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

Giovanni Di Santi
Vrije Universiteit Amsterdam

Andrea Di Dio
Vrije Universiteit Amsterdam

Cristiano Giuffrida
Vrije Universiteit Amsterdam

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*: the latency of the attacker's own page faults reflects contention with co-located GIs. Specifically, we show that, despite strong hardware isolation, black-box 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 Model (LLM) workloads across GIs with 96% accuracy. While direct PCIe monitoring reaches comparable accuracy on noise-free workloads, *MIGraine* also stays accurate under PCIe noise and captures GPU-internal memory activity. We further provide the first characterization of containerized versus virtualized MIG deployments and show that *MIGraine* still fingerprints ML inference workloads under the stronger memory isolation of vGPUs.

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, 7–9, 25]. 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 [26, 27, 35, 72, 76]. Despite these challenges, researchers have successfully demonstrated side-channel attacks on GPUs. Similarly to their CPU counterparts [21, 29, 33, 64, 70], these attacks target shared components such as caches, translation lookaside buffers (TLBs), and other microarchitectural components [3, 13, 15, 28, 39, 40, 80], interconnects such as PCIe [65] and NVLink [12, 77, 78], and interrupts [36]. The net result is the ability to fingerprint a co-located GPU workload, for instance to mount ML *model fingerprinting* attacks.

NVIDIA MIG. To support GPU multi-tenancy, NVIDIA introduced *Multi-Instance GPU* (MIG) for its 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" [46]. However, prior work has identified components, namely the L3-TLB [79] and the PCIe interface [37], 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 [79]. Prior PCIe attacks [37], 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) [53, 81] programming model. UVM is widely used by modern GPU workloads [4,14,42,54,60], enabled by default by the NVIDIA driver [43, 49], and adopted in the major cloud environments [5, 16]. 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 [11] classifier.

MIGraine achieves 96% accuracy fingerprinting LLM workloads running through vLLM [31] on co-located GIs. Direct PCIe monitoring reaches comparable accuracy on these workloads without noise. However, *MIGraine* stays at 96% under PCIe noise, where PCIe monitoring drops to about 60%, and it distinguishes GPU kernels that PCIe monitoring confuses.

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, *MIGraine* still fingerprints ML inference workloads in vGPU environments with over 90% accuracy, exploiting hardware contention on the PCIe bus and GPU memory subsystems that neither deployment architecture mitigates. Our analysis also uncovered a vulnerability in the hypervisor's vGPU plugin, which NVIDIA rated high severity and assigned CVE-2025-23352 [52].

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 ML and LLM workloads on co-located GIs with 89% and 96% accuracy, the latter also under PCIe noise.

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. This analysis also uncovered a high-severity vulnerability in NVIDIA's vGPU plugin (CVE-2025-23352). We demonstrate that our side channel also fingerprints ML inference workloads in vGPU deployments, highlighting isolation gaps in MIG that persist across both environments.

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 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

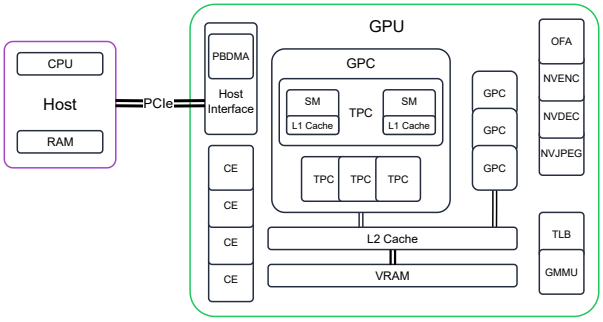


Figure 1: High-level overview of a modern NVIDIA GPU, with the host system (CPU and RAM) connected to the GPU via PCIe. Internally, the GPU features a compute hierarchy (GPCs, TPCs, SMs), a memory hierarchy (L1/L2 caches, TLB, VRAM), 4 Copy Engines (CEs), one PBDMA engine, and various specialized engines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

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 [56, 57].

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 [79].

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 in MIG environments [45]. 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 creation. 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 a dedicated set of compute engines (GPCs), a partition of the L2 cache (its own L2 slices), memory controllers, and DRAM bandwidth, ensuring strong, predictable performance. L2-bound accesses nevertheless experience cross-GI interference, as Section 6.1 shows. 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.

MIG deployment models. MIG reaches tenants through two deployment models, which differ in what tenants share. In *containerized* deployments, all tenants run on one host kernel and share a single NVIDIA driver stack, including its page fault handling. In *vGPU* deployments, each VM runs its own guest driver on a dedicated GI exposed through SR-IOV. Both models leave the PCIe link and internal GPU memory paths shared (Sections 6 and 9), and Section 9.1 details the vGPU architecture.

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 [68] and VFIO [69], 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)?*

Deployments in practice. This setting matches how providers deploy MIG today. NVIDIA positions MIG for cloud service providers with multi-tenant use cases [46]. The managed Kubernetes services of AWS, Google Cloud, and Microsoft Azure partition A100- and H100-class GPUs into GIs assigned to separate containers [6, 17, 38], and AWS recommends MIG for multi-tenant inference, stating that a workload on one instance cannot affect a workload on another [6]. For VMs, NVIDIA's vGPU software places each MIG-backed vGPU on its own GI [47]. In these services, the cluster operator decides which workloads share a GPU, so co-location arises whenever one cluster serves mutually distrusting users, as in shared inference services and ML platforms.

Experimental setup. Our experimental setup uses the open-source kernel driver [48] (version 570.133.07) for containerized environments and NVIDIA AI Enterprise 6.2 [47] (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. Our containers share the host driver, as pods do in the Kubernetes services above, and the A30 and H100 cover the Ampere and Hopper MIG generations these services expose. Production GPUs host up to seven GIs, and Section 8.2 shows that *MIGraine* maintains accuracy above 90% when a third GI runs background inference.

4 Overview

We briefly outline our *MIGraine* side-channel attack, which enables an adversary on one GI to fingerprint ML/LLM workloads running on a co-located GI.

Building the primitive. *MIGraine* measures UVM page fault latency. A workload on one GI changes the time required to service faults on another GI through 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 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 an 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

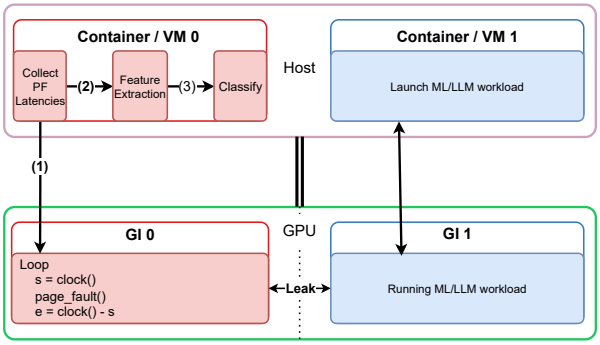


Figure 2: High-level overview of our *MIGraine* attack.

fed into a classifier (XGBoost [11]), which predicts the identity of the victim's workload (Step 3). We analyze the side channel in detail in Section 8.

Analysis roadmap. We reverse-engineer NVIDIA's UVM driver (Section 5) and identify the fault buffer as a software resource shared across GIs. We then test, in both directions, interference from the software and hardware components in the page fault path (Section 6). Finally, we use this interference to build a covert channel and the *MIGraine* side channel (Sections 7–8), and evaluate *MIGraine* in vGPU deployments (Section 9).

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 [44], 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 [51, 55].

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 [59]; and (2) *confidential computing channels*, which are dedicated to Confidential Computing, a Trusted Execution Environment (TEE) mechanism introduced on Hopper+ architectures [20].

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` [79] 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, which 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 Graphics Engines [50]. 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 vice versa. 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 [66] training on the CIFAR-10 dataset [30]) and PCIe bus saturation through continuous host-GPU data transfers using `cudaMemcpy`. The latter is included because prior work [37] identified the (direct) PCIe interface as a shared resource between GIs 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 GIs.

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

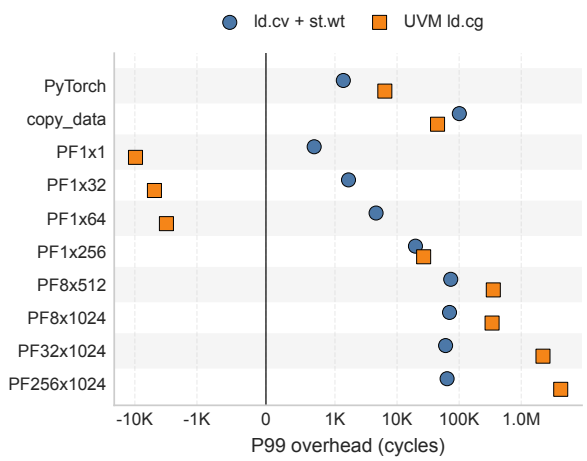


Figure 3: 99th percentile overhead (in clock cycles) for two GPU kernels on GI 0 under various interference workloads on GI 1, measured relative to an idle baseline. Each row corresponds to one interference workload on GI 1 and reports two independent probe kernels on GI 0. Blue markers show a kernel 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.

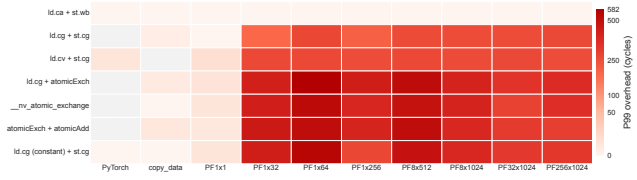


Figure 4: 99th percentile overhead of seven GI 0 memory-access probes under co-located interference workloads on GI 1, measured relative to an idle GI 1 baseline. Probe kernels cover an L1-sensitive `ld.ca+st.wb` configuration, L2-oriented `ld.cg+st.cg` and `ld.cv+st.cg` accesses, mixed cache+atomic 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.

Takeaway 2

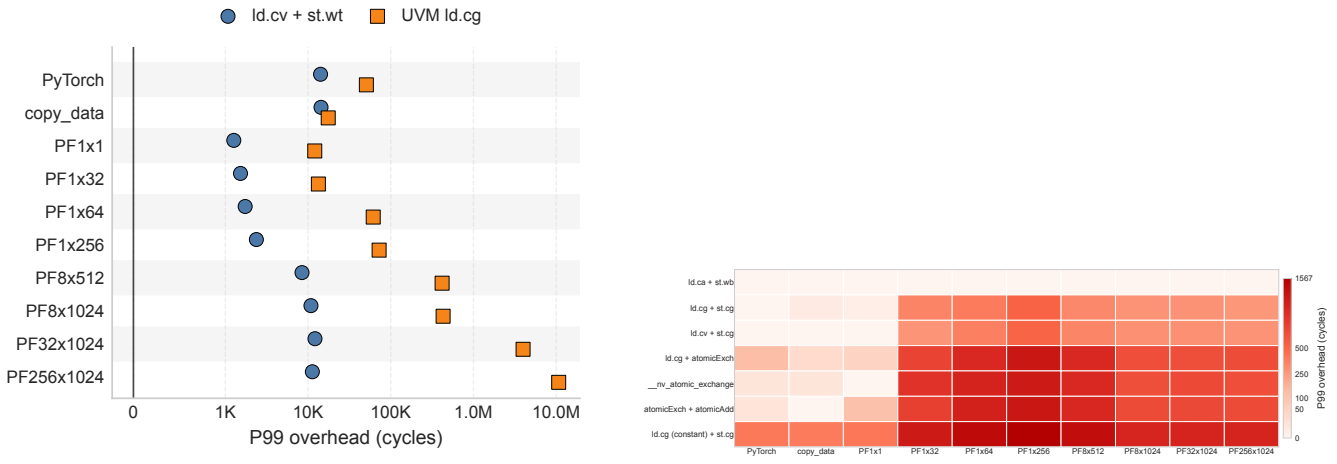
Shared kernel-level resources in the UVM fault handling path cause cross-GI contention in containerized deployments, significantly increasing memory access latency for UVM-intensive workloads.

100,000 clock cycles of (moderate) overhead when the co-located GI performs PCIe data transfers, consistent with prior (direct) PCIe contention results [37].

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 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 GIs. 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. However, the extent of this effect depends heavily on the CPU scheduling and hardware configuration of the machine, and may therefore vary across devices.

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 accesses to 256 distinct cache lines while varying the memory access operations. Our kernels systematically vary along 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.



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

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

Figure 5: H100 cross-GI interference results.

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 GPU-internal memory accesses.

H100 results. Figure 5 presents our cross-GI interference analysis results on the H100. Consistent with the A30 results in Figures 3 and 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.

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 in 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 `nvml DeviceGetPcieThroughput`, abused in prior work [37].

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.

Takeaway 4

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

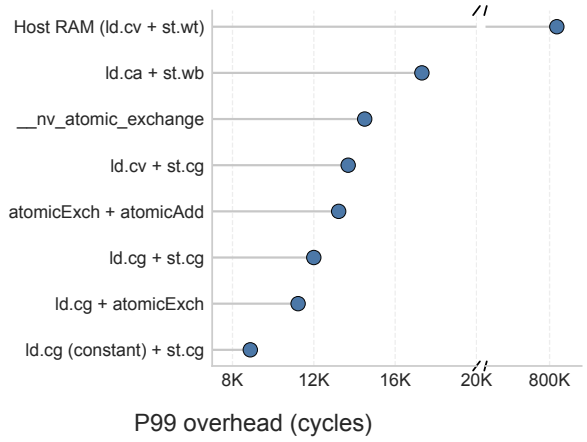


Figure 6: 99th percentile overhead of page fault latency on GI 0 under various GPU kernels running on GI 1. Y-axis labels report the load/store or atomic instruction pattern used by each kernel. Host RAM accesses cause the highest overhead due to PCIe contention, while GPU-internal operations show mild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

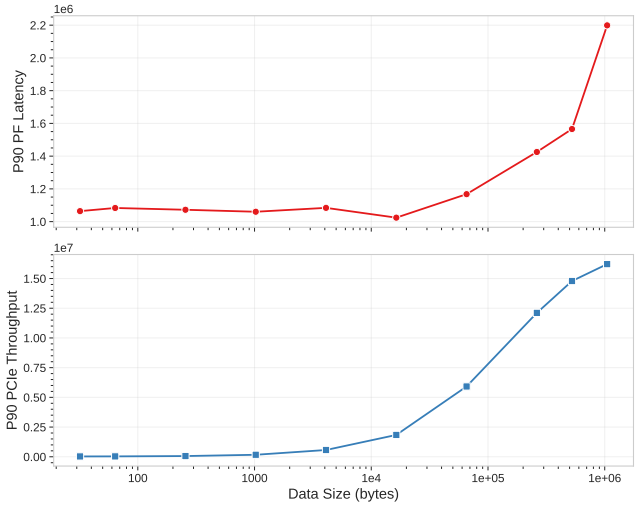


Figure 7: 90th percentile page fault latency on GI 0 (upper panel) and PCIe throughput on GI 0 measured via `nvmlDeviceGetPcieThroughput` (lower panel), while varying `cudaMemcpy` traffic on GI 1

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 10 ms. This rate of 100 bps is comparable to the most reliable configuration of the PCIe covert channel across GIs by Miao et al. [37], which reaches 93 bps at a 2.2% error rate and up to 6.8 kbps at 3.3%. The L3-TLB channel of Zhang et al. [79] reaches up to 31 kbps, building on reverse-engineered TLB eviction sets. Our channel uses a simple on-off encoding and targets error-free transmission, demonstrating page fault contention as a communication medium across GIs.

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

8 Page Fault Side Channel

In Section 6.2, we showed that different workloads induce measurable variations in UVM page fault latency (Takeaways 1 and 4). In this section, we show that an attacker

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 ← get_latencies_gpu_kernel(N_l)
    p99_latency ← p99(latency_values)
    if p99_latency > τ then
        message[i] = 1
    else
        message[i] = 0
    end if
end for
```

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 [63] model; (3) *vLLM* [31], a widespread framework for LLM inference, running the Phi-3-Mini [2] model; and (4) *cuDF* [62], a GPU DataFrame library, performing UVM-intensive join operations [54] that extensively utilize unified memory. For each workload, we compute the mean of the top 20% highest-latency samples (tail 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 increases, to 70K and 65K cycles, respectively, compared to the inactive baseline of approximately 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.

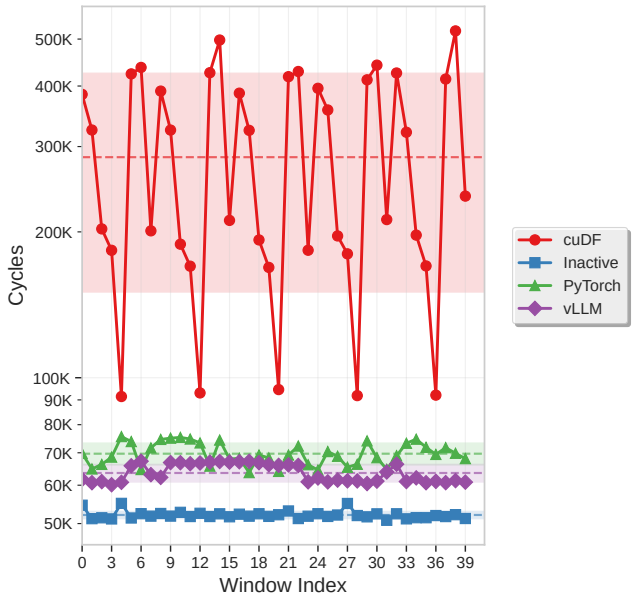


Figure 8: Tail-average latency of page fault handling on GI 0 under different workloads on GI 1. Each point on the log-scale y-axis represents the mean of the top 20% highest-latency samples within the sample window.

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 [66], ResNet [22], DenseNet [24], and MobileNetV2 [63] 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 [37, 79], 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 [11] on statistical features extracted from 10,000-sample windows

Table 1: Classification results for ML fingerprinting.

Model	Precision	Recall	F1	#
DenseNet [24]	0.79	0.82	0.80	40
MobileNetV2 [63]	0.95	0.97	0.96	40
ResNet [22]	0.95	0.88	0.91	40
VGG [66]	0.88	0.88	0.88	40
Accuracy	0.89			

Table 2: Classification results for LLM fingerprinting.

Model	Precision	Recall	F1	#
GPT2-Large [61]	0.98	0.98	0.98	321
GPT2-Medium [61]	0.98	0.97	0.98	321
OLMo-1B [18]	0.85	0.91	0.88	321
OPT-125m [75]	0.99	1.00	1.00	321
Phi-3-Mini [2]	0.96	0.98	0.97	321
Qwen2-1.5B [73]	0.96	0.90	0.93	321
StableLM [10]	1.00	1.00	1.00	321
StarCoder2-3B [34]	1.00	0.99	0.99	321
TinyLlama-1B [74]	0.92	0.90	0.91	321
Accuracy	0.96			

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.

LLM fingerprinting. Next, we further extend our testbed to classify Large Language Models (LLMs) under inference workloads. We use vLLM [31] 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 [37]. 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 and a 3,000-sample step size. We split each trace chronologically, using its first 80% for training and its last 20% for testing, so that no window spans both sets. Table 2 presents our results, with our LLM classifier achieving 96% overall accuracy. We obtained similar classification accuracy when we sampled prompts uniformly at random from the Alpaca dataset [67].

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. 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 evaluate the resilience of *MIGraine* to PCIe interference, we repeat the LLM fingerprinting experiment while generating synthetic noise with `cudaMemcpy` transfers at three intensities: low (1KB–1MB every 10–20ms), medium (1–10MB every 5–10ms), and high (10–50MB every 1–5ms). A classifier trained only on noise-free data is not robust: its accuracy drops from the 96% baseline to 58%, 15%, and 20% under low, medium, and high noise, respectively. When training and testing are performed under the same fixed noise level, accuracy returns to approximately 96%, showing that the workload-dependent signal remains distinguishable. More importantly, a single classifier trained on a composite dataset containing all noise levels also achieves 96% accuracy on a mixed-noise test set. In contrast, a direct PCIe-based classifier trained on the same mixed-noise data achieves only approximately 60% accuracy. We therefore use the mixed-noise experiment as the primary robustness test, while interpreting the per-noise-level results as diagnostic upper bounds.

We additionally evaluate a realistic three-GI configuration. The attacker and victim occupy two GIs, while a separate steady-state vLLM workload runs on the third and generates background activity. After retraining with traces collected under this configuration, *MIGraine* maintains accuracy above 90%. Because steady-state vLLM inference produces relatively stable PCIe traffic, the synthetic `cudaMemcpy` experiment remains the stronger stress test: it deliberately sweeps a wider range of contention levels and demonstrates robustness beyond what a purely PCIe-based classifier can provide.

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% ac-

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	

curacy, it struggles to distinguish between cache-intensive operations such as `ld.cv+st.cg` and `ld.ca+st.wb` (F1 scores of 0.16 and 0.25, 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.

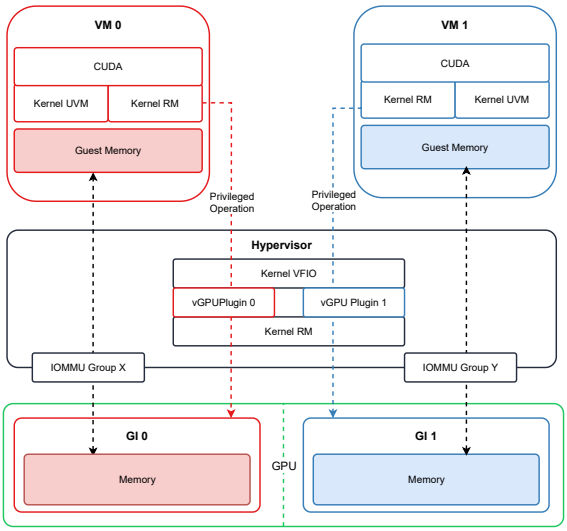


Figure 9: High-level overview of a vGPU+MIG setup with two VMs, each connected to their own GPU Instance. Each GPU Instance uses a separate IOMMU group for DMA memory transfers with the guest VM.

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 [58, 59]. 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 the question of where vGPU enforces memory isolation. If the guest driver, the hypervisor, or both enforced it purely in software, a malicious guest could remap its own page tables to access another GI's physical memory. 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 [44] and reads 1 MiB 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` [79], 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 its 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* (GPAs). The GPU translates each GPA to a *system physical address* (SPA), the actual location in VRAM, using a GI-specific offset. For instance, in our configuration with two 12 GiB GIs, all GPAs are translated to SPAs by adding `0x4000000` to GI 0 guest addresses and `0x4000000+0x2f8000000` to GI 1 addresses. According to the open GPU kernel driver, which uses the same terminology, a dedicated memory management unit in the GPU, the *VMMU*, performs this translation [41].

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 GPAs. 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-2025-23352. NVIDIA's advisory lists code execution, denial of service, privilege escalation, information disclosure, and data tampering as potential impacts [52]. We leave exploitation beyond the crash to future work.

Takeaway 5

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 5).

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 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.

This result clarifies what the vGPU setting adds. The driver-level contention of containerized deployments disappears, and the remaining leakage flows through PCIe and GPU-internal contention, which MIG leaves shared in both deployment models. What changes is the attacker's vantage point: a guest VM with its own driver stack, whose privileged operations the hypervisor mediates. From this vantage point, the attacker's own page faults still reveal the co-located workload, with no microarchitectural reverse engineering. Our vGPU experiment covers ML inference, and we leave the LLM and noise evaluations of Section 8 in vGPU to future work.

Takeaway 6

Despite vGPU's isolation of kernel-level software resources and its hardware-enforced memory isolation through the VMMU, our side channel still fingerprints ML inference workloads with over 90% accuracy. It exploits black-box contention on PCIe traffic and GPU memory operations, which neither containerized nor vGPU deployments isolate.

10 Limitations

MIGraine uses page fault latency as a black-box sensor. The latency combines contention from the host driver (in containerized deployments), the PCIe link, and internal GPU memory paths, which frees the attacker from modeling the underlying architecture but prevents a full attribution of the signal. Section 8.2 separates PCIe from GPU-internal contention, and Table 3 shows that host activity alone cannot explain the signal: the victim kernels there differ only in their

GPU memory instructions, yet page fault latency tells them apart. We did not quantify the remaining host-side share, such as CPU scheduling of the fault handler.

Our evaluation is closed-world. It covers four CNN families and nine LLMs, the latter being all license-free models that vLLM supports and that fit the victim GI (Section 8.2). Production services host many more models, including fine-tuned variants and adapters, so an open-world attacker must also reject unknown workloads. Our classifiers also draw training and test windows from the same executions, so run-specific artifacts such as warm-up state may contribute to their accuracy. Accuracy may also vary with the hyperparameters of the victim's training runs.

Like prior cross-GI side channels [37,79], *MIGraine* identifies workloads rather than their data: it reveals which model a tenant runs, but not prompts, generated tokens, or weights. Accuracy stays similar when prompts vary (Section 8.2), which suggests that the model, rather than the prompt, dominates the signal. Finally, we evaluated two GPU models, the A30 and the H100. Device-dependent effects require profiling on the target GPU, which our attacker already performs (Section 8.2).

11 Mitigations

Mitigations map onto the three contention sources behind *MIGraine*. The driver-level source disappears once each GI receives its own fault buffer and fault-servicing path. The vGPU software stack already provides this separation (Section 9), so operators hosting mutually distrusting tenants can prefer vGPU+MIG over containers, and NVIDIA could extend the same partitioning to containerized deployments. The PCIe and GPU-internal sources persist in both settings, as our vGPU results show. Restricting UVM for untrusted tenants removes the attacker's probe altogether, since a tenant without managed memory cannot trigger the replayable faults that *MIGraine* times. This measure breaks workloads that rely on UVM. Limiting the page fault rate per GI offers a middle ground that caps the attacker's sampling rate.

Closing the remaining sources requires hardware support. NVIDIA could arbitrate PCIe bandwidth per GI and partition the fault-handling units inside the GPU, trading utilization for isolation. Noise injection offers limited protection: in our experiments, a classifier trained on traces with varying PCIe noise still reached 96% accuracy (Section 8.2). Until hardware defenses exist, cloud providers can avoid co-locating mutually distrusting tenants on one physical GPU, at the cost of the consolidation that motivates MIG.

12 Related Work

Zhang et al. leveraged the shared L3-TLB to fingerprint ML models across GIs [79], but their classification worked only during the model startup phase and required learning TLB-specific eviction sets for each GPU generation. Miao et al. [37] explored PCIe contention on MIG setups without vGPUs, using the `nvmlDeviceGetPcieThroughput` interface to identify LLM workloads across GIs. 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 GPUs shared without MIG. Side et al. [65] timed the attacker's own host-GPU transfers to sense PCIe contention, building a cross-VM covert channel on time-sliced vGPUs and a website fingerprinting attack on personal computers. Jin et al. [28] mounted cross-VM side-channel attacks on the TLBs of time-sliced vGPUs. The two sharing models expose different resources. Under time slicing, tenants take turns on the entire GPU: they share every on-GPU resource, but their kernels never run concurrently. MIG gives each tenant dedicated SMs, L2 slices, and memory, and runs tenants concurrently, so leakage must flow through the resources MIG leaves shared. MIGraine observes three of them through a single signal, page fault latency: the PCIe link, internal GPU memory paths, and, in containerized deployments, the host driver's fault-handling path. We expect its PCIe component to carry over to time-sliced vGPU, where Side et al. already observed PCIe leakage, whereas its GPU-internal component relies on the concurrent execution that MIG enables.

In contrast, we demonstrate that page-fault-driven contention can fingerprint both ML and LLM workloads even without weight offloading and 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).

Concurrently with our work, Gu et al. [19] showed that memory barriers issued in one GI affect the latency of certain loads in another, despite MIG's L2 cache partitioning. On Hopper GPUs, kernel launches issue such barriers, so an attacker who times its own loads recovers the kernel launch pattern of a victim and, from it, the LLM in use and the lengths of its inputs and outputs. This kernel-level signal exposes finer-grained information than our workload fingerprints, such as output lengths at single-token granularity. The two attacks observe different shared paths. Their attack probes the GPU memory system with specific memory instructions, whereas MIGraine uses the page fault path, which spans the host driver, the PCIe link, and the GPU memory subsystem, as a black-box sensor. The paths also differ in reach. On Ampere GPUs, kernel launches issue no memory barriers, so their side

channel depends on victims that frequently invoke CUDA memory-management APIs. MIGraine operates on both Ampere and Hopper GPUs and extends to vGPU deployments. Together, the two works show that MIG leaves shared state both inside the GPU memory system and along the host-GPU path.

Recent GPU Rowhammer attacks [23, 32, 71] target GDDR6 page tables to enable memory access and host compromise, whereas we study side channels on HBM-based MIG GPUs. Our vGPU analysis further suggests that such page-table attacks would be ineffective in this setting (Section 9.2).

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 GIs with 96% accuracy. We showed that the attack also fingerprints ML inference workloads in 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.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by AWS through the "Themis" project, by NWO through project "INTERSECT" and the Dutch Prize for ICT research, and by the European Union's Horizon Europe programme under grant agreement No. 101120962 ("Rescale").

References

[1] Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed A. Badawy. Demystifying the NVIDIA Ampere architecture through microbenchmarking and instruction-level analysis. In *HPEC*, 2022.

[2] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024.

[3] Jaeguk Ahn, Jiho Kim, Hans Kasan, Leila Delshadtehrani, Wonjun Song, Ajay Joshi, and John Kim. Network-on-chip microarchitecture-based covert channel in GPUs. In *MICRO*, 2021.

[4] Tyler Allen and Rong Ge. In-depth analyses of unified virtual memory system for GPU accelerated computing. In *SC*, 2021.

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

[6] AWS. Use multi-instance GPUs (MIG) with NVIDIA GPUs on Amazon EKS. https://docs.aws.amazon.com/eks/latest/userguide/device-management-nvidia-mig.html, 2026.

[7] Joshua Bakita and James H Anderson. Hardware compute partitioning on NVIDIA GPUs. In *RTAS*, 2023.

[8] Joshua Bakita and James H Anderson. Demystifying NVIDIA GPU internals to enable reliable GPU management. In *RTAS*, 2024.

[9] Joshua Bakita and James H Anderson. Hardware compute partitioning on NVIDIA GPUs for composable systems. In *ECRTS*, 2025.

[10] Marco Bellagente, Jonathan Tow, Dakota Mahan, Duy Phung, et al. Stable LM 2 1.6B technical report. *arXiv preprint arXiv:2402.17834*, 2024.

[11] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *KDD*, 2016.

[12] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Spy in the GPU-box: Covert and side channel attacks on multi-GPU systems. In *ISCA*, 2023.

[13] Zibo Gao, Junjie Hu, Feng Guo, Yixin Zhang, Yinglong Han, Siyuan Liu, Haiyang Li, and Zhiqiang Lv. I know what you said: Unveiling hardware cache side-channels in local large language model inference. In *USENIX Security*, 2025.

[14] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. Traversing large graphs on GPUs with unified memory. *PVLDB*, 2020.

[15] Lukas Giner, Roland Czerny, Christoph Gruber, Fabian Rauscher, Andreas Kogler, Daniel De Almeida Braga, and Daniel Gruss. Generic and automated drive-by GPU cache attacks from the browser. In *AsiaCCS*, 2024.

[16] Google. Deep Learning VM release notes. https://cloud.google.com/deep-learning-vm/docs/release-notes#June_26_2023, 2023.

[17] Google. Running multi-instance GPUs. https://docs.cloud.google.com/kubernetes-engine/docs/how-to/gpus-multi, 2026.

[18] Dirk Groeneveld, Iz Beltagy, Evan Walsh, Akshita Bhagia, et al. OLMo: Accelerating the science of language models. In *ACL*, 2024.

[19] Cheng Gu, Reese Levine, Zhenkai Zhang, Tyler Sorensen, and Yanan Guo. Behind bars: A side-channel attack on NVIDIA MIG cache partitioning using memory barriers. In *USENIX Security*, 2026.

[20] Zhongshu Gu, Enriquillo Valdez, Salman Ahmed, Julian James Stephen, Michael Le, Hani Jamjoom, Shixuan Zhao, and Zhiqiang Lin. Blueprint, bootstrap, and bridge: A security look at NVIDIA GPU confidential computing. In *MLSys*, 2026.

[21] MD Mahady Hassan, Shanto Roy, and Reza Rahaeimehr. Memory under siege: A comprehensive survey of side-channel attacks on memory. *Computers & Security*, 2026.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[23] Yichang Hu, Noah Brown, Yuhang Chen, Joshua Bakita, Tianlong Chen, Daniel Genkin, and Andrew Kwong. GDDRHammer: Greatly disturbing DRAM rows — cross-component Rowhammer attacks from modern GPUs. In *IEEE S&P*, 2026.

[24] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

[25] Rodrigo Huerta, Mojtaba Abaie Shoushtary, José-Lorenzo Cruz, and Antonio González. Dissecting and modeling the architecture of modern GPU cores. In *MICRO*, 2025.

[26] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. Dissecting the NVIDIA Turing T4 GPU via microbenchmarking. *arXiv preprint arXiv:1903.07486*, 2019.

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

[28] Hongyue Jin, Yanan Guo, and Zhenkai Zhang. Exploiting TLBs in virtualized GPUs for cross-VM side-channel attacks. In *NDSS*, 2026.

[29] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *CACM*, 2020.

[30] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

[31] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *SOSP*, 2023.

[32] Chris S. Lin, Yuqin Yan, Guozhen Ding, Joyce Qu, Joseph Zhu, David Lie, and Gururaj Saileshwar. GPUBreach: Privilege escalation attacks on GPUs using Rowhammer. In *IEEE S&P*, 2026.

[33] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[34] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, et al. StarCoder 2 and The Stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

[35] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. Benchmarking and dissecting the NVIDIA Hopper GPU architecture. In *IPDPS*, 2024.

[36] Haoyu Ma, Jianwen Tian, Debin Gao, and Chunfu Jia. On the effectiveness of using graphics interrupt as a side channel for user behavior snooping. *IEEE TDSC*, 2021.

[37] Yuanqing Miao, Yingtian Zhang, Dinghao Wu, Danfeng Zhang, Gang Tan, Rui Zhang, and Mahmut Taylan Kandemir. Veiled pathways: Investigating covert and side channels within GPU uncore. In *MICRO*, 2024.

[38] Microsoft. Create a multi-instance GPU node pool in Azure Kubernetes Service (AKS). https://learn.microsoft.com/en-us/azure/aks/gpu-multi-instance, 2026.

[39] Hoda Naghibijouybari, Khaled N Khasawneh, and Nael Abu-Ghazaleh. Constructing and characterizing covert channels on GPGPUs. In *MICRO*, 2017.

[40] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: GPU side channel attacks are practical. In *CCS*, 2018.

[41] NVIDIA. Address translation identifiers. https://github.com/NVIDIA/open-gpu-kernel-modules/blob/1893c6c8fd17c79d17706d8382af09d360b5b703/src/nvidia/generated/g_mem_desc_nvoc.h#L136, 2025.

[42] NVIDIA. Beyond GPU memory limits with unified memory on Pascal. https://developer.nvidia.com/blog/beyond-gpu-memory-limits-unified-memory-pascal/, 2025.

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

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

[45] NVIDIA. Multi-Instance GPU user guide: Application considerations. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html#application-considerations, 2025.

[46] NVIDIA. Multi-Instance GPU user guide: Introduction. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/introduction.html, 2025.

[47] NVIDIA. NVIDIA AI Enterprise 6.2. https://archive.docs.nvidia.com/ai-enterprise/release-6/6.2/appendix/vgpu.html, 2025.

[48] NVIDIA. NVIDIA Linux open GPU kernel module source. https://github.com/NVIDIA/open-gpu-kernel-modules, 2025.

[49] NVIDIA. NVIDIA transitions fully towards open-source GPU kernel modules. https://developer.nvidia.com/blog/nvidia-transitions-fully-towards-open-source-gpu-kernel-modules/, 2025.

[50] NVIDIA. 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, 2025.

[51] NVIDIA. RM privileged channels. https://github.com/NVIDIA/open-gpu-kernel-modules/blob/c5e439fea4fe81c78d52b95419c30cabe44e48fd/src/nvidia/generated/g_mem_mgr_nvoc.h#L304, 2025.

[52] NVIDIA. Security bulletin: NVIDIA GPU display drivers – October 2025. https://nvidia.custhelp.com/app/answers/detail/a_id/5703, 2025.

[53] NVIDIA. Unified memory for CUDA beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/, 2025.

[54] NVIDIA. Unified virtual memory supercharges pandas with RAPIDS cuDF. https://developer.nvidia.com/blog/unified-virtual-memory-supercharges-pandas-with-rapids-cudf, 2025.

[55] NVIDIA. UVM privileged channels. https://github.com/NVIDIA/open-gpu-kernel-modules/blob/c5e439fea4fe81c78d52b95419c30cabe44e48fd/kernel-open/nvidia-uvm/uvm_channel.h#L596, 2025.

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

[57] NVIDIA. NVIDIA H100 GPU datasheet. https://resources.nvidia.com/en-us-hopper-architecture/nvidia-tensor-core-gpu-datasheet, 2026.

[58] NVIDIA Developer Forums. UNIX graphics feature deprecation schedule. https://forums.developer.nvidia.com/t/unix-graphics-feature-deprecation-schedule/60588, 2018.

[59] NVIDIA GitHub Discussions. Discussion on open GPU kernel modules #312. https://github.com/NVIDIA/open-gpu-kernel-modules/discussions/312#discussioncomment-11145687, 2024.

[60] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. vAttention: Dynamic memory management for serving LLMs without PagedAttention. In *ASPLOS*, 2025.

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

[62] RAPIDS. RAPIDS cuDF. https://docs.rapids.ai/api/cudf/stable/, 2025.

[63] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

[64] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[65] Mert Side, Fan Yao, and Zhenkai Zhang. LockedDown: Exploiting contention on host-GPU PCIe bus for fun and profit. In *IEEE EuroS&P*, 2022.

[66] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[67] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford Alpaca: An instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[68] The Linux Kernel Community. How to use PCI IOV (single root I/O virtualization). https://docs.kernel.org/PCI/pci-iov-howto.html, 2009.

[69] The Linux Kernel Community. VFIO - virtual function I/O. https://docs.kernel.org/driver-api/vfio.html, 2025.

[70] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *IEEE S&P*, 2019.

[71] Junpeng Wan, Yanan Guo, Zhi Zhang, Zhuo Li, Dave (Jing) Tian, and Zhenkai Zhang. GeForge: Hammering GDDR memory to forge GPU page tables for fun and profit. In *IEEE S&P*, 2026.

[72] Da Yan, Wei Wang, and Xiaowen Chu. Optimizing batched Winograd convolution on GPUs. In *PPoPP*, 2020.

[73] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, et al. Qwen2 technical report, 2024.

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

[75] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, et al. OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[76] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. Understanding the GPU microarchitecture to achieve bare-metal performance tuning. In *PPoPP*, 2017.

[77] Yicheng Zhang, Ravan Nazaraliyev, Sankha Baran Dutta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Beyond the bridge: Contention-based covert and side channel attacks on multi-GPU interconnect. In *SEED*, 2024.

[78] Yicheng Zhang, Ravan Nazaraliyev, Sankha Baran Dutta, Andres Marquez, Kevin Barker, and Nael Abu-Ghazaleh. NVBleed: Covert and side-channel attacks on NVIDIA multi-GPU interconnect. *arXiv preprint arXiv:2503.17847*, 2025.

[79] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. TunneLs for bootlegging: Fully reverse-engineering GPU TLBs for challenging isolation guarantees of NVIDIA MIG. In *CCS*, 2023.

[80] Zhenkai Zhang, Kunbei Cai, Yanan Guo, Fan Yao, and Xing Gao. Invalidate+Compare: A timer-free GPU cache attack primitive. In *USENIX Security*, 2024.

[81] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. Towards high performance paged memory for GPUs. In *HPCA*, 2016.

A Open Science

The artifact is hosted at https://github.com/meowmeowxw/nvidia-mig-pf-side-channel. It contains: (i) CUDA probes and interferers such as `attack.cu`, `pf_latency.cu`, `pf_latency_llm.cu`, `cache_accesses_victim.cu`, `cache_accesses_loop.cu`, `pcie_monitor.c`, and `pcie_noiser.cu`; (ii) workload drivers such as `torch_ml.py`, `vllm_runner.py`, and `cudf_example.py`; (iii) orchestration scripts `run_docker_*.sh`; (iv) analysis scripts such as `plotter_cache.py`, `plotter_simple.py`, `plotter_side_channel.py`, `classifier_ml.py`, and `classifier_llm.py`; and (v) the container and dependency files needed to rebuild the software environment (`Dockerfile`, `requirements.txt`, `Makefile`, and `setup.sh`).

We do *not* provide the vGPU artifacts. Reproducing the vGPU experiments requires a valid NVIDIA AI Enterprise license and access to proprietary guest and hypervisor components that cannot be redistributed.

Environment. We validated the artifact on Ubuntu 24.04 with an NVIDIA A30 (Ampere) GPU, NVIDIA Open GPU Kernel Modules v570.133.07, CUDA 12.8, Docker, and NVIDIA Container Toolkit. The helper script `setup.sh` installs the software dependencies, enables MIG, creates the two 12 GiB slices used in the paper, builds the `ai_image` container, and downloads the public workloads used by the containerized experiments.

Main experiments. To reproduce Figures 3 and 4, run `./run_docker_cross_gi.sh`. This collects the raw cross-GI interference traces under `logs_cache/`. The plots are then generated with

```
./plotter_cache.py -s p99 -o figs/ \
--log_scale on
```

Figure 8 is reproduced by running `./run_docker_simple.sh`, which writes page fault latency traces for the inactive, PyTorch, vLLM, and cuDF cases to `logs_simple/`. The corresponding plot is generated with

```
./plotter_simple.py --logs_dir ./logs_simple \
--plot_type tail_avg --window_size 3000 \
--fig_output ./figs/pf_latency_simple.pdf \
--show_uncertainty --log_scale on
```

Figure 6 uses the same page fault measurement primitive, but swaps the co-located workload for synthetic GPU memory kernels. Reviewers can reproduce it by keeping the inactive baseline from `logs_simple/inactive`, running `./run_docker_side_channel_cache.sh`, and then plotting the resulting traces with

```
./plotter_side_channel.py --cache-dir \
logs_side_channel_cache --baseline-file \
logs_simple/inactive --output \
figs/side_channel_p99_overhead.pdf.
```

Fingerprinting models. Run `./run_docker_ml.sh` to reproduce the basic ML fingerprinting experiment. The script collects execution traces for VGG, ResNet, DenseNet, and MobileNetV2 and stores them in `logs_ml/`. Then, train the classifier using:

```
./classifier_ml.py --logs_dir ./logs_ml
--window_size 10000
```

For a vLLM-based fingerprinting reproduction, run `./run_docker_vllm.sh` to collect execution traces in `logs_llm/`. Then, train the classifier using:

```
./classifier_llm.py --logs_dir ./logs_llm
--window_size 40000 --step 3000
--columns 0,1
```

The `--columns0,1` option uses both the GPU-side and host-side timing columns, matching the setup described in Section 8.

B Ethical considerations

We reported the UVM 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-2025-23352 to the vGPU bug (Section 9.2), and acknowledged the side-channel risk.