Security and Performance Implications of GPU Cache Eviction Priority Hints

Qizhong Wang
Computer Science and Engineering
University of California, Santa Cruz
Santa Cruz, USA
qwang148@ucsc.edu

Xiangyue Huang
Computer Science and Engineering
University of California, Santa Cruz
Santa Cruz, USA
hxiangyu@ucsc.edu

Yanan Guo
Computer Science
University of Rochester
Rochester, USA
yguo51@cs.rochester.edu

Yuanchao Xu
Computer Science and Engineering
University of California, Santa Cruz
Santa Cruz, USA
yxu314@ucsc.edu

Abstract

NVIDIA provides cache eviction priority hints such as *evict_first* and *evict_last* on recent GPUs. These hints allow users to specify the eviction priority that should be used for individual cache lines to improve cache utilization. However, NVIDIA does not disclose the microarchitectural details of these hints or cache eviction behaviors when using them, which makes their security and performance implications unclear.

In this paper, we first reverse engineer the detailed comprehensive behaviors of these eviction priority hints. Then based on our findings, we analyze their impact on system security and performance. First, we found that these priority hints introduce new security problems. Specifically, we develop a new covert channel using the evict_first priority hint, which is more efficient than existing GPU covert channels. We also demonstrate a performance degradation attack using the evict_last priority hint, which is more stealthy compared to the known methods. Second, from the performance perspective, we show that marking a cache line as evict_last does not always keep it in the cache. In fact, if more than 12/16 (or 3/16, depending on the driver version) of the L2 cache size worth of data are marked as evict_last, cache thrashing can occur, which leads to performance degradation for real GPU workloads.

ACM Reference Format:
Qizhong Wang, Xiangyue Huang, Yanan Guo, and Yuanchao Xu. 2025. Security and Performance Implications of GPU Cache Eviction Priority Hints. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3725843.3756116

1 Introduction

Graphics Processing Units (GPUs) were originally designed for high-quality graphics rendering in video games and multimedia applications. However, they have since evolved into general-purpose computing platforms capable of handling a broad range of workloads. Their massive number of cores and high memory bandwidth allow them to perform complex computations far more efficiently than traditional CPUs. Today, GPUs have become a core part of the computing infrastructure in many fields, such as deep learning [4, 5, 17, 20, 42], bioinformatics analysis [25, 28, 48], encryption [2, 46], and climate science [14, 32]. Their ability to accelerate large-scale computations continues to drive scientific and technological advancements.

Cache plays a critical role in GPU architecture and is essential for achieving high performance in GPU applications. NVIDIA GPUs, for example, employ a two-level cache hierarchy: each Streaming Multiprocessor (SM) has a private L1 cache, and all SMs share the L2 cache. Similar to CPUs, GPU caches utilize a Least Recently Used (LRU) replacement policy [8, 56]. This policy estimates which cache line is least likely to be reused in the near future based on access history: the cache line unused for the longest time is considered least useful and selected for eviction when space is required. However, data reuse patterns can be complex, and the LRU policy's predictions are not always accurate. Sole reliance on this strategy may lead to suboptimal cache utilization and, consequently, reduced performance. To address this, NVIDIA introduced *cache eviction priority hints* in recent GPUs (starting with the Volta microarchitecture [36]).

Common cache eviction priority hints include *evict_first* and *evict_last*. These hints can be attached to load and store instructions to guide the hardware's eviction decisions. Specifically, when a cache set requires eviction, the hardware considers these hints along with the replacement policy to determine which cache line should be removed. According to the NVIDIA document [38], data cached with the *evict_first* priority is given the highest eviction priority and will likely be evicted first when cache eviction is necessary. This makes it suitable for data accessed in a streaming manner. Conversely, data cached with the *evict_last* priority has the lowest eviction priority and will likely be evicted only after data with other priorities has already been removed. This hint is intended for data that should be retained in cache due to frequent reuse. However, NVIDIA has not disclosed specific details about how these hints operate at the microarchitectural level, raising two key concerns:

First, from a security perspective, these priority hints give software users more direct control over the cache eviction and state, which may simplify the construction of covert channels by attackers. In addition, the *evict_last* priority hint, in particular, could be exploited to launch performance degradation attacks in multi-tenant environments. Second, without a clear understanding of how these hints are implemented in hardware, it is difficult for users to determine the most effective way to use them for performance optimization.

Therefore, in this paper, we address these concerns by reverse engineering the microarchitectural implementation of the cache eviction priority hints and analyzing their security and performance implications. Specifically, we conduct a series of experiments to study the microarchitectural behavior of these hints. We focus on two key aspects: 1) the maximum number of *evict_first* and *evict_last* cache lines that can simultaneously exist in an L2 cache set, and 2) how these *evict_first* and *evict_last* cache lines interact with each other and with the "normal" cache lines in the set. We examine these behaviors in both single-process and multi-process scenarios. For multi-process scenarios, we consider both single-GPU systems with Volta-MPS enabled and multi-GPU systems. Based on several critical observations obtained from reverse engineering, we further investigate how these priority hints influence system security and performance.

Security implications. We found that, in most cases, only one *evict_first* cache line can exist in an L2 set at a time. When eviction is required, this cache line is always selected for removal. This property enables a highly efficient conflict-based cache covert channel on GPUs. Specifically, in traditional conflict-based covert channels, the sender and receiver must manipulate the entire cache set to generate conflicts. This process usually requires many operations and is relatively inefficient. In contrast, using the *evict_first* priority hint, the sender and receiver can target the single slot allocated for *evict_first* cache lines, creating conflicts with far fewer operations. This leads to a significantly more efficient covert channel. Our experiments demonstrate that this channel achieves a bandwidth of 40.14 Mb/s, which is substantially higher than the bandwidths of the existing GPU covert channels [1, 8, 31].

In addition, we found that multiple *evict_last* cache lines can exist in an L2 set, and these *evict_last* cache lines cannot be evicted by "normal" cache lines. This can lead to performance degradation attacks: in a multi-tenant system, if one user marks several cache lines per L2 set with the *evict_last* priority, they can effectively "pin" those cache lines in the L2 cache set, which significantly reduces the cache availability for other users. Unlike common performance degradation attacks that rely on resource contention and thus require frequent data accesses to cause contention, this method requires minimal access activity, which makes it much more stealthy. Our results show that, in a multi-GPU system, an attacker using this technique can degrade another user's performance by nearly 40%.

Performance implications. As explained earlier, NVIDIA recommends using the *evict_last* priority for cache lines that are frequently reused, to effectively "pin" them in the L2 cache. However, they do not specify how many cache lines should—or can—be marked in this way. Intuitively, one might assume it is safe to mark up to the full L2 cache size worth of data as *evict_last*, to keep all of it resident in the cache and maximize cache utilization. However, we found that GPUs only allow a maximum of 12 (or 3, depending on the GPU driver version) *evict_last* cache lines per 16-way L2 cache set. If more cache lines are marked with this priority hint, they cannot remain pinned. Instead, accessing them is likely to cause cache thrashing, leading to a situation where all accesses—originally expected to be L2 hits—become L2 misses. Our experiments show that the amount of data marked as *evict_last* has a significant impact on the performance of real-world workloads. Marking more than 12/16 (or 3/16) of the L2 cache size worth of data not only fails to improve performance but can actually degrade it, even if the data is frequently reused and should be marked as *evict_last* according to the NVIDIA document [38].

2 Background

2.1 GPU Architecture

Figure 1 shows the architecture overview of a typical GPU [36, 37]. Note that we follow NVIDIA's terminology since in this paper we mainly focus on the security and performance of NVIDIA GPUs. The basic processing units in a GPU are called streaming multiprocessors (SMs). Modern GPUs usually have tens to hundreds of SMs, and each SM can execute a group of parallel threads (known as a warp) in a Single-Instruction Multiple-Thread (SIMT) fashion. An SM comprises essential components that collaboratively manage and execute threads efficiently. The warp scheduler dynamically allocates active warps, optimizing resource utilization within the SM. Registers from the SM's register file are allocated to each thread in a warp. Standard arithmetic and logical operations are conducted on the CUDA cores, while specialized operations are handled by the Special Function Units (SFUs). Memory operations are handled by the Load/Store (LD/ST) units.

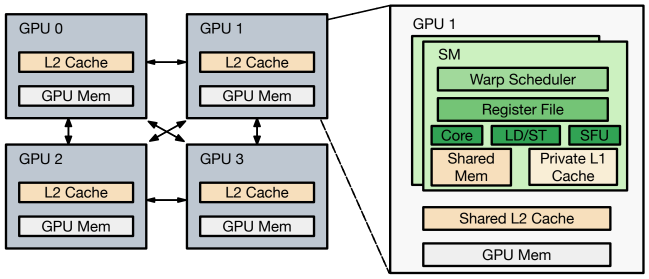

Figure 1: NVIDIA GPU architecture.

GPUs have their own dedicated cache/memory system. As shown in Figure 1, each SM has its own private L1 cache. SMs are connected to the shared L2 cache through a hierarchical on-chip network; the L2 cache is further connected to memory controllers, which interface with the off-chip device memory. Similar to host memory on the CPU side, device memory is also based on DRAM.

Many modern applications have computational demands that exceed the capabilities of a single GPU. To meet these needs, multi-GPU systems equipped with high-speed interconnects such as NVLink are increasingly used to enhance computational bandwidth.

NVLink facilitates a peer-to-peer communication mechanism, allowing one GPU to directly access another GPU's memory without involving the host. Note that although data in one GPU's memory can be accessed by a remote GPU, it cannot be stored in the remote GPU's L2 cache. Instead, it is always stored in the local L2 cache [8].

2.2 Volta-MPS

Historically, GPUs were limited to time-sharing among multiple processes; different processes could not run on the GPU in parallel, even when sufficient computing and memory resources were available. This model significantly restricted the utilization of GPU hardware resources. To address this limitation, NVIDIA introduced the Multi-Process Service (MPS), a technology that enables multiple processes to run simultaneously on a GPU.

However, the initial version of MPS required all concurrent processes to share the same address space, which raised substantial security concerns. To overcome this issue, NVIDIA later introduced Volta-MPS, ***which allows workloads to run in separate address spaces in parallel***. Subsequent NVIDIA GPUs can now utilize Volta-MPS to enhance GPU utilization while maintaining security.

2.3 GPU Cache Eviction Priority Hints

Different types of data have different access patterns, and thus a single cache replacement policy may not be optimal for all types of data. To enhance cache utilization, NVIDIA offers several cache eviction priority hints such as *evict_first*, *evict_last*, and *evict_normal*, which can be applied to load/store instructions [38] to guide cache eviction behavior. These hints influence the order in which data is evicted from the corresponding cache set.

Specifically, according to the NVIDIA document [38], *evict_normal* is the default priority; when no priority is specified in the instruction, the data is flagged as *evict_normal*. In addition, data cached with the *evict_first* priority will be placed first in the eviction priority order and will likely be evicted when cache eviction is required. This priority is suitable for streaming data. In contrast, data cached with the *evict_last* priority will be placed last in the eviction priority order and will likely be evicted only after data with the *evict_normal* or *evict_first* eviction priority has been evicted. This priority is suitable for data that should remain persistent in the cache.

```
1 /* Create an evict_last policy for addr to addr+128.*/
2 createpolicy.range.L2::evict_last.b64 cache_policy, [addr
      ], 128, 128;
3
4 /* Load data with the evict_last priority into L2 */
5 ld.global.L2::cache_hint.u64 val, [addr], cache_policy;
```

Listing 1: An example of using the GPU cache eviction priority hints (in PTX).

Listing 1 provides an example of how to implement and utilize the eviction hint priorities. Specifically, one must first establish the eviction policy for a specific address range (Line 2). Afterward, this eviction policy can be applied to subsequent load and store instructions that target data (cache lines) within this address range (Line 5).

2.4 Covert Channels

A covert channel exploits unintended communication pathways [10, 35], such as shared resources like shared cache and memory, to secretly transfer information. This unauthorized data transfer typically involves two main components: a sender, who encodes information into the shared resources by altering their state, and a receiver, who decodes this information by monitoring changes in those resources. Covert channels enable the transmission of sensitive or private data between users who are not authorized to communicate, thus raising significant security concerns. Cache conflict-based timing covert channels are among the most powerful covert channels on modern processors. In these channels, the sender transmits information by causing conflicts in a certain cache set, and the receiver retrieves the information by measuring these conflicts using timing information.

For example, in Prime+Probe [26], to transfer one bit of information, the receiver first primes the cache set by filling the set with the receiver's cache lines. Then, the sender sends a bit "1" by bringing the sender's cache lines into this set, or sends "0" by not bringing the cache lines. Later the receiver receives the bit by probing this set (i.e., re-accessing the cache lines from the priming stage) and measuring the probing latency: if the sender sent "1", then at least one of the receiver's cache lines were evicted and it now takes longer to probe; if the sender sent "0", then none of the receiver's cache lines was evicted and it is now faster to probe. Prime+Probe was originally implemented on the CPUs; however, recently researchers have proven that it works on the GPUs as well [8, 33, 56].

3 Motivation

NVIDIA provides only high-level documentation on cache eviction priority hints (cf. Section 2.3) and does not disclose their microarchitectural implementation details. Without this information, the security and performance implications of these operators remain unclear. From a security perspective, these hints give users more direct control over the cache state, potentially enabling new and more efficient cache-based covert channels. In addition, in multi-tenant scenarios, an attacker could exploit the *evict_last* priority hint to degrade cache performance for other users, potentially leading to performance degradation attacks. From a performance perspective, NVIDIA suggests applying the *evict_last* hint to data that "should remain in cache". However, NVIDIA does not specify how much data can be marked this way, or whether doing so guarantees the data will stay in the cache. Without understanding these details, it is challenging for users to decide how to use these hints effectively to improve performance. Therefore, the goal of this paper is to reverse engineer the implementation of these operators and analyze their security and performance implications.

4 Characterization of Eviction Priority Hints

The experiments in this paper are conducted on three systems, including two single-GPU systems and one multi-GPU system, as listed in Table 1. Due to space limitations, we primarily show the results obtained on the single-GPU system with one NVIDIA RTX 3080 GPU (unless specified otherwise). Note that by default we use CUDA 11.6 with driver version 510.39.01 for the experiments. However, in Section 4.2.5, we change the CUDA and driver versions as specified in Table 1 to explore how different GPU configurations affect the behavior of the eviction priority hints. These hints apply to both L1 and L2 caches; however, in this paper, we focus on the L2 cache, as it plays a critical role in GPUs.

Table 1: System specifications.

System Type	GPU Model	CUDA	Driver	L2	Link
Single-GPU System #1	RTX 3080	11.6–12.0	510–550.x	5MB	–
Single-GPU System #2	A40-48GB	11.8–12.0	525–550.x	6MB	–
Single-GPU System #3	RTX 4090	11.8	535–570.x	72MB	–
Multi-GPU System	A100-80GB (SXM4)	11.8	550.x	40MB	NVLink

4.1 The *evict_first* Priority Hint

In this section, we conduct experiments to characterize the detailed behavior of the *evict_first* priority hint. Specifically, we focus on answering three questions: 1) whether an *evict_first* cache line is actually the "first one" that is evicted in the set; 2) whether an *evict_first* cache line becomes *evict_normal* when it is accessed by a regular load or store instruction; and 3) how the *evict_first* cache lines and *evict_normal* cache lines interact within an L2 cache set. We first characterize this eviction priority hint in a single-process scenario, and then verify the conclusions in multi-process scenarios.

4.1.1 Eviction policy. We first verify that in an L2 set, if there is one cache line marked as *evict_first*, it will indeed be the one evicted when eviction occurs in this set. We achieve this using a four-step experiment. To prepare for the experiment, we construct an eviction set, i.e., a group of cache lines mapped to a specific L2 cache set. We build the eviction set using methodologies proposed in prior studies [8, 56]. The detailed steps of the experiment are as follows:

- Step 1: We fill a specific L2 set with 16 cache lines from the eviction set (l_0 to l_{15}); we load these cache lines without specifying the eviction priority (i.e., they are marked as *evict_normal*).
- Step 2: We use the `discard` instruction to remove one of these cache lines (l_i, where $0 \le i \le 15$) from the L2 cache. Then, we load a new cache line l_{16} from the eviction set with the *evict_first* priority to fill the vacated position.
- Step 3: We reload l_i to bring it back into the L2 set and trigger eviction.
- Step 4: We check the status of l_0 to l_{16} (except l_i) to determine which cache line was evicted in Step 3.

Results. We repeat the above experiment with different values of i, and we find that regardless of the value of i, l_{16} is always the one evicted. This confirms that using the *evict_first* priority indeed causes the cache line to be the first one evicted from the L2 set.

4.1.2 Update policy. In this section, we study how the eviction state of a cache line is updated. Specifically, we examine whether, when a cache line is initially loaded into the cache with the *evict_first* priority, this *evict_first* status remains after the cache line is accessed with a regular load or store instruction (without the priority hint). We achieve this with an experiment similar to the one explained in Section 4.1.1. However, in Step 2 of the experiment, after loading l_{16} as *evict_first*, we access it again without the priority hint.

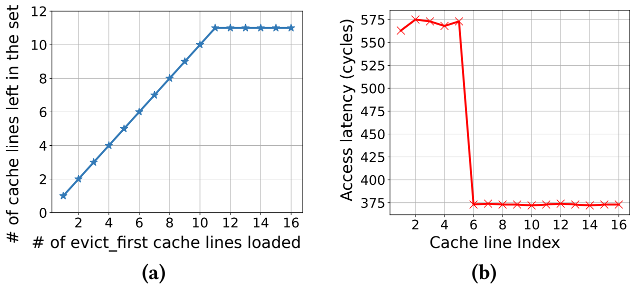

Figure 2: (a) The number of cache lines left in the L2 set after loading 1 to 16 evict_first lines; (b) the latency of re-accessing each cache line, after 16 evict_first lines are loaded.

Results. We find that in this experiment, l_{16} is no longer the cache line that is evicted in Step 4. Instead, l_0, the LRU cache line, is the one evicted. We further expand the experiment by accessing multiple cache lines (and triggering multiple evictions) in Step 3; we find that l_{16} is not evicted until all other cache lines in the set have been evicted. This indicates that a regular load or store instruction clears the *evict_first* flag for l_{16} and makes it the most recently used cache line in the set.

Takeaway 1: When a single cache line in an L2 set is marked as *evict_first*, it is the next cache line to be evicted in the set, regardless of its LRU state. However, accessing this cache line with a regular load or store instruction (without a priority hint) changes its status to *evict_normal*.

4.1.3 Interactions among evict_first cache lines. The previous section examined scenarios with only one *evict_first* cache line in an L2 set. In this section, we aim to explore the possibility and implications of having multiple *evict_first* cache lines within a single set. Specifically, we aim to answer two questions: 1) how many *evict_first* cache lines can simultaneously exist in an L2 set, and 2) when there are multiple *evict_first* cache lines in a set, which one is evicted first. We achieve this goal through a two-step experiment. In the first step, we prepare an empty L2 set,[1] and then load l_0 to l_i from the eviction set with the *evict_first* priority ($0 < i \le 15$). Next, we check the status of each of these cache lines (i.e., whether they remain in the L2 cache) by re-accessing them and collecting the access latency.

Results We repeat the experiment with varying values of i, and the results are shown in Figure 2a. We find that, starting with an empty L2 set, we can load a maximum of 11 *evict_first* cache lines without causing eviction; loading more than 11 cache lines with the *evict_first* priority causes these cache lines to evict each other (despite the set not being full). Additionally, we observe that these 11 cache lines follow an LRU-based policy. For example, as shown in Figure 2b, when loading 16 cache lines with this priority, the first 5 cache lines loaded are evicted.

[1]This can be achieved by using the `discard` instruction to flush all the cache lines in the set.

Takeaway 2: Starting with an empty L2 set, the maximum number of *evict_first* cache lines that can concurrently reside in a single cache set is 11; these cache lines follow an LRU-based eviction policy.

4.1.4 Interactions among evict_first and evict_normal cache lines. Here we explore how evict_first and evict_normal cache lines influence each other within an L2 set. Again, we conduct a two-step experiment: in the first step, we load 11 *evict_first* cache lines into an empty L2 set, and then we load n_0 *evict_normal* cache lines that are also mapped to this set ($0 < n_0 \leq 17$). In the second step, we check the state of each cache line using timing information to determine the number of *evict_first* and *evict_normal* cache lines that remain in the L2 cache, respectively.

Note that the maximum value of n_0 is set to 17 instead of 16 because we find that on our GPU, filling an L2 set using normal load operations requires 17 cache lines. Specifically, when loading the cache lines one by one into an empty L2 set, after the 15th cache line is loaded, the 16th cache line evicts the first one, even though the set is not yet full. Loading the 17th cache line then completes the filling of all 16 entries of the cache set. This observation aligns with the replacement policy that was reverse engineered in prior work [56].

Results. We repeat the above experiment with different values of n_0, and Figure 4a shows the number of *evict_first* and *evict_normal* cache lines that remain in the set (collected from Step 2). The results indicate that when continuously introducing *evict_normal* cache lines into the set, these cache lines initially replace the *evict_first* cache lines even if the set is not yet full. After all the *evict_first* cache lines are evicted, additional *evict_normal* cache lines occupy the previously unoccupied slots in the set. Specifically, as shown in Figure 4a, loading four *evict_normal* cache lines causes four of the *evict_first* cache lines to be evicted (even though the set has five unoccupied slots); loading 14 *evict_normal* cache lines causes the eviction of all 11 *evict_first* cache lines and fills three of the unoccupied slots.

Figure 3 shows the number of (a) *evict_first*, (b) *evict_normal*, and (c) total (*evict_first* + *evict_normal*) cache lines remaining in the set, collected from Step 2. The sub-captions indicate the type of remaining cache lines shown in each sub-figure. When there are fewer than 11 total cache lines in the set (*evict_first* + *evict_normal*), loading *evict_normal* cache lines does not cause the *evict_first* cache lines to be evicted. However, once this number reaches 11 or more, eviction occurs. For example, as highlighted in the figures, with four *evict_first* cache lines initially in the set, up to seven *evict_normal* cache lines can be loaded without triggering eviction. Loading more cache lines starts evicting the existing *evict_first* lines.

Note that to better understand the interactions between *evict_first* and *evict_normal* cache lines, we expand the above experiments by varying the number of initial *evict_first* cache lines in the set (rather than fixing it at 11). The results are shown in Figure 3. We observe that when there are fewer than 11 total cache lines in the set, loading *evict_normal* cache lines does not cause the *evict_first* cache lines to be evicted. However, once this number reaches 11 or more, eviction occurs. For example, as highlighted in the figure, with four *evict_first* cache lines initially in the set, up to seven *evict_normal* cache lines can be loaded without triggering eviction. Loading more than seven cache lines starts evicting the existing *evict_first* lines.

Takeaway 3: When an L2 set is not fully occupied (but contains *evict_first* cache lines), loading *evict_normal* cache lines may cause eviction of those *evict_first* cache lines rather than filling the available slots.

The above experiment explains how *evict_normal* cache lines evict the *evict_first* ones. To understand how the *evict_first* cache lines evict the *evict_normal* ones, we reverse the loading order in the above experiment: we load n_0 *evict_normal* cache lines ($0 < n_0 \leq 16$) into an empty L2 set, and then load 17 *evict_first* cache lines. Finally, we check how many cache lines remain in the cache.

From Figure 4b, we have two observations. First, when $n_0 < 11$ (meaning fewer than 11 *evict_normal* cache lines are loaded), we can load $11 - n_0$ *evict_first* cache lines into the L2 set without triggering eviction. Loading additional *evict_first* cache lines leads to self-eviction among them. This finding aligns with the earlier observation. Second, if 11 or more *evict_normal* cache lines are loaded, only one *evict_first* cache line can be loaded into the set (evicting exactly one *evict_normal* cache line); attempting to load more causes self-eviction among the *evict_first* cache lines.

Takeaway 4: Multiple *evict_first* cache lines can exist in an L2 set only when there are fewer than 11 cache lines present. When the total number of cache lines in an L2 set exceeds 11, the set can maintain only one *evict_first* cache line.

4.1.5 Multi-process and multi-GPU scenarios. In this section, we aim to test whether the properties observed above apply to scenarios in which multiple processes are running. We consider two specific cases: 1) multiple processes running in parallel on a single GPU using Volta-MPS, and 2) multiple processes running in parallel on a system with multiple GPUs.

To achieve this, we replicate the experiments from Section 4.1.1 through Section 4.1.4. However, rather than conducting all experimental steps within a single process, we distribute them across multiple processes. For instance, in the experiment described in Section 4.1.4, we let one process load the *evict_normal* cache lines and another process load the *evict_first* cache lines. Our results show that the conclusions drawn from single-process experiments remain valid in multi-process scenarios, both when using Volta-MPS and when using multiple GPUs.

4.2 The *evict_last* Priority Hint

In this section, we study another eviction priority hint, *evict_last*. Similar to Section 4.1, we first verify whether an *evict_last* cache line is indeed the "last one" evicted from an L2 set, and then we characterize the implementation details of this priority hint.

4.2.1 Eviction and update policy. According to the NVIDIA document [38], a cache line marked as *evict_last* is assigned the lowest eviction priority and will likely be evicted only after the *evict_normal* and *evict_first* cache lines have been evicted. To verify

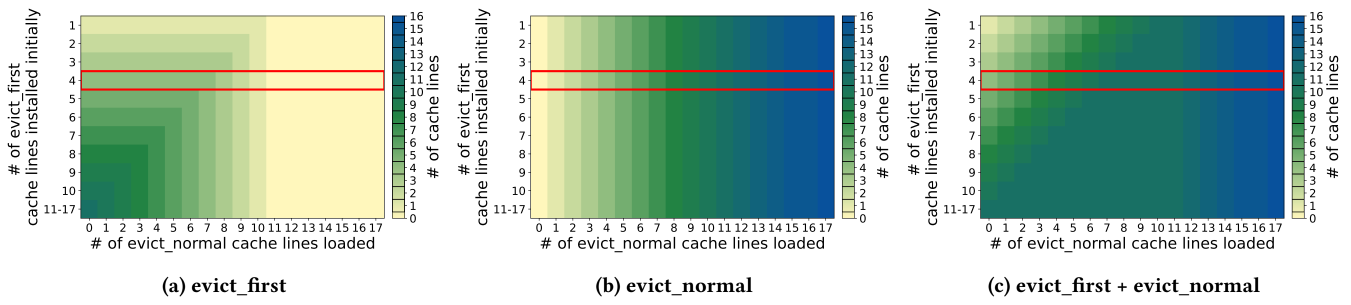

Figure 3: The number of (a) evict_first, (b) evict_normal, and (c) evict_first + evict_normal cache lines remaining in the set after loading 1 to 17 evict_first cache lines and then 0-17 evict_normal cache lines.

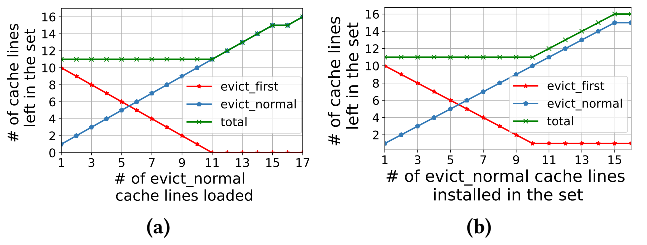

Figure 4: The number of evict_first and evict_normal cache lines left in the set after (a) loading 1 to 17 evict_normal cache lines into a set initially installed with 11 evict_first cache lines, or (b) loading 17 evict_first cache lines into a set initially installed with 1 to 16 evict_normal cache lines.

this, we conduct an experiment similar to the one in Section 4.1.1: we fill an L2 set with *evict_normal* and *evict_first* cache lines, and then replace one line with an *evict_last* cache line. Next, we load additional cache lines to cause evictions in this set and determine whether the *evict_last* cache line is evicted from the set.

Results. From the experiment, we find that as long as there are *evict_first* or *evict_normal* cache lines in the L2 set, the *evict_last* cache line is not considered for eviction. This means that in practice, when a single cache line in an L2 set is marked as *evict_last*, it is effectively "pinned" in the L2 cache. Note that we also observe that in multi-process scenarios, if an *evict_last* cache line has not been accessed for an extended period, it may be removed from the L2 cache by the GPU system (see Section 4.2.6 for more details).

In addition, we conduct an experiment similar to the one in Section 4.1.2 to examine how the state of an *evict_last* cache line is updated. We find that, unlike scenarios involving the *evict_first* priority hint, when an *evict_last* cache line is accessed by a regular load or store instruction (without the hint), it remains an *evict_last* cache line.

Takeaway 5: When a single cache line in an L2 set is marked as *evict_last*, it is the last cache line to be evicted (effectively "pinned") in the L2 set. Additionally, accessing it with a regular load or store instruction without a priority hint does not change its status.

4.2.2 *Interactions among evict_last cache lines.*

In this section, we investigate the maximum number of *evict_last* cache lines that can simultaneously reside in an L2 set. Following the methodology from Section 4.1.3, we conduct a two-step experiment. In the first step, we prepare an empty L2 set and then load i cache lines (where $i > 0$) into the set with the *evict_last* priority hint. In the second step, we check the status of each cache line to determine whether they remain in the L2 cache or have been evicted.

Results. We repeat this experiment with different values of i, and the results are shown in Figure 5a. Interestingly, we find there can be up to 16 *evict_last* cache lines in an L2 set, meaning the set can be completely filled with *evict_last* cache lines. This differs from the behavior observed with *evict_first* cache lines.

Note that, as shown in Figure 5a, filling the 16-way L2 set requires 17 cache lines: loading the 16th cache line causes the 15th cache line to be evicted. This behavior matches what was observed with *evict_normal* cache lines (cf. Section 4.1.4).

Takeaway 6: Starting with an empty set, the maximum number of *evict_last* cache lines that can simultaneously reside in an L2 set is 16.

4.2.3 *Interactions among evict_last and evict_normal cache lines.*

Here, we examine how the *evict_last* and *evict_normal* cache lines interact within an L2 set. Specifically, we aim to understand two aspects: 1) how loading *evict_normal* cache lines impacts the *evict_last* cache lines in the set, and 2) the reverse—how loading *evict_last* cache lines impacts the *evict_normal* cache lines. Note that while Section 4.2.1 demonstrated that when only a single *evict_last* cache line exists in an L2 set, it is never evicted by an *evict_normal* cache line, this conclusion may not hold when multiple *evict_last* cache lines are present.

To determine how loading *evict_normal* cache lines affects the *evict_last* cache lines, we conduct a two-step experiment. In the first step, we fill an L2 set with *evict_last* cache lines, and then we load n_0 *evict_normal* cache lines ($0 < n_0 \leq 17$) into this set. In the second step, we check the status of each loaded *evict_normal* and *evict_last* cache line to determine whether it remains in the L2 cache.

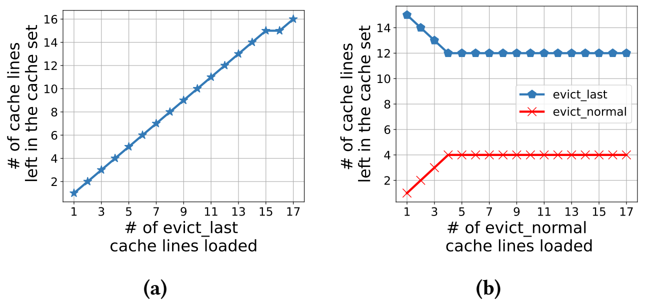

Figure 5: (a) The number of cache lines left in the set after loading 1 to 17 evict_last cache lines; (b) the number of evict_last and evict_normal cache lines left in the set after loading 1 to 17 evict_normal cache lines into a set initially installed with 16 evict_last cache lines.

Results. Figure 5b shows the number of *evict_normal* and *evict_last* cache lines that remain in the set after running the above experiment with varying values of n_0. We find that we can bring in up to four *evict_normal* cache lines (thereby evicting four *evict_last* cache lines from the set); attempting to bring in additional *evict_normal* cache lines only results in evictions among the *evict_normal* cache lines themselves. As a result, 12 *evict_last* cache lines can reliably remain in the set without being affected by the *evict_normal* cache lines.

Takeaway 7: The *evict_normal* cache lines can evict *evict_last* cache lines only when there are more than 12 *evict_last* cache lines in the set; 12 or fewer *evict_last* cache lines can reliably remain in an L2 set without being affected by *evict_normal* cache lines.

Building on the findings in Figure 5b, we now explore how the state of an L2 set changes when new *evict_last* and *evict_normal* cache lines are introduced into a set that already contains both types. Specifically, we start with a set containing 12 *evict_last* cache lines and 4 *evict_normal* cache lines (based on the earlier results). We then alternately introduce new *evict_normal* and *evict_last* cache lines by accessing them and checking which existing cache line is evicted after each new addition.

We find that accessing a new *evict_last* cache line always evicts the LRU cache line in the set, regardless of its eviction hint. In contrast, accessing a new *evict_normal* cache line can evict only another *evict_normal* cache line when the set contains 12 or fewer *evict_last* cache lines. However, if there are more than 12 *evict_last* cache lines in the set, accessing a new *evict_normal* cache line evicts the LRU cache line among all the *evict_last* cache lines present in the set.

Later, in Section 6.1, we demonstrate that this observation has a critical impact on the performance of using the *evict_last* priority hint. Applying the priority hint naively to more than 12 cache lines per L2 set can lead to cache thrashing, resulting in a cache miss every time an *evict_last* cache line is accessed.

Takeaway 8: Loading an *evict_last* cache line into an L2 set always evicts the LRU cache line in the set; loading an *evict_normal* cache line into an L2 set can evict either type of cache line, depending on the number of *evict_last* cache lines currently present in the set.

4.2.4 *Multi-process and multi-GPU scenarios.*

In this section, we verify the behavior of the *evict_last* priority in scenarios where multiple processes are running on the GPU(s). Similar to Section 4.1.5, we focus on two configurations: Volta-MPS and multiple GPUs. Specifically, we aim to determine how *evict_last* and *evict_normal* cache lines from different processes interact. To achieve this, we replicate the experiments for the *evict_last* priority with two processes involved. Our results indicate that all previous conclusions regarding the *evict_last* priority remain valid in these scenarios.

Table 2: The maximum number of evict_last cache lines in an L2 set with different GPU driver versions.

Driver Version	Number of evict_last
510, 515, 525	12
535, 550, 560, 565, 570	3

4.2.5 *The effect of GPU driver/architecture.*

We repeated all experiments from Section 4.2 across various GPU driver versions using the GPUs listed in Table 1 to determine whether the conclusions remain consistent. We find that the driver version affects Takeaway 7. As shown in Table 2, older drivers (e.g., versions 515 and 525) allow up to 12 *evict_last* cache lines to remain in an L2 set without being evicted by *evict_normal* cache lines. With newer drivers, starting from version 535, this number drops to just 3. Importantly, this behavior appears to be independent of GPU architecture—on all tested GPUs (listed in Table 1), results are consistent as long as the driver version is the same. Note that not all driver–GPU combinations were tested, as some are incompatible. Additionally, we did not test Volta GPUs (sm_70) because they only support cache hints for L1 caches; L2 cache hint support requires Ampere (sm_80) or later [38].

Takeaway 9: The GPU driver version affects the maximum number of *evict_last* cache lines that can reliably reside in an L2 set without being evicted by *evict_normal* cache lines.

4.2.6 *"Unusual" eviction of evict_last cache lines.*

In multi-process scenarios with Volta-MPS or multiple GPUs, if one process extensively uses the *evict_last* hint to pin cache lines in the L2 cache, it can significantly impact the cache utilization and performance of other concurrently running processes. Thus, we investigate whether the GPU system or hardware has implemented any techniques to prevent this—i.e., whether it removes *evict_last* cache lines from the L2 cache under certain circumstances.

We find that if one process marks a cache line as *evict_last*, and at least one other process is actively using the L2 cache (instead of just the L1), this cache line is eventually removed from the L2 cache if it has not been accessed for a long period. Table 3 shows

Table 3: The access latency of an evict_last cache line after waiting for a different amount of time.

Waiting Time (cycles)	0	10^5	10^7	10^8	10^9
Access Latency (cycles)	347	348	346	347	545

the access latency for such a cache line after it remains untouched over various periods of time. It shows that the cache line is removed from the L2 cache after it has not been accessed for more than 10^8 cycles. Importantly, this behavior does not occur when no other process is using the L2 cache.

Takeaway 10: In multi-process scenarios, if two or more processes are actively using the L2 cache, an *evict_last* cache line is removed from the L2 cache after it has not been accessed for approximately 10^8 cycles.

4.3 Overall Performance/Security Implications

From the characterization results, the eviction priority hints may lead to both security and performance issues. Intuitively, *evict_first* facilitates creating set conflicts and may therefore enable more efficient covert channels. Additionally, *evict_last* allows a user to pin data in the cache, which can lead to performance degradation in multi-tenant scenarios. Moreover, the replacement policy for *evict_last* cache lines (Takeaways 7 and 8) may cause unexpected evictions when programmers intend to use *evict_last* to keep data resident in the cache. We explore these implications in detail in Sections 5 and 6.

5 Security Implications

In this section, we present the security implications of using these priority hints, specifically examining how attackers could leverage them to introduce new security concerns. In particular, we focus on two issues: cache covert channels and performance degradation attacks. We demonstrate that these priority hints enable 1) the design of new and more efficient GPU covert channel protocols, and 2) the introduction of new and more stealthy performance degradation attacks.

5.1 Covert Channel

Based on the reverse engineering results from Section 4, we construct a new GPU covert channel utilizing the *evict_first* priority hint. In this section, we first introduce the threat model and then discuss the details of this covert channel.

5.1.1 Threat model. We assume a threat model similar to that used in previous GPU covert channels [1, 8]. Specifically, the two essential parties in the covert channel—the sender and the receiver—are two unprivileged processes running simultaneously in either: 1) the same GPU with Volta-MPS enabled[2], or 2) different GPUs connected via interconnects like NVLink. The sender and receiver use different computing units (SMs) on the GPU(s) but share the L2 cache. We assume that the sender and receiver can identify a group of addresses mapped to the same cache set (e.g., using methods proposed in prior studies [8, 56]). They can also use the cache priority hints provided by NVIDIA. We further assume that the sender and receiver synchronize using the clock counter on GPUs, as in prior GPU covert channels [1, 8]. Additionally, we do not require any shared data between the sender and receiver.

5.1.2 The Load+Load covert channel. We develop a covert channel based on the *evict_first* priority hint, which we name Load+Load. Similar to previous conflict-based cache covert channels on GPUs, such as Prime+Probe, Load+Load operates by causing cache set conflicts. However, with Load+Load, the sender and receiver do not need to manipulate the entire L2 set to create conflicts. Instead, they trigger conflicts using the *evict_first* priority hint: since only one *evict_first* cache line can exist in a full L2 set, the *evict_first* cache lines of the sender and the receiver compete for this single slot, resulting in conflicts within this slot.

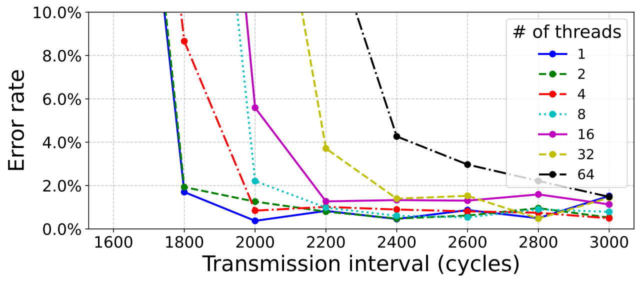

Figure 6: The channel error rates under various transmission intervals and numbers of threads used in the channel.

The detailed channel protocol is shown in Algorithm 1. Here we assume the sender and receiver are both single-threaded and utilize a single L2 set to transfer secret information. Multi-threaded scenarios are discussed later. The sender and the receiver must ensure that the sender's cache line l_s and the receiver's cache line l_r are mapped to the same L2 set. To prepare the channel, the receiver initializes the cache set by bringing 16 evict_normal cache lines into the set, and then loading l_r with the evict_first priority. After this setup, in each iteration of the data transfer, the sender sends "1" by accessing l_s with the evict_first priority, and sends "0" by not accessing it. The receiver then receives this bit by accessing l_r with the evict_first priority and timing the access. Specifically, if the sender sends "1", then l_s evicts l_r from the L2 cache, resulting in a longer access latency for l_r. In contrast, if the sender sends "0", then l_r remains in the L2 cache, resulting in a much shorter access latency.

Multi-threaded Covert Channels. In Algorithm 1, the sender and receiver transfer data using only a single L2 set. However, since GPUs support parallel execution across multiple threads, the sender and receiver can leverage multiple L2 sets to transfer multiple bits of data simultaneously, with each thread handling the data transfer using a different L2 set.

[2]Note that unlike MPS, Volta-MPS allows GPU kernels to run in different address spaces (cf. Section 2.2).

Algorithm 1: Load+Load

```
$l_s$: the sender's data (cache line) for transmitting signals
$l_r$: the receiver's data (cache line) for transmitting signals
$message[n]$: the n-bit long message to be transferred
$Th0$: the timing threshold for distinguishing hit and miss

Sender Algorithm
  for $i = 0$ to $n$ do
      Synchronization();
      if $message[i] = 1$ then
          load $l_s$ with evict_first cache_hint;
      else
          Do not load;
      end
      wait_for_receiver();
  end
end

Receiver Algorithm
  initialization();
  for $i = 0$ to $n$ do
      Synchronization();
      load $l_r$ with evict_first and time the load;
      if $load\_time > Th0$ then
          Received a bit "1";
      else
          Received a bit "0";
      end
  end
end
```

5.1.3 *Channel evaluation.* We implement Load+Load on two platforms: 1) a single-GPU system with Volta-MPS enabled, and 2) a multi-GPU system, to evaluate its efficiency and reliability. For the single-GPU experiments, we use the system equipped with an NVIDIA RTX 3080 GPU, as detailed in Table 1. For the multi-GPU tests, we use the system equipped with two NVIDIA A100 GPUs.

Single-GPU results. We measure the error rate of the channel using various numbers of threads and transmission intervals. The single-GPU results are shown in Figure 6. With a long transmission interval (i.e., a low transmission rate), the error rate remains low. However, when the transmission interval falls below a certain threshold, the error rate rises significantly. This threshold increases when more threads are used. We believe this is due to the limited L2 cache access bandwidth. When multiple threads access the L2 cache simultaneously, the access latency significantly increases, necessitating a longer interval to complete the access. Figure 7 shows the highest bandwidth achieved with 1 to 64 threads while maintaining an error rate below 1

Peak bandwidth. We find that to maintain a low error rate (e.g., ≤ 1.5%), the maximum bandwidth achievable by Load+Load is 59.3 Mb/s, using 160 threads spread across 5 thread blocks (and thus 160 L2 sets). In comparison, the maximum bandwidth of Prime+Probe

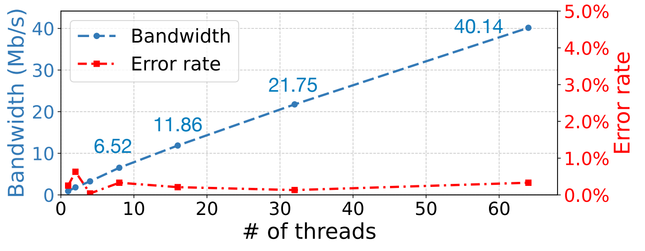

Figure 7: The bandwidths and error rates of Load+Load.

Table 4: The bandwidths and error rates of covert channels.

Channel	Leakage Source	Bandwidth	Error Rate
Load+Load	GPU L2 Cache	40.14 Mb/s	0.33%
GPU Prime+Probe [8, 33]	GPU L2 Cache	4.62 Mb/s	0.62%
Veiled Pathways [31]	GPU DRAM Frequency	0.5 bit/s	N/A
	GPU NVENC	0.5 bit/s	N/A
	GPU NVDEC	2 bit/s	N/A
	GPU NVJPEG	2 bit/s	N/A
	GPU-CPU PCIe	6.8 kb/s	3.3%
Network-on-Chip [1]	GPU NoC	24 Mb/s	0.6%
Streamline [41]	CPU LLC	14.41 Mb/s	0.35%
NTP + NTP [12]	CPU LLC	2.42 Mb/s	0.5%

is 6.1 Mb/s, using 128 threads. Further increasing the number of threads does not improve channel performance: as explained earlier, using more threads (e.g., 256) increases contention within the L2 cache and slows down each access. Consequently, to maintain a low error rate, the transmission interval must be lengthened to ensure accesses complete fully, which in turn reduces the overall channel bandwidth.

Compared to Other GPU Covert Channels. Table 4 compares the bandwidth and error rates of Load+Load with prior GPU covert channels, as well as several CPU-based channels (discussed in detail later). Note that we exclude channels that operate only within an SM, such as those based on functional units and the L1 cache [33]. The results show that Load+Load achieves the highest bandwidth while maintaining a comparable error rate. Since Load+Load is most closely related to GPU Prime+Probe, we provide a more detailed comparison between these two channels in Table 5. Similar to other conflict-based channels, Prime+Probe operates by triggering conflicts within a cache set; however, it requires the sender and receiver to access all 16 slots (ways) in the set. In contrast, the Load+Load approach requires accessing only a single way, making it significantly more efficient than Prime+Probe. As shown in Table 5, Load+Load consistently delivers substantially higher bandwidths than Prime+Probe, with slightly lower error rates. For example, using 32 threads, Load+Load achieves a bandwidth of 21.74 Mb/s and an error rate of 0.13%, whereas Prime+Probe achieves a bandwidth of only 2.39 Mb/s with an error rate of 0.78%.

Note that numerous cache covert channels have been implemented on the CPU side, and some achieve high bandwidths as well. Specifically, Streamline [41], which utilizes the entire CPU LLC for data transmission, achieves a bandwidth of 14.41 Mb/s. It is possible to implement Streamline on GPUs as well; however,

Table 5: The detailed bandwidths and error rates of Load+Load and Prime+Probe on a single-GPU system with Volta-MPS enabled.

Threads	Load+Load Bw. (Mb/s)	Prime+Probe Bw. (Mb/s)	Load+Load Err. Rate (%)	Prime+Probe Err. Rate (%)
1	0.90	0.10	0.25	0.10
2	1.81	0.19	0.63	0.17
4	3.26	0.37	0.04	0.38
8	6.52	0.72	0.33	0.43
16	11.86	1.27	0.21	0.75
32	21.74	2.39	0.13	0.78
64	40.14	4.62	0.33	0.62

this channel requires data sharing between the sender and receiver, making it significantly less practical than Load+Load. Additionally, NTP+NTP [12] is a covert channel similar to Load+Load, but designed for Intel CPUs. However, this channel only functions with inclusive LLCs, and modern Intel CPUs have largely transitioned to non-inclusive LLCs, significantly limiting the applicability of NTP+NTP. Furthermore, NTP+NTP utilizes the (non-temporal) prefetch instruction, which the hardware may sometimes ignore, such as during periods of high memory pressure.

Multi-GPU results. Table 6 shows the bandwidths and error rates of Load+Load and Prime+Probe in multi-GPU systems, with a single thread used by the sender and receiver. Here, we let the sender and receiver run on two different GPUs connected via NVLink. Similar to the single-GPU scenario, Load+Load achieves a higher bandwidth and a lower error rate compared to Prime+Probe.

Table 6: The bandwidths and error rates of Load+Load and Prime+Probe on a multi-GPU system.

	Bandwidth (Mb/s)	Error Rate (%)
Load+Load	0.125	0.97
Prime+Probe	0.015	0.92

Noise tolerance. Similar to other conflict-based cache covert channels such as Prime+Probe, Load+Load is affected by interference from co-running applications. Specifically, when the sender transmits a "0" by not triggering evictions, cache activity from other applications may still cause evictions and make the receiver misinterpret a "0" as a "1". To address this problem and improve channel reliability, the sender and receiver can use multiple sets to encode each bit: a bit is decoded as "1" only if eviction is detected in all sets. We implemented this approach using 32 L2 sets and measured the error rate while a workload is running concurrently with the channel. The results are shown in Table 7. Some workloads have small impact on the error rate. However, workloads with dense L2 activity–such as Vector-Add–can raise the error rate to around 50%, making the channel unusable.

Table 7: Error rates of Load+Load with different workloads; the variation is due to phase changes.

Workloads	Time/ Spin-ALU	SpMV	GEMM	Histogram	kNN	Vector Add
Err. Rate (%)	0.9–2.4	0.8–8.2	0.6–9.6	3.4–10.4	4.7–13.7	50.0

5.2 Stealthy Performance Degradation Attack

In Section 4.2.4, we showed that in multi-tenant environments, one GPU user can use the *evict_last* priority hint to effectively pin data in the L2 cache. When this happens, it can significantly affect cache utilization and therefore the performance of another GPU user running on the same system. In this section, we demonstrate that an attacker could exploit this feature to significantly degrade the performance of another user (i.e., the victim), resulting in a performance degradation attack.

We demonstrate this attack in a multi-GPU system. The victim is a benign application running on GPU0, handling tasks such as GEMM, QR decomposition, and Convolution. The attacker is a malicious application running on GPU1; the attacker's goal is to reduce the victim's performance. The attacker achieves this by pinning cache lines in GPU0's L2 cache using the *evict_last* priority hint. Note that the attacker can access GPU0's L2 cache using the method demonstrated in prior work [8]; this method does not require shared data between the attacker and the victim.

The attacker can also degrade performance by repeatedly scanning GPU0's L2 cache, that is, continuously bringing new data into the cache. This scanning causes the victim's data to be evicted from the L2 cache and increases the victim's cache access latency due to heightened contention for the limited L2 bandwidth. However, this method requires the attacker to access GPU0's L2 cache very frequently to achieve significant performance degradation. In contrast, the attack method leveraging the *evict_last* priority hint does not require frequent cache accesses; instead, the attacker only needs to refresh the pinned cache lines approximately every 10^8 cycles before they are removed from the L2 cache by the GPU system (see Section 4.2.6). This makes this method more stealthy and practical than the scanning-based method.

Table 8: The performance impact of using the evict_last priority and cache scanning on different applications.

Applications	Performance Degradation	
	Priority Hint	Scanning
GEMM	32.82%	11.87%
QR	39.77%	1.25%
Conv	33.58%	10.86%

We test the two attack methods, `scanning` and `priority_hint`, across several different victim applications. For the `priority_hint` method, the attacker uses this hint to pin three cache lines in each set of the L2 cache. For comparison, we implement these two methods using the same access frequency; specifically, the attacker either scans the L2 cache or re-accesses the pinned cache lines approximately every 10^8 cycles. For the `priority_hint` method, each round of access takes about 4.50×10^5 cycles (with 256 threads), causing the attacker to remain idle for 97.75% of the time. For the `scanning` method, each round of access takes about 4.57×10^5 cycles (with 1024 threads), resulting in a similar idle rate. We intentionally use more threads in the `scanning` method to maintain similar idle rates between the two methods (ensuring a fair comparison). As shown in Table 8, using the *evict_last* priority hint, the attacker can significantly degrade the victim's performance, reducing it by up to approximately 40%. In contrast, scanning the cache causes only about a 10% performance impact. In fact, we find that to achieve the same level of performance degradation as the priority hint method, the attacker would need to scan the cache over 40× more frequently.

Performance impact on streaming workloads. Workloads with streaming access patterns typically experience frequent L2 cache misses even without interference. Thus, pinning data in the L2 cache has minimal impact on their performance. To degrade their performance, the attacker can significantly increase its access rate to create contention for L2 bandwidth (effectively causing the `pinning` method to fall back to the `scanning` method). This approach results in approximately a 9% performance degradation for such workloads.

6 Performance Implications

In this section, we demonstrate the significance of our reverse engineering results from a performance perspective: we show that the benefits of the priority hints are conditional and require strategic application to be effective. As explained earlier, according to the NVIDIA document [38], marking a cache line as *evict_last* ensures it will be the last to be evicted from the L2 set, effectively pinning the cache line within the L2 cache. Therefore, to improve performance, one might intuitively mark data that will be frequently used as *evict_last*, ensuring the data remain in the L2 cache. However, NVIDIA does not specify how much data can or should be pinned. Intuitively, one might assume it is possible to pin as much data as the L2 cache can hold. Ideally, this would prevent data with low reuse potential, such as streaming data, from polluting the cache, while keeping frequently reused data consistently resident in the L2 cache.

However, according to our reverse engineering results (see Section 4.2), attempting to pin more than 12 cache lines per set (with older GPU drivers) or more than 3 cache lines per set (with newer GPU drivers) can lead to the "pinned" cache lines being evicted. In the worst case, this may result in cache thrashing for the "pinned" cache lines, where all accesses to these lines, expected to result in cache hits, actually result in misses. This scenario not only fails to improve performance, but can even degrade it.

In this section, we first design micro-benchmarks to demonstrate these scenarios. Then, we show the performance impact of pinning different amounts of data using the *evict_last* priority hint on several real-world workloads.

6.1 Analysis of Performance Impact of Varying Sizes of *evict_last* Data

Section 4.2 revealed two key insights. First, when there are more than 12 (or 3 with newer GPU drivers) *evict_last* cache lines in an L2 set, loading *evict_normal* cache lines can evict the existing *evict_last* cache lines from the set. Second, introducing a new *evict_last* cache line might cause the eviction of an existing *evict_last* cache line. This indicates that when both *evict_last* and *evict_normal* cache lines coexist in an L2 set, applying the *evict_last* priority hint to more than 12 (or 3) cache lines can lead to a high miss rate for these cache lines, rather than keeping them effectively "pinned" in the cache.

For example, consider a program that uses 14 cache lines accessed with the *evict_last* priority and other cache lines accessed with the *evict_normal* priority; all of these cache lines are mapped to the same L2 set. In the worst-case scenario, where the program frequently accesses the *evict_normal* cache lines, these lines remain relatively "young" in the set. Concurrently, if the program repeatedly scans the 14 *evict_last* cache lines at a much lower frequency, the LRU cache line in the set is likely to be one of the *evict_last* cache lines. This situation leads to cache thrashing among the *evict_last* cache lines: it essentially resembles a scenario where the program repeatedly scans 14 cache lines mapped to a 12-way (or 3-way) L2 set with an LRU replacement policy, resulting in cache misses for each access to these cache lines.

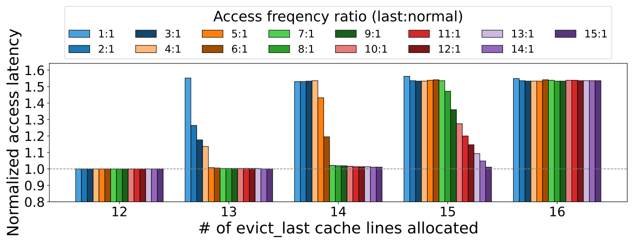

Figure 8: Access latency of evict_last cache lines with different numbers of evict_last cache lines per set and varying access frequency ratios between evict_last and evict_normal lines; the results are normalized to the access latency when all accesses to the evict_last cache lines hit in the L2 cache.

In more typical scenarios, where the *evict_normal* cache lines are accessed with a frequency comparable to or lower than that of the *evict_last* cache lines, the interaction between these cache lines can still hurt performance. When the program loads an *evict_last* cache line, it may cause an *evict_normal* cache line to be evicted. However, subsequent accesses to *evict_normal* cache lines can lead to the eviction of the oldest *evict_last* cache lines (once more than 12 *evict_last* lines reside in the set). This activity negatively affects the cache hit rate for the *evict_last* cache lines, which ideally should be 100%.

To understand the performance impact of this issue, we design microbenchmarks that alternately access two groups of cache lines, all of which are mapped to the same L2 set. The cache lines in the first group are marked as *evict_last* and are repeatedly scanned by the program. The cache lines in the second group are marked as *evict_normal*; each of these lines is accessed only once and not reused. Each microbenchmark varies the number of *evict_last* cache lines and the ratio between the access frequencies of *evict_last* and *evict_normal* cache lines. We measure the average access latency of the *evict_last* cache lines, and the results are shown in Figure 8. Note that the experiments are conducted on the RTX 3080 GPU using driver version 510 (see Table 1).

When the number of *evict_last* cache lines in the set is 12 or fewer, all accesses to these cache lines result in L2 cache hits, delivering the expected performance (low access latency). However, when this number exceeds 12, accesses to these cache lines may result in L2 misses, significantly increasing the average access latency. This issue intensifies as the ratio of access frequency between the *evict_last* and *evict_normal* cache lines decreases. Notably, having

16 *evict_last* cache lines in the set leads to a roughly 50% increase in average access latency, even when the access frequency ratio between the *evict_last* and *evict_normal* cache lines is as high as 15:1.

6.2 Performance Evaluation on Real-World Workloads

Building on the results in Section 6.1, this section demonstrates how the amount of data marked as *evict_last* can significantly impact the performance of real-world workloads. Specifically, we focus on four workloads: PageRank, Breadth-First Search (BFS), Fully Connected Layer (FC), and Recurrent Neural Network (RNN). We mark data that will be frequently reused as *evict_last*. For instance, in PageRank, we apply *evict_last* to high-degree vertices and their edges. Figure 9 illustrates the performance improvements achieved by using *evict_last* on different data sizes compared to the baseline where no priority hint is used. We only vary the size between 0 and 5 MB because the L2 cache size on the GPU used in our experiments is 5 MB (see Table 1); intuitively, pinning more than 5 MB would not yield additional benefits.

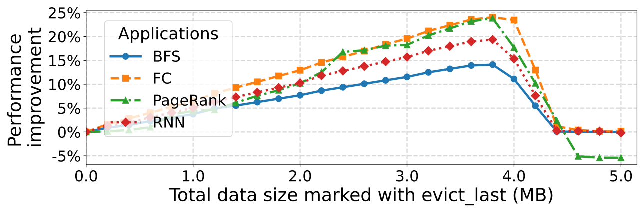

Figure 9: The performance improvement with different sizes of data marked as evict_last.

According to the figure, when the size of data marked as *evict_last* is relatively small, the performance gain increases as more data is marked. The optimal performance across all workloads occurs when approximately 3.75 MB of data is marked as *evict_last*. Beyond this threshold, further increasing the amount of *evict_last* data results in diminishing performance gains. This supports our conclusion that pinning more than 12 cache lines in each 16-way L2 set can adversely affect performance. Note that we are using an older GPU driver, where pinning approximately 12 out of 16 ways in each set yields optimal performance. With a newer driver, the optimal proportion might shift to 3 out of 16 ways.

7 Discussion

7.1 Data Prefetching

According to the NVIDIA document [38], eviction priority hints can also be used with the prefetch instruction; however, only the *evict_last* hint is supported for this instruction, not *evict_first*. We tested the properties of *evict_last* described in Section4.2 using the prefetch instruction and confirmed that these properties still hold.

7.2 Covert Channel Countermeasures

Noise Injection/Hint Remapping. An effective way to defend against Load+Load is to inject noise into the use of the *evict_first* priority hint, particularly when covert channel activity is detected. For instance, the driver can remap certain *evict_first* hints to *evict_normal* before launching a suspicious kernel. Additionally, the hardware can selectively reinterpret some *evict_first* hints as *evict_normal* during execution. These modifications introduce noise into the covert channel, potentially preventing its operation entirely. At the same time, they preserve the utility of *evict_first* for benign workloads, particularly those that do not exhibit access patterns similar to the covert channel.

Trade-off Between Security and Performance. The hardware implementation of the *evict_first* priority hint can be modified to better balance security and performance. Specifically, instead of allowing only one *evict_first* cache line per L2 set (when full), the hardware could permit multiple *evict_first* cache lines to coexist—similar to how *evict_last* is handled. When eviction is required, one of the *evict_first* lines would be selected for replacement. This approach reduces the bandwidth of Load+Load, as the cache line marked as *evict_first* is no longer guaranteed eviction. At the same time, it still allows programs to achieve performance gains by marking streaming data as *evict_first*, although the benefits may be reduced compared to the current implementation.

Cache Randomization. As with other conflict-based cache covert channels, Load+Load relies on the sender's and receiver's ability to construct eviction sets. Prior work on CPUs has proposed several designs to prevent eviction-set construction, such as cache mapping randomization [39, 44]; these designs can be adapted to GPUs to similarly mitigate Load+Load.

7.3 Side-Channel Attacks

The *evict_last* priority hint can be exploited to facilitate side-channel attacks. We discuss two specific types of such attacks below

Efficient application fingerprinting attack. The *evict_last* priority hint can enable a more efficient side channel than Prime+Probe. Specifically, an attacker can pin n cache lines in the target L2 set using the *evict_last* priority hint. Then, to prime or probe the set, the attacker only needs to access $16 - n$ cache lines in the set, rather than accessing all 16 cache lines, as in Prime+Probe. This approach results in a more efficient side-channel attack than Prime+Probe.

We implemented an application fingerprinting attack using this method on the RTX 3080 GPU listed in Table 1. Similar to prior work, the attacker monitors evictions in each L2 set and collects a memorygram—a time-series trace of cache activity—while the victim application runs. This trace is then used to identify the application. In each L2 set, the attacker pins 8 cache lines and uses the remaining 8 cache lines to perform a Prime+Probe-style monitoring of the victim's accesses.

To evaluate the attack, we selected six victim applications (following prior work [8]): 2D Convolution, SpMV, Walsh Transform, Bitonic Sort, Radix Sort, and Vector-Add. For each application, we collected a set of memorygram samples and trained a CNN-based classifier. We evaluated the classifier using 5-fold cross-validation. The results demonstrate that the attack achieves an accuracy comparable to Prime+Probe [8], with over 99.9% precision, recall, and F1 score across all classes. Notably, it requires only 8 accesses per

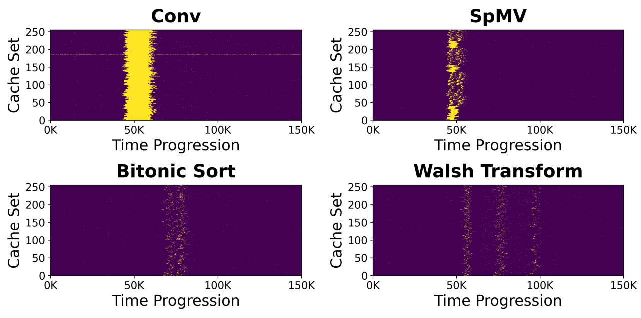

Figure 10: Example memorygrams for four applications.

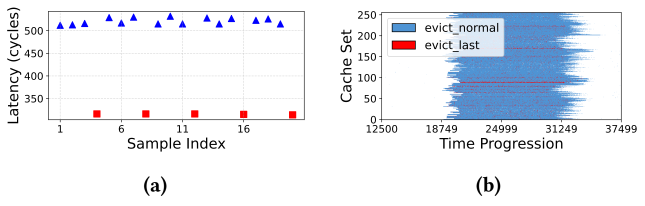

Figure 11: (a) Access latency to the LRU line in the set when the victim repeatedly issues three evict_last accesses followed by one evict_normal access; (b) detected accesses from a victim running matrix multiplication.

L2 set per iteration, compared to 16 in Prime+Probe. Example memorygrams are shown in Figure 10.

Cache eviction hints detection attack. Takeaways 7 and 8 show that when there are at most 12 (or 3) *evict_last* cache lines in an L2 set, loading an *evict_last* line evicts the set's overall LRU line, while loading an *evict_normal* line evicts only the LRU line within the *evict_normal* lines. Thus, by monitoring which line is evicted, the attacker can detect the victim's use of eviction priority hints: 1) an eviction of the overall LRU line indicates an *evict_last* access by the victim, and 2) an eviction of the LRU line within the *evict_normal* group indicates an *evict_normal* access by the victim. For example, Figure 11a shows the access latency for the LRU line of the set when the victim repeatedly issues three *evict_last* accesses followed by one *evict_normal* access. Figure 11b shows the detected *evict_last* and *evict_normal* accesses from a victim running matrix multiplication. The victim uses the *evict_last* hint to access one of the input matrices and keeps it in the L2 cache to enhance performance. Since the attacker can distinguish between *evict_last* and *evict_normal* accesses, it can obtain finer-grained information than traditional Prime+Probe, potentially revealing additional secrets like the matrix size.

7.4 Novelty and Impact

To our knowledge, this is the first work to reverse-engineer the implementation of GPU eviction priority hints and analyze their security and performance implications. From a security standpoint, we are the first to show that these hints can be exploited to construct efficient GPU covert channels and performance degradation attacks. From a performance perspective, we demonstrate that eviction hints–originally designed to help pin data in cache–can instead lead to cache thrashing in many cases. Without this finding, GPU users may place undue trust in eviction hints and experience unexpected performance degradation.

7.5 Related Work

Cache eviction priority hints have been used by several previous studies [15, 27, 57] to improve the performance of GPU applications. For example, [57] uses *evict_last* to pin parameters in batch normalization in the cache due to their temporal reuse in subsequent operations; [15] prefetches and marks embedding data as *evict_last* to reduce the access latency. However, none of these studies have looked into the implementation details of the eviction priority hints.

On the security side, CPU microarchitectural covert channels, especially cache covert channels, [6, 9, 10, 13, 24, 29, 40, 43, 50, 52] have been extensively studied over the years. These channels exploit different features of the CPU cache to secretly transmit information. For example, the LRU-channel [51], Reload+Refresh [3], and Prime+Scope [40] utilize the replacement policy. Prefetch+Prefetch and Prefetch+Reload [13] exploit the cache coherence protocol. Many covert channels utilize cache conflicts and cache reuse [3, 10, 26, 41, 53].

Over the past decade, significant efforts have been dedicated to uncovering the microarchitectural details of modern GPUs and explore the security issues caused by these microarchitecture designs [18, 19, 22, 23, 30, 35, 45, 49]. These studies provided insights into components like cache [7, 8, 11, 13, 16, 33], functional units [33], NVLink [54], on-chip network [1, 21] and translation lookaside buffer (TLB) [35, 55], and enabled the construction of many GPU covert channels. Additionally, prior work [34, 47] has studied the covert channels based on CUDA APIs or performance counters.

8 Conclusion

In this paper, we reverse engineered the cache behavior of the evict_first and evict_last priority hints on NVIDIA GPUs. We showed that evict_first can be used to build a conflict-based covert channel which achieves significantly higher bandwidth than prior GPU channels, while evict_last enables a more stealthy performance degradation attack. Beyond security implications, we also found that evict_last can improve performance by keeping frequently reused data in cache. However, marking too many cache lines as evict_last leads to cache thrashing and performance degradation in real-world workloads.

Acknowledgments

We sincerely thank the anonymous reviewers from MICRO 2025 and ISCA 2025 for their insightful feedback, which greatly contributed to improving the final version of this paper. This work is supported in part by the National Science Foundation (NSF) under Grant No. CNS-2350228. Any opinions, findings, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] 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. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 565–577.

[2] Ahmad Al Badawi, Bharadwaj Veeravalli, Jie Lin, Nan Xiao, Matsumura Kazuaki, and Aung Khin Mi Mi. 2020. Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters. *IEEE Transactions on Parallel and Distributed Systems* 32, 2 (2020), 379–391.

[3] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. 2020. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*. 1967–1984.

[4] Zhilu Chen, Jing Wang, Haibo He, and Xinming Huang. 2014. A fast deep learning system using GPU. In *2014 IEEE international symposium on circuits and systems (ISCAS)*. IEEE, 1552–1555.

[5] Hyeonseong Choi and Jaehwan Lee. 2021. Efficient use of GPU memory for large-scale deep learning model training. *Applied Sciences* 11, 21 (2021), 10377.

[6] Yujie Cui, Chun Yang, and Xu Cheng. 2022. Abusing cache line dirty states to leak information in commercial processors. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 82–97.

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

[8] 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.

[9] Dmitry Evtyushkin and Dmitry Ponomarev. 2016. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 843–857.

[10] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: a fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*. Springer, 279–299.

[11] Yanan Guo, Dingyuan Cao, Xin Xin, Youtao Zhang, and Jun Yang. 2023. Uncore encore: Covert channels exploiting uncore frequency scaling. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 843–855.

[12] Yanan Guo, Xin Xin, Youtao Zhang, and Jun Yang. 2022. Leaky way: a conflict-based cache covert channel bypassing set associativity. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 646–661.

[13] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. 2022. Adversarial prefetch: New cross-core cache side channel attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1458–1473.

[14] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. 2021. Domain-specific multi-level IR rewriting for GPU: The Open Earth compiler for GPU-accelerated climate simulation. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 4 (2021), 1–23.

[15] Rishabh Jain, Vivek M Bhasi, Adwait Jog, Anand Sivasubramaniam, Mahmut T Kandemir, and Chita R Das. 2024. Pushing the Performance Envelope of DNN-based Recommendation Systems Inference on GPUs. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1217–1232.

[16] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. 2019. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 29–41.

[17] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 947–960.

[18] 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).

[19] 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).

[20] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 463–479.

[21] Zhixian Jin, Jaeguk Ahn, Jiho Kim, Hans Kasan, Jina Song, Wonjun Song, and John Kim. 2024. Ghost Arbitration: Mitigating Interconnect Side-Channel Timing Attacks in GPU. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1138–1152.

[22] Zhixian Jin, Christopher Rocca, Jiho Kim, Hans Kasan, Minsoo Rhu, Ali Bakhoda, Tor M Aamodt, and John Kim. 2024. Uncovering Real GPU NoC Characteristics: Implications on Interconnect Architecture. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 885–898.

[23] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, and Wolfgang Lehner. 2017. Big data causing big (TLB) problems: Taming random memory accesses on the GPU. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*. 1–10.

[24] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. NetCAT: Practical cache attacks from the network. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 20–38.

[25] Tao Liao, Yongjie Zhang, Peter M Kekenes-Huskey, Yuhui Cheng, Anushka Michailova, Andrew D McCulloch, Michael Holst, and J Andrew McCammon. 2013. Multi-core CPU or GPU-accelerated multiscale modeling for biomolecular complexes. *Computational and Mathematical Biophysics* 1, 2013 (2013), 164–179.

[26] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*. IEEE, 605–622.

[27] Haoxuan Liu, Vasu Singh, Michał Filipiuk, and Siva Kumar Sastry Hari. 2024. ALBERTA: ALgorithm-Based Error Resilience in Transformer Architectures. *IEEE Open Journal of the Computer Society* (2024).

[28] Antonio Maciá-Lillo, Tamai Ramírez, Higinio Mora, Antonio Jimeno-Morenilla, and José-Luis Sánchez-Romero. 2023. GPU Cloud Architectures for Bioinformatic Applications. In *International Work-Conference on Bioinformatics and Biomedical Engineering*. Springer, 77–89.

[29] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud.. In *NDSS*, Vol. 17. 8–11.

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

[31] 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 *57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. To appear.

[32] John Michalakes and Manish Vachharajani. 2008. GPU acceleration of numerical weather prediction. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–7.

[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] Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. (Mis) managed: A novel TLB-based covert channel on GPUs. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 872–885.

[36] NVIDIA. 2017. NVIDIA Tesla V100 GPU Architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[37] NVIDIA. 2021. NVIDIA Ampere GA102 GPU Architecture. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf.

[38] NVIDIA. 2024. Parallel Thread Execution ISA. https://docs.nvidia.com/cuda/pdf/ptx_isa_8.5.pdf.

[39] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. 2021. Systematic analysis of randomization-based protected cache architectures. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 987–1002.

[40] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2021. Prime+ Scope: Overcoming the observer effect for high-precision cache contention attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2906–2920.

[41] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. 2021. Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1077–1090.

[42] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*. PMLR, 31094–31116.

[43] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. 2018. Microarchitectural Minefields: 4K-Aliasing Covert Channel and Multi-Tenant Detection in Iaas Clouds.. In *NDSS*.

[44] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. 2020. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization.. In *NDSS*.

[45] Vasily Volkov and James W Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE, 1–11.

[46] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. 2012. Accelerating fully homomorphic encryption using GPU. In *2012 IEEE conference on high performance extreme computing*. IEEE, 1–5.

[47] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. 2020. Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 125–137.

[48] Anuradha Welivita, Indika Perera, Dulani Meedeniya, Anuradha Wickramarachchi, and Vijini Mallawaarachchi. 2018. Managing complex workflows in bioinformatics: an interactive toolkit with gpu acceleration. *IEEE transactions on nanobioscience* 17, 3 (2018), 199–208.

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

[50] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *21st USENIX Security Symposium (USENIX Security 12)*.

[51] Wenjie Xiong and Jakub Szefer. 2020. Leaking information through cache LRU states. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 139–152.

[52] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. 2018. Are coherence protocol states vulnerable to information leakage?. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 168–179.

[53] Yuval Yarom and Katrina Falkner. 2014. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*. 719–732.

[54] 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:2503.17847 [cs.CR] https://arxiv.org/abs/2503.17847

[55] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. 2023. T unne L s for B ootlegging: 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.

[56] 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.

[57] Donglin Zhuang, Zhen Zheng, Haojun Xia, Xiafei Qiu, Junjie Bai, Wei Lin, and Shuaiwen Leon Song. 2024. {MonoNN}: Enabling a New Monolithic Optimization Space for Neural Network Inference Tasks on Modern {GPU-Centric} Architectures. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 989–1005.