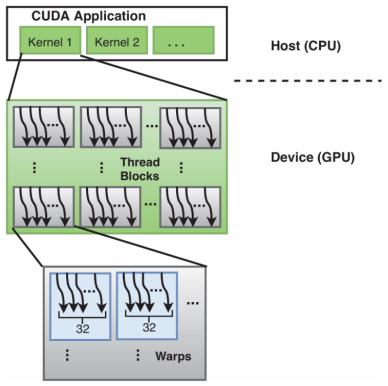

Figure 2: CUDA Basic Execution and Programming Model.

2 BACKGROUND

2.1 NVIDIA GPGPU Hardware Architecture

NVIDIA's General Purpose Graphics Processing Units (GPGPUs), based on the Single Instruction, Multiple Threads (SIMT) architecture, are engineered to handle general computing tasks alongside traditional graphics processing. The architectural lineage that includes the Volta, Turing, and Ampere series demonstrates a consistent design philosophy, as depicted in Fig. 2 which outlines the structure of NVIDIA GPGPUs. Each GPGPU interfaces with memory via controllers and connects to other processors through NVLink or Peripheral Component Interconnect Express (PCIe) interfaces. At its core, the GPGPU comprises multiple Graphic Processing Clusters (GPCs) and Texture Processing Clusters (TPCs), with each GPC housing several TPCs that in turn contain two SMs. These SMs are subdivided into four processing blocks, each equipped with its own pipeline for executing instructions on warps—groups of 32 threads. These blocks possess dedicated resources such as register files, CUDA cores, tensor cores, and load/store units, and share L1 cache and shared memory.

The hierarchical memory design is crucial, featuring a unified L2 cache across the GPGPU and segregated caches at various levels—L1 data cache/shared memory, L1.5 constant cache/L1 instruction cache, and L1 constant cache are exclusive to each SM. Register files and L0 instruction caches are localized within each SM processing block. Notably, the configuration of the L1 data cache and shared memory, which share the same static random access memory (SRAM), can be adjusted to optimize performance based on specific application needs.

2.2 CUDA Execution and Programming Model

The CUDA execution model, integral to NVIDIA's approach, involves three main operations: (i) copying data and program code from the host CPU to the GPU's memory; (ii) executing computational kernels on the GPU; (iii) transferring the resulting data back to the host CPU's memory. These operations are encapsulated as commands within the GPU's scheduling hardware, allowing for efficient task management. The execution follows a SIMT paradigm, where identical instructions are dispatched to multiple threads, each working on distinct data pieces. These threads are organized into warps, the fundamental schedulable units within the GPU.

Exploiting multiple levels of parallelism, CUDA allows kernels to be executed as grids of thread blocks, or Cooperative Thread Arrays (CTAs), which enable simultaneous warp execution. Thread blocks can be structured in 1D, 2D, or 3D grids, and each is managed as a warp multiple regardless of the actual number of threads specified. The warp schedulers in each SM dynamically allocate threads to the hardware resources, optimizing resource utilization.

2.3 Advanced CUDA Execution Model and Programming Constructs

To transcend the basic model's limitations, CUDA employs advanced constructs such as CUDA streams for interleaving compute and data operations. A CUDA stream, essentially a command queue for the GPU, is created and specified for each operation. Commands within the same stream execute in a FIFO sequence and do not overlap, while commands in different streams can execute concurrently if possible. Stream operations can be synchronous, where the CPU waits for GPU task completion, or asynchronous, allowing concurrent CPU computations.

Additionally, CUDA leverages pinned host memory and device shared memory to enhance performance. For example, the cudaMallocHost function allocates pinned memory, which bypasses paging and improves data transfer bandwidth, allowing overlap of compute and copy operations across different streams. Commands are enqueued using cudaMemcpyAsync for data transfers, and the kernel launch syntax is augmented to specify dynamic shared memory allocations.

These programming constructs are designed to optimize data handling and computational efficiency, ensuring that memory accesses—especially in device code—are coalesced to maximize memory bandwidth utilization. This setup significantly impacts the performance of memory-intensive operations on the GPGPU.

2.4 Covert Channel Attacks on GPUs: The Significance of CUDA Workload Scheduling

Covert channel attacks on GPUs [5, 12, 13] pose significant security challenges by exploiting unintended communication paths within GPU architectures. These types of attacks are particularly concerning in environments where GPUs, utilizing the CUDA programming model, process sensitive data.

The success of covert channel attacks largely depends on the ability of a malicious process (spy) to achieve co-location with a legitimate target process (trojan) on the same GPU. This co-location is crucial as it allows the spy to subtly manipulate and measure the GPU's resource utilization, thereby extracting sensitive information through variations in system performance and resource availability.

One key aspect that facilitates these attacks is the concurrent scheduling of CUDA kernels, which can inadvertently provide opportunities for covert communication. When CUDA workloads from different processes run simultaneously, they share GPU resources, potentially allowing a spy process to synchronize its execution with a trojan process. By monitoring memory allocations and utilizing GPU performance counters, a spy can gather extensive data about the computational behaviors and resource usage of co-located tasks. Additionally, precise analysis of execution timing can reveal operational patterns and anomalies that signal the activities of other processes on the GPU.

Understanding the nuances of how CUDA tasks are scheduled and executed on NVIDIA GPUs is therefore essential. This knowledge helps in recognizing the risks associated with covert channel attacks and forms the basis for developing robust security measures to safeguard sensitive computations against potential security breaches.

3 RELATED WORK

The understanding of NVIDIA GPGPU architectures, including memory hierarchy, computational units, and thread block scheduling, has evolved significantly due to focused research efforts. A body of literature has dissected the internals of NVIDIA GPGPUs [1, 2, 8, 9, 11, 16, 17], employing specialized methodologies to deeply analyze various features of each architecture. These studies have substantially enhanced the comprehension of GPGPU architectures, aiding in both performance modeling and optimization.

Particularly, the domain of thread block scheduling has seen notable attention. The "leftover policy" [4, 18] prioritizes kernel resource allocation, dedicating resources to a kernel until its scheduling is complete. Further studies have investigated the overall scheduling hierarchy, particularly thread block-to-SM scheduling constrained by the maximum threads per SM [15]. The "most room policy" [7] proposes thread block placement policies that prioritize scheduling thread blocks onto the SM that can accommodate the highest number. However, these investigations often lack a comprehensive analysis of thread block-to-SM scheduling mechanisms and do not offer a precise, validated scheduling algorithm that addresses all scenarios comprehensively. These studies, while advancing our understanding of the relationship between thread block scheduling and SM resource limitations, often do not provide a thorough examination of SM resource monitoring and allocation.

Beyond the internal scheduling mechanisms, the management of real-time GPU operations has also been explored. Historically, due to the black-box nature of GPUs, programs utilizing GPUs have had to lock an entire GPU or individual execution engines while performing operations [6, 10]. This approach typically disallows concurrent kernel execution to prevent adverse interference, potentially leading to wasted GPU processing cycles, especially on less-capable integrated GPUs.

4 SCHEDULING POLICY

This section describes the GPU scheduling policy utilized in our experiments, with graphical representations to clarify the deployment of blocks and operations within kernels.

In the notation used, the j^{th} block of kernel K_k is denoted as $K_k : j$, and b_i denotes the i^{th} block in the grid. Furthermore, C_ki and C_ko signify the input and output copy operations for kernel K_k, respectively.

4.1 General Scheduling Mechanisms

Our comprehensive analysis of NVIDIA GPU scheduling has elucidated specific mechanisms that dictate how computational tasks

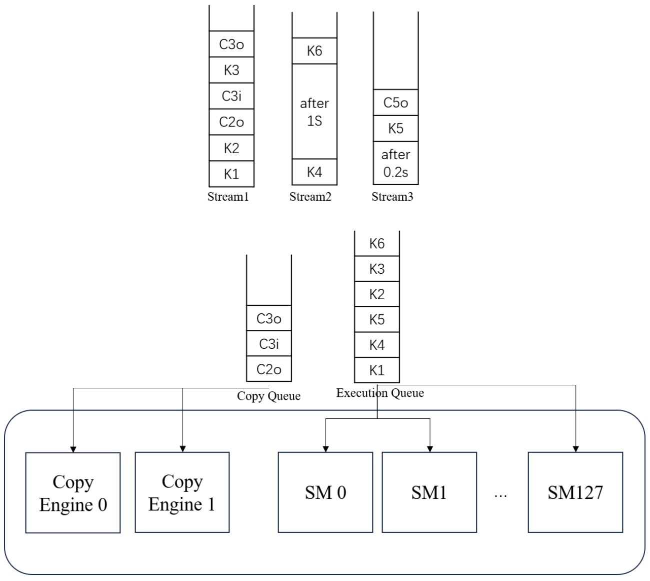

Figure 3: Stream Settings for the General Scheduling Experiment.

Table 1: Kernel Launch Details for General Scheduling Experiment

Kernel	Stream	Blocks	Threads/Block	Shared Mem/Block
K1	S1	128	1024	0
K2	S1	1	512	0
K3	S1	1	1024	0
K4	S2	336	256	32 KB
K5	S3	144	256	32 KB
K6	S2	1	384	0

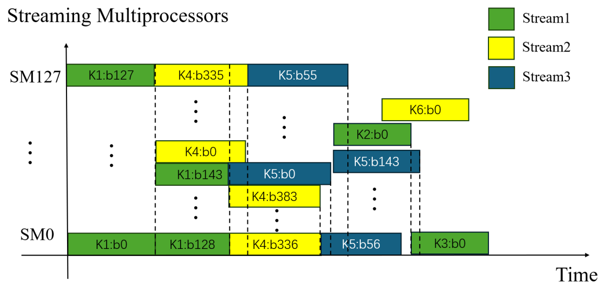

Figure 4: Timeline for the General Scheduling Experiment.

are managed within the GPU architecture. You can see the stream setting3, kernel configuration 1 and the timeline of this experiment4 above.

These mechanisms are foundational to ensuring efficient task execution and optimal resource utilization:

- **Kernel Queueing and Execution:** Kernels enter the Execution Engine (EE) queue as they reach the front of their

stream queue, adhering to a FIFO sequence. They remain in the EE queue until all blocks complete, ensuring orderly execution and resource release.
- **Copy Operation Management:** Copy operations are added to the Copy Engine (CE) queue upon reaching the stream queue's head, synchronizing data transfers with corresponding kernel executions for smooth processing.

In our experiment, two threads, thread_0 and thread_1, execute CUDA operations within a shared address space. Specifically, thread_0 utilizes a single stream, $S1$, while thread_1 operates two streams, $S2$ and $S3$. The system configuration includes 128 SMs and a single CE. The experiment chronicles the time each kernel, or its associated input copy operation, is issued, with output copy operations immediately following the completion of their respective kernels.

Initially, thread_0 issues kernels $K1$, $K2$, and $K3$, along with copy operations $C2o$, $C3i$, and $C3o$ to stream $S1$. Concurrently, thread_1 issues kernels $K4$ and $K5$ to streams $S2$ and $S3$, respectively, and a copy operation $C5o$ to stream $S3$. Operations within streams $S1$ and $S3$ are enqueued following the order of CUDA command execution. At the forefront of their respective stream queues, kernels $K1$, $K4$, and $K5$ are transferred to the EE queue. $K1$ is initially dispatched to the GPU, occupying all available SMs due to its block count (144 blocks), with each SM processing a single block of $K1$.

Subsequently, as more resources become available, the remaining blocks of $K1$ and all blocks of $K4$ are assigned to the GPU. This results in both $K1$ and $K4$ being removed from the EE queue, though they remain within their original stream queues. Despite the availability of resources on each SM that would allow for concurrent execution with blocks from $K4$, this kernel is not at the head of the EE queue and thus its blocks are not eligible for assignment at this time.

Following the completion of $K4$ and $K5$, their respective streams, $S1$ and $S3$, advance such that the copy operations $C2o$ and $C5o$ are now at the head. These are enqueued on the CE queue. $C5o$ is immediately assigned to the CE and is promptly dequeued upon assignment. Since the CE can process only one operation at a time, $C2o$ remains pending in the CE queue until the completion of $C5o$. Since $K2$ and $K3$ are both in stream $S1$, they can only commence execution after their predecessors in the same stream have completed. Conversely, as $K6$ is in a different stream, $S3$, it can initiate execution independently and does not need to wait for the completion of operations in stream $S5$.

4.2 Resource-Based Scheduling Criteria

The assignment of blocks to SMs is intricately tied to the availability of critical resources, which is governed by a set of clear rules:

- **Resource Availability for Block Assignment:** Blocks are assigned to an SM only if sufficient shared-memory resources are available, ensuring each block can execute effectively without contention.
- **Determining SM Occupancy:** SM occupancy is determined by the highest demand from three resources: thread, shared memory, and register occupancy. This approach prioritizes blocks based on their resource intensity within existing constraints.
- **Shared Memory Prioritization:** Blocks requesting over 3KB of shared memory receive scheduling priority. Below this threshold, thread count primarily influences scheduling, balancing memory use and processing efficiency.

During our experiments, we observed specific patterns in the allocation and execution of kernels based on available resources within NVIDIA's GPU architecture. For instance, when Kernel K1 is at the forefront of the EE queue, it becomes eligible for assignment. However, despite requiring 1024 threads per block, only 128 blocks of K1 are initially assigned across the SMs. This partial assignment results from each SM's limitation to allocate a maximum of 1536 threads at any given time. Consequently, only one block of K1 can be scheduled per SM, leaving 16 blocks pending until resources free up after the initial 128 blocks complete execution.

Following this, Kernel K4, which is next in the EE queue, requires fewer resources (four blocks of 512 threads each), allowing up to three of its blocks to concurrently fit on one SM. This results in a total of 336 blocks of K4 being scheduled alongside the remaining 16 blocks of K1.

Memory Constraints and Scheduling. The RTX 4090 allows up to 100KB of shared memory per SM. Given that each block of K4 requires 30KB and K5 requires 40KB of shared memory, at most three blocks of K4 and two blocks of K5 can be concurrently assigned to an SM. This sharing of resources is carefully managed to optimize the utilization across the 128 SMs available.

Interactions Between Different Streams. As the execution progresses, after K2 and K5 complete, their associated copy operations, C2*o* and C5*o*, advance to the head of their respective streams. These operations are then queued in the CE, with C5*o* being immediately assigned and executed due to its higher priority in the queue. This sequencing ensures that copy operations do not bottleneck computational progress.

Theoretical Model for Resource Utilization. The CUDA Occupancy Calculator provides a theoretical framework for understanding resource utilization on NVIDIA GPUs. It considers various factors including the number of warps per thread block, shared memory per block, and the number of registers per warp. The utilization formula is given by:

$$\text{GPU Occupancy} = \max\left(\left\lfloor \frac{\text{Max\_Threads\_Per\_SM}}{\text{Total\_Warps} \cdot 32}, \frac{\text{Max\_Shared\_Memory}}{(\text{Used\_SHM} + \delta) \cdot \text{Num\_Blocks}}, \frac{\text{RegFile\_Capacity}}{\text{Regs\_Per\_Warp} \cdot \text{Total\_Warps}} \right\rfloor\right)$$

Here, Max_Threads_Per_SM denotes the maximum number of threads that an SM can support at one time, ensuring that the SM does not exceed its threading capacity. Total_Warps, calculated as $\left\lceil \frac{\text{Total\_Threads}}{32} \right\rceil$, represents the number of warps required by the kernel, with each warp comprising up to 32 threads, thus encapsulating the threading needs of the kernel. Total_Threads is the sum of all threads launched by the kernel. The Max_Shared_Memory indicates the total shared memory per SM, while Used_SHM is the amount of this memory already claimed by active blocks, and δ is a small constant to cover

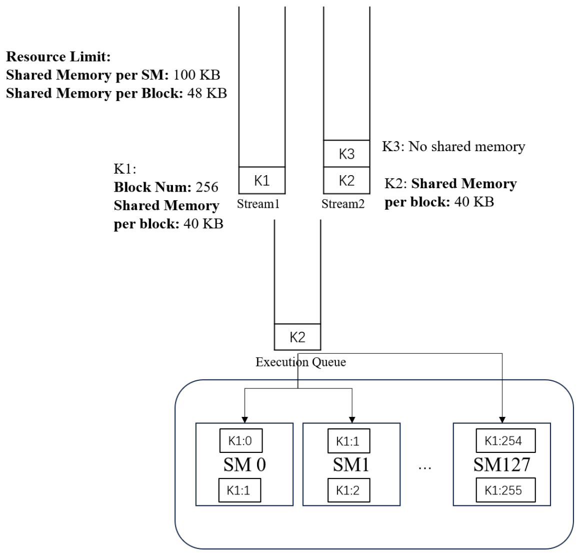

Figure 5: Stream Settings for the Null Stream Scheduling Experiment.

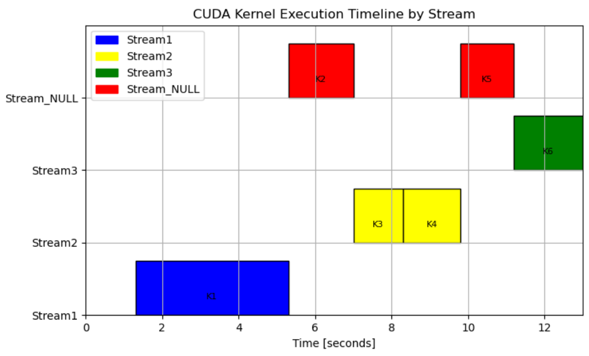

Figure 6: Timeline for the Null Stream Scheduling Experiment.

system overhead or a minimum reserve. Num_Blocks is the count of thread blocks active on the SM. RegFile_Capacity reflects the total register file size available per SM, which constrains the aggregate register usage by all threads, and Regs_Per_Warp specifies the register consumption per warp, influenced by the kernel's register usage pattern.

Impact of Memory Allocation on Scheduling. Our experiments further illustrate that the scheduling decisions heavily prioritize configurations requesting over 3KB of shared memory. For allocations under this threshold, thread count predominantly influences scheduling decisions. We observed that for every 256-byte increment above the 3KB threshold, the scheduling behavior adjusts, reflecting the granularity with which shared memory is allocated and managed within the GPU.

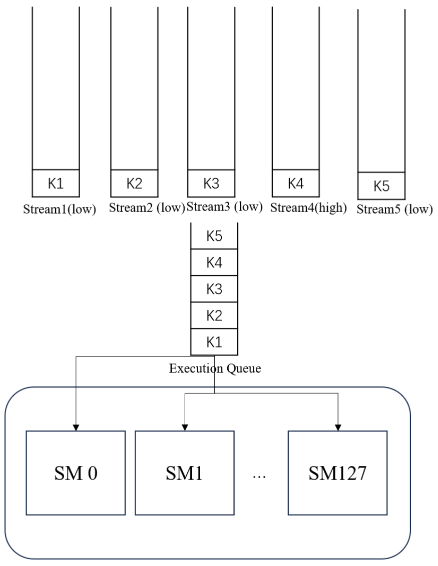

Figure 7: Stream Settings for the Stream Priority Scheduling Experiment.

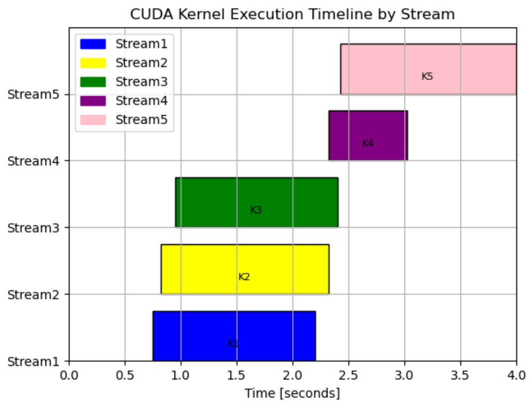

Figure 8: Timeline for the Stream Priority Scheduling Experiment.

4.3 Effects of NULL Stream and Stream Priority

In CUDA, the null stream [] functions as the default stream and exhibits unique behaviors not shared by other streams. According to the CUDA documentation, concurrent execution of two kernels is prohibited if any operations are submitted to the null stream in the interim. However, the documentation does not clarify how the order of kernel execution is influenced by such submissions. Our experimental observations have provided significant insights into how NULL streams 56 and stream priorities76 impact scheduling:

- **NULL Stream Interaction:** Concurrent kernel execution is prohibited if NULL stream operations interrupt. A kernel in the NULL stream is queued in the EE only if other streams are empty or if their head kernels started later. This rule

highlights the NULL stream's role in enforcing sequential execution.
- **Stream Priority Effects:** Kernels are queued in the EE based on stream priority. Blocks from the foremost kernel in the EE are assigned only after higher-priority streams are cleared, ensuring prioritized processing of critical tasks.

Consider an experiment where Kernel $K1$ is launched first, utilizing all available threads on the GPU. In theory, if a portion of $K1$'s blocks complete execution, Kernel $K2$ should be eligible for dispatch as it requires only one SM thread. However, due to $K2$ being in the null stream, it does not transition to the EE queue until $K1$ has entirely finished execution. This observation leads to the formulation of our second rule. Similarly, $K5$ does not advance to the EE queue until both $K3$ and $K4$ have completed, illustrating the sequential processing enforced by the null stream.

Moreover, while $K2$ remains at the head of the null stream queue, no subsequent kernel launched can move to the EE queue. This behavior of the null stream kernels results in unnecessary blocking of $K6$, which could not execute concurrently with $K1$, $K3$, or $K4$, leading to significant underutilization of SM capacity.

Additional experimentation focused on stream priority and its impact on scheduling when resource contention is present. The prevailing assumption about stream priority suggests that blocks from higher-priority streams are preferentially assigned when new blocks are eligible for scheduling. Our experiments sought to determine whether kernels in high-priority streams, experiencing resource blocking, would preempt those in low-priority streams or if they would be indefinitely delayed by earlier-queued kernels from low-priority streams that consume available resources.

In a targeted experiment, we explored whether kernels in high-priority streams could be indefinitely delayed by resource blocking from kernels in low-priority streams. Kernels $K1$ through $K3$ were rapidly deployed across seven distinct low-priority streams. Subsequently, Kernel $K4$ was launched in a high-priority stream. At that moment, none of the initial three kernels had completed, thereby leaving inadequate threads available for $K4$. Despite requiring three-quarters of the available threads, only one-quarter was available, preventing the immediate scheduling of $K4$, despite its high priority. This scenario underscored that a block at the head of any Execution Engine (EE) queue is only eligible for assignment if all higher-priority EE queues are empty, highlighting a critical dynamic in priority-based scheduling.

4.4 Analysis of Block-to-SM Scheduling

Utilizing the kernel presented in 1, we explore block-to-SM scheduling dynamics. This kernel allows for varying kernel sizes, thread counts, shared memory allocations, and register usage to adapt to different testing scenarios. Unfortunately we can not find an exact prediction algorithm for GPU scheduling, but our experiments provide insights into block-to-SM scheduling behaviors.

```
__global__ void TestKernel(int *smIDs, int *bIDs, int *tIDs,
                           int *bDims, int *memSizes,
                           float *durations, clock_t rate) {
    int bID = blockIdx.x, tID = threadIdx.x, smID;
    asm("mov.u32 %0, %%smid;" : "=r"(smID)); // Retrieve SM ID
    if (tID == 0) { // First thread actions
        smIDs[bID] = smID;
        bIDs[bID] = bID;
        bDims[bID] = blockDim.x;
        memSizes[bID] = blockDim.x * sizeof(int);
    }
    tIDs[bID * blockDim.x + tID] = tID; // Record IDs
    extern __shared__ int sMem[]; // Shared memory
    sMem[tID] = tID; // Initialize shared memory
    __syncthreads(); // Sync threads
    // Computation loop
    const int regCount = 256, memCount = memSizes[bID] / sizeof(
        int);
    int regs[regCount];
    for (int i = 0; i < memCount; i++) {
        regs[i % regCount] += sMem[i];
    }
    if (tID < memCount) sMem[tID] = regs[tID];
    // Delay to simulate execution
    clock_t start = clock64();
    while ((clock64() - start) < durations[bID] * rate * 1e-3f) {}
}
```

Listing 1: Kernel code used for block-to-SM scheduling tests.

4.4.1 *Thread-Dominated Block Scheduling.*

In examining thread utilization's impact on GPU scheduling, we intentionally minimized register and shared-memory constraints in kernel configurations across various streams. This approach, facilitated by setting shared memory size and register count to the minimal value of one, ensures that the scheduling decisions are predominantly influenced by thread counts, as defined by the first term of Equation below.

Our experimental methodology involved ten trials, each with randomly configured block and thread sizes for the kernel described in 1. In each trial, we assessed the actual distribution of blocks to SMs using a round-robin approach, with SM assignments identified via the 'sm_id' retrieved from the SMID register. We then compared these empirical results against the expected RR distribution. While deviations occurred, the overall prediction accuracy was remarkably high. As illustrated in Figure 9, the prediction accuracy remained robust, achieving up to 96% even when the number of streams increased to 8. This high level of accuracy highlights the effectiveness of the resource-aware round-robin scheduling approach in scenarios dominated by thread allocation.

Our findings reveal that mispredictions occur predominantly as resources near saturation, leading to random block allocations to SMs, as illustrated in 2. This randomness becomes more pronounced past specific saturation points, resulting in unpredictable scheduling. We hypothesize that when all SM resources are fully utilized, incoming blocks are temporarily blocked until the next clock cycle. At this point, the GPU re-evaluates SM availability and assigns blocks accordingly. The random completion times of tasks across various SMs contribute to sporadic block allocations, injecting unpredictability into the scheduling outcomes post-saturation.

Table 2: Disordered block-to-SM mapping observed under maximal thread utilization conditions across 6 streams.

Block ID	124	125	126	127	128	129	130	131	132	133
SM ID	121	123	125	127	19	84	41	18	106	5

4.4.2 *Considerations for Shared Memory and Register-Intensive Configurations.*

Scheduling complexity increases significantly when the constraints on shared memory and register usage are factored into the configuration. In our experiments, each kernel configuration manipulates these resources differently, controlled via the

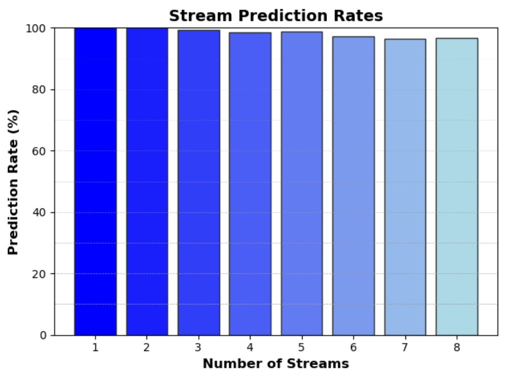

Figure 9: Prediction rates in block-to-SM scheduling under thread-dominated configurations.

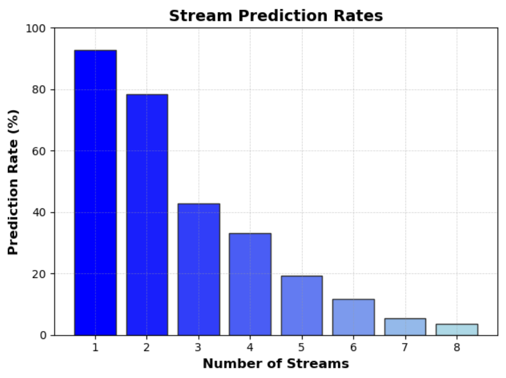

Figure 10: Misprediction rates in block-to-SM scheduling with intensive shared memory and register usage.

compiler flag *'-maxrregcount'* to adjust resource consumption intentionally, allowing precise modulation of shared memory allocations and register limits.

Despite integrating resource occupancy parameters into our scheduling model , our adapted round-robin approach frequently leads to substantial mispredictions in block execution times. Figure 10 demonstrates these misprediction rates, charted against the number of streams per kernel.

As the result shows, with an increase to eight streams, misprediction rates can soar to as much as 97%. These inaccuracies in scheduling extend beyond mere saturation of available SMs, affecting the broader predictability of our scheduling algorithm. We hypothesize that while the allocated size of shared memory substantially influences scheduling decisions, the latency associated with memory access emerges as an equally critical factor. This introduces an additional layer of complexity to the scheduling dynamics, affecting the effective allocation of blocks to SMs.

4.4.3 TPC-Based Block-to-SM Scheduling Hypothesis. Another interesting finding on scheduling behavior is round-robin mechanism preferentially assigns blocks to SMs with even IDs before those with odd IDs, as evidenced by Tables 4a and 4b.

Table 3: Example of Block-to-SM mapping for memory intensive experiments.

Block ID	90	91	92	93	94	95	96	97	98	99
SM ID	6	81	121	100	117	40	42	25	107	79

In NVIDIA GPUs, the architecture is organized into SMs, which are grouped into larger units known as Graphics Processing Clusters (GPCs). Each GPC contains a raster engine and six Texture Processing Clusters (TPCs), with each TPC housing two SMs. For instance, the integrated GPU in the RTX 4090 is configured with 128 SMs across 64 TPCs, with the first TPC encompassing SMs labeled as IDs 0 and 1, and subsequent TPCs following in a similar pattern.

We hypothesize that the primary purpose of this scheduling strategy is to minimize memory contention. This contention, we believe, is not due to shared Arithmetic Logic Units (ALUs)—since each block is processed on separate SMs—but rather stems from memory access conflicts. Particularly within a TPC, warps share the same bus to access both the GPU's L2 cache and system RAM, leading to significant contention.

Based on our observations, we propose that interference within the GPU memory hierarchy can critically influence the response times of tasks, particularly when multiple streams are involved. The Worst Case Execution Time (WCET) of a CUDA block can significantly escalate if it is scheduled in a TPC alongside another memory-intensive block. Although interference diminishes when blocks are allocated to different TPCs, it remains a concern. Our hypothesis extends to the initial distribution strategy employed by the NVIDIA block scheduler, which systematically assigns blocks starting with SMs of even indices, followed by those with odd indices. This method is theorized to reduce potential memory interference among SMs within the same TPC, mitigating adverse effects under scenarios of high computational load.

Table 4: Block-to-SM mappings to even and odd sets of SMs from our experiments.

(a) Scheduling for even SM IDs

Block ID	0	1	2	3	4	5	6	7	8	9
SM ID	0	2	4	6	8	10	12	14	16	18

(b) Scheduling for odd SM IDs

Block ID	64	65	66	67	68	69	70	71	72	73
SM ID	1	3	5	7	9	11	13	15	17	19

5 STRATEGIES FOR ENHANCING CO-LOCATION IN COVERT CHANNELS

Based on our experimental insights, we propose strategies to optimize co-location between Trojan and spy processes. Effective co-location is crucial for successful resource contention-based communication.

5.1 Prioritizing Functional Unit Channels

Our results indicate that thread-domain scheduling is relatively predictable, whereas scheduling involving shared memory is considerably more complex and less predictable. In light of this, leveraging Functional Unit Channels rather than relying predominantly on Cache and Memory Covert Channels could simplify achieving co-location. This approach minimizes the unpredictability associated with memory-intensive scheduling, facilitating more reliable setup of covert channels.

5.2 Minimizing Stream Usage

The complexity of the GPU scheduler increases with the number of streams involved, which can hinder effective co-location. By limiting the number of streams utilized by both the trojan and the spy, it is possible to reduce scheduling variability, thereby enhancing the likelihood of successful co-location.

5.3 Strategic Block Placement

Considering the GPU scheduler's tendency to schedule odd and even indexed SMs separately, strategic placement of Trojan and spy processes is imperative. To ensure that both contend for the same TPC resources, avoid placing them in consecutive blocks, which would lead to different TPC allocations. For example, on a GPU with 128 SMs, placing a Trojan in the 0th block and a spy in the 128th block ensures that both are within the same TPC, specifically spanning the 0th and 1st SMs. This strategic alignment increases the chances of effective resource contention, which is key for covert communication.

6 WORST-CASE GPU SCHEDULING ANALYSIS

In this section, we discuss worst-case GPU scheduling, exploring two types of experiments. The first investigates how thread layout within the kernel can lead to worst-case GPU scheduling. The second examines how the selection and partitioning of GPU SMs can impact worst-case scheduling.

6.1 Experiment Setup

6.1.1 Hardware and Software Platform. The experiments were conducted on a single Nvidia A100 (40 GB PCIe) with 30 CPU cores, 205.4 GB RAM, running Linux version 146-235-238-45, GPU Driver Version: 535.129.03, and CUDA Version: 12.2. The GA100 GPU comprises the following units:

- 8 GPCs, 8 TPCs/GPC, 2 SMs/TPC, 16 SMs/GPC, 128 SMs per full GPU
- 64 FP32 CUDA Cores/SM, 8192 FP32 CUDA Cores per full GPU

6.1.2 Testing Kernel. All experiments measured the response times of a matrix multiplication task, where two 1024x1024 matrices are multiplied, producing a third matrix of the same size. Each GPU thread is responsible for computing one element of the result matrix, with all elements initialized as 32-bit floating-point numbers between 0 and 1. All source code and data are available online.

Matrix multiplication was chosen due to its constant computational and memory access requirements, making it a relevant and common operation in many AI and graphics applications. It is also easy to configure matrix multiplication kernels to use different thread block sizes without affecting the total GPU computation required.

We defined two tasks for our experiments:

- **MM1024**: Uses blocks of 1024 threads (32x32 2D blocks), launching exactly 1024 blocks for the matrix.
- **MM256**: Uses blocks of 256 threads (16x16 2D blocks), requiring 4096 blocks to cover the matrix.

6.2 Worst-Case Scheduling: Thread Layout

The maximum number of block threads per SM imposes restrictions on kernel block and thread layout. Improper design can lead to resource idling. For instance, if each block's thread count slightly exceeds half the maximum threads per SM, scheduling one block to an SM may leave insufficient capacity for a second block, causing inefficient use of SM resources. We compared an underutilized kernel (block size slightly over half the max threads per SM plus warp size) with a fully utilized kernel (block size optimized to fully utilize SM capacity) for MM1024, maintaining consistent overall thread count.

Experiment 1: Different Thread and Block Numbers

- **Underutilized Kernel**: This configuration has a block size slightly exceeding half the maximum threads per SM, leading to underutilization.
- **Fully Utilized Kernel**: This setup maximizes resource usage by selecting block sizes that fully utilize SM capacity.

The underutilized kernel significantly underperforms compared to the fully utilized kernel, both in terms of average performance and variability. This highlights the potential for worst-case scheduling when block sizes are not optimized to match the hardware's thread capacity, leading to resource idling and reduced throughput.

Experiment 2: Single Large Kernel vs. Multiple Small Kernels We compared a single large kernel execution with multiple small kernels executed within one or more streams while maintaining the overall thread count.

- **Single Large Kernel**: Represents a monolithic kernel execution.
- **Multiple Small Kernels with 1 Stream**: Shows a slight overhead from managing multiple kernels but remains relatively efficient.
- **Multiple Small Kernels with 2 Streams**: A considerable increase in mean execution time, indicating that parallelism with two streams introduces overhead.
- **Multiple Small Kernels with 4 Streams**: The highest mean execution time, suggesting the overhead of synchronizing and managing multiple streams outweighs potential parallelism benefits.

6.3 Worst-Case Scheduling: SM Selection

This section explores how SM partitioning influences worst-case scheduling by masking specific SMs during kernel launch to enforce specific patterns.

6.3.1 SM Masking Method. CUDA does not natively support SM masking, but thanks to the work of Joshua Bakita and James H. Anderson [3], we know it is possible to manipulate the kernel

Configuration	**# Samples**	**Min (ms)**	**Max (ms)**	**Median (ms)**	**Mean (ms)**	**Std Dev (ms)**
Experiment 1: Different Thread and Block Numbers						
Underutilized Kernel	100	1.9466	3.8246	2.2917	2.2571	0.1921
Fully Utilized Kernel	100	1.2063	2.5221	1.3271	1.2998	0.1322
Experiment 2: Single Large Kernel vs. Multiple Small Kernels						
Single Large Kernel	100	1.1950	2.4166	1.3261	1.3006	0.1233
Multiple Small Kernels with 1 Stream	100	1.2175	2.5108	1.3280	1.3029	0.1311
Multiple Small Kernels with 2 Streams	100	2.3182	3.7737	2.5897	2.5659	0.1499
Multiple Small Kernels with 4 Streams	100	4.5384	5.9607	5.1384	5.0882	0.1679

Table 5: Performance metrics for different thread layout configurations

launch parameters. They discovered a masking variable located between 84 and 92 bytes in the CUDA kernel launch function's input parameters. (This is the reason why we chose to conduct the experiments detailed in this section on a Linux platform. The reverse-engineering findings is specific to the Linux version of the CUDA kernel launch function. For the Windows version, however, this variable does not appear within this byte range, rendering their outcome incompatible with Windows systems.) This mask operates at the TPC (Texture Processing Cluster) level rather than the SM level. Each 64-bit mask can control up to 64 TPCs (each TPC consists of 2 SMs).

Using this information, we hook into the CUDA kernel launch process by retrieving the export table using `cuGetExportTable`, which provides function pointers for callback registration and enabling. We then register the `KernelLaunchCallback` function as a callback for kernel launch events and enable it. In the callback function, the kernel launch parameters are extracted and modified to apply the desired SM configuration.

6.3.2 *GPU Partitioning.*

We explore several SM partitioning configurations:

- **Full GPU Sharing**: Both tasks have unrestricted access to all SMs.
- **Even Partitioning**: Tasks are restricted to separate, non-overlapping partitions containing half of the SMs each.
- **Uneven Partitioning**: Tasks are restricted to separate, non-overlapping partitions containing 1/4 and 3/4 of the SMs, respectively.

Two kernels are launched concurrently, each assigned different SM partitions to evaluate performance. We use separate streams to ensure parallel execution.

Table 6 shows the results. The baseline is established by isolating the two kernels and launching them individually, resulting in execution times of about 2.63 ms. When both kernels run concurrently with full SM sharing, the performance remains similar (2.62 ms), indicating the workload does not overwhelm the GPU.

However, when SMs are partitioned, performance degrades, especially with uneven partitioning. This indicates that specific partitioning configurations can exacerbate worst-case scheduling scenarios by inefficiently utilizing the available SM resources.

6.3.3 *SM Locality.*

To evaluate the impact of limiting partitions to specific SM groups, we contrast two partitioning approaches:

- **GPC-packed**: Packs as many SMs as possible into each GPC before occupying additional GPCs, minimizing the number of GPCs used.
- **GPC-distributed**: Distributes SMs evenly across all GPCs, ensuring even use of all available GPCs.

Table 7 shows that GPC-packed configurations generally outperform GPC-distributed ones. When SMs are packed into a single GPC, operations benefit from low-latency access to shared L1 cache and shared memory, with minimal inter-GPC communication overhead. However, packing too many SMs into a single GPC can lead to resource contention. For the workloads tested, GPC-packed appears superior, but as workloads scale, this approach may face challenges due to resource bottlenecks within a single GPC.

6.4 Summary

From these experiments, we conclude that for the GEMM testing kernel, the worst scheduling strategy involves partitioning the kernel into multiple streams and assigning unevenly partitioned GPU SMs. When multiple GEMM kernels run concurrently, enforcing specific SM configurations and distributed SM usage can exacerbate worst-case scheduling scenarios, leading to significant performance degradation.

7 CONCLUSION

This paper presents a comprehensive investigation of the scheduling mechanisms and policies governing the NVIDIA RTX 4090 GPU, focusing on block-to-SM scheduling dynamics. Our findings reveal the complex interplay of factors such as kernel queueing, resource availability, and stream priorities in determining GPU scheduling behavior.

Experiments under thread-dominated configurations showed high prediction accuracy using a round-robin approach, while resource-intensive scenarios, particularly those involving shared memory and registers, exhibited significant unpredictability. We also uncovered a distinct scheduling pattern favoring even-indexed SMs over odd-indexed ones, possibly to minimize memory contention within Texture Processing Clusters (TPCs).

Additionally, we explored worst-case GPU scheduling by analyzing how improper thread layout and SM partitioning exacerbate

Configuration	# Samples	Min (ms)	Max (ms)	Median (ms)	Mean (ms)	Std Dev (ms)
Isolated MM1024	100	2.3798	5.0340	2.6562	2.6395	0.2567
Isolated MM256	200	2.2744	5.6677	2.6542	2.6352	0.2944
MM1024 and MM256 - Full Sharing	300	2.2744	5.6677	2.6495	2.6209	0.2938
MM1024 (vs. MM1024) - Even Partitioning	400	2.2744	5.6677	2.6542	2.7143	0.3175
MM1024 (vs. MM1024) - Quarter Partitioning	500	2.2744	7.5939	2.6563	3.1286	0.8849
MM1024 (vs. MM256) - Even Partitioning	600	2.2744	7.5939	2.6757	3.1108	0.8149
MM1024 (vs. MM256) - Quarter Partitioning	700	2.2744	7.5939	3.0188	3.3134	0.9070
MM256 (vs. MM256) - Even Partitioning	800	2.2744	7.5939	3.0228	3.2793	0.8591
MM256 (vs. MM256) - Quarter Partitioning	900	2.2744	7.5939	3.0592	3.4242	0.9112
MM256 (vs. MM1024) - Even Partitioning	1000	2.2744	7.5939	3.0587	3.3837	0.8757
MM256 (vs. MM1024) - Quarter Partitioning	1100	2.2744	7.5939	3.0653	3.5090	0.9266

Table 6: Performance metrics for different SM partitioning configurations

Configuration	# Samples	Min (ms)	Max (ms)	Median (ms)	Mean (ms)	Std Dev (ms)
Full GPU Sharing (Unpartitioned)	100	2.3112	5.3616	2.6327	2.6125	0.2953
GPC-packed, Equal Partitions	200	2.3112	7.5110	4.5076	3.5882	1.0197
GPC-packed, Unequal Partitions	300	2.3112	7.5110	4.3551	3.8536	0.9287
GPC-distributed, Equal Partitions	400	2.3112	14.5994	4.5076	5.7237	3.3415
GPC-distributed, Unequal Partitions	500	2.3112	14.5994	4.5476	6.4524	3.3278

Table 7: Performance metrics for different SM locality configurations

performance degradation. The underutilized kernel configurations and uneven SM partitioning scenarios demonstrated significant negative impacts on performance, underscoring the importance of careful kernel design and SM allocation.

These insights have important implications for optimizing co-location strategies in covert channels on GPGPUs. By prioritizing functional unit channels, minimizing stream usage, and employing strategic block placement, the reliability and effectiveness of covert communication can be enhanced.

Our work contributes to a deeper understanding of the complex scheduling mechanisms within modern GPUs, which is crucial for optimizing performance and fortifying systems against covert channel attacks. Future research could explore the impact of memory hierarchy on task response times and investigate advanced techniques for mitigating covert channel vulnerabilities across different GPU architectures and programming models.

In conclusion, our study unravels the complexities of GPU scheduling in the NVIDIA RTX 4090, empowering developers and security professionals to build more robust and secure GPU-based systems while highlighting potential avenues for optimizing co-location strategies in covert channel attacks.

ACKNOWLEDGMENTS

We thank Professor Simha Sethumadhavan and teaching assistant Ryan R. Piersma for their invaluable guidance and support throughout this project.

REFERENCES

[1] Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed A Badawy. 2022. Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.

[2] Tanya Amert, Nathan Otterness, Ming Yang, James H Anderson, and F Donelson Smith. 2017. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 104–115.

[3] Joshua Bakita and James H Anderson. 2023. Hardware compute partitioning on nvidia gpus. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 54–66.

[4] Brett W Coon, John R Nickolls, John Erik Lindholm, Robert J Stoll, Nicholas Wang, and Jack Hilaire Choquette. 2014. Thread group scheduler for computing on a parallel thread processor. US Patent 8,732,713.

[5] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2021. Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 972–984.

[6] Glenn A Elliott, Bryan C Ward, and James H Anderson. 2013. GPUSync: A framework for real-time GPU management. In *2013 IEEE 34th Real-Time Systems Symposium*. IEEE, 33–44.

[7] Guin Gilman, Samuel S Ogden, Tian Guo, and Robert J Walls. 2021. Demystifying the placement policies of the NVIDIA GPU thread block scheduler for concurrent kernels. *ACM SIGMETRICS Performance Evaluation Review* 48, 3 (2021), 81–88.

[8] Guin Gilman and Robert J Walls. 2022. Characterizing concurrency mechanisms for nvidia gpus under deep learning workloads. *ACM SIGMETRICS Performance Evaluation Review* 49, 3 (2022), 32–34.

[9] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).

[10] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. 2011. RGEM: A responsive GPGPU execution model for runtime engines. In *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE, 57–66.

[11] Xinxin Mei and Xiaowen Chu. 2016. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2016), 72–86.

[12] Hoda Naghibijouybari, Khaled N Khasawneh, and Nael Abu-Ghazaleh. 2017. Constructing and characterizing covert channels on gpgpus. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 354–366.

[13] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered insecure: Gpu side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2139–2153.

[14] NVIDIA Corporation. 2024. *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html Available online.

[15] Ignacio Sanudo Olmedo, Nicola Capodieci, Jorge Luis Martinez, Andrea Marongiu, and Marko Bertogna. 2020. Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 213–225.

[16] Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H Anderson, F Donelson Smith, Alex Berg, and Shige Wang. 2017. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 353–364.

[17] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 235–246.

[18] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. 2016. Warped-slicer: Efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 230–242.

Received 4 May 2024; revised 4 May 2024; accepted 4 May 2024