GPUHammer: Rowhammer Attacks on GPU Memories are Practical

Chris S. Lin[†]
University of Toronto
shaopenglin@cs.toronto.edu

Joyce Qu[†]
University of Toronto
joyce.qu@mail.utoronto.ca

Gururaj Saileshwar
University of Toronto
gururaj@cs.toronto.edu

Abstract

Rowhammer is a read disturbance vulnerability in modern DRAM that causes bit-flips, compromising security and reliability. While extensively studied on Intel and AMD CPUs with DDR and LPDDR memories, its impact on GPUs using GDDR memories, critical for emerging machine learning applications, remains unexplored. Rowhammer attacks on GPUs face unique challenges: (1) proprietary mapping of physical memory to GDDR banks and rows, (2) high memory latency and faster refresh rates that hinder effective hammering, and (3) proprietary mitigations in GDDR memories, difficult to reverse-engineer without FPGA-based test platforms.

We introduce GPUHammer, the first Rowhammer attack on NVIDIA GPUs with GDDR6 DRAM. GPUHammer proposes novel techniques to reverse-engineer GDDR DRAM row mappings, and employs GPU-specific memory access optimizations to amplify hammering intensity and bypass mitigations. Thus, we demonstrate the first successful Rowhammer attack on a discrete GPU, injecting up to 8 bit-flips across 4 DRAM banks on an NVIDIA A6000 with GDDR6 memory. We also show how an attacker can use these to tamper with ML models, causing significant accuracy drops (up to 80%).

1 Introduction

Modern DRAM systems are increasingly vulnerable to Rowhammer attacks, a hardware vulnerability that enables attackers to induce bit-flips in memory cells by rapidly accessing neighboring rows [39]. Such attacks have been extensively explored on DDR and LPDDR memories in Intel and AMD CPUs [10, 13, 19, 30, 31, 37, 40], and even on integrated GPUs in ARM CPUs [18], where a CPU-based LPDDR memory was attacked using the integrated GPUs. Such attacks demonstrate that attackers can tamper sensitive data, and even escalate to kernel-level privileges on CPUs. In modern systems, discrete GPUs increasingly play a critical role in running high-performance computing, artificial intelligence applications, and graphics processing; however, thus far, there has been limited investigation of the Rowhammer vulnerability on GPUs. Moreover, machine learning models have been shown to be susceptible to bit-flips, allowing bit-flipping attackers to dramatically lower accuracy [26, 46, 48, 62, 82] and inject backdoors [7, 74] into these models. As GPUs are heavily used for ML model inference, understanding whether discrete GPUs are susceptible to Rowhammer is essential to ensure the security of ML systems. Thus, this paper seeks to explore the vulnerability of NVIDIA GPUs to Rowhammer.

Unlike CPUs, NVIDIA GPUs use Graphics-DDR (GDDR) memory in client and server GPUs, and High Bandwidth Memory (HBM) only in server GPUs. Moreover, with GPUs being designed as throughput-oriented processors, they have a distinct architecture, instruction set, and programming model, compared to CPUs. These differences make adopting existing Rowhammer attacks to GPUs challenging.

Challenges for GPU Rowhammer. Rowhammer attacks require hammering DRAM rows with rapid activations. To do so, an attacker is required to (a) evict addresses from on-chip caches, (b) access conflicting DRAM rows in a memory bank to ensure DRAM row activations, (c) activate the rows at a sufficiently high rate, and (d) develop access patterns that fool the in-DRAM Rowhammer mitigations. NVIDIA GPUs, since the Ampere generation (sm-80), support the `discard` instruction, which evicts any address from all on-chip GPU caches, satisfying (a). However, the other requirements (b-d) introduce several challenges (C1-C3) for a successful attack.

C1. First, the mappings of virtual to physical addresses, and of physical addresses to DRAM rows and banks in NVIDIA GPUs are proprietary and unknown. This is necessary to know to activate rows in the same bank for Rowhammer.

C2. Second, as GPUs are optimized for throughput, they have up to 4$\times$ [44] the memory latency compared to CPUs, limiting hammering intensity. Moreover, GPU memories have faster refresh periods (e.g., 32ms or less) compared to CPU memories (32ms - 64ms), limiting the time available for hammering.

[†]Equal contribution.

Combined, these factors result in insufficient hammering rates to trigger Rowhammer bit-flips on GPUs.

C3. Third, GPU memories may have proprietary in-DRAM defenses that need to be defeated by attacks to trigger bit-flips.

In this paper, we develop techniques to overcome these challenges and launch Rowhammer attacks on NVIDIA GPUs. We focus on NVIDIA A6000 GPU with GDDR6 memory, a popular workstation GPU also available in the cloud, but our analysis is equally applicable to other client or server GPUs.

GDDR Row Addressing. Rowhammer attacks on CPUs typically fully reverse-engineer the DRAM bank address function given a physical address, leveraging DRAMA [59]. However, unlike CPUs, on GPUs, the physical addresses are not exposed even to a privileged user, making reverse engineering the closed-form bank address function challenging. However, we discovered that GPU virtual-to-physical memory mapping is typically stable for large memory allocations. Consequently, we directly reverse engineer the mapping of virtual addresses (offsets within an array) to DRAM banks and rows within the GPU DRAM by utilizing the latency increase on DRAM row-buffer conflicts, similar to DRAMA [59]. Based on this, we identify addresses that map to unique rows, for all rows in a bank, and use these addresses to perform hammering.

Increasing Activation Rates. Since GPUs have almost a $4\times$ higher memory latency [44] compared to CPUs, the naive hammering of GDDR memory from a single thread, like in CPU-based attacks, cannot achieve sufficiently high activation intensity to flip bits. This is even more important for GPUs, as the refresh period (tREFW) for GDDR6 is just 22ms, as we reverse engineer in Section 6, 30% less than that of DDR5, limiting the time available for performing activations. We address this challenge by increasing the activation rates through parallelized hammering, leveraging the throughput-oriented programming model in GPUs. We develop k-warp m-thread-per-warp hammering kernels optimized for GPUs, achieving activation rates close to the theoretical maximum of 500,000 activations per tREFW, which is $5\times$ higher than naive single-thread hammering kernels adapted from CPU-based attacks.

Synchronization to Refreshes. As shown by prior attacks [13, 19, 30], defeating in-DRAM mitigations requires not only hammering N-aggressors [19, 30], but also bypassing the in-DRAM TRR mitigation consistently on each tREFI by synchronizing the attack pattern to REF commands [13]. Similar to prior work [13], we insert NOPs to each thread to add delays to the hammering kernel, synchronizing it with REF commands. However, we observe that synchronization weakens as the number of warps increases. To address this, we design GPU kernels that utilize multiple warps and multiple threads per warp, to effectively hammer more aggressors while maintaining synchronization with REF commands.

Demonstrating Rowhammer Exploits. Based on the techniques above, we develop our attack GPUHammer, with which we launch the first systematic Rowhammer campaign on NVIDIA GPUs, specifically on an A6000 with 48GB GDDR6 DRAM. We use n-sided synchronized hammering patterns ($n = 8, 12, 16, 20, 24$), similar to TRRespass [19] and SMASH [13], targeting 4 DRAM banks on the GDDR6 memory. On the A6000, we observe for the first time a total of 8 bit-flips (one per row) in GDDR6 DRAM, including at least one bit-flip in each bank we hammered.

Using these bit-flips, we demonstrate for the first time accuracy degradation attacks on ML models running on a GPU, that affect a wide range of deep learning models. We show that bit-flips can occur in the most-significant bit of the exponent in FP16-representation weights, significantly altering the value of the parameter and degrading the accuracy. Across five different models, including AlexNet, VGG16, ResNet50, DenseNet161, and InceptionV3, we observe accuracy drops of between 56% to 80% due to Rowhammer bit-flips. Our observations highlight the pressing need for both system-level mitigations against Rowhammer attacks on GPUs and algorithmic resilience in ML models against bit-flipping attacks, given the real-world threat posed by Rowhammer on GPUs.

Overall, this paper makes the following contributions:

1) We demonstrate the first Rowhammer attack on discrete GPUs, capable of flipping bits in GDDR6 memories.
2) We reverse-engineer unknown details about the address to bank and row mappings, to efficiently activate DRAM rows from the same bank in GPU memories.
3) We propose new techniques for parallelized hammering on GPUs that achieve high activation rates and precise synchronization with refresh operations to bypass in-DRAM Rowhammer defenses in GDDR6 memory.
4) Using GPUHammer, we demonstrate the *first* Rowhammer exploit on a discrete GPU (NVIDIA A6000 GPU), that targets ML models and degrades model accuracy by 56% to 80% with a single bit-flip.

Responsible Disclosure. We responsibly disclosed the Rowhammer vulnerability on A6000 to NVIDIA on 15th January, 2025, and subsequently also to the major cloud service providers (AWS, Azure, GCP) who may be affected. NVIDIA has confirmed the problem, and at the time of writing is still investigating a fix. After the expiry of an embargo requested by NVIDIA till 12th August, 2025, our code will be available on Github at `https://github.com/sith-lab/gpuhammer`.

2 Background and Motivation

2.1 GPU Execution Model and Architecture

GPUs are throughput-oriented processors designed for large-scale data processing through parallel execution. Their architecture, composed of numerous smaller cores, allows for the concurrent execution of thousands of threads, making them well-suited for operations like matrix multiplications, image processing, and machine learning.

Execution Model. NVIDIA GPUs are designed for parallel execution, where computations are divided into *threads*, the smallest units of execution. Threads are grouped into *warps*, each consisting of 32 threads that execute the same instruction in lockstep. Multiple warps are further organized into thread *blocks*, with each block assigned to a single Streaming Multiprocessor (SM) on the GPU. Different thread blocks are spatially mapped across the available SMs (e.g., the A6000 GPU has 80 SMs) and run concurrently. Within an SM, the warps of a thread block execute in a time-multiplexed manner. When a GPU kernel (a function to be executed on the GPU) is launched, the computational workload, represented as a grid of blocks $\times$ warps $\times$ threads, is distributed across the available SMs for execution.

Memory Hierarchy. Like CPUs, the GPU memory hierarchy includes L1 and L2 caches on-chip, and an off-chip DRAM, which could be a GDDR or HBM-based memory. Typically, GDDR memories are used in client (RTX3080) or server (e.g., A6000) GPUs, whereas HBM is used exclusively in server-class GPUs (e.g., A100 or H100). The L1 cache is private to each Streaming Multiprocessor (SM) and the L2 cache is shared across all SMs on the GPU. As GPU systems are optimized for memory bandwidth and have massive die sizes, their memory system latencies can be an order of magnitude higher than CPUs. A recent study [44] reports the L1 latency for an RTX 3090 close to 20ns, L2 latency close to 100ns, and memory latency close to 300ns, with the memory latency on GPUs being almost 4$\times$ the latency on CPUs [44].

2.2 GPU DRAM Architecture

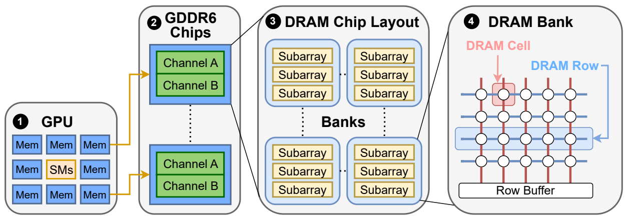


Figure 1: GDDR6 DRAM Organization

DRAM Architecture. Modern GPUs typically use GDDR or HBM-based memories. While the GDDR DRAM has a planar structure, the HBM has a 3D structure with DRAM dies layered one above the other. However, within the DRAM chip, the architecture is similar across form factors.

Figure 1 shows the organization of GDDR6 DRAM in A6000 GPU, where multiple DRAM chips surround the GPU core containing the SMs. The memory controllers (MCs) on the GPU core route memory requests from the SMs to the appropriate DRAM chip. Each MC connects to a DRAM chip via a memory bus, operating as an independent memory channel. Each DRAM chip is organized into 16 banks, and each bank consists of arrays of DRAM cells. Cells are grouped into DRAM rows that are 1KB to 2KB in size. On a memory access, an entire DRAM row is activated and read into a row buffer, before the requested portion is sent over the data bus.

The A6000 GDDR6 has a 384-bit memory bus with 12 DRAM chips, each providing 32 bits. A chip has two channels, which may be used as a single 32-bit channel or two 16-bit memory channels, resulting in 12 to 24 channels. Each channel has 16 banks, and each bank has 16K to 32K rows.

DRAM Operations. To access data in DRAM, the memory controller (MC) issues a sequence of commands. First, an ACTIVATE command (ACT) is sent to activate or open a row and read the charge into the row-buffer. Then, the MC can perform READ or WRITE operations by specifying the column within the row-buffer that is to be read or written to. Finally, before a new row can be activated in the same bank, the MC must send a PRECHARGE command (PRE) to close and deactivate the row, before accessing another row in the bank. Between two ACTs to a single bank, the MC is required to wait for at least tRC (typically about 45ns). To maintain data integrity, the memory controller periodically issues REFRESH commands (REF) on average at least once every 1.9 μs (tREFI) [34], which replenishes the charge in a subset of rows at a time. Typically, the entire DRAM is refreshed within 8K or 16K REF commands, ensuring that each row is refreshed at least once every 16ms or 32ms [34].

2.3 Rowhammer

Rowhammer is a read-disturbance phenomenon in which rapid activations of an aggressor row in DRAM can cause charge leakage in a physically adjacent victim row, ultimately inducing bit-flips [39]. The minimum number of ACTs to a row required to trigger bit-flips, known as T_{RH}, has been shown to be decreasing over time in CPU-based DRAM. Prior works [37, 39] using DRAM testing infrastructure show that the T_{RH} has dropped from 139K in DDR3 [39] to 10K in DDR4 and 4.8K in LPDDR4 [37]. Similar FPGA-based testing [57] has shown HBM2 to have a T_{RH} of around 20K. No prior work has studied the Rowhammer vulnerability on GPU memories like GDDR6 or HBM2e/3, due to the lack of any open-source testing infrastructure for such memories.

Mitigations in CPU-Based DRAM. In DDR4 and DDR5 memories, Target Row Refresh (TRR) [19, 24, 31] has been widely adopted as an in-DRAM Rowhammer mitigation. TRR operates by tracking frequently activated aggressor rows and proactively refreshing their neighboring victim rows during REF commands (e.g., once every *n* REF), to prevent bit-flips.

Recently, JEDEC, the consortium of DRAM vendors, has standardized more advanced mitigations such as RFM (Refresh Management) [32] and PRAC (Per Row Activation

Counter) [53], to address Rowhammer vulnerabilities. However, thus far, to our knowledge, no commercial systems have adopted such sophisticated mitigations.

Bypasses. Recent attacks on DDR4 and DDR5 have demonstrated that TRR is vulnerable to sophisticated attack patterns. TRRespass [19] showed that *n*-sided attack patterns could evade the TRR mitigation and induce Rowhammer bit-flips in DDR4 memories, by overflowing its tracking mechanism and preventing certain aggressor rows from being tracked and mitigated. Blacksmith [30] observed that TRR samples aggressor rows only at certain time instances within tREFI, and thus creates patterns where certain aggressor rows can evade the tracker. SMASH [13] further shows that aligning the hammering patterns with tREFI intervals, when mitigations are issued, can increase the chance of a Rowhammer bit-flip, as TRR continues to be fooled in a predictable manner. Zenhammer [31] shows that similar attacks continue to be applicable to some DDR5 DRAM.

2.4 Motivation and Goal

The growing demand for machine learning models has caused a surge in GPU deployment in the cloud, with NVIDIA GPUs commanding a market share of almost 90% [65]. While machine learning model inference is vulnerable to a wide variety of threats, such as adversarial examples [20], model inversion attacks [17], and jailbreaking attacks [6], more recently, they have also been shown to be vulnerable to tampering via bit-flipping attacks like Rowhammer, that can cause accuracy degradation [26,46,48,62,82] or backdoor injection into these models [7,74]. As ML inference is predominantly performed on GPUs, it is important to examine whether GPUs are susceptible to Rowhammer to fully evaluate the threat landscape for machine-learning systems. Thus, this paper studies the potential for Rowhammer attacks on NVIDIA GPUs.

2.5 Threat Model for GPU Rowhammer

Setting. We assume an attacker aiming to tamper with a victim's data in GPU memory via Rowhammer. This requires co-location of multi-tenant data within the same DRAM bank. Such a setting is possible in the case of time-sliced usage of GPUs in the cloud using NVIDIA GPU Operator [14]. Tools such as Google Kubernetes Engine (GKE) [21] can also enable memory isolation across time-slices allowing kernels of different users to time-share on a GPU. Multi-tenant data colocation is also feasible in case GPUs are spatially shared in a fine-grained manner, as in emerging serverless settings [11]. We assume ECC is disabled on GPUs.[1] We leave Rowhammer attacks on GPUs with ECC enabled for future work.

[1]Cloud service providers like AWS and GCP allow guest VMs to disable ECC on GPUs. Enabling ECC on GDDR-based GPUs can introduce slowdown [15,16], *c.f.*, Section 10, and reduce memory capacity [71].

Attacker Capabilities. We assume the attacker can execute CUDA kernels natively on the GPU with user-level privileges. The attacker can execute kernels that generate high-intensity memory access patterns for inducing Rowhammer. Additionally, GPU memories might deploy in-DRAM mitigations, like TRR [34,35,57] that act in the shadow of a refresh command. These kernels may attempt to surpass these mitigations, by activating multiple dummy rows [19] and synchronizing their attack patterns with respect to refresh commands [13].

The attacker can identify vulnerable rows in the GPU memory by targeting memory allocated by itself. Then, while colocated with the victim, the attacker may force sensitive victim data to be mapped to vulnerable locations through memory massaging. We provide more details in Section 8.

3 Building Blocks of GPUHammer Attack

In this section, we introduce the minimum primitives required to perform a Rowhammer attack on NVIDIA GPU systems, as well as the challenges that come with them.

3.1 Primitives for Activating DRAM Rows

Rowhammer attacks require rapid activations of a DRAM row. To obtain a DRAM row activation on successive loads to the same address, after each load, the address needs to be (a) evicted from the on-chip cache, and also (b) evicted from the row-buffer in the DRAM.

Cache Eviction. Since the `sm_80` architecture, i.e., Ampere generation GPUs, NVIDIA PTX introduced the `discard` [56] instruction to its ISA. `discard` clears an address from the L2 cache, making it an effective primitive for hammering. However, we still need to ensure the address is not cached in the L1 cache, to ensure a discard followed by a load is always fetched from memory. PTX provides multiple modifiers for load instructions to modify their caching behavior. So we measure the latency for performing a discard, followed by a load with these modifiers, to test which one enables consistent memory accesses. Figure 2 shows the result.

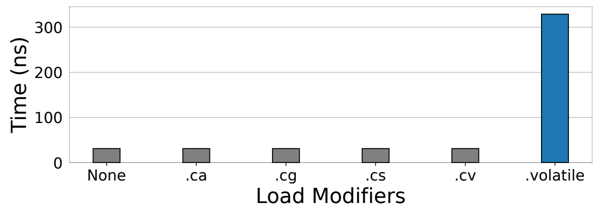


Figure 2: Latency of `discard` and load with different modifiers. On Ampere GPUs, only `.volatile` brings data into L2 and not into L1 cache, thus incurring memory access latency.

As shown in Figure 2, of all the `ld` modifiers in PTX, including cache hints such as `.ca` (cache at all levels), `.cg` (cache in L2, not L1), `.cs` (evict-first policy), `.cv` (don't cache), and `.volatile` (no reordering of loads), on the A6000 GPU, all the cache hints are ignored, and only the `.volatile` modifier ensures the load does not install the address into the L1 cache. Only the combination of `discard` and `ld.global.volatile` results in a latency close to memory access (around 300ns [44]), and other ld modifiers result in cache access latency (around 40ns [44]). Thus, memory accesses for an address are ensured with the following code:

```
discard.global.L2 [%0], 128;
ld.global.volatile %0, [%1];
```

Listing 1: Memory access using `discard` and `ld.volatile`

Row Buffer Eviction. To ensure row activations, after each memory access, the accessed row must be evicted from the row buffer. This can happen in two ways: (1) when a row is closed due to a period of inactivity, or (2) when another row in the same bank is accessed, causing a row buffer conflict that closes the current row [59]. As the first method can be slow, we rely on the second method to achieve activations. If one accesses addresses mapped to different rows in the same bank (`row1` and `row2`), a row-buffer conflict can evict `row1` from the row-buffer as follows:

```
ld.global.volatile %0, [%row1];
ld.global.volatile %0, [%row2]; // row1 evicted
```

Listing 2: Row-buffer eviction via row-buffer conflict.

3.2 Naive Rowhammer CUDA Kernel

Using the cache and row-buffer eviction primitives described above, we can construct a CUDA kernel to perform a n-sided Rowhammer attack similar to TRRespass [19]. We perform a `discard` and a `ld` to n successive addresses that map to different rows in a bank, in a loop of it iterations, to ensure it/n activations to each of the n rows. Our CUDA kernel to perform n-sided hammers on the GPU memory is as follows:

```
__global__ void hammer(size_t **addrs, size_t N,
     size_t it)
{
  size_t dummy, i = 0;
  for (; it--;)
  {
    asm ("discard.global.L2 [%0], 128;"
    ::"l"(addrs[i]));

    asm ("ld.u64.global.volatile %0, [%1];"
    : "=l"(dummy) : "l"(addrs[i]));

    i = (i + 1) % N;
  }
}
```

Listing 3: Naive single-threaded Rowhammer kernel

Thus, GPUs provide all the major building blocks required for a Rowhammer attack on GPU memories.

3.3 Challenges

To launch a successful Rowhammer attack, there are a few additional challenges introduced by the architectural constraints of GPUs, that we need to overcome.

Challenge-1: Reversing GPU Address to Bank Mappings. To engineer row buffer conflicts, we first need to reverse engineer the mapping of addresses to DRAM banks and rows. Unlike CPUs, whose functions have been reverse-engineered in prior works [31,59,78], no prior work has reverse-engineered the proprietary mapping functions used in GPUs. Moreover, unlike CPUs, where the physical addresses are exposed to privileged software, NVIDIA GPUs do not expose the virtual to physical mappings even to privileged software, further complicating this task. In Section 4, we show how to recover the bank and row address mappings, overcoming this challenge.

Challenge-2: Achieving High Activation Rates on GPUs. As DRAM cells are refreshed periodically, the Rowhammer attack is a time-bound attack. For a successful attack, the number of activations must cross the T_{RH} for a given row, before the refresh period (e.g., 32 ms) completes. However, achieving such rates on GPUs is challenging for two reasons.

First, GPU memory latencies (around 300ns) are around $4\times$ to $5\times$ higher than CPUs (around 60ns) [44] – due to this, our naive CUDA hammering kernel in Section 3.2 is only able to achieve 100K activations in 32ms, and less than 7K activations per row in a 16-sided pattern, which is lower than T_{RH} for most CPU-based DRAM. Second, the refresh intervals in GPU memory (e.g., 32ms or lower for GDDR6) are shorter compared to CPU DRAM (32–64 ms), further limiting the time available for hammering. Together, these factors contribute to hammering rates on GPUs that are insufficient to trigger Rowhammer-induced bit-flips. In Section 5, we show how we can overcome such limitations and hammer with significantly high intensity to induce bit-flips.

Challenge-3: Synchronizing Hammering to REFs. Recent CPU-based attacks have demonstrated that Rowhammer aggressor patterns should not only be n-sided [19], but also be synchronized to memory REF commands [13] to fool the in-DRAM mitigations in an effective manner.

Like CPU-based DDR memories, GPU memories like GDDR6 are also likely to be equipped with proprietary in-DRAM Rowhammer mitigations, that may be fooled by synchronized hammering patterns. However, little is known about the REF command frequency on NVIDIA GPUs, or the capability of memory accesses in CUDA kernels to be synchronized to REF commands of GPU memory controllers. In Section 6, we reverse-engineer details about GPU memory refresh, and show how one can achieve synchronized many-sided aggressor patterns on GPUs.

In subsequent sections (Sections 4 to 6), we present our attack primitives that address these challenges.

4 Reversing GPU DRAM Row Addressing

Understanding the mapping of memory addresses to DRAM banks and rows in GPU memory is crucial to generating row-buffer conflicts and activations. For this, we adopt the approach from DRAMA [59], tailoring it to NVIDIA GPUs. However, one key challenge is that unlike CPUs, where the virtual to physical memory translations are accessible to privileged software, allowing the reverse engineering of the closed-form function mapping the physical addresses to bank and row IDs, NVIDIA GPUs make the physical addresses inaccessible making the reversing of the exact functions challenging.

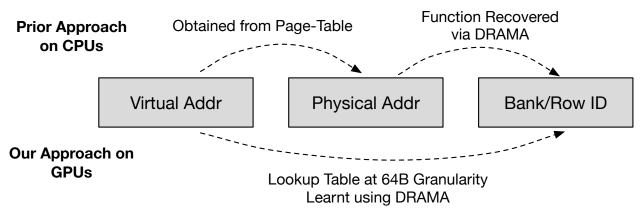


Figure 3: Prior approaches on CPUs reverse engineer the exact function of physical addresses to DRAM bank/row IDs. Without access to physical addresses on GPUs, we directly learn the mapping of virtual addresses to row and bank IDs.

To overcome this challenge, our approach instead directly learns a lookup table of virtual addresses to bank and row IDs, as shown in Figure 3, by measuring the increase in latency due to row-buffer conflicts similar to DRAMA [59]. This enables us to identify virtual addresses mapping to distinct rows in the same bank without recovering the exact row-addressing function itself. Below, we outline our approach in more detail.

4.1 Approach

To construct a lookup table mapping addresses at a 64B granularity to DRAM rows and banks, we use the following steps.

① **Allocate a large array.** We allocate an array just smaller than the GPU DRAM, using `cudaMalloc`. The virtual addresses of this array let us access the entire GPU DRAM.

② **Check for conflicts with a reference address.** We fix a reference address from the array, and identify all the addresses that have row-buffer conflicts with it. We do this by accessing pairs of memory addresses and measuring their latency, as shown in Figure 4. Here the first address is the fixed reference address, and the second is varied by iterating through the memory in 64B increments. When both addresses map to the same bank, the row-buffer-conflict causes a higher latency, compared to when both addresses map to different banks.

③ **Generate a Conflict-Set.** We collect all addresses in memory that have a high latency when accessed with the reference address, i.e., have a row-buffer conflict, and map to the same bank as the reference address. We call this set a *Conflict-Set*.

④ **Generate a Row-Set.** From the conflict-set, we filter out duplicates, i.e., addresses that map to the same row. We do this by iterating through the conflict-set, and only retaining an address if it conflicts with all the addresses already present in the set. The set of remaining addresses, consists of one address per row, which we call a *Row-Set*. We map each address in the Row-Set linearly to successive RowIDs. For each RowID, we also maintain a vector of addresses (*RowSet-Vec*) that do not conflict with the row-identifying address, *i.e.*, thus obtaining all the addresses that map to the same row.

The *Row-Set* obtained using a reference address from one bank, provides a lookup table mapping an address to a RowID for that bank. By changing the reference address, and repeating this process, we can get the Row-Set for successive banks.

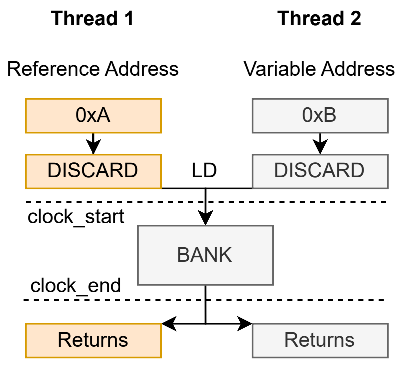
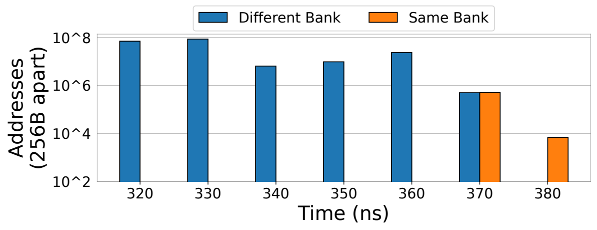


Figure 4: Conflict testing primitive. We measure the time to access a pair of addresses: the first address is a reference address, and the second is varied by iterating through the entire memory. As instructions in both threads execute in lockstep, the `clock_end` records the time when the longer of the two memory access completes.

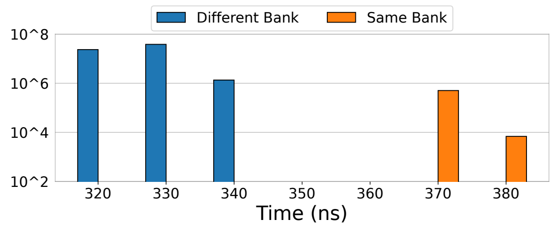
Results of Timing Measurements. We evaluated our conflict testing primitive on an NVIDIA RTX A6000 GPU equipped with 48GB of GDDR6 DRAM. We allocated a 47GB array and selected the starting address of the array as the *reference* address. Figure 5a shows the results of our timing measurements. Although pairs of addresses mapping to the same-bank with row-buffer conflicts have higher latency (360 ns - 380 ns), we observe that pairs without row-buffer conflicts[2] have a wide range of latencies (from 320 ns to 370 ns), and some overlap with conflicting pairs. This makes separating the same-bank address pairs from different-bank pairs using a cutoff threshold difficult.

We discover that this is because of non-uniform memory access latency in GPUs, similar to Non-Uniform Memory Access (NUMA) effects in servers. In server-grade GPUs with large memory, DRAM is often deployed in a distributed man-

[2]We identified the absence of row-buffer conflicts and row-activations in these pairs by adding an additional address from this set, and not observing an increase in latency corresponding to tRC, which is expected on an activation.



(a) Before Accounting for NUMA effects



(b) After Eliminating NUMA effects

Figure 5: Memory access latency measurements for address-pairs, categorized as same-bank and different-bank. (a) Naively considering pairs of reference addresses with all addresses in the memory, causes variation in the different-bank latencies (320ns - 370ns) due to NUMA effects, that overlap with the same-bank latencies (370-380ns). (b) Only considering addresses that have similar single memory access latency as the reference address, allows us to eliminate NUMA effects.

ner around the GPU SoC, introducing variation in memory access latency. As our conflict-testing primitive measuring latency of address-pairs measures the greater of the two latencies, it is highly susceptible to such NUMA effects. We discuss and address this next.

4.2 Eliminating Noise due to NUMA Effects

We characterize the NUMA effects in our A6000 GPU, by performing single uncached memory accesses to different addresses in the GPU DRAM and measuring their latencies. Figure 6 shows the histogram of measured memory access latency. There is a significant variation in the latency from 280ns to 370ns, with the higher end of this range overlapping with the latency of row-buffer conflicts on our original bank (360-380ns). However, we observe that our reference address has a latency of 320ns, and therefore all the addresses belonging to the same bank as the reference address, must also have a similar memory access latency in Figure 6.

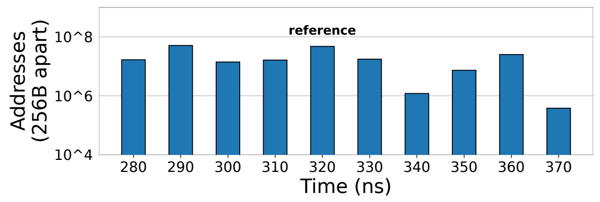


Figure 6: NUMA effects in A6000 GPU. Memory access latency for single memory access for each address in allocated memory (47GB). There is significant variation in the latencies, due to NUMA effects. Our *reference* address has a memory access latency of 320ns.

Thus, we eliminate the noise due to NUMA effects in our conflict testing process, by only considering the addresses that have a similar latency as our *reference* address (within ± 10 ns of the *reference*'s latency). This leverages a key observation that addresses belonging to the same bank and therefore the same DRAM chip should exhibit similar access delays. Figure 5b shows the results of our conflict-testing process after we only consider the addresses that have a similar latency as the reference address. Now, we see a clear separation between the row-buffer-hit latency and the row-buffer-conflict latency. We set the threshold as 350ns, i.e., 30ns higher than the reference address latency (320ns), and automatically classify any address pairs with higher latency as row-buffer conflicts and mapping to the same bank.

4.3 Evaluation of Row Addressing Recovery

After allocating 47GB out of 48GB GDDR6 DRAM on our A6000, we derive the conflict set and row set for 4 different banks using different reference addresses. Analysis of the *row set* revealed the following observations:

Observation 1. In GPU memories, contiguous chunks of 256 bytes are mapped to a single row and bank, and successive 256-byte chunks map to different banks.

Leveraging the observation that row and bank mappings were performed at 256-byte granularity, we significantly reduced the execution time of our row set recovery, by testing at 256B granularity instead of 64B; this reduced the runtime for our row set recovery to 4 hours per bank.

Observation 2. The row-sets show that an A6000 GDDR6 has a row size of 2KB and 64K rows per bank.

We repeat a similar row-set recovery for an RTX3080 with GDDR6 memory and an A100 with HBM2e memory and find that both have 32K rows per bank and a row size of 2KB and 1KB, respectively.

Observation 3. The gap between two successive 256B chunks mapping to the same row varies from row-to-row; as does the gap between addresses of two rows.

We hypothesize that the mapping of addresses to DRAM banks uses a complex XOR function with inputs from many bits of the address, although this is hard to definitively conclude without access to physical addresses. We provide a visualization of this behavior in Appendix B.

Observation 4. For large memory allocations, such as 47GB out of 48GB, the Virtual-to-Physical memory mapping is consistent, allowing the row-sets to be reused.

Repeated executions of the row-set recovery process, across system or program restarts, resulted in different virtual addresses, as ASLR is enabled, but the mapping of the offsets within the array to the rows remained the same. This implies that the physical memory allocated remains the same, and that we can simply reuse the row-sets for subsequent hammering.

By accessing the addresses from a single row set in a loop, we are able to activate rows and launch Rowhammer attacks.

5 Hammering with High Activation Rates

Successful Rowhammer attacks require a high activation rate that crosses the T_{RH} for a given row within a refresh period. The high memory latency (around 300ns as shown in Figure 6) and the reduced refresh window (32ms or lower) [34] in GPUs make it difficult to achieve sufficiently high activation rates.

5.1 Problem with Single-Thread Hammering

A naive CUDA kernel for hammering GPU memory, inspired by CPU-based hammering, uses a single thread to activate aggressor rows in a loop, as shown in Listing 3. However, with GPU memory latency of around 300 ns, such a kernel can only achieve a maximum of around 106,000 activations within 32ms. For n-sided attacks which require accessing multiple aggressor rows simultaneously to bypass in-DRAM mitigations like TRR [19], this rate is insufficient. For example, a 20-sided attack activating 20 rows at a time results in only 5K activations per row, whereas the T_{RH} for modern memories is at least 10K to 20K [37, 57].

Notably, the tRC, *i.e.*, time between two activations for the GPU memories GDDR6 [34] is only around 45ns, implying that it is theoretically possible to achieve over 700,000 activations in 32ms, *i.e.*, 7× the activation rates of our naive single-thread hammering. This discrepancy indicates significant idle time at the memory controller, caused by the high round-trip latency from SMs to the memory controller due to the GPU's large chip size, as shown in Figure 7(a). We seek to reduce this idle time at the memory controller by parallelizing the hammering process.

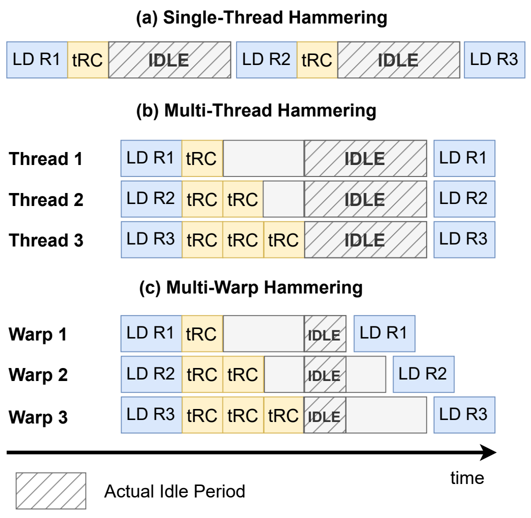


Figure 7: Memory controller utilization with (a) single-thread, (b) multi-thread, and (c) multi-warp hammering. Multi-warp hammering minimizes idle time, maximizing activation rates.

5.2 Parallel Hammering for More Activations

To reduce the idle time at the memory controller and increase DRAM activation rates, we explore two techniques for parallelized hammering: multi-thread and multi-warp hammering.

Multi-Thread Hammering. In this approach, each aggressor is assigned a dedicated thread in a single warp. As shown in Figure 7(b), the `load` instructions to each hammered row are executed in parallel in lockstep. Thus, the ACT to each row is issued back-to-back every tRC, reducing the idle time at the memory controller and increasing activation rates. However, this still incurs some idle time due to the lock-step execution of the loads, as the SM must wait for all loads in the current iteration to return before issuing the next round of loads.

Multi-Warp Hammering. To address the above limitation, we adopt multi-warp hammering, where each aggressor is handled by a thread in a different warp. When one warp is stalled due to memory access, the SM schedules the next warp in a round-robin manner with minimal overhead. Unlike threads within a warp, threads in different warps do not execute in lock-step; thus, the SM can issue a memory request as soon as a request from a previous iteration returns, without waiting for all the memory requests of the current iteration to complete. This further reduces idle time at the memory controller and increases the activation rate during hammering.

We implement n-sided multi-warp hammering, *i.e.*, one aggressor per warp hammering with n aggressors, using a CUDA kernel with $32 \times n$ threads (as each warp has 32 threads). Each aggressor is assigned to a thread $0, 32, 64, \ldots$, and so on.

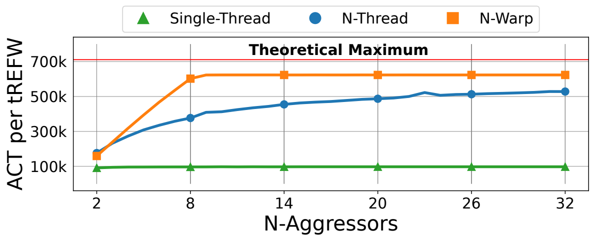


Figure 8: Activation rates (ACTs per tREFW) with three different hammering techniques. Multi-warp hammering achieves the best activation rate of 620K per tREFW (32ms), reaching close to the theoretical maximum (around 700K).

5.3 Evaluation of Parallelized Hammering

We evaluate the different hammering techniques, including single-thread, multi-thread, and multi-warp hammering, on an NVIDIA A6000 GPU with 48GB GDDR6 DRAM, and compare the number of activations in a refresh period (ACTs per tREFW) as the number of aggressor rows increases. We assume a refresh period (tREFW) of 32ms and a time per ACT (tRC) of 45ns. Thus, the theoretical maximum ACTs per tREFW is around 700K.

As shown in Figure 8, the single-thread hammering only reaches an ACT per tREFW of approximately 90K, or roughly 12% of the theoretical maximum. The multi-thread hammering (N-Thread) significantly increases the activation rate, which grows as the number of aggressor rows increases. However, even with 32 aggressors, it reaches a maximum of less than 520K ACTs per tREFW. In contrast, multi-warp hammering (N-Warp) reaches around 620K ACTs per tREFW with just 8 aggressors, reaching close the theoretical maximum. This is due to the reduced idle time at the memory controller compared to the other approaches.

Multi-warp hammering, with as few as 8 warps, achieves an activation rate almost 7× higher than single-thread hammering, and sustains close to 20K activations per tREFW per row even with 32 aggressors. Therefore, we adopt multi-warp hammering with 8 warps by default to ensure the maximum ACT intensity for our attack.

6 Synchronization with Refresh Commands

Synchronizing hammering patterns with DRAM refresh commands (REFs) is crucial for bypassing in-DRAM mitigations like TRR [13, 31]. TRR mitigates Rowhammer by tracking frequently accessed rows and issuing victim refreshes coupled with regular refreshes. Without synchronization, the TRR sampler state is unpredictable, giving each victim row a chance to be refreshed. As a result, our attack can be mitigated even if we overflow the sampler's capacity with multi-warp hammering in Section 5.2. By contrast, a synchronized pattern forces the TRR sampler into a well-defined state before regular refresh. This technique ensures that specific aggressor rows repeatedly escape mitigation in the presence of TRR.

To synchronize with REF commands, there are two key requirements: (1) A bubble must be inserted at the memory controller at the end of a hammering iteration, sparing time for the REF command, and (2) synchronization must not alter the order of aggressors. Prior works [13, 31] achieved synchronization in single-thread setting by strategically inserting NOPs after a hammering iteration, timed to align with tREFI intervals. However, in multi-warp hammering on GPUs, we observe that simply using `__syncthreads` for thread-synchronization at the end of a hammering iteration introduces uncertainty to warp ordering, complicating the synchronization to REFs.

Per-Warp Synchronization Design. In our design, we avoid using `__syncthreads` and instead add delays within each warp after i hammering iterations using the `add` instruction. When delays overlap across all warps, a bubble is implicitly created at the memory controller, allowing a REF to be inserted in alignment with the hammering pattern. Figure 9 illustrates how this overlap achieves synchronization without disrupting warp order. We insert `add` instructions every $i = 1, 2,$ or 3 hammering iterations, depending on the n-sided pattern, to ensure that a whole number of iterations are completed within a tREFI while limiting activations to no more than 24 per tREFI. Thus, each *round* of hammering consists of $n \times i$ activations followed by some `add` delays. We provide a sample implementation in Appendix A.

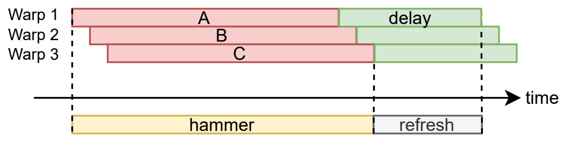
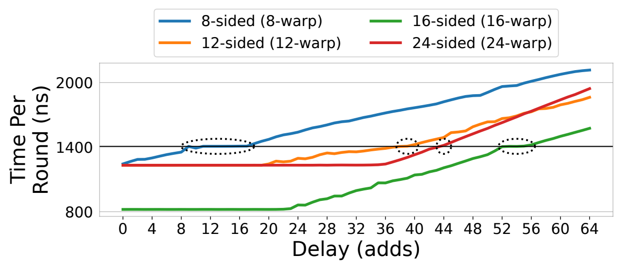


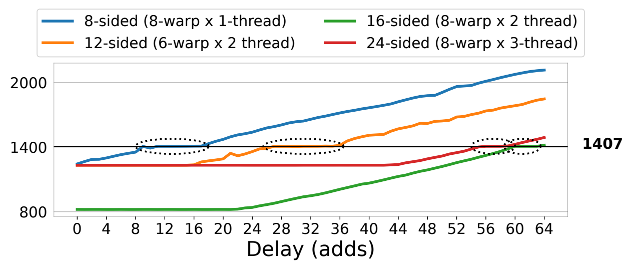
Figure 9: Per-Warp delays inserted by `add` for synchronizing to REF. When the delays overlap for all the warps, the REF is inserted in alignment with the hammering pattern.

Evaluation of Synchronization. Our synchronization approach relies on the overlap of delays across warps, which is heavily influenced by the warp configuration. Figure 10(a) shows synchronization results for multi-warp hammering patterns (1 thread per warp) using 8-,12-,16-, and 24-sided aggressor patterns. We observe that the 8-warp configuration achieves strong synchronization with tREFI, as shown by a flat-lined time-per-round at approximately 1407ns. However, the 12- to 24-warp configurations have weak or no synchronization, likely because the overlap of delays diminishes, and eventually vanishes, as the number of warps increases.

To ensure synchronization for patterns with a larger number of aggressors (n), we adopt configurations with k warps and m threads per warp, such that $n = k \times m$. We further restrict the number of warps k to be 8 or lower, in order to



(a) Multi-Warp Hammering (1 thread per warp).



(b) Multi-Warp & Thread Hammering (*k*-warps, *m*-threads-per-warp)

Figure 10: Evaluation of tREFI Synchronization plotting Time per Round (ns), where a round consists of 1, 2, or 3 iterations of n-sided hammering with up to 24 activations. (a) With delays added to naive multi-warp hammering, synchronization works well for 8-sided (8-warp), but is weaker for 12-sided or 24-sided (12/24 warps), likely due to delays not overlapping sufficiently. (b) By using 8 warps or less and multiple threads per warp, we ensure synchronization for all *n*-sided patterns.

maintain a sufficient overlap of delays for synchronization. Figure 10(b) shows the synchronization for *n*-sided patterns, using *k*-warp *m*-thread-per-warp configurations. We achieve sufficient synchronization for all tested *n*-sided patterns. We list the code for our final kernel with synchronized, many-sided hammering with *k* warps and *m* threads per warp, in Listing 4 in Appendix A.

Additionally, based on the synchronization instances, we observe that the GDDR6 memory in the A6000 GPU has a tREFI of 1407ns and 16K tREFIs per refresh period. Thus, our *n*-sided hammering can hammer up to 24 aggressor activations per tREFI and a total of 393K activations per refresh period.

Observation 5. The refresh interval (tREFI) for A6000 GDDR6 is 1407ns, leading to a refresh period of 23ms.

7 GPUHammer Evaluation

Using the row-sets from Section 4, the high-intensity hammering kernels from Section 5, and the synchronized many-sided hammering from Section 6, we launch GPUHammer attacks. To evaluate our attack, we answer the following questions:

1. In systematic hammering campaigns, which NVIDIA GPUs are vulnerable to bit flips? (*c.f.*, Section 7.1)
2. What aggressors (*c.f.*, Section 7.2) and data patterns (*c.f.*, Section 7.6) are most effective in flipping bits?
3. What is the directionality of the bit-flips (*c.f.*, Section 7.5), the Rowhammer threshold (*c.f.*, Section 7.3), and the in-DRAM TRR sampler size (*c.f.*, Section 7.4)?
4. Are the bit-flips realistically exploitable? (*c.f.*, Section 8)
5. What are the possible mitigations for GPUHammer and their associated costs? (*c.f.*, Section 10)

7.1 Systematic Hammering Campaigns

We conduct systematic hammering campaigns across four DRAM banks on three GPUs with ECC disabled: an RTX A6000 GPU with GDDR6 memory (in a workstation), an A100 with HBM2e memory (in the cloud), and an RTX 3080 with GDDR6 (in a laptop). We used *n*-sided patterns, starting with distance-2 between aggressors (hammering rows $R_i, R_{i+2}, \ldots, R_{i+2n}$) where we did not observe bit-flips, and then with distance-4 between aggressors (hammering $R_i, R_{i+4}, \ldots, R_{i+4n}$) when we started to see bit-flips.

Table 1 shows the results for 8-, 12-, 16-, 20-, and 24-sided hammering with distance-4 aggressors, using victim data of `0x55...` or `0xAA...` (aggressor data was inverted). For each *n*-sided pattern, we iterated through the entire DRAM bank, incrementing 3 rows at a time. We hammered each pattern for 128ms (∼5 refresh periods), with a total run time of 30 hours per bank per GPU. The A6000 with GDDR6 memory exhibited a total of 8 unique Rowhammer bit-flips (one per row), with at least one bit-flip in each hammered bank and all of them were single bit-flips. The A100 (HBM2e) and the RTX3080 (GDDR6) GPUs did not exhibit any bit-flips – we discuss potential reasons in Section 9.

Takeaway 1. A6000 GDDR6 DRAM is vulnerable to Rowhammer, with bit-flips observed on every bank hammered; No flips were observed on the A100 or RTX3080.

Table 1: Number of bit-flips observed across 4 banks on different GPUs. A6000 with GDDR6 had bit-flips on every bank.

GPU	Bank	*n*-Sided Patterns				
		8	12	16	20	24
RTX A6000	A	0	0	0	0	**1**
	B	0	0	0	**1**	**3**
	C	0	0	0	0	**1**
	D	0	0	0	0	**3**
A100	E / F / G / H	0	0	0	0	0
RTX 3080	I / J / K / L	0	0	0	0	0

7.2 Critical Aggressors Characterization

For the A6000 GPU where we observed bit-flips, we characterize the critical aggressor rows whose presence is sufficient to trigger a bit-flip in the victim. For this, we hammered the 8 vulnerable locations we identified in the A6000 (banks A, B, C, and D), starting with distance-4 sequential aggressor patterns around the victim, and replacing rows from the pattern with random rows, to identify the critical aggressor rows whose presence is sufficient to trigger a bit-flip. Table 2 shows the critical aggressor rows for each victim.

Table 2: Critical Aggressor Rows. Assuming the vulnerable victim row is R_i, a tick under R_j indicates that hammering R_j in the aggressor pattern is sufficient to trigger bit-flip at R_i.

Bit-flip	Critical Aggressor Row					
No.	R_{i-3}	R_{i-2}	R_{i-1}	R_{i+1}	R_{i+2}	R_{i+3}
A_1		✓				
B_1		✓	✓			
B_2					✓	✓
B_3					✓	
C_1		✓	✓			
D_1		✓	✓			
D_2					✓	
D_3				✓	✓	

Our first observation is that all our bit-flips are due to *single-sided* hammering: our critical aggressor rows are either $+1$, $+2$, $+3$ neighbors of the victim row R_i, or -1, -2 neighbors of R_i, but never on both sides. Second, some of our bits (B_2) flip with aggressors at R_{i+2} and R_{i+3}, but not with R_{i+1}. This indicates that physical rows may not be linearly laid out within the DRAM, and there could be some remapping internally.

Finally, we observe that hammering the R_{i+2} or R_{i-2} rows produces the most reliable bit-flips, whereas $R_{i\pm1}$ or R_{i+3} produces bit-flips less reliably. Thus, we hypothesize that $R_{i\pm2}$ may be neighbors of R_i, and $R_{i\pm1}$ or R_{i+3} may be in the neighborhood, but not directly adjacent to R_i.

Takeaway 2. On the A6000, bit-flips are triggered by single-sided hammering on critical aggressor rows from only one side of the victim, with $R_{i\pm2}$ being the most effective, suggesting non-contiguity in DRAM row layout.

7.3 Rowhammer Threshold Characterization

The Rowhammer threshold (T_{RH}) is the minimum number of activations to an aggressor row required to induce bit-flips in adjacent rows. We characterize the T_{RH} for the bit-flips on the A6000 using 24-sided aggressor patterns. We do so by hammering the aggressor pattern for x tREFIs out of every $x+y$ tREFIs, issuing dummy accesses for the remaining y tREFIs. We gradually lower the hammering intensity until it stops triggering bit flips. Thus, each aggressor receives $x/(x+y)$ times the activations it originally received in a refresh window (tREFW), *i.e.*, $16\text{K} \times x/(x+y)$. We vary x and y from 0 and 100, to find the lowest value of $x/(x+y)$ for which the bit-flip is triggered, to estimate the T_{RH} for a bit flip.

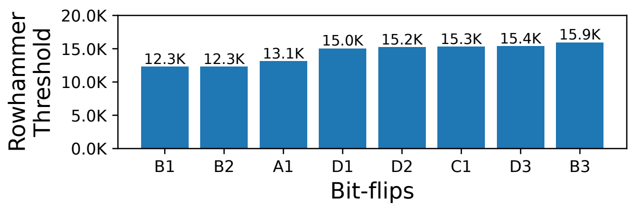


Figure 11: Rowhammer threshold (T_{RH}) for each of the 8 observed bit-flips on the A6000. The minimum T_{RH} observed for the A6000 GPU across all the bit flips is 12.3K activations.

Figure 11 shows the T_{RH} for the different bit flips. Across all bit flips, the minimum T_{RH} is around 12.3K, close to the previously reported values for DDR4 [37]. We observe that 3 bit-flips (in banks A and B) have T_{RH} of 12–13K, while 4 others (in banks C and D) have it around 15–16K. This suggests that T_{RH} may have some correlation with the bank. One outlier is B3, whose T_{RH} of 15.9K is higher than that of B1 and B2 (12.3K). However, B3's bit-flip direction is opposite to that of B1 and B2, as shown in Section 7.5. Therefore, it might be in a different sub-array within the bank.

Takeaway 3. The A6000 GPU has an observed T_{RH} of 12.3K activations. An attacker requires at least 12.3K activations to a single DRAM row to observe a bit-flip.

7.4 TRR Sampler Size Characterization

Prior works show that DRAM implements Target Row Refresh (TRR), an in-DRAM Rowhammer mitigation that tracks frequently activated rows by a fixed-size sampler and accordingly issues mitigative refreshes [19, 24, 30]. Following prior works like TRRespass and SMASH [13, 19], we design our aggressor patterns to test for the memory's Rowhammer defenses. We observe, in GDDR6, bit-flips only occur when using patterns with a large number of distinct aggressor rows issued in synchronization with REF commands, suggesting the presence of a TRR-like mitigation that can only track a limited number of aggressors.

We studied 8- to 24-sided attack patterns on a victim row with a known bit-flip, A_1. In each attack pattern, we included a critical aggressor row and filled the remaining positions with random dummy rows from the same bank. For each n-sided pattern, we also varied the position of the critical aggressor row within the pattern randomly and hammered each configuration 50 times. Figure 12 shows the fraction of hammers that triggered a bit-flip with a given n-sided pattern.

We observe an absence of bitflips for patterns with 16 or fewer aggressor rows ($n \leq 16$); whereas patterns with 17 or more aggressor rows (($n \geq 17$) consistently induce bit-flips.

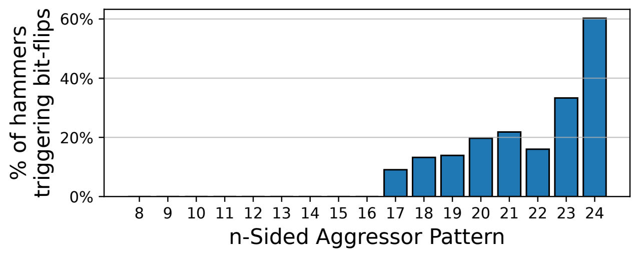


Figure 12: Fraction of hammers that trigger bit-flips with different n-sided aggressor patterns. No bit-flips are observed when the number of aggressors is less than 17.

This behavior mirrors that in prior work on TRR-protected DDR4 modules [19], where insufficient aggressors fail to trigger bitflips. The presence of bitflips only after $n = 16$ suggests that the TRR sampler in A6000's GDDR6 can track at most 16 rows per bank and issue mitigative refreshes.

Takeaway 4. A TRR-like fixed-size sampler is present in A6000's GDDR6 memory, tracking 16 rows per bank.

7.5 Bit-Flip Direction Characterization

Table 3 shows the bit-flip directions for each of the bit-flips we observed on A6000. A majority of the bit-flips are $0 \to 1$ flips (in anti cells) compared to flips from $1 \to 0$ (true cells), which are fewer in number. Interestingly, we observe Bank B has bit-flips in both directions, indicating that the GDDR6 DRAM bank architecture probably consists of multiple sub-arrays with different types of cells, similar to that observed for DDR4 DRAM [54]. Additionally, we notice that a majority of these flips are in the 0th, 6th, or 7th bit in a given byte, which may impact their exploitability, as we discuss in Section 8.

Table 3: Direction of Bit-Flips and Bit Location in a Byte.

Bit-flip No.	Original Data	Flipped Data	Direction $0 \to 1$	$1 \to 0$	Bit Location
A_1	0x55	0x45		✓	4
B_1	0xAA	0xAB	✓		0
B_2	0xAA	0xEA	✓		6
B_3	0xAA	0x2A		✓	7
C_1	0xAA	0xAB	✓		0
D_1	0xAA	0xEA	✓		6
D_2	0xAA	0xAB	✓		0
D_3	0xAA	0xEA	✓		6

7.6 Data Pattern Characterization

We analyzed the impact of data values in victim and aggressor rows on the frequency of bit-flips, using a representative $0 \to 1$ flip (D_1) and a $1 \to 0$ flip (A_1). We observe that the aggressor bits directly above or below the flippy victim bit can influence the bit-flip frequency for both directions. But, the aggressor bit diagonal to the flippy victim bit can only influence the $0 \to 1$ bit-flip frequency. Moreover, victim data patterns around the flippy bit influence the bit flip frequency to a greater extent than aggressor data. These insights can be used to launch Rambleed [43] attacks on GPU memories to leak sensitive bit values based on data-dependence in bit-flip frequency. We provide more details about these results in Appendix C.

8 End to End Exploit with GPU Rowhammer

8.1 Attack Overview

In this section, we demonstrate a proof-of-concept exploit using the Rowhammer bit-flips and use them to tamper with Deep Neural Network (DNN) weights on an A6000 GPU. We achieve an *accuracy degradation* attack during DNN inference like Terminal Brain Damage (TBD) [26], which simulated such attacks. TBD showed that a single bit-flip ($0 \to 1$) surgically inserted in the most significant bit (MSB) of the exponent of a DNN weight, can cause significant damage to model accuracy, reducing it by up to 99%, and almost 50% of the weights are susceptible to such attacks. In our exploit, we show for the first time that such an attack can be executed using our Rowhammer-induced bit-flips on GPUs, and the resultant tampering of the DNN weights resident in the GPU memory can impact the DNN accuracy significantly.

8.2 Attack Setup

Threat Model. We assume a multi-tenant setting where the attacker and victim are co-located on the same GPU and execute CUDA kernels in a time-multiplexed manner. Such co-location is feasible on time-sliced GPUs, as enabled by NVIDIA's GPU Operators and schedulers like RunAI [14], which default to 250ms slices ($\sim$10 tREFIs). This interval is enough for multiple hammering iterations and inducing bit-flips. Memory allocations can persist across slices, allowing attackers to retain access. We assume the GPU memory allocations are managed via the RAPIDS Memory Manager (RMM) [1], a widely used allocator in ML applications endorsed by NVIDIA for its speed and efficiency [23]. We assume ECC is disabled, as discussed in Section 2.5. Using our GPUHammer primitives, an attacker can profile DRAM for bit-flips, identifying both their location and direction.

Memory Massaging. To map victim data to vulnerable bits in GPU DRAM, the attacker must perform memory massaging. We achieve this by exploiting the fact that RAPIDS immediately reuses any freed memory, making it easier for an attacker to control the victim's memory allocation.

Specifically, the attacker first allocates large contiguous memory regions, then frees the chunk that contains a known flippy bit. When the victim runs and requests memory, this chunk is allocated to the victim process (e.g., PyTorch), mapping the victim's data adjacent to attacker-controlled aggressor rows. As RAPIDS can allocate in 256-byte increments, the attacker has fine-grained control over this placement.

Unlike RAPIDS, which reuses memory immediately, `cudaMalloc` reuses memory after a longer period; exploration of similar attacks with `cudaMalloc` is left for future work.

Attack Targets and Metrics. As the victim, we choose an ML inference process in PyTorch using pre-trained ImageNet models: AlexNet [42], VGG16 [70], ResNet50 [25], DenseNet161 [27], and InceptionV3 [72], whose weights we tamper with bit-flips. We use ILSVRC2012 dataset [66] as validation set to calculate the accuracy before ($\text{Acc}_{\text{pristine}}$) and after ($\text{Acc}_{\text{corrupted}}$) the bit-flip, and the Relative Accuracy Drop (RAD) as, $RAD = (\text{Acc}_{\text{pristine}} - \text{Acc}_{\text{corrupted}})/\text{Acc}_{\text{pristine}}$.

8.3 Attack Evaluation

We launch the bit-flipping attack on PyTorch models using FP16 weights. We make 50 attempts for the attack, which takes around 30 minutes per model, moving the victim memory by a randomly chosen distance in each attempt so that the bit flip maps to a new model weight. We report the highest accuracy degradation (highest RAD) across all 50 attempts and repeat it for 4 different $0 \rightarrow 1$ bit flips on the A6000 GPU.

Table 4 shows that bit flip D_1 and D_3 both cause considerable damage, reducing top-1 accuracy from 80% to less than 0.5% (RAD of 0.99). The attack is highly effective with D_1 and D_3 as PyTorch stores FP16 weights as 2-byte aligned, and both map to the Most Significant Bit (MSB) of the FP16 exponent (the 4th bit of exponent, E^4).[3] A $0 \rightarrow 1$ flip here exponentially increases the weight's value, causing much damage. On the other hand, the other two bit-flips, B_1 and B_2, map to bits in the Mantissa (M^0 and M^6), and only cause a small increase in the weights, with negligible accuracy impact.

For the bit flips that map to the MSB of the exponent (e.g., D_1 or D_3), not all 50 attempts are necessary. Figure 13 compares the average RAD across all models with D_1, as the number of exploit attempts increases. We see substantial degradation (RAD of 20%) after only 2 attempts, and RAD >99% after 8 attempts, highlighting the efficacy of our attack.

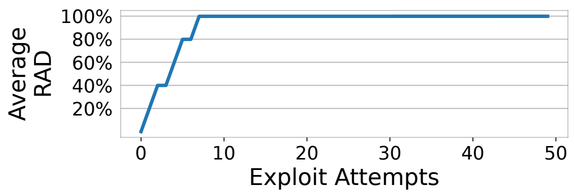


Figure 13: Average Relative Accuracy Drop (RAD) across all models as exploit attempts vary for bitflip D_1. Average RAD is 20% after 2 attempts, and $> 99\%$ after only 8 attempts.

[3]In NVIDIA GPUs, with little-endian architecture, the 6th bit of the 2nd physical byte, where D_1 and D_3 are located, maps to the Most Significant Bit (MSB) of the FP16 exponent, i.e., 4th bit of the exponent, E^4.

We attempted a similar attack on FP32 models but found that no bit-flips mapped to the MSB of the FP32 exponent (6th bit of the 4th byte), limiting accuracy damage. However, with more time and profiling of banks, it is likely that we will find exploitable bit-flips even for FP32 weights.

Table 4: Accuracy degradation attack on ImageNet models using FP16 weights, with Rowhammer bit-flips on NVIDIA A6000 GPU. We report the top-1 / top-5 accuracy without (Base Acc) and with (Degraded Acc) the bit-flip, and the Relative Accuracy Drop (RAD) for top-1 accuracy.

Flip	Network	Base Acc. (%)	Degraded Acc. (%)	RAD
D_1 (E^4)	**AlexNet**	56.66 / 79.78	0.10 / 0.64	0.99
	VGG16	72.22 / 90.90	0.08 / 0.52	0.99
	ResNet50	80.26 / 94.94	0.08 / 0.60	0.99
	DenseNet161	77.20 / 93.58	0.08 / 0.52	0.99
	InceptionV3	69.92 / 88.74	0.04 / 0.58	0.99
	Average	71.26 / 89.59	0.08 / 0.58	0.99
D_3 (E^4)	**AlexNet**	56.66 / 79.78	0.10 / 0.66	0.99
	VGG16	72.22 / 90.90	0.12 / 0.42	0.99
	ResNet50	80.26 / 94.94	0.02 / 0.34	0.99
	DenseNet161	77.20 / 93.58	0.08 / 0.56	0.99
	InceptionV3	69.92 / 88.74	0.10 / 0.50	0.99
	Average	71.26 / 89.59	0.09 / 0.50	0.99
B_1 (M^0)	**AlexNet**	56.66 / 79.78	56.66 / 79.78	0.00
	VGG16	72.22 / 90.90	72.22 / 90.90	0.00
	ResNet50	80.26 / 94.94	80.24 / 94.94	< 0.01
	DenseNet161	77.20 / 93.58	77.20 / 93.58	0.00
	InceptionV3	69.92 / 88.74	69.88 / 88.74	< 0.01
	Average	71.26 / 89.59	71.24 / 89.59	< 0.01
B_2 (M^6)	**AlexNet**	56.66 / 79.78	56.64 / 79.80	< 0.01
	VGG16	72.22 / 90.90	72.20 / 90.90	< 0.01
	ResNet50	80.26 / 94.94	80.24 / 94.94	< 0.01
	DenseNet161	77.20 / 93.58	77.20 / 93.58	0.00
	InceptionV3	69.92 / 88.74	69.84 / 88.76	< 0.01
	Average	71.26 / 89.59	71.24 / 89.60	< 0.01

E = Exponent, M = Mantissa, $(E/M)^{x}$ = x**th** bit in E/M

Takeaway 5. A single Rowhammer bit-flip in the MSB of the exponent of a model weight can significantly degrade accuracy from 80% down to 0.02% (RAD > 0.99).

9 Discussion

Additional Targets and Attack Vectors. While we target ML models in our Rowhammer attack, another potential target can be GPU page tables, which are also stored in the device memory [29], similar to how CPU Rowhammer exploits have targeted CPU PTEs [10, 19, 30, 40]. Another potential attack vector can be the GPU-based renderer in browsers [8] that may co-locate the pixel data of different websites. Future works can also attempt to steal sensitive pixels via Rambleed [43], leveraging the data-dependent flips we observe in Section 7.6.

Absence of Bit-flips on A100 and RTX3080. The Rowhammer threshold for DRAM devices varies with technology, process variations, and runtime conditions (e.g., temperature). Consequently, the vulnerability levels can have chip-to-chip variations as well as be form factor dependent. This makes pinpointing the root cause of the absence of bit flips difficult. The RTX3080, a laptop GPU, uses GDDR6 chips distinct from the workstation-grade A6000, which possibly have a higher Rowhammer threshold, making it less vulnerable. The A100, equipped with HBM2e memory, features a fundamentally different DRAM architecture compared to GDDR. Thus it might not only have a higher Rowhammer threshold, but also a different in-DRAM mitigation mechanism immune to the attack patterns we tested. Future works can systematically study the vulnerability of these GPUs using more sophisticated attack patterns [30, 40, 50]. Additionally, on-die ECC in HBM2e can also mask observable bit flips, as discussed next.

On-Die ECC in HBM2e and Future GPU Memories. GPUs like the A100 (HBM2e), H100/B100 (HBM3/e), and RTX5090 (GDDR7) integrate on-die ECC [2, 9, 33] that can correct single bit-flips and detect double bit-flips by default. This can conceal any Rowhammer-based single bit-flips that may occur. However, if more than two bit-flips occur within the same code word, the on-die ECC can also mis-correct the data. Thus, on-die ECC can make Rowhammer attacks more difficult on emerging GPUs, but not impossible.

Impact on NVIDIA MIG and Confidential Computing. In cloud-based GPUs, NVIDIA's Multi-instance-GPU (MIG) [55] allows shared usage of a GPU by spatially partitioning GPU memory along channels for MIG instances assigned to different users. Meanwhile, NVIDIA Confidential Computing (CC) assigns an entire GPU or multiple GPUs to a single confidential VM. Both configurations prevent multi-tenant data co-location in the same DRAM bank required for our exploits, thwarting our Rowhammer-based multi-tenant data tampering exploits in such environments. We leave the exploration of Rowhammer exploits on such environments for future work.

10 Mitigations

Enabling ECC. GDDR-based GPUs like A6000 can optionally enable memory-controller-based ECC, which can detect and correct single bit-flips and mitigate our attack. However, systems disable ECC by default as it is stored out-of-band, in a separate region of memory, and enabling it causes a memory overhead of 6.5% [71] and slowdown. We evaluate the slowdown of ECC on our A6000 GPU, using CUDA samples [12] and MLPerf v4.1 Inference benchmarks [64]. Figure 14 shows that enabling ECC results in memory bandwidth loss of up to 12% and slowdown of 3%-10% in ML applications. Thus, ECC can effectively mitigate our attacks at reasonable costs.

Randomizing Virtual to Physical Mappings. In the absence of access to physical addresses on the GPU, our attack relies on directly reversing the virtual address to DRAM row

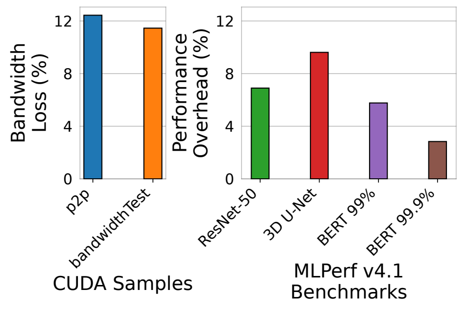


Figure 14: Overheads of enabling ECC in A6000 GPU for MLPerf Inference and CUDA samples benchmarks.

mappings. This leverages the fact that the virtual-to-physical mappings on NVIDIA GPUs are the same across different runs for large memory allocations spanning the entire GPU RAM. If the GPU driver randomized the virtual to physical mapping on each run, it would require the attacker to repeatedly profile the DRAM row mappings each time. While this would significantly increase the cost of the attack, this may have some impact on performance due to TLB fragmentation.

Making Memory Massaging Unpredictable. Our exploit requires precise memory massaging to map victim memory to the vulnerable bits in DRAM. RAPIDS memory manager, designed for memory efficiency, immediately reallocates memory released by one allocation, enabling such precise memory massaging. Using allocators that quarantine deallocated memory [69] or randomize the reallocation makes memory massaging harder; however such allocators may increase the memory usage, making them less desirable.

Adopting Modern Rowhammer Mitigations. Beyond TRR, advanced Rowhammer mitigations, such as Refresh Management (RFM), have been proposed for HBM3 [35] and DDR5 [36], and Per Row Activation Counting (PRAC) [5, 53, 60, 80] has been proposed in the DDR5 specification. Adopting such mitigations in emerging GDDRx memories can address the risk of Rowhammer in future GPUs.

11 Related Work

A. Rowhammer Exploits: Prior Rowhammer exploits targeted CPU-based DDRx and LPDDRx memories. In DDR3, Flip Feng Shui (FFS) [63] used memory de-duplication in virtualized settings to compromise cryptographic code, while Drammer [75] enabled privilege escalation on mobile platforms. ThrowHammer [73] demonstrated Rowhammer exploits over the network, while Rowhammer.js [22] demonstrated these from JavaScript. Attacks like ECCploit [10] defeated ECC, while TRRespass [19], Blacksmith [30] and SMASH [13] defeated in-DRAM defenses like TRR in DDR4. Zenhammer [31] recently showed modern AMD Zen CPUs and DDR5 memories are also vulnerable. Grand Pwning Unit [18] used integrated GPUs on mobile platforms that share the CPU DRAM to perform Rowhammer attacks on LPDDR3 CPU DRAM. In contrast to all these attacks, we enable the first Rowhammer attacks on GDDR6 GPU memory in discrete NVIDIA GPUs, widely used in workstations and servers.

B. Rowhammer Mitigations: Prior software-based Rowhammer mitigations have focused on CPUs. GuardION [76] uses guard rows between security domains, while CATT [4], RIP-RH [3], ZebRAM [41], and Siloz [49] isolate security domains in separate DRAM regions. These methods could potentially defend against GPU-based attacks with NVIDIA driver support. Hardware-based mitigations, including Rowhammer trackers [28, 38, 51, 58, 61], aggressor relocation [67, 68, 79, 81], delayed activations [83], and refresh-generating activations [52], designed for CPU DRAM, might also be adapted to GPUs, though their overheads may vary depending on the GPU latency and bandwidth constraints.

C. Bit-Flipping Attacks on DNNs: Many exploits have shown that bit-flips in CPUs can tamper with DNN weights, causing accuracy degradation [26, 46, 48, 62, 82] and enabling backdoor injection [7, 74]. Our GPU-based exploit injects a single bit flip in the MSB of the exponent in DNN weights, similar to TBD [26]. However, more sophisticated exploits with more than one bit-flip can also be adapted to GPUs. Recent works propose ML model-based defenses against bit-flips, including usage of early-exits [77], honeypot neurons [47], and checksums [45]. Such defenses can also be used on GPUs to mitigate bit-flipping attacks on ML models.

12 Conclusion

This paper introduces GPUHammer, the first Rowhammer attack on NVIDIA GPUs with GDDR6 DRAM. Overcoming challenges like high memory latency and faster refresh, GPUHammer achieves bit-flips by reverse-engineering DRAM bank mappings and leveraging GPU parallelism. We show how these bit-flips dramatically degrade ML model accuracy by up to 80% on GPUs, highlighting the vulnerability of GPUs to Rowhammer attacks and the need for mitigations.

13 Acknowledgments

We thank the anonymous reviewers and our shepherd for their helpful suggestions. We also acknowledge NVIDIA and the CSPs for their prompt responses following our report. This research was supported by Natural Sciences and Engineering Research Council of Canada (NSERC) under funding reference number RGPIN-2023-04796, and an NSERC-CSE Research Communities Grant under funding reference number ALLRP-588144-23. Any research, opinions, or positions expressed in this work are solely those of the authors and do not represent the official views of the Communications Security Establishment Canada or the Government of Canada.

14 Ethics Considerations

We responsibly disclosed the Rowhammer vulnerability in the A6000 GPU to NVIDIA on 15th January, 2025, sharing proof-of-concept code and a draft of our paper. Subsequently, we also disclosed our findings to major cloud providers (AWS, Azure, GCP) who may be affected, so that the vulnerability is addressed before public dissemination. Our experiments were conducted locally or in isolated environments, and targeted open-source models, mitigating risks to production systems.

15 Open Science

Upon expiry of the embargo requested by NVIDIA on 12th August 2025, our code will be publicly available on `https://github.com/sith-lab/gpuhammer`. Our artifact is archived at `https://doi.org/10.5281/zenodo.15612689` and includes the code to reproduce the key results on the A6000 GPU including the Rowhammer campaigns (Table-1), bit flip characterization (Figure 11, Figure 12, Table 3), and the exploit on ML applications (Figure 13, Table 4).

References

[1] RAPIDS AI. Rapids memory manager. `https://github.com/rapidsai/rmm`.

[2] AnandTech. JEDEC Publishes GDDR7 Memory Spec: Next-Gen Graphics Memory Adds Faster PAM3 Signaling and On-Die ECC. `https://www.anandtech.com/show/21287/jedec-publishes-gddr7-specifications-pam3-ecc-higher-density`, 2024. Accessed: 2025-01-22.

[3] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahamd-Reza Sadeghi. Rip-rh: Preventing rowhammer-based inter-process attacks. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 561–572, 2019.

[4] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 117–130, 2017.

[5] Oğuzhan Canpolat, A Giray Yağlıkçı, Geraldo F Oliveira, Ataberk Olgun, Nisa Bostancı, Ismail Emir Yuksel, Haocong Luo, Oğuz Ergin, and Onur Mutlu. Chronus: Understanding and securing the cutting-edge industry solutions to dram read disturbance. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025.

[6] Patrick Chao, Alexander Robey, Edgar Dobriban, Hamed Hassani, George J Pappas, and Eric Wong. Jailbreaking black box large language models in twenty queries. *arXiv preprint arXiv:2310.08419*, 2023.

[7] Huili Chen, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. Proflip: Targeted trojan attack with progressive bit flips. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 7718–7727, 2021.

[8] Google Chrome. Webgpu: Unlocking modern gpu access in the browser. https://developer.chrome.com/blog/webgpu-io2023, 2023. Accessed: 2025-01-22.

[9] Ki Chul Chun, Yong Ki Kim, Yesin Ryu, Jaewon Park, Chi Sung Oh, Young Yong Byun, So Young Kim, Dong Hak Shin, Jun Gyu Lee, Byung-Kyu Ho, et al. A 16-GB 640-GB/s HBM2E DRAM with a data-bus window extension technique and a synergetic on-die ECC scheme. *IEEE Journal of Solid-State Circuits*, 56(1):199–211, 2020.

[10] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 55–71, 2019.

[11] Marcin Copik, Alexandru Calotoiu, Pengyu Zhou, University of Toronto, Lukas Tobler, Torsten Hoefler, ETH Z¨urich, and AYES. Magnificient: Fast, isolated, and gpu-enabled serverless functions. *SC '24: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*.

[12] NVIDIA Corporation. Cuda samples. https://github.com/NVIDIA/cuda-samples, 2023. Accessed: 2025-01-22.

[13] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized many-sided rowhammer attacks from JavaScript. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1001–1018. USENIX Association, August 2021.

[14] NVIDIA Run:ai Docs. GPU Time Slicing. https://docs.run.ai/v2.17/Researcher/scheduling/GPU-time-slicing-scheduler/. Accessed: 2025-01-22.

[15] NVIDIA Developer Forums. Ecc on vs ecc off. https://forums.developer.nvidia.com/t/ecc-on-vs-ecc-off/20315. Accessed: 2025-01-22.

[16] NVIDIA Developer Forums. Impact of enabling ecc on power and performance. https://forums.developer.nvidia.com/t/impact-of-enabling-ecc-on-power-and-performance/174567. Accessed: 2025-01-22.

[17] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1322–1333, 2015.

[18] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand pwning unit: Accelerating microarchitectural attacks with the GPU. In *2018 ieee symposium on security and privacy (sp)*, pages 195–210. IEEE, 2018.

[19] Pietro Frigo, Emanuele Vannacc, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Trrespass: Exploiting the many sides of target row refresh. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 747–762, 2020.

[20] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[21] Google. Share GPUs across workloads with GPU timesharing. https://cloud.google.com/kubernetes-engine/docs/how-to/timesharing-gpus. Accessed: 2025-01-22.

[22] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*, pages 300–321. Springer, 2016.

[23] Mark Harris and Mark Harris. Fast, flexible allocation for nvidia cuda with rapids memory manager. https://developer.nvidia.com/blog/fast-flexible-allocation-for-cuda-with-rapids-memory-manager/. Accessed: 2025-01-22.

[24] Hasan Hassan, Yahya Can Tugrul, Jeremie S. Kim, Victor van der Veen, Kaveh Razavi, and Onur Mutlu. Uncovering in-dram rowhammer protection mechanisms:a new methodology, custom rowhammer patterns, and implications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1198–1213, New York, NY, USA, 2021. Association for Computing Machinery.

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

[26] Sanghyun Hong, Pietro Frigo, Yigitcan Kaya, Cristiano Giuffrida, and Tudor Dumitras. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 497–514, Santa Clara, CA, August 2019. USENIX Association.

[27] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869*, 2014.

[28] Aamer Jaleel, Gururaj Saileshwar, Stephen W. Keckler, and Moinuddin Qureshi. Pride: Achieving secure rowhammer mitigation with low-cost in-dram trackers. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 1157–1172, 2024.

[29] Sungbin Jang, Junhyeok Park, Osang Kwon, Yongho Lee, and Seokin Hong. Rethinking page table structure for fast address translation in gpus: A fixed-size hashed page table. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, PACT '24, page 325–337, New York, NY, USA, 2024. Association for Computing Machinery.

[30] Patrick Jattke, Victor Van Der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 716–734, 2022.

[31] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. ZenHammer: Rowhammer attacks on AMD zen-based platforms. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1615–1633, Philadelphia, PA, August 2024. USENIX Association.

[32] JEDEC. DDR4 SDRAM standard (JESD79-4B). 2017.

[33] JEDEC. JEDEC Publishes HBM3 Update to High Bandwidth Memory (HBM) Standard. https://www.jedec.org/news/pressreleases/jedec-publishes-hbm3-update-high-bandwidth-memory-hbm-standard, 2022. Accessed: 2025-01-22.

[34] JEDEC. Graphics Double Data Rate (GDDR6) SGRAM Standard (JESD250D). 2023.

[35] JESD238A. HBM3 Specification. 2023.

[36] JESD79-5. DDR5 Specification. 2020.

[37] Jeremie S. Kim, Minesh Patel, A. Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting rowhammer: An experimental analysis of modern dram devices and mitigation techniques. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 638–651, 2020.

[38] M. Kim, J. Park, Y. Park, W. Doh, N. Kim, T. Ham, J. W. Lee, and J. Ahn. Mithril: Cooperative row hammer protection on commodity dram leveraging managed refresh. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1156–1169, Los Alamitos, CA, USA, apr 2022. IEEE Computer Society.

[39] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: an experimental study of dram disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 361–372. IEEE Press, 2014.

[40] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-Double: Hammering from the next row over. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3807–3824, Boston, MA, August 2022. USENIX Association.

[41] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. Zebram: comprehensive and compatible software protection against rowhammer attacks. In *13th USENIX - (OSDI 18)*, pages 697–710, 2018.

[42] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[43] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 695–711. IEEE, 2020.

[44] Chester Lam. Measuring gpu memory latency. https://chipsandcheese.com/p/measuring-gpu-memory-latency, 2021. Accessed: 2025-01-22.

[45] Jingtao Li, Adnan Siraj Rakin, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. Radar: Run-time adversarial weight attack detection and accuracy recovery. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 790–795, 2021.

[46] Shaofeng Li, Xinyu Wang, Minhui Xue, Haojin Zhu, Zhi Zhang, Yansong Gao, Wen Wu, and Xuemin Sherman Shen. Yes, one-bit-flip matters! universal dnn model inference depletion with runtime code fault injection. In *Proceedings of the 33th USENIX Security Symposium*, 2024.

[47] Qi Liu, Jieming Yin, Wujie Wen, Chengmo Yang, and Shi Sha. NeuroPots: Realtime proactive defense against Bit-Flip attacks in neural networks. In *32nd USENIX Security Symposium (USENIX Security)*, 2023.

[48] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. Fault injection attack on deep neural network. In *Proceedings of the 36th International Conference on Computer-Aided Design (ICCAD)*, 2017.

[49] Kevin Loughlin, Jonah Rosenblum, Stefan Saroiu, Alec Wolman, Dimitrios Skarlatos, and Baris Kasikci. Siloz: Leveraging dram isolation domains to prevent inter-vm rowhammer. In *29th Symposium on Operating Systems Principles (SOSP)*, 2023.

[50] Haocong Luo, Ataberk Olgun, Abdullah Giray Yağlıkçı, Yahya Can Tuğrul, Steve Rhyner, Meryem Banu Cavlak, Joël Lindegger, Mohammad Sadrosadati, and Onur Mutlu. Rowpress: Amplifying read disturbance in modern dram chips. In *50th International Symposium on Computer Architecture (ISCA)*, 2023.

[51] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. Protrr: Principled yet optimal in-dram target row refresh. In *IEEE Symposium on Security and Privacy (SP)*, pages 735–753. IEEE, 2022.

[52] Michele Marazzi, Flavien Solt, Patrick Jattke, Kubo Takashi, and Kaveh Razavi. REGA: Scalable Rowhammer Mitigation with Refresh-Generating Activations. In *IEEE Symposium on Security and Privacy (SP)*, 2023.

[53] JEDEC. JESD79-5C. https://www.jedec.org/document_search?search_api_views_fulltext=jesd79-5c. Accessed: 2025-01-22.

[54] Hwayong Nam, Seungmin Baek, Minbok Wi, Michael Jaemin Kim, Jaehyun Park, Chihun Song, Nam Sung Kim, and Jung Ho Ahn. Dramscope: Uncovering dram microarchitecture and characteristics by issuing memory commands. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 1097–1111, 2024.

[55] NVIDIA. Nvidia multi-instance gpu and nvidia virtual compute server. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/solutions/resources/documents1/Technical-Brief-Multi-Instance-GPU-NVIDIA-Virtual-Compute-Server.pdf, 2020. Accessed: 2025-01-22.

[56] NVIDIA. PTX: Parallel Thread Execution ISA, Version 8.5. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, 2025. Accessed: 2025-01-22.

[57] Ataberk Olgun, Majd Osseiran, A Giray Yağlıkçı, Yahya Can Tuğrul, Haocong Luo, Steve Rhyner, Behzad Salami, Juan Gomez Luna, and Onur Mutlu. An experimental analysis of rowhammer in hbm2 dram chips. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pages 151–156. IEEE, 2023.

[58] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. Graphene: Strong yet lightweight row hammer protection. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2020.

[59] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, Austin, TX, August 2016. USENIX Association.

[60] Moinuddin Qureshi and Salman Qazi. Moat: Securely mitigating rowhammer with per-row activation counters. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 698–714, 2025.

[61] Moinuddin Qureshi, Aditya Rohan, Gururaj Saileshwar, and Prashant J. Nair. Hydra: enabling low-overhead mitigation of row-hammer at ultra-low thresholds via hybrid tracking. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 699–710, New York, NY, USA, 2022. Association for Computing Machinery.

[62] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1211–1220, 2019.

[63] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *25th USENIX Security Symposium (USENIX Security)*, 2016.

[64] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[65] Jon Peddie Research. NVIDIA Market Share. https://www.jonpeddie.com/news/shipments-of-graphics-add-in-boards-decline-in-q1-of-24-as-the-market-experiences-a-return-to-seasonality/. Accessed: 2025-01-22.

[66] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vision*, 115(3):211–252, December 2015.

[67] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J. Nair. Randomized row-swap: mitigating row hammer by breaking spatial correlation between aggressor and victim rows. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[68] Anish Saxena, Gururaj Saileshwar, Prashant J. Nair, and Moinuddin Qureshi. Aqua: Scalable rowhammer mitigation by quarantining aggressor rows at runtime. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 108–123, 2022.

[69] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference (USENIX ATC)*, 2012.

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

[71] Michael B Sullivan, Mohamed Tarek Ibn Ziad, Aamer Jaleel, and Stephen W Keckler. Implicit memory tagging: No-overhead memory safety using alias-free tagged ecc. In *50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[72] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[73] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[74] M Caner Tol, Saad Islam, Andrew J Adiletta, Berk Sunar, and Ziming Zhang. Don't knock! rowhammer at the backdoor of dnn models. In *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2023.

[75] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[76] Victor Van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. Guardion: Practical mitigation of dma-based rowhammer attacks on arm. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 92–113. Springer, 2018.

[77] Jialai Wang, Ziyuan Zhang, Meiqi Wang, Han Qiu, Tianwei Zhang, Qi Li, Zongpeng Li, Tao Wei, and Chao Zhang. Aegis: Mitigating targeted bit-flip attacks against deep neural networks. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2329–2346, Anaheim, CA, August 2023. USENIX Association.

[78] Minghua Wang, Zhi Zhang, Yueqiang Cheng, and Surya Nepal. Dramdig: A knowledge-assisted tool to uncover dram address mapping. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[79] Minbok Wi, Jaehyun Park, Seoyoung Ko, Michael Jaemin Kim, Nam Sung Kim, Eojin Lee, and Jung Ho Ahn. SHADOW: Preventing Row Hammer in DRAM with Intra-Subarray Row Shuffling. In *HPCA*, 2023.

[80] Jeonghyun Woo, Shaopeng Chris Lin, Prashant J Nair, Aamer Jaleel, and Gururaj Saileshwar. Qprac: Towards secure and practical prac-based rowhammer mitigation using priority queues. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025.

[81] Jeonghyun Woo, Gururaj Saileshwar, and Prashant J Nair. Scalable and secure row-swap: Efficient and safe row hammer mitigation in memory systems. In *HPCA*, 2023.

[82] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. DeepHammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips. In *USENIX Security*, 2020.

[83] A. Giray Yağlıkçi, Minesh Patel, Jeremie S. Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. Blockhammer: Preventing rowhammer at low cost by blacklisting rapidly-accessed dram rows. In *HPCA*, 2021.

Appendix

A Synchronized Many-Sided Hammer Kernel

Our synchronized many-sided hammering uses a CUDA kernel with *k* warps and *m* threads per warp, hammering `k × m` rows. The kernel is launched as `hammer<<<1, 1024>>>` to ensure a sufficient number of warps. Each warp contains 32 threads, of which the first *m* threads are used for hammering.

```
void hammer(size_t **aggr_addr, size_t it, size_t
    k, size_t m, size_t round, size_t delay) {
  size_t dummy, dummy_sum;
  size_t warpId = threadIdx.x / 32;
  size_t threadId_in_warp = threadIdx.x % 32;

  if (warpId < k && threadId_in_warp < m) {
    size_t *addr = *(aggr_addr + threadId_in_warp
    + warpId * m);
    asm volatile("discard.global.L2 [%0], 128;"
                 ::"l"(addr));
    __syncthreads();
    /* Hammers performed for 128ms */
    for (;count--;) {
      /* perform 'round' ACTs before sync delay */
      for (size_t i = round; i--;) {
        asm volatile("discard.global.L2 [%0],
            128;" ::"l"(addr));
        /* Aggressor Row Activation*/
        asm volatile("ld.u64.global.volatile %0,
             [%1];" : "=l"(dummy) : "l"(addr));
        /* Ordered memory access within threads */
        __threadfence_block();
      }
      /* Delay Added to Synchronize to REF */
      for (size_t i = delay; i--;)
        dummy_sum += dummy;
    }
  }
}
```

Listing 4: *k*-warps, *m*-threads per warp Hammering Kernel

B Virtual Address to DRAM Row Mapping

A 2KB DRAM row is divided into 256-byte chunks, with each chunk mapped to a different physical bank and row. As we cannot access GPU physical memory, we allocate a large array in virtual memory (47GB out of 48GB), and identify the 256B chunks in virtual addresses (VA) that map to unique rows, as shown in Figure 15. A 256-byte chunk maps to a new DRAM row within a bank only after all the chunks of the prior row have been mapped. However, the distance between successive rows in a bank does not appear to have any obvious trends. This suggests that a complex function might be used to map memory addresses to DRAM bank/rows.

C Characterizing Bit-Flip Data Patterns

We analyze the impact of data patterns on bit-flip frequency by varying the victim and aggressor data from `0x00` to `0xFF`, with a step size of `0x11`, using a 24-sided pattern on our A6000 GPU. We measure the bit flip frequency as the data values vary for a $0 \rightarrow 1$ flip (D_1) and a $1 \rightarrow 0$ flip (A_1), as shown in Figure 16a and Figure 16b respectively.

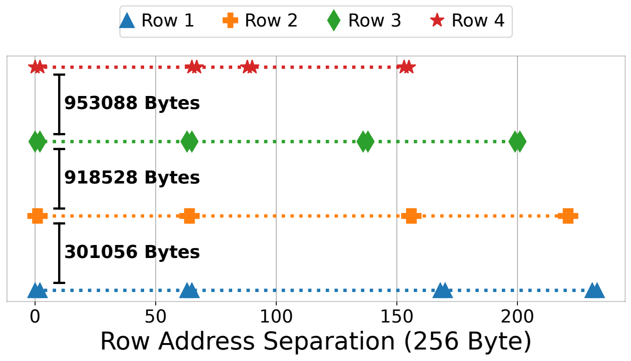
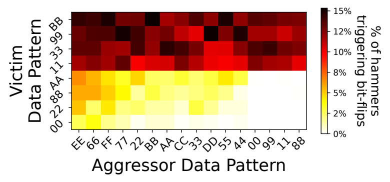
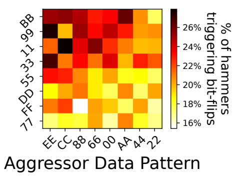


Figure 15: Mapping of virtual addresses to DRAM rows in a given bank, for four different rows (Rows 1 to 4) at 256-byte granularity. Each 256 byte chunk is one point on the graph.



(a) A $0 \rightarrow 1$ bit-flip at the 6th bit (D_1)



(b) A $1 \rightarrow 0$ bit-flip at the 4th bit (A_1)

Figure 16: Heatmap showing fraction of hammers triggering bit-flips with varying victim and aggressor data patterns.

Aggressor Data Influence. The aggressor bits directly above and below the victim bit strongly influence the rate of bit-flips, with the highest rate achieved when these bits are the inverse of the victim bit. For example, in the $1 \rightarrow 0$ flip (A_1, 4th bit), when victim data is `0x55` (0b010**1**0101), the aggressor data with `0xEE` (0b111**0**1110) or `0xCC` (0b110**0**1100) achieves high bit-flip rates. For the $0 \rightarrow 1$ flip (D_1, 6th bit), additionally, the aggressor bit diagonally opposite the vulnerable victim bit has an influence. For instance, when victim data is `0xAA` (0b1**0**101010), even the aggressor data `0x33` (0b0**01**10011) triggers bit flips in addition to `0xEE` (0b1**1**101110).

Victim Data Influence. The victim data has a stronger influence on the rate of bit-flips than aggressor data. For both the $1 \rightarrow 0$ flip (A_1, 4th bit) and the $0 \rightarrow 1$ flip (D_1, 6th bit), the victim data of `0xBB` (0b**1**0**1**1**1**011) and `0x99` (0b**1**0**0**1**1**001) are the most effective. Interestingly, for the $0 \rightarrow 1$ flip (D_1, 6th bit), (D_1), `0x11` (0b**0**0**0**10001), which surrounds the victim bit with two bits of the same charge, is more effective than the gridded pattern `0xAA` (0b**1**0**1**01010), where the victim bit is surrounded by two bits of the opposite charge.