MIGraine: Practical Side-channel Attacks on NVIDIA Multi-Instance GPUs via Page Fault Contention

Abstract

Multi-tenant GPU deployments require strong isolation to prevent cross-tenant data leakage. NVIDIA introduced Multi-Instance GPU (MIG) to partition a single GPU into separate GPU Instances (GIs), advertising strong isolation and non-interference across clients. Although recent work has challenged the microarchitectural isolation guarantees of NVIDIA MIG, state-of-the-art side-channel attacks remain limited in accuracy and portability in practice.

In this paper, we present *MIGraine*, a practical cross-GI side channel based on *page fault contention*. Specifically, we show that, despite strong hardware isolation, blackbox side-channel attacks with high accuracy and portability are still possible on modern NVIDIA GPUs. Unlike prior efforts, our attacks are GPU generation-agnostic and do not rely on microarchitectural reverse engineering. To substantiate our claims, we focus on *Unified Virtual Memory* (UVM), NVIDIA's programming model for automatic host-GPU data movement. We first present a deep-dive analysis of the UVM driver to reverse-engineer page fault handling behavior. We then show that UVM page faults on one GI increase latency on a co-located GI, while the reverse direction turns page fault latency into a cross-GI side channel. Finally, we demonstrate exploitability by fingerprinting steady-state Large Language Models (LLMs) workloads across GIs with 96% accuracy. We further provide the first characterization of containerized versus virtualized MIG deployments, showing that MIG's isolation guarantees fall short in both cases, even under the stronger memory isolation provided by vGPUs.

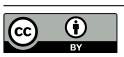
Keywords

GPU security, side channels, multi-tenant GPUs, NVIDIA MIG, Unified Virtual Memory

1 Introduction

Graphics Processing Units (GPUs) emerged in the 2000s as specialized processors for accelerating graphics workloads, exploiting their massive parallelism to process millions of pixels simultaneously. Over time, GPUs have been popularized for an increasingly diverse set of tasks, including cryptocurrency mining and Machine Learning (ML). Their growing deployment across mobile devices, desktops, and especially servers, has prompted researchers to examine the security implications of GPUs in modern systems.

GPU side channels. Similarly to CPUs, GPU components are proprietary and kept secret by vendors such as NVIDIA, raising the bar for security research which often relies on reverse engineering [1, 6–8, 21]. Unlike most CPUs, however, NVIDIA GPUs do not provide a public instruction set architecture (ISA). Considerable effort has been dedicated towards documenting the ISA for various GPU generations [22, 23, 30, 67, 70]. Despite these challenges, researchers have successfully demonstrated side-channel attacks on GPUs. Similarly to their CPU counterparts [17, 25, 28, 59, 65], these attacks target shared components such as caches, translation lookaside buffers (TLBs), and other microarchitectural components [3, 11, 13, 24, 33, 34, 74], interconnects such as PCIe [60] and NVLink [10, 71, 72], and interrupts [31]. The net result is the ability to fingerprint a co-located GPU workload, for instance to mount ML *model fingerprinting* attacks.



This work is licensed under a Creative Commons Attribution 4.0 International License.

NVIDIA MIG. To support GPU multi tenancy, NVIDIA introduced *Multi-Instance GPU* (MIG) for their server-grade GPUs. MIG partitions a single physical GPU into multiple *GPU Instances* (GIs), each with dedicated compute, memory, and cache resources. MIG can be deployed in both containerized and virtualized (vGPU) cloud environments, with each tenant assigned a separate GI. NVIDIA documents MIG as providing strong multi-tenant isolation, stating that it "ensures one client cannot impact the work or scheduling of other clients" and that each GI has "separate and isolated paths through the entire memory system" [39]. However, prior work has identified components, namely the L3-TLB [73] and the PCIe interface [32], which remain shared across GIs and enable side-channel attacks. Nonetheless, existing attacks remain limited in accuracy and portability in practical settings. Specifically, prior L3-TLB attacks rely on reverse engineered TLB details and can only accurately fingerprint ML models at load time [73]. Prior PCIe attacks [32], in turn, are more accurate, but only apply to containerized deployments and rely on the victim ML model to use special features such as weight offloading.

In this paper, we show that page fault interrupts generate cross-GI interference that can be abused to mount side-channel attacks across GIs despite MIG-enforced isolation. To craft page fault-triggering primitives, we focus our attention on NVIDIA's Unified Virtual Memory (UVM) [45, 75] programming model. UVM is widely used by modern GPUs workloads [4, 12, 35, 46, 53], enabled by default by the NVIDIA driver [36, 42], and adopted in the major cloud environments [5, 14]. UVM enables automatic page migration between host and GPU memory and is fundamentally driven by page fault interrupts.

Reverse engineering and contention analysis. First, we present a deep-dive analysis of the UVM driver to understand page fault handling behavior (Section 5). Our analysis shows that the fault buffer and page fault handling code are shared across GIs in containerized environments and that said code interacts with multiple hardware components including the PCIe bus and internal GPU memory subsystems. We then systematically characterize cross-GI interference by first measuring how page fault storms on one GI affect the execution of different workloads on a co-located GI (Section 6.1). We specifically design workloads to test *contention* on different resources: (i) GPU-Host memory accesses that stress the PCIe interface and shared software buffers, and (ii) GPU-internal memory operations targeting the L1, L2, and Constant caches as well as VRAM. Our experiments confirm that a page fault storm can induce measurable latency increases on a neighboring GI across

all tested workloads, with particularly severe degradation (up to millions of cycles) for UVM-intensive operations. Next, we reverse the direction of our contention analysis to demonstrate that GPU workloads running on one GI affect the page fault latency of another GI (Section 6.2).

Page fault covert channel. With insights from our analysis, we demonstrate that the contention generated by page fault storms on GPU-internal memory operations can be exploited to construct a cross-GI covert channel (Section 7). Specifically, we exploit the increased latency in GPU memory loads during page fault storms as a signaling mechanism.

MIGraine. Next, with insights from our reversed contention analysis, we present *MIGraine*, a practical cross-GI side-channel attack (Section 8). *MIGraine* exploits the interference that GPU workloads running on a co-located GI induce on UVM page fault latency to leak information across GIs. Specifically, we show that an attacker continuously measuring page fault latency can fingerprint sensitive victim workloads, such as Machine Learning (ML) or Large Language Models (LLMs), on a neighboring GI. This is feasible by extracting simple statistical features from timings and training an XGBoost [9] classifier.

MIGraine achieves 96% accuracy fingerprinting LLM workloads running through vLLM [27] on co-located GIs.

vGPU isolation analysis. Since containerized and virtualized MIG deployments employ fundamentally different isolation architectures, we also present a separate analysis of vGPU environments (Section 9). Specifically, we reverse-engineer the vGPU address translation scheme and show that vGPU adds hardware-enforced memory isolation, unlike containerized setups. Despite this stronger isolation, we show that our *MIGraine* attack remains effective in vGPU environments, exploiting hardware contention on the PCIe bus and GPU memory subsystems that neither deployment architecture mitigates.

Contributions. We make the following contributions:

(1) We present *MIGraine*, a MIG side-channel attack based on UVM page fault latency, demonstrating its effectiveness in fingerprinting both ML and LLM workloads on co-located GIs with high accuracy (> 90%).
(2) We conduct a root-cause analysis of the side channel, identifying contention on the PCIe bus, shared software buffers, and cache coherency operations as the primary sources of cross-GI interference.
(3) We analyze architectural differences between containerized and vGPU MIG deployments, showing that vGPU adds software resource partitioning and hardware memory isolation. We demonstrate that our side channel remains effective across both environments, highlighting fundamental isolation gaps in MIG.

2 Background

2.1 NVIDIA GPU Hardware Architecture

A modern NVIDIA GPU is a complex, heterogeneous system with several specialized hardware units designed for massively parallel processing. Each GPU generation introduces distinct hardware

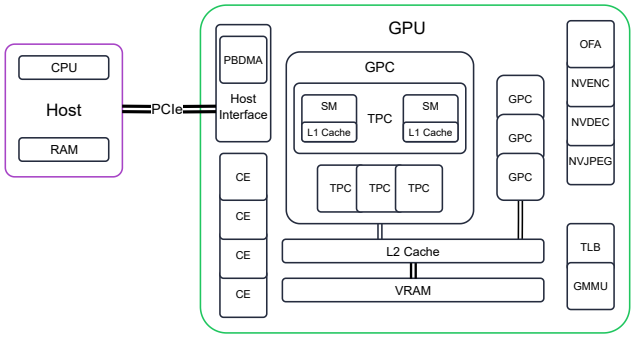


Figure 1: High-level overview of a modern NVIDIA GPU, with the host system (CPU and RAM) connected to the GPU via PCIe. Internally, the GPU features a compute hierarchy (GPCs, TPCs, SMs), a memory hierarchy (L1/L2 caches, TLB, VRAM), 4 Copy Engines (CEs), one PBDMA engine, and various specialized engines (NVJPEG, NVDEC, etc).

features and improvements, but at their core, they all include computational units, a memory subsystem, and high-speed interconnects. Ampere GPUs were the first to support MIG partitioning, followed by Hopper, and Blackwell. Throughout this work, we use the term *Ampere+* to indicate that a given feature is available on all GPU architectures starting with Ampere. Figure 1 presents a high-level overview of the NVIDIA GPU architecture.

GPU engines. The primary processing power of an NVIDIA GPU comes from its *Compute/Graphics (GR) Engine*, which is built upon a hierarchy of fundamental computational units. At the lowest level are the *Streaming Multiprocessors (SMs)*, each containing simple cores that execute threads concurrently in a Single-Instruction, Multiple-Thread (SIMT) fashion. These SMs are grouped into *Texture Processing Clusters (TPCs)*, which in turn form *General Processing Clusters (GPCs)*. A high-end GPU may contain dozens of SMs distributed across several GPCs.

In addition to the main GR engine, NVIDIA GPUs include specialized engines for different tasks: *Copy Engines (CEs)* handle memory transfers, dedicated units perform image and video encoding, and *Push Buffer (PBDMA)* engines fetch, parse, and dispatch commands from host memory to the appropriate GPU engines.

Memory subsystem. An NVIDIA GPU has its own memory subsystem. Each SM has a private, fast L1 cache, while all SMs share a larger, unified L2 cache that serves as the last-level cache (LLC). There is also a *Constant* cache, although its placement and hierarchy details are not publicly available. Beyond these on-chip caches lies the GPU's main DRAM memory, also called *Video RAM (VRAM)*. In the server-grade NVIDIA platforms relevant to this paper, MIG-capable GPUs use *High Bandwidth Memory (HBM)* rather than GDDR; for example, the NVIDIA A30 uses HBM2, while the H100 uses HBM3 [49, 50].

Furthermore, the GPU has its own *GPU Memory Management Unit (GMMU)*, handling virtual-to-physical address translation. The GMMU relies on a multi-level page table structure, similar to that of a CPU. Recent NVIDIA architectures use a 5-level page table

hierarchy to map the large virtual address spaces required by applications. The virtual memory system supports different page types depending on the target physical memory; Page Directory/Table Entries (PDEs/PTEs) that point to VRAM typically map large pages of 2 MB or 64 kB, whereas PTEs that point to the host system's RAM use the standard 4 kB page size. To accelerate these translations, GPUs are equipped with *Translation Lookaside Buffers (TLBs)*. On Ampere GPUs, the TLB itself is a three-level hierarchy, culminating in a large L3-TLB that is shared across all GPCs [73].

Interconnects. The GPU is typically connected to the host system (CPU and main memory) via a *Peripheral Component Interconnect Express (PCIe)* bus. For multi-GPU servers requiring higher bandwidth and lower latency, NVIDIA provides a proprietary high-speed interconnect called *NVLink*. NVLink establishes direct communication paths between GPUs.

NVIDIA GPU software stack. Running programs on the GPU requires a complex software stack that spans from user-space libraries to kernel drivers. NVIDIA GPUs can be programmed using graphics APIs or *CUDA (Compute Unified Device Architecture)*, a proprietary parallel computing model. We focus on CUDA, since graphics APIs are disabled on MIG environments [38]. The CUDA compiler toolchain processes programs, known as *GPU kernels*, in two stages. First, it compiles the code into an intermediate representation called *PTX (Parallel Thread Execution)*. Then, it compiles the PTX code into *SASS (Shader Assembly)*, the native instruction set for the target GPU architecture. Because the SASS instruction set is specific to each GPU generation, this final compilation step is tailored to a particular architecture for optimal hardware utilization. To launch a kernel, a developer specifies a grid of thread blocks. Each block contains a group of threads that run concurrently on a single Streaming Multiprocessor (SM) and can synchronize with one another. CUDA provides two primary memory allocation strategies. The `cudaMalloc` function allocates memory exclusively in GPU space, accessible only from GPU kernels. In contrast, `cudaMallocManaged` creates unified virtual memory accessible from both CPU and GPU through page fault handling mechanisms managed by the *Unified Virtual Memory (UVM)* driver. When either processor attempts to access a managed page residing in the other's memory space, the UVM driver intercepts the page fault and coordinates automatic data migration. The CUDA runtime also depends on the *Resource Manager (RM)* driver, which handles low-level GPU initialization and command queue creations. On modern GPUs, some RM functionality is offloaded to the *GPU System Processor (GSP)*, an onboard microcontroller running its own firmware.

2.2 NVIDIA GPU Virtualization

Virtual (vGPU). To provide Virtual Machines (VMs) with GPU access, NVIDIA offers its *virtual GPU (vGPU)* technology. Early vGPU implementations enabled GPU sharing primarily by partitioning memory across multiple VMs. The compute resources, however, were not spatially divided but were shared through time slicing, where each VM's workload ran in turn.

Multi-Instance GPU (MIG). Introduced with the Ampere architecture, *Multi-Instance GPU (MIG)* is a hardware technology that partitions GPU memory and its computational units. MIG spatially divides a single GPU into multiple, fully isolated GPU Instances (GIs). Each GI receives its own dedicated set of compute engines (GPCs), L2 cache, memory controllers, and DRAM bandwidth, ensuring strong, predictable performance. Modern deployments often use vGPU in conjunction with MIG to assign a dedicated GI to each VM, supporting virtualization with strong isolation guarantees. Throughout this paper, we use the term *vGPU* to refer to this vGPU+MIG setup, unless otherwise specified.

3 Threat Model

We consider a multi-tenant GPU deployment scenario where multiple users share the same physical GPU, leveraging NVIDIA's Multi-Instance GPU (MIG) technology. Each user operates within a dedicated GPU Instance (GI), which is designed to provide isolation from other GIs. We consider the isolation properties of MIG under different deployment configurations. Each configuration carries distinct implications for isolation and potential attack surfaces.

First, we consider a *containerized* environment in which each container is assigned a dedicated GI. Containers share the same host GPU kernel driver, but lack direct access to GPU MMIO regions and cannot modify the shared kernel driver. Second, we consider a *virtualized* environment where each Virtual Machine (VM) is assigned a dedicated GI using SR-IOV [64] and VFIO [63], assuming a KVM-based hypervisor. In this setup, we assume an adversary with full control over the guest, including the ability to freely modify guest kernel drivers.

For both configurations, we aim to answer the following research question: *Can an attacker with code execution in a container or a VM on GI 0 infer information about the workload running on GI 1 (i.e., a co-located GI)?*

Experimental setup. Our experimental setup uses the open-source kernel driver [41] (version 570.133.07) for containerized environments and NVIDIA AI Enterprise 6.2 [40] (version 570.133.20) with proprietary drivers for vGPU-based ones. We conducted our experiments on an NVIDIA A30 (Ampere) GPU connected via PCIe 4.0 to a host equipped with an Intel i9-12900K CPU and 132 GiB of RAM. We later replicated the experiments with consistent results on an NVIDIA H100 (Hopper) GPU. We configure MIG with two GIs, each provisioned with 12 GiB of VRAM and 28 SMs.

4 Overview

In this section, we provide a high-level overview of our *MIGraine* side-channel attack, which enables an adversary on one GI to fingerprint ML/LLM workloads running on a co-located GI. We first describe the core technical primitive that enables the attack, then outline the end-to-end attack flow, and conclude with a roadmap that frames our subsequent contributions as the scientific validation of our attack.

Building the primitive. At the core, *MIGraine* is based on the observable latency when handling UVM page faults. As we will show, the time needed for the GPU and driver stack to serve each page fault is influenced by concurrent activity on a co-located GI. Specifically, a workload running on one GI affects the page fault latency on the other GI due to contention over shared hardware and software resources.

The side channel. The three core steps of our attack are depicted in Figure 2. Assuming the attacker has control over GI 0, they run a

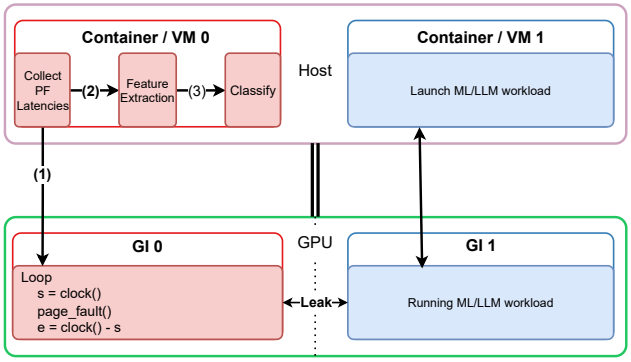


Figure 2: High-level overview of our *MIGraine* attack.

simple GPU kernel that continuously collects timings of page faults on their own pages (Step 1). The victim runs a separate (benign) workload such as a ML or LLM model on GI 1, which is co-located on the same physical GPU with MIG enabled. All the timings collected by the attacker are grouped into time windows and a vector of statistical features (e.g., mean, standard deviation, percentiles) is extracted from each window (Step 2). Finally, the feature vectors are fed into a classifier (XGBoost [9]), which predicts the identity of the victim's workload (Step 3). We analyze the side channel in detail in Section 8.

Root cause analysis. In the remainder of the paper, we provide the foundational analysis to explain and validate our findings. We first perform a deep dive into NVIDIA's UVM driver (Section 5), reverse engineering the page fault handling process and identifying components shared across GIs (e.g., the fault buffer). We then systematically characterize cross-GI interference in face of different GIs contending on such shared components. Specifically, we study how (i) page faults on one GI impact various memory access patterns on a neighboring GI and (ii) viceversa, that is how a given workload running on one GI impacts the page fault latency on another GI (Section 6). We later use the former form of interference to build a contention-based covert channel (Section 7) and the latter to build our contention-based side channel (Section 8).

Scope of analysis. The analysis in Sections 5–8 focuses primarily on **containerized MIG deployments**, where containers share the host kernel driver. In Section 9, we then investigate how these findings translate to **virtualized (vGPU) environments**, examining first the fundamental differences in memory isolation enforcement between the two deployment models and then the portability of our side-channel attack to vGPU environments.

5 UVM Driver Internals

We first analyze the UVM kernel driver to gain deeper knowledge of the inner workings of memory management, with a particular focus on page fault handling due to its complex interaction between the host and the GPU.

5.1 UVM Channels

Channels allow user-space programs (e.g., CUDA) to submit work without constantly invoking system calls [37], and enable the kernel driver to submit work asynchronously. Through our analysis of the UVM driver, we found that it distinguishes between *user* and *kernel* channels. The latter are also defined as *UVM Channels* when they are used by the UVM driver.

Each GI maintains a dedicated Push Buffer (PB) in the UVM driver allocated in Host RAM, implemented as a contiguous memory region from which instructions are fetched by the PBDMA engines. We further observed that UVM Channels are backed by Copy Engines, and by default are privileged, which means that they can address physical memory and execute other privileged methods [44, 47].

However, UVM also employs two special types of channels: (1) *proxy channels* which are privileged channels used by the guest UVM driver in vGPU environments where commands are pushed by the guest and intercepted by the hypervisor or directly handled by the vGPU component inside the GSP on Ada+ architectures [52]; and (2) *confidential computing channels* which are dedicated to Confidential Computing, a Trusted Execution Environment (TEE) mechanism introduced on Hopper+ architectures [16].

5.2 UVM Page Table Management

We identified that page tables are managed jointly by the UVM and RM drivers. Each GPU channel defines its own root page table directory, which the GMMU uses to resolve virtual memory addresses. We further found that there are two sets of page tables namely, those for kernel mappings that are accessed by the kernel-level drivers (i.e., UVM and RM drivers), and those for user mappings that are accessed by user-space applications (i.e., CUDA programs).

To understand how MIG enforces isolation in containerized setups, we analyzed the physical address space partitioning scheme. Our analysis revealed that each GI receives a disjoint partition of the physical address space, with page tables configured such that no virtual address in GI X can map to a physical address belonging to GI Y. To verify that memory isolation relies exclusively on page table configuration, we extended `gpu-tlb` [73] to dump and extract page tables from MIG-configured GPUs. We then modified the UVM kernel driver to enable arbitrary PTE updates, allowing us to map a virtual address of a CUDA program running on GI 0 to a physical address belonging to GI 1. This successfully reads GI 1's memory, confirming the absence of additional isolation. As we will demonstrate in Section 9.2, this is not the case in vGPU setups.

5.3 UVM Page Fault Handling

We discovered that the GPU records fault information in a fault buffer, which is used by the kernel driver to handle page faults. The RM driver allocates this buffer in system memory and maps it to both the CPU (Kernel) and the GPU via `nvGpuOpsInitFaultInfo`. It also maps the Fault Buffer GET and PUT MMIO registers to the CPU, which track respectively the last fault entry processed by the CPU and the next entry to be written by the GPU. Our analysis further revealed that, in non-vGPU setups, this fault buffer is **shared** across GIs.

The sequence below illustrates what happens when the GPU first executes a load instruction targeting a memory address allocated with `cudaMallocManaged`:

(1) The GPU's GMMU detects that the virtual address (VA) does not have a valid translation. It writes fault information into the fault buffer at index PUT, such as the faulting VA, instance pointer, type of access, engine type (in this case GR), and increments the PUT register. Additionally, it also stalls the current GPU process and its associated channels.
(2) The GPU then raises an interrupt to notify the host. The RM intercepts it and delegates it to the UVM driver's top-half Interrupt Service Routine (ISR), which in turn schedules the bottom-half ISR to service the fault.
(3) The UVM driver reads the PUT value. Since it differs from the cached GET value, it parses the data in a `uvm_fault_entry` structure by reading from the fault buffer at index GET.
(4) The UVM driver updates the GET value to the most recently parsed entry, informing the GPU that the entry has been read successfully.
(5) UVM keeps track of instance pointers (GPU processes) in a Red Black tree, that is used in this step to find the respective *User Channel* to get the right virtual address space structure and the GI that triggered the page fault.
(6) The driver allocates a 64 kB page on the GI and updates the GPU page tables (PDEs or PTEs) accordingly. If a corresponding host page is present, the driver also copies its contents to the GPU memory.
(7) Once the page has been allocated and the page tables are updated, the driver invalidates the TLB entry and replays the faulting instruction.

When the page is subsequently accessed by the Host, the driver invalidates the corresponding GPU TLB entry and page table mappings (PTE/PDE). Upon re-access by the GPU, the driver only needs to rewrite the PTE/PDE, copy the page from the Host, and replay the instruction, without re-allocating the page on the GPU. At any given moment there could be multiple fault entries that belong to different GIs, but ultimately they are all serviced from the same bottom-half ISR that is **shared** between GIs.

Note that this fault is referred to as a *Replayable Fault* on Graphic Engines [43]. Other types of faults can occur on CEs and PBDMA engines; these are classified as *Non-Replayable Faults* and are serviced by both UVM and RM.

Takeaway 1

In containerized environments, the fault buffer and kernel page fault handling are not partitioned by MIG, yielding **shared resources across all GIs**.

6 Cross GPU Instance Interference

In the previous section, our analysis identified shared resources and complex GPU-Host interactions during page fault handling, which can result in potential vectors for cross-GI interference. Beyond the kernel-level contention identified in Takeaway 1 (i.e., the shared fault buffer), the page fault handling path involves multiple hardware components: the PCIe bus for host-GPU communication, GPU engines for privileged operations (TLB invalidations, page table setup), and other components used for stalling the current process and writing fault information. To systematically characterize cross-GI interference, we must therefore test contention from both software and hardware resources.

This section examines the extent to which contention on the UVM page fault handling path in one GI degrades the performance of different workloads on a co-located GI and viceversa. First, we systematically measure latency on GI 0 workloads targeting two separate memory subsystems: (1) *GPU memory accesses that require Host communication* and (2) *GPU internal memory accesses*. Specifically, we evaluate whether repeated page faults (i.e., a page fault storm) running on GI 1 can affect the execution time of programs running on GI 0. The storm intensity is controlled by configuring the number of threads and blocks, allowing us to scale page fault frequency precisely. To further validate that observed performance degradation stems specifically from page fault handling rather than other contention sources, we conduct additional experiments with alternative interference patterns. We separately evaluate the impact of a compute-intensive ML workload (VGG model [61] training on the CIFAR-10 dataset [26]) and PCIe bus saturation through continuous host-GPU data transfers using `cudaMemcpy`. The latter is included because prior work [32] identified the (direct) PCIe interface as a shared resource between MIG instances and thus a cross-GI contention source.

Next, we change the direction of the experiments, by measuring the latency of page faults on GI 0, while GI 1 runs the workloads. To stabilize latency measurements, we set the GPU clock frequency to a fixed value (i.e., 600 MHz, but we observed similar results across the board).

6.1 Impact of Page Faults on Target Workload

GPU-Host memory accesses. We now examine how page fault interrupts affect a target containerized workload's memory operations requiring host communication. Our analysis focuses on two common scenarios: direct access to host RAM and UVM-managed memory that triggers page faults. To test host RAM accesses, we allocate pinned memory using `cudaHostAlloc` and `cudaHostGetDevicePointer`, then create a GPU kernel that measures the latency of load (`ld.cv`) and store (`st.wt`) operations across 256 distinct cache lines. To test UVM memory accesses, we develop a similar kernel that loads from a large `cudaMallocManaged` region, randomly accessing pages to hinder UVM prefetcher optimizations. Our UVM test effectively measures contention between *concurrent* page faults across GI.

Figure 3 presents our results on the A30. As shown in the figure, our results evidence moderate-to-significant cross-GI interference. In particular, Host RAM accesses incur up to 100,000 clock cycles of (moderate) overhead when the co-located GI performs PCIe data transfers, consistent with prior (direct) PCIe contention results [32].

In contrast, UVM unmapped memory accesses on GI 0 suffer much more significant performance degradation, experiencing delays of up to millions of clock cycles during page fault storms with a high number of threads on GI 1. We attribute this severe

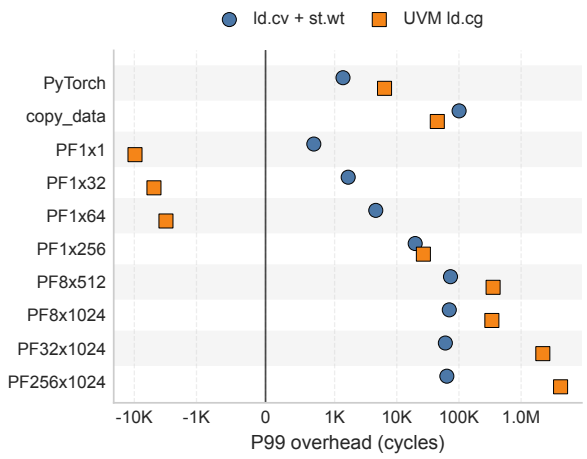


Figure 3: 99th percentile overhead (in clock cycles) for two GPU kernels on GI 0 under various interference workloads on GI 1, measured relative to an idle baseline. Each row corresponds to one interference workload on GI 1 and reports two independent probe kernels on GI 0. Blue markers show a kernel accessing pinned Host RAM via the PCIe bus (`ld.cv+st.wt`), while orange markers show a kernel accessing unmapped UVM memory that triggers page faults (`UVM ld.cg`). Interference sources include: PCIe-intensive data transfers (`copy_data`), ML model training (`pytorch`), and page fault storms with varying block × thread.

performance degradation to bottlenecks in the GPU's shared fault-handling mechanism. The kernel driver can only service a limited number of page faults concurrently and critical resources such as the fault buffer are shared among all GIs. Consequently, the high volume of page faults from GI 1 creates a bottleneck, delaying the servicing of UVM faults generated by GI 0. Interestingly, page fault storms involving fewer threads appear to speed up page fault handling across GPU instances. This counterintuitive behavior likely arises because the interrupts triggered by the page fault storm awaken the ISR, while the fault entry is asynchronously added to the fault buffer. Subsequently, when the interrupt handler processes the buffer, it handles GI 0's fault without the overhead of waking the kernel's fault-handling path, as GI 1 has already done so.

Takeaway 2

Shared kernel-level resources in the UVM fault handling path cause cross-GI contention in containerized deployments, significantly increasing memory access latency for UVM-intensive workloads.

GPU memory accesses. We demonstrated that page fault storms affect memory accesses that require host communication. We now investigate their impact on cache behavior and VRAM access latency by developing multiple specialized GPU kernels. These single-thread kernels measure the latency of distinct 256 cache lines accesses, when varying the memory access operations. Our kernels

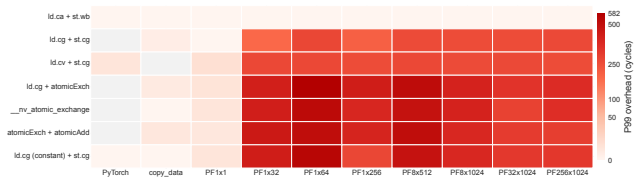


Figure 4: 99th percentile overhead of seven GI 0 memory-access probes under co-located interference workloads on GI 1, measured relative to an idle GI 1 baseline. Probe kernels cover an L1-sensitive `ld.ca+st.wb` configuration, L2-oriented `ld.cg+st.cg` and `ld.cv+st.cg` accesses, mixed cache+atomic accesses (`ld.cg+atomicExch`), pure atomic operations (`__nv_atomic_exchange` and `atomicExch+atomicAdd`), and constant-cache loads with L2 stores (`ld.cg(constant)+st.cg`). Rows are GI 0 probes; columns are GI 1 interferers.

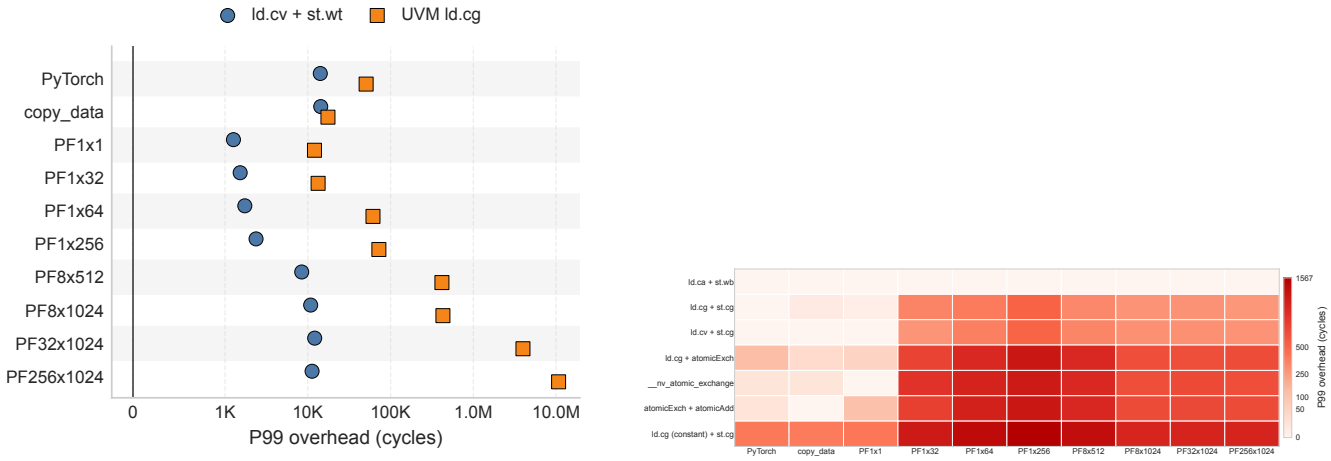
systematically varied across two key dimensions: (1) the load instruction cache hints (`ld.ca`, `ld.cg`, `ld.cv`), and (2) the store instruction cache hints (`st.cg`) or atomic operations (`atomicExch`, `atomicAdd`, `__nv_atomic_exchange`).

Figure 4 presents our results on the A30. As shown in the figure, the `ld.ca+st.wb` probe, which is designed to favor L1-cached accesses in this configuration, shows no measurable interference. In contrast, other memory access types that target L2 cache, Constant cache, and VRAM all exhibit increased latency under page fault storms, with atomic operations showing the highest sensitivity. The 99th percentile overhead heatmap reveals worst-case interference, with some atomic operations experiencing up to 500 cycles of additional latency under intensive page fault storms (1×64, 8×512). The consistent pattern across multiple memory operation types demonstrates that page fault handling on one GI creates contention on internal GPU memory paths that are not fully isolated by MIG.

Takeaway 3

The latency of L2/Constant cache and VRAM accesses on one GI increases when another GI executes a page fault storm. This shows that page faults contend not only on shared hardware (PCIe) and kernel resources, but also on on-GPU memory accesses.

H100 Results. Figure 5 presents our cross-GI interference analysis results on the H100. Consistent with the A30 results, in Figure 3 and Figure 4, GPU-Host and UVM-backed probes exhibit visible interference under page-fault storms, while GPU-internal probes show limited impact on L1-oriented accesses and measurable contention on L2-, constant-cache-, atomic-, and VRAM-related accesses. Since, we consistently observed similar results across both GPU generations throughout our evaluation, for brevity, we focus on the A30 results hereafter.



(a) *GPU-Host and UVM-backed memory probes on H100.*

(b) *GPU-internal memory probes on H100.*

Figure 5: H100 cross-GI interference results.

6.2 Impact of Target Workload on Page Faults

After establishing that page fault storms affect both GPU-Host memory accesses (Takeaway 2) and GPU-internal memory operations (Takeaway 3), we invert the direction of our analysis: we test whether the page fault latency on one GI changes based on the memory access patterns and resources used in another GI.

First, we systematically measure how different GPU kernels affect the GPU page fault latency on a co-located GI. The setup involves modifying memory-intensive GPU kernels from Section 6.1 to run on an infinite loop on GI 1 (using 8 blocks of 256 threads). Simultaneously, on GI 0, we measure page fault latency by accessing unmapped memory pages and recording the clock cycles required to complete each page load operation. These kernels target the L2 cache, Constant cache, VRAM and Host RAM using various cache hints and atomic operations. For brevity, we did not include the GPU kernels involving concurrent page faults because we already presented the results of this scenario in Figure 3.

Figure 6 presents our results. As shown in the figure, our results reveal that all GPU kernel types affect page fault latency, albeit to varying degrees. The Host RAM access kernel yields the most significant overhead due to its intensive PCIe traffic, while GPU-internal memory operations reveal more moderate effects.

To further analyze the impact of PCIe bus traffic on page faults, we launch a stress test on GI 1, repeatedly invoking `cudaMemcpy` with varying data sizes, while measuring page fault latency on GI 0. The results, shown in Figure 7, reveal a clear correlation: as PCIe throughput increases, so does the page fault handling latency on the adjacent GI. In other words, this shows that page fault handling latency is a proxy metric to (indirectly) measure PCIe contention, unlocking the ability to exploit PCIe-based side channels without resorting to direct (possibly unavailable/disabled) PCIe interfaces such as `nvmlDeviceGetPcieThroughput`, abused in prior work [32].

In Section 8.2, we conduct a more detailed analysis of how GPU-internal memory operations targeting L2 cache, Constant cache, and VRAM memory accesses contribute to page fault latency variations, isolating their effects from PCIe-related contention.

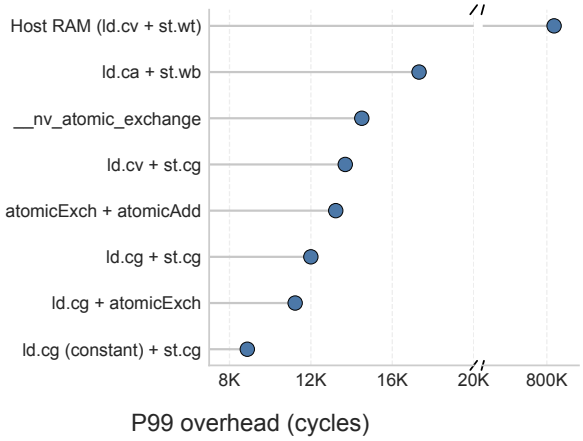


Figure 6: 99th percentile overhead of page fault latency on GI 0 under various GPU kernels running on GI 1. Y-axis labels report the load/store or atomic instruction pattern used by each kernel. Host RAM accesses cause the highest overhead due to PCIe contention, while GPU-internal operations show mild but consistent interference.

Takeaway 4

GPU workloads on one GI affect page fault latency on co-located GIs. PCIe traffic is an important source of this contention, albeit other hardware resources also contribute.

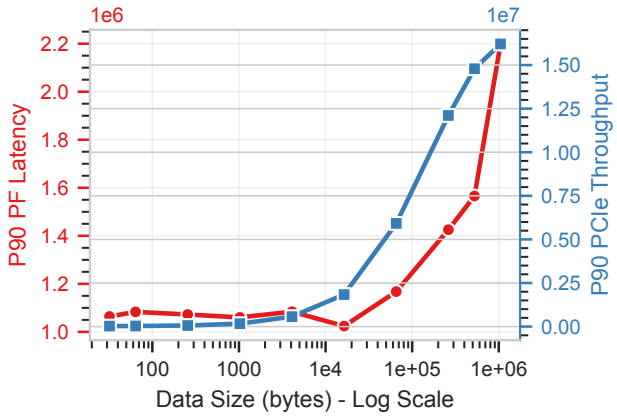


Figure 7: Combined view of the 90th percentile page fault latency on GI 0 (left axis, red) and PCIe throughput on GI 0 measured via `nvmlDeviceGetPcieThroughput` (right axis, blue) while varying `cudaMemcpy` traffic on GI 1.

7 Page Fault Covert Channel

Using the observations made in the previous section, we can construct a covert channel across GIs.

7.1 Primitive

For demonstration purposes, we focus on our first contention analysis scenario—i.e., memory access latency during page fault storms. Two approaches are possible: (1) exploiting increased latency in UVM-backed loads during page fault storms (Takeaway 2), or (2) exploiting increased latency in GPU memory loads during such storms (Takeaway 3). The former UVM-based approach can potentially encode more information in the signal, but also incur extreme delays. For this reason, we selected the latter GPU-only memory approach. Algorithm 1 and Algorithm 2 detail our (simple) sender and receiver implementations (respectively). In short, the sender encodes a '1' bit by launching a page fault storm on its GI, while a '0' bit is encoded by idling for a fixed time. The receiver continuously measures the latency of a GPU memory-access kernel (e.g., L2 loads) on its GI, and decodes a '1' bit if the 99th-percentile latency exceeds a predefined threshold, and a '0' bit otherwise.

7.2 Evaluation

To evaluate our covert channel, we repeatedly transmitted a series of messages from the sender to the receiver, recording the resulting transmission rate and error rate. The hyperparameters, such as the number of page faults for the sender (N_p), the number of latency samples for the receiver (N_l), and the detection threshold (τ), were manually tuned to ensure reliable synchronization between the sender and receiver. Our results show that, when targeting error-free communication (i.e., an error rate of 0%), we can successfully transmit one bit every 10ms.

Algorithm 1 Covert channel sender

Require: $target\_time$: nanoseconds to sleep, N_p: number of page faults to execute
  for $i = 0$ **to** $sizeof(message) - 1$ **do**
    if $message[i] = 1$ **then**
      $page\_fault\_storm(N_p)$
    else
      $sleep(target\_time)$
    end if
  end for

Algorithm 2 Covert channel receiver

Require: N_l: number of latencies to collect for each bit, τ: detection threshold
  for $i = 0$ **to** $sizeof(message) - 1$ **do**
    $latency\_values \leftarrow get\_latencies\_gpu\_kernel(N_l)$
    $p99\_latency \leftarrow p99(latency\_values)$
    if $p99\_latency > \tau$ **then**
      $message[i] = 1$
    else
      $message[i] = 0$
    end if
  end for

8 Page Fault Side Channel

In Section 6.2, we showed that different workloads induce measurable variations in UVM page fault latency (Takeaway 1 and 4). In this section, we show that an attacker can exploit this capability to craft a *MIGraine* side-channel attack primitive, able to fingerprint diverse ML and LLM workloads running on a co-located GI.

8.1 Primitive

We first seek to establish whether page fault latency variations can reliably distinguish between different types of real-world workloads running on a co-located GI. Our measurement methodology employs a memory access pattern that deliberately triggers and times page faults on GI 0 in order to monitor various workloads executing on GI 1. We launch the measurement GPU kernel with 32 threads. Each thread performs a single global memory load pointing to a page that is not yet mapped on the GPU and measures the latency incurred to handle the resulting page fault. Crucially, we do not initially map the accessed pages on the host, avoiding page migration operations and ensuring each iteration triggers a fresh page mapping and allocation on the GPU. At each iteration, we trigger a new page fault by accessing a page that is not yet mapped on the GPU.

We collect all the timing measurements on GI 0 while executing a diverse set of workloads on GI 1. These include: (1) an *Inactive* state to serve as a baseline; (2) *PyTorch* training a MobileNetV2 [58] model; (3) *vLLM* [27], a widespread framework for LLM inference, running the Phi-3-Mini [2] model; and (4) *cuDF* [56], a GPU DataFrame library, performing UVM-intensive join operations [57] that extensively utilize unified memory. For each workload, we compute the mean of the top 20% highest-latency samples (tail

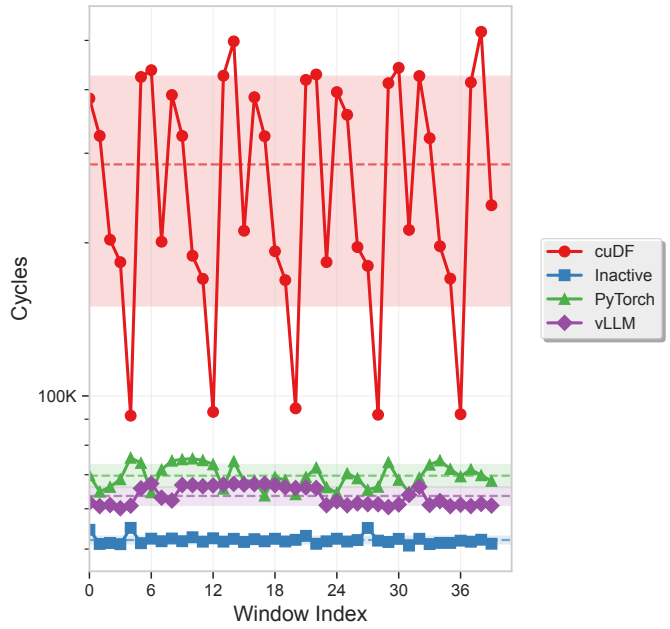


Figure 8: Tail-average latency of page fault handling on GI 0 under different workloads on GI 1. Each point on the log-scale y-axis represents the mean of the top 20% highest-latency samples within the sample window.

average) within consecutive sample windows to capture the characteristic latency distribution while filtering transient noise. Figure 8 presents our results.

As shown in the figure, our experimental results reveal distinct latency signatures for each workload type. The UVM-intensive cuDF workload exhibits the highest tail- average latency (100K–500K cycles), consistent with our findings in Section 6.1: concurrent page fault handling on shared kernel resources creates significant contention. The PyTorch and vLLM workloads show moderate latency increase, i.e., 70K and 65K cycles respectively, compared to the inactive baseline of 50K cycles. These distinguishable patterns across windows indicate that page fault handling latency on GI 0 is consistently influenced by the type of workload running on the co-located GI 1, enabling cross-GI page fault side-channel-based fingerprinting attacks.

8.2 Evaluation

ML fingerprinting. To evaluate our attack end-to-end, we first extend our testbed by training different machine learning models on GI 1 while measuring page fault handling latency on GI 0. We choose VGG [61], ResNet [18], DenseNet [20], and MobileNetV2 [58] because they are standard CNN families with distinct compute and memory footprints while fitting comfortably within a single GI. The goal is to assess whether page fault latency can distinguish realistic workloads drawn from commonly used model families. As in prior fingerprinting settings [32, 73], the attacker trains the classifier offline by profiling candidate workloads on a controlled system with the same GPU, driver, and MIG configuration as the target deployment. At attack time, the attacker only collects timings from its own GI; retraining is only needed after substantial platform changes, such as GPU generation, driver, CUDA/runtime, or major framework updates. To show that accurate fingerprinting is possible based solely on page fault latency measurements, we train an XGBoost classifier [9] on statistical features extracted from 10,000-sample windows of timing measurements. including standard deviation, percentiles (P90, P95, etc.), mean, and median. We use 80% of windows for training and 20% of them for testing, achieving an overall accuracy of 89%, as shown in Table 1. We also observe comparable classification accuracy when the models are running constant inference, under both static and dynamic GPU frequency scaling configurations.

Table 1: Classification results for ML fingerprinting.

Model	Precision	Recall	F1	#
DenseNet [20]	0.79	0.82	0.80	40
MobileNetV2 [58]	0.95	0.97	0.96	40
ResNet [18]	0.95	0.88	0.91	40
VGG [61]	0.88	0.88	0.88	40
Accuracy	0.89			

Table 2: Classification results for LLM fingerprinting.

Model	Precision	Recall	F1	#
GPT2-Large [55]	0.98	0.98	0.98	321
GPT2-Medium [55]	0.98	0.97	0.98	321
OLMo-1B [15]	0.85	0.91	0.88	321
OPT-125m [69]	0.99	1.00	1.00	321
Phi-3-Mini [2]	0.96	0.98	0.97	321
Qwen2-1.5B [54]	0.96	0.90	0.93	321
StableLM [62]	1.00	1.00	1.00	321
Starcoder2-3B [29]	1.00	0.99	0.99	321
TinyLlama-1B [68]	0.92	0.90	0.91	321
Accuracy	0.96			

LLM fingerprinting. Next, we further extend our testbed to classify Large Language Models (LLMs) under inference workloads. We use vLLM [27] to run the models and evaluate all vLLM-supported, license-free models that fit within the victim GI's VRAM budget, reflecting the operational constraints of a single MIG slice and avoiding special victim configurations such as weight offloading [32]. Each LLM is executed in a continuous loop using a fixed prompt to stabilize the steady-state inference regime and isolate model-dependent behavior from prompt-dependent variance. As in the ML setting, the attacker builds the classifier offline by profiling candidate models on a controlled system matching the target deployment. To strengthen the signal, the attacker times their UVM page faults from both the GPU and host sides, yielding two features per sample. Our dataset employs a sliding window approach with 40,000 samples per window, a 3,000-sample step size, and a 80/20 train/test window split per model. Table 2 presents our results, with our LLM classifier achieving 96% overall accuracy.

Understanding fingerprinting sources. To better understand the sources of cross-GI interference that enable our *MIGraine* attack, we systematically evaluate the contribution of two distinct contention sources: PCIe traffic and on-GPU cache coherence operations.

First, we test whether PCIe bandwidth alone could serve as a reliable fingerprinting signal. Using the `nvmlDeviceGetPcieThroughput` interface, which reports bidirectional PCIe bandwidth usage between host and GPU, we train a simple XGBoost classifier on the same LLM workloads from Table 2. Our results confirm that PCIe traffic patterns alone can fingerprint LLMs with comparable accuracy, demonstrating that PCIe usage alone can be used to fingerprint LLM workloads. However, as we will see next, a direct PCIe-based side channel is significantly less resilient to noise than our page fault latency-based approach.

To test the resilience of our page fault side channel to PCIe noise, we run the LLM fingerprinting experiment again while concurrently generating synthetic PCIe noise (through `cudaMemcpy` transfers) of varying intensities: low (1KB–1MB in 10–20ms intervals), medium (1–10MB in 5–10ms intervals), and high (10–50MB in 1–5ms intervals). We then evaluate our classifier's performance in three distinct scenarios. First, we train the classifier exclusively on noise-free data and test it against each of the three noise levels. The accuracy degrades significantly from the baseline of 96% to 58% (low noise), 15% (medium), and 20% (high), demonstrating that a noise-unaware model is not robust. Second, we train and test separate classifiers for each noise level (e.g., training on medium noise, testing on medium noise). In this case, the accuracy remains high, around 96% across all levels. This shows that the fingerprinting signal is still prominent even when substantial noise is present. Finally, to create a truly robust classifier, we train it on a composite dataset containing samples from all noise levels. When testing against a similarly mixed dataset, the classifier achieves an accuracy of 96%. This result proves that our side channel is highly resistant to varying levels of PCIe noise. In contrast, using the same robust mixed-noise training with a direct PCIe-based side channel causes classification accuracy to drop to approximately 60% in presence of noise. We use synthetic `cudaMemcpy` traffic here to sweep the level of PCIe contention in a controlled manner. We use this setup as a robustness stress test. Accordingly, the per-noise-level classifiers should be interpreted only as diagnostic upper bounds showing that the signal remains separable within each controlled noise regime, whereas the mixed-noise model evaluates whether a single classifier can remain effective across multiple contention levels.

To isolate the contribution of on-GPU memory operations from PCIe traffic, we perform a controlled experiment using the memory-intensive GPU kernels from Section 6.1. These kernels, which target L2 cache, Constant cache, and VRAM using various cache hints and atomic operations, are executed on GI 1 while we measure page fault latency on GI 0 and the host. We first verify that these kernels generate minimal PCIe traffic by attempting to classify them using a direct PCIe-based classifier. While this strategy achieves 70% accuracy, it struggles to distinguish between cache-intensive operations such as `ld.cv+st.cg` and `ld.ca+st.wb` (16 and 25% F1-score respectively), indicating that PCIe monitoring alone cannot fully capture cache-level memory behavior. However, when we apply our page fault latency-based classifier to the same workloads, it successfully differentiates all GPU kernel variants with over 80% accuracy (Table 3), including the cache-intensive operations that the PCIe-based approach failed to distinguish.

Table 3: Classification results for different GPU kernels.

Memory Accessed | Instructions	F1	#
`L1|ld.ca+st.wb`	0.88	324
`L2|ld.cg+st.cg`	0.77	324
`L2|ld.cg+atomicExch`	0.83	324
`L2|nv_atomic_exchange`	0.64	324
`L2|atomicExch+atomicAdd`	0.94	324
`Constant+L2|ld.cg+st.cg`	0.87	324
`L2+VRAM|ld.cv+st.cg`	0.71	324
Accuracy	0.81	

9 vGPU Applicability

Our analysis has thus far focused on containerized MIG deployments, where, for instance, a shared kernel driver presents a major source of cross-GI contention. This section extends our investigation to virtualized (vGPU) environments to analyze their differences and determine the portability of our findings across both deployment models.

9.1 vGPU Environment Details

Setup. Research on Ampere vGPU environments is challenging due to the proprietary nature of the ecosystem. Both the guest RM driver and the hypervisor vGPU plugin are closed-source. The guest RM driver is only available in binary form, as the open-source kernel modules do not yet support vGPU functionality on Ampere [51, 52]. However, access to the proprietary UVM driver source code, together with the open-source version available for newer architectures, allows us to reverse engineer the behavior of specific memory management operations on Ampere's vGPU implementations.

On a vGPU setup, certain privileged operations initiated by the guest kernel driver, such as physical-address-based memory operations (e.g., `memcpy` or `memset`), TLB invalidations, and fault replays, are restricted by the GPU. Consequently, to execute these operations, the driver relies on proxy channels (Section 5.1). There, a per-VM process known as the vGPU plugin validates and forwards them to the host RM Driver, which can communicate directly with the GPU. The scheme in Figure 9 depicts a high-level overview of the components involved. To support page migrations despite limits on physically addressed `memcpy` and `memset`, the UVM guest driver builds a linear virtual mapping of system and video memory.

9.2 vGPU Memory Isolation Analysis

Given that containers share the host kernel and lack direct hardware access, page table management is exclusively performed by the kernel driver (Section 5.2). Hence, an attacker cannot arbitrarily modify page tables without first compromising privileged code.

In contrast, our vGPU threat model (Section 3) assumes an attacker with root access inside a guest VM, including the ability to modify guest kernel drivers at will. This raises a critical concern because a malicious guest might be able to remap its own page

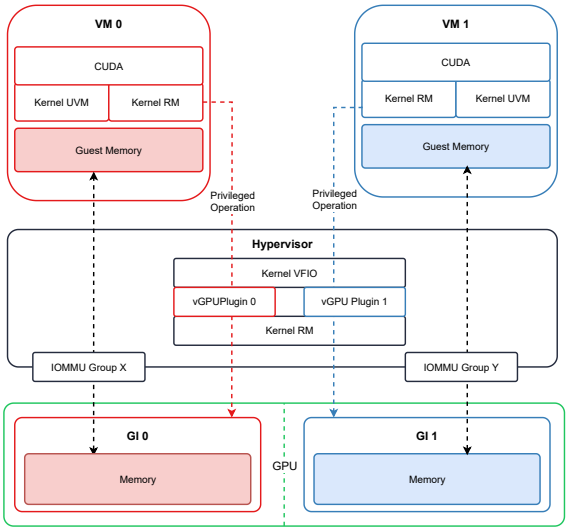


Figure 9: High-Level overview of a vGPU+MIG setup with two VMs, each connected to their own GPU Instance. Each GPU Instance uses a separate IOMMU Group for DMA memory transfers with the Guest VM.

tables to access another GI's physical memory without proper isolation (i.e., if the isolation is enforced exclusively in software by the guest driver, the hypervisor, or both). To better understand the isolation architecture, we now turn our attention to the vGPU memory management architecture. Specifically, we analyze how page tables are structured, how the GMMU performs address translation in virtualized setups, and what role proxy channels (Section 5.1) play in mediating privileged memory operations.

vGPU address translation. To understand the differences in address translation within vGPU environments, our first step was to extract and analyze guest page tables from the hypervisor's perspective. We implemented a memory dumper that specifies the target physical address through the `NV_PBUS_BAR0_WINDOW` BAR0 register [37] and reads 1MiB blocks from `NV_PRAMIN`, iteratively shifting the base address to cover the entire memory space. We then extended an existing open-source tool, `gpu-tlb` [73], to parse vGPU page tables from the resulting memory dump. To validate our findings and understand the addressing scheme, we instrumented the UVM guest driver to log page table updates, aggregating them with the data extracted from the dump.

Our findings reveal that each VM performs page table updates on their view of GPU memory, which we refer to as *GPU guest physical memory*. Consequently, two VMs see similar guest physical regions that are in use by the GPU. Moreover, the VRAM physical addresses found in GPU memory dumps (such as those in PTEs and PDEs) are actually guest physical addresses. We refer to these as *VRAM Guest Physical Addresses* (VGPAs). Each VGPA is translated to a *VRAM Physical Address* (VPA) using a GI-specific offset. For instance, in our configuration with two 12 GiB GIs, all VGPA addresses are translated to VPA by adding `0x4000000` to GI 0 guest addresses and `0x4000000+0x2f8000000` to GI 1 addresses. According to the open GPU kernel driver, the GPU internally has a memory management unit (*VMMU*), which is responsible for the translation between VGPA and VPA [48].

Testing vGPU page table isolation. With knowledge of vGPU address translation in mind, we next evaluated whether page table manipulation could allow one GI to access another GI's memory. To this end, we added a custom ioctl to the guest UVM driver to issue controlled PTE updates and tested multiple remapping scenarios. These experiments showed that the vGPU plugin mediates guest-visible PTE updates and enforces software checks on the requested VGPAs. More importantly, even when we edited page tables directly from the hypervisor, the subsequent GPU accesses still failed to read memory belonging to another GI, instead triggering an internal GMMU error. This result is consistent with the two-stage translation described above and indicates that vGPU memory isolation is not enforced purely in software: cross-GI VRAM isolation ultimately also relies on hardware checks in the VMMU/GMMU path.

During our analysis, we also found that malformed guest-issued PTE updates can crash the vGPU plugin; NVIDIA confirmed this as a high-severity vulnerability, assigning it CVE-XXXX-XXXXX.

Takeaway 6

Containerized MIG enforces memory isolation purely via software-managed page tables; vGPUs use both software- (hypervisor) and hardware-level (VMMU) checks.

9.3 Evaluation

We next evaluate whether our side-channel attack remains effective in vGPU deployments, despite their per-VM kernel drivers and hypervisor-mediated operations (Takeaway 6).

To assess the portability of our side channel, we replicated the cross-GI interference experiments from Section 6 in a vGPU environment. We observed that, for non-UVM GPU kernels, the resulting latency profiles matched those from our containerized experiments (Figure 4). However, the extreme latency we observed with the UVM-based kernel (Figure 3) did not manifest in the vGPU setup. This confirms that kernel-level software resources, such as the fault buffer, are properly isolated by the vGPU software stack.

Despite this isolation of such software-level resources, we found that our our side-channel attack remains effective. To support this claim, we tested the attack with one VM repeatedly measuring page fault latency while a second VM ran various PyTorch models in inference mode. The models remained clearly distinguishable with over 90% accuracy, demonstrating the side channel's continued effectiveness even in a vGPU deployment configuration.

Takeaway 7

Despite vGPU's proper isolation of kernel-level software resources and hardware-enforced memory isolation with the VMMU, our side channel's fingerprinting capabilities remain effective. This is because our attack exploits black-box contention on PCIe traffic and GPU memory operations, which are not isolated by either containerized or vGPU architectures.

10 Limitations

We identified that the page fault latency is influenced by multiple competing factors that are difficult to completely isolate, including PCIe bus contention, kernel-level software resources, and internal GPU memory accesses. This complexity makes it challenging to attribute latency increases to specific architectural components or identify the dominant contributing factor in real-world scenarios. At the same time, this evidences *MIGraine*'s blackbox nature, which does not require deep knowledge of the underlying architecture to successfully mount attacks.

Although we successfully fingerprinted ML models and LLMs running across GIs, several variables could hinder classification accuracy. These include different hyperparameters used during ML model training, varying frequencies of prompts sent to LLMs, and the complexity introduced by having three or more GIs concurrently running different workloads. In such scenarios, more traces would likely be needed to effectively fingerprint separate models.

11 Mitigations

To mitigate the page-fault side channel, NVIDIA could consider implementing stricter (MIG) isolation mechanisms between GIs, such as adopting a zero-sharing policy in the hardware/software page fault handling path. A stop-gap mitigation is to disallow UVM in multi-tenant environments or otherwise implement rate-limiting mechanisms for UVM page faults. Nonetheless, these mitigations cannot alone tackle other possible forms of cross-GI interference, which may still lead to practical contention-based side-channel attacks across GIs.

12 Related Work

Zhang et al. leveraged the shared L3-TLB to fingerprint ML models across MIG GPU instances [73], but their classification worked only during the model startup phase and requires learning TLB-specific eviction sets for each GPU generation. Miao et al. [32] explored PCIe contention on MIG setups without vGPUs, using the `nvml DeviceGetPcieThroughput` interface to identify LLM workloads across GPU instances. However, their method was evaluated only in scenarios involving weight offloading, where model weights must be continuously exchanged between the host and the GPU.

Related work also studied virtualized GPUs without MIG. Side et al. [60] exploited PCIe contention in standard time-sliced vGPU deployments to perform website fingerprinting, and Jin et al. [24] exploited GPU TLBs to mount cross-VM side-channel attacks in the same setting. These works are complementary to ours but study a fundamentally different sharing model: tenants interfere because they time-share one GPU, whereas we study vGPU+MIG deployments where tenants receive separate GIs and still leak information across the advertised isolation boundary.

In contrast, we demonstrate that page-fault-driven contention can fingerprint both ML and LLM workloads even without weight offloading, without access to NVML PCIe monitoring interfaces such as `nvmlDeviceGetPcieThroughput`, and in both containerized MIG and vGPU+MIG environments. Our method also remains effective beyond startup and can distinguish workloads that exhibit similar aggregate PCIe traffic but differ in their GPU-memory behavior. In summary, ours is the first effort to fingerprint ML models and LLMs across different setups, without environmental limitations (bare-metal/container or vGPU) or temporal constraints (works beyond startup).

Recent GPU Rowhammer attacks [19, 66], show that bit flips on GDDR6-based NVIDIA GPUs can corrupt GPU page tables, ultimately enabling arbitrary memory access and host compromise. In contrast, our work studies side channels on HBM-based MIG GPUs. Moreover, our vGPU+MIG analysis suggests that page-table bit-flip attacks would be ineffective in this setting (Section 9.2). These results further motivate studying GPU page-table isolation across deployment models.

13 Conclusion

In this paper, we explored how Unified Virtual Memory impacts the security guarantees provided by NVIDIA Multi-Instance GPU (MIG) technology. Our investigation revealed that MIG does not fully isolate internal GPU memory paths, kernel-level software components, and the PCIe bus, causing information leakage between GPU Instances (GIs).

We developed *MIGraine*, a novel side-channel attack that exploits page fault latency to fingerprint LLMs across GPU instances with >96% accuracy. We showed the attack is effective on both containerized and virtualized GPU (vGPU) MIG deployments. We also validated our results on both A30 (Ampere) and H100 (Hopper) GPUs. In other words, *MIGraine* persists across different GPU generations and deployment models, despite significant differences in the underlying architecture. For instance, our reverse engineering efforts uncovered important differences in the isolation architecture of vGPU and containerized MIG deployments, yet the side channel remains effective throughout.

14 Disclosure

We reported the page-fault side-channel vulnerability to NVIDIA in April 2025 and the vGPU plugin segmentation fault in June 2025. NVIDIA placed the vulnerabilities presented in the paper under (now lifted) embargo, assigned CVE-XXXX-XXXXX to the vGPU bug (Section 9.2), and acknowledged the side-channel risk.

References

[1] Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed Badawy. 2022. Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis. arXiv:2208.11174 [cs.AR] https://arxiv.org/abs/2208.11174

[2] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Weizhu Chen, Yen-Chun

Chen, Yi-Ling Chen, Hao Cheng, Parul Chopra, Xiyang Dai, Matthew Dixon, Ronen Eldan, Victor Fragoso, Jianfeng Gao, Mei Gao, Min Gao, Amit Garg, Allie Del Giorno, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Wenxiang Hu, Jamie Huynh, Dan Iter, Sam Ade Jacobs, Mojan Javaheripi, Xin Jin, Nikos Karampatziakis, Piero Kauffmann, Mahoud Khademi, Dongwoo Kim, Young Jin Kim, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Xihui Lin, Zeqi Lin, Ce Liu, Liyuan Liu, Mengchen Liu, Weishung Liu, Xiaodong Liu, Chong Luo, Piyush Madan, Ali Mahmoudzadeh, David Majercak, Matt Mazzola, Caio César Teodoro Mendes, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Liliang Ren, Gustavo de Rosa, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Yelong Shen, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Praneetha Vaddamanu, Chunyu Wang, Guanhua Wang, Lijuan Wang, Shuohang Wang, Xin Wang, Yu Wang, Rachel Ward, Wen Wen, Philipp Witte, Haiping Wu, Xiaoxia Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Jilong Xue, Sonali Yadav, Fan Yang, Jianwei Yang, Yifan Yang, Ziyi Yang, Donghan Yu, Lu Yuan, Chenruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. arXiv:2404.14219 [cs.CL] https://arxiv.org/abs/2404.14219

[3] Jaeguk Ahn, Jiho Kim, Hans Kasan, Leila Delshadtehrani, Wonjun Song, Ajay Joshi, and John Kim. 2021. Network-on-Chip Microarchitecture-Based Covert Channel in GPUs (MICRO'21).

[4] Tyler Allen and Rong Ge. 2021. In-depth analyses of unified virtual memory system for GPU accelerated computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[5] AWS. 2023. Important NVIDIA driver changes to DLAMIs. https://docs.aws.amazon.com/dlami/latest/devguide/important-changes.html. Accessed: 2025-08-26.

[6] 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, San Antonio, Texas, 54–66.

[7] Joshua Bakita and James H Anderson. 2024. Demystifying NVIDIA GPU Internals to Enable Reliable GPU Management. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 294–305.

[8] Joshua Bakita and James H Anderson. 2025. Hardware Compute Partitioning on NVIDIA GPUs for Composable Systems. In *37th Euromicro Conference on Real-Time Systems (ECRTS 2025)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 21–1.

[9] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[10] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2023. Spy in the gpu-box: Covert and side channel attacks on multi-gpu systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.

[11] Zibo Gao, Junjie Hu, Feng Guo, Yixin Zhang, Yinglong Han, Siyuan Liu, Haiyang Li, and Zhiqiang Lv. 2025. I Know What You Said: Unveiling Hardware Cache Side-Channels in Local Large Language Model Inference. *arXiv preprint arXiv:2505.06738* (2025).

[12] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. 2020. Traversing large graphs on GPUs with unified memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1119–1133.

[13] Lukas Giner, Roland Czerny, Christoph Gruber, Fabian Rauscher, Andreas Kogler, Daniel De Almeida Braga, and Daniel Gruss. 2024. Generic and Automated Drive-by GPU Cache Attacks from the Browser. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. 128–140.

[14] Google. 2023. Deep Learning VM release notes. https://cloud.google.com/deep-learning-vm/docs/release-notes#June_26_2023. Accessed: 2025-08-26.

[15] Dirk Groeneveld, Iz Beltagy, Pete Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Harsh Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, Shane Arora, David Atkinson, Russell Authur, Khyathi Raghavi Chandu, Arman Cohan, Jennifer Dumas, Yanai Elazar, Yuling Gu, Jack Hessel, Tushar Khot, William Merrill, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew E. Peters, Valentina Pyatkin, Abhilasha Ravichander, Dustin Schwenk, Saurabh Shah, Will Smith, Emma Strubell, Nishant Subramani, Mitchell Wortsman, Pradeep Dasigi, Nathan Lambert, Kyle Richardson, Luke Zettlemoyer, Jesse Dodge, Kyle Lo, Luca Soldaini, Noah A. Smith, and Hannaneh Hajishirzi. 2024. OLMo: Accelerating the Science of Language Models. arXiv:2402.00838 [cs.CL] https://arxiv.org/abs/2402.00838

[16] Zhongshu Gu, Enriquillo Valdez, Salman Ahmed, Julian James Stephen, Michael Le, Hani Jamjoom, Shixuan Zhao, and Zhiqiang Lin. 2025. NVIDIA GPU Confidential Computing Demystified. arXiv:2507.02770 [cs.CR] https://arxiv.org/abs/2507.02770

[17] MD Hassan, Shanto Roy, and Reza Rahaeimehr. 2025. Memory Under Siege: A Comprehensive Survey of Side-Channel Attacks on Memory. *arXiv preprint arXiv:2505.04896* (2025).

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[19] Yichang Hu, Noah Brown, Yuhang Chen, Joshua Bakita, Tianlong Chen, Daniel Genkin, and Andrew Kwong. 2026. GDDRHammer: Greatly Disturbing DRAM Rows — Cross-Component Rowhammer Attacks from Modern GPUs. In *2026 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA.

[20] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[21] Rodrigo Huerta, Mojtaba Abaie Shoushtary, José-Lorenzo Cruz, and Antonio González. 2025. Analyzing Modern NVIDIA GPU cores. *arXiv preprint arXiv:2503.20481* (2025).

[22] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the nvidia turing t4 gpu via microbenchmarking. *arXiv preprint arXiv:1903.07486* (2019).

[23] 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).

[24] Hongyue Jin, Yanan Guo, and Zhenkai Zhang. 2026. Exploiting TLBs in Virtualized GPUs for Cross-VM Side-Channel Attacks. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, USA. https://doi.org/10.14722/ndss.2026.231480

[25] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.

[26] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. University of Toronto.

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[28] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[29] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE]

[30] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Hongyuan Liu, Qiang Wang, and Xiaowen Chu. 2025. Dissecting the NVIDIA Hopper Architecture through Microbenchmarking and Multiple Level Analysis. *arXiv preprint arXiv:2501.12084* (2025).

[31] Haoyu Ma, Jianwen Tian, Debin Gao, and Chunfu Jia. 2021. On the effectiveness of using graphics interrupt as a side channel for user behavior snooping. *IEEE Transactions on Dependable and Secure Computing* 19, 5 (2021), 3257–3270.

[32] Yuanqing Miao, Yingtian Zhang, Dinghao Wu, Danfeng Zhang, Gang Tan, Rui Zhang, and Mahmut Taylan Kandemir. 2024. Veiled Pathways: Investigating Covert and Side Channels Within GPU Uncore. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1169–1183.

[33] 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.

[34] 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.

[35] NVIDIA. 2025. Beyond GPU Memory Limits with Unified Memory on Pascal. https://developer.nvidia.com/blog/beyond-gpu-memory-limits-unified-memory-pascal/. Accessed: 2025-08-26.

[36] NVIDIA. 2025. Default kernel modules installed on NVIDIA open source driver. https://github.com/NVIDIA/open-gpu-kernel-modules/blob/c5e439fea4fe81c78d52b95419c30cabe44e48fd/kernel-open/Makefile#L94. Accessed: 2025-08-26.

[37] NVIDIA. 2025. Documentation of NVIDIA RAM. https://nvidia.github.io/open-gpu-doc/manuals/ampere/ga100/dev_ram.ref.txt. GA100 Development RAM Reference.

[38] NVIDIA. 2025. MIG User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html#application-considerations. Accessed: 2025-07-17.

[39] NVIDIA. 2025. Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/introduction.html. Accessed: 2026-04-03.

[40] NVIDIA. 2025. NVIDIA AI Enterprise 6.2. https://docs.nvidia.com/ai-enterprise/release-6/6.2/appendix/vgpu.html. Accessed: 2025-08-26.

[41] NVIDIA. 2025. NVIDIA Linux Open GPU Kernel Module Source. https://github.com/NVIDIA/open-gpu-kernel-modules. Accessed: 2025-08-26.

[42] NVIDIA. 2025. NVIDIA Transitions Fully Towards Open-Source GPU Kernel Modules. https://developer.nvidia.com/blog/nvidia-transitions-fully-towards-open-source-gpu-kernel-modules/. Accessed: 2025-08-26.

[43] NVIDIA. 2025. Replayable Faults in NVIDIA GPUs. https://github.com/NVIDIA/open-gpu-kernel-modules/blob/c5e439fea4fe81c78d52b95419c30cabe44e48fd/kernel-open/nvidia-uvm/uvm_gpu_non_replayable_faults.c#L38. Accessed: 2025-08-26.

[44] NVIDIA. 2025. RM Privileged Channels. https://github.com/NVIDIA/open-gpu-kernel-modules/blob/c5e439fea4fe81c78d52b95419c30cabe44e48fd/src/nvidia/generated/g_mem_mgr_nvoc.h#L304. Accessed: 2025-10-03.

[45] NVIDIA. 2025. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/. Accessed: 2025-08-26.

[46] NVIDIA. 2025. Unified Virtual Memory Supercharges pandas with RAPIDS cuDF. https://developer.nvidia.com/blog/unified-virtual-memory-supercharges-pandas-with-rapids-cudf. Accessed: 2025-08-26.

[47] NVIDIA. 2025. UVM Privileged Channels. https://github.com/NVIDIA/open-gpu-kernel-modules/blob/c5e439fea4fe81c78d52b95419c30cabe44e48fd/kernel-open/nvidia-uvm/uvm_channel.h#L596. Accessed: 2025-10-03.

[48] NVIDIA. 2025. VRAM Guest Physical Addresses (VGPAs). https://github.com/NVIDIA/open-gpu-kernel-modules/blob/1893c6c8fd17c79d17706d8382af09d360b5b703/src/nvidia/generated/g_mem_desc_nvoc.h#L136. Accessed: 2025-08-26.

[49] NVIDIA. 2026. NVIDIA A30 Tensor Core GPU Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/data-center/products/a30-gpu/pdf/a30-datasheet.pdf. Accessed: 2026-04-06.

[50] NVIDIA. 2026. NVIDIA H100 GPU Datasheet. https://resources.nvidia.com/en-us-hopper-architecture/nvidia-tensor-core-gpu-datasheet. Accessed: 2026-04-06.

[51] NVIDIA Developer Forums. 2018. UNIX Graphics Feature Deprecation Schedule. https://forums.developer.nvidia.com/t/unix-graphics-feature-deprecation-schedule/60588. Accessed: 2025-07-14.

[52] NVIDIA GitHub Discussions. 2024. Discussion on Open GPU Kernel Modules #312. https://github.com/NVIDIA/open-gpu-kernel-modules/discussions/312#discussioncomment-11145687. Accessed: 2025-07-14.

[53] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vattention: Dynamic memory management for serving llms without pagedattention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 1133–1150.

[54] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] https://arxiv.org/abs/2412.15115

[55] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[56] RAPIDS. 2025. RAPIDS cuDF. https://docs.rapids.ai/api/cudf/stable/. Accessed: 2025-10-03.

[57] RAPIDS. 2025. Unified Virtual Memory Supercharges pandas with RAPIDS cuDF. https://developer.nvidia.com/blog/unified-virtual-memory-supercharges-pandas-with-rapids-cudf/. Accessed: 2025-10-03.

[58] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[59] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 753–768.

[60] Mert Side, Fan Yao, and Zhenkai Zhang. 2022. Lockeddown: Exploiting contention on host-gpu pcie bus for fun and profit. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 270–285.

[61] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[62] Stability AI Language Team. [n. d.]. Stable LM 2 1.6B. https://huggingface.co/stabilityai/stablelm-2-1.6b

[63] The Linux Kernel Community. [n. d.]. VFIO - Virtual Function I/O. https://docs.kernel.org/driver-api/vfio.html. Accessed: 2025-07-19.

[64] The Linux Kernel Community. 2009. How to Use PCI IOV (Single Root I/O Virtualization). https://docs.kernel.org/PCI/pci-iov-howto.html. Accessed: 2025-07-19.

[65] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 88–105.

[66] Junpeng Wan, Yanan Guo, Zhi Zhang, Zhuo Li, Dave (Jing) Tian, and Zhenkai Zhang. 2026. GeForge: Hammering GDDR Memory to Forge GPU Page Tables for Fun and Profit. In *2026 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA.

[67] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Optimizing batched winograd convolution on GPUs. In *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*. 32–44.

[68] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. Tinyllama: An open-source small language model. *arXiv preprint arXiv:2401.02385* (2024).

[69] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL] https://arxiv.org/abs/2205.01068

[70] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the gpu microarchitecture to achieve bare-metal performance tuning. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 31–43.

[71] Yicheng Zhang, Ravan Nazaraliyev, Sankha Baran Dutta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. 2024. Beyond the bridge: Contention-based covert and side channel attacks on multi-gpu interconnect. In *2024 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 35–36.

[72] Yicheng Zhang, Ravan Nazaraliyev, Sankha Baran Dutta, Andres Marquez, Kevin Barker, and Nael Abu-Ghazaleh. 2025. NVBleed: Covert and Side-Channel Attacks on NVIDIA Multi-GPU Interconnect. *arXiv preprint arXiv:2503.17847* (2025).

[73] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. 2023. TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 960–974.

[74] Zhenkai Zhang, Kunbei Cai, Yanan Guo, Fan Yao, and Xing Gao. 2024. {Invalidate+ Compare}: A {Timer-Free}{GPU} Cache Attack Primitive. In *33rd USENIX Security Symposium (USENIX Security 24)*. 2101–2118.

[75] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 345–357.