TLB-pilot: Mitigating TLB Contention Attack on GPUs with Microarchitecture-Aware Scheduling

BANG DI, Hunan University and Alibaba Group
DAOKUN HU, Hunan University
ZHEN XIE, University of California, Merced
JIANHUA SUN and HAO CHEN, Hunan University
JINKUI REN, Alibaba Group
DONG LI, University of California, Merced

Co-running GPU kernels on a single GPU can provide high system throughput and improve hardware utilization, but this raises concerns on application security. We reveal that translation lookaside buffer (TLB) attack, one of the common attacks on CPU, can happen on GPU when multiple GPU kernels co-run. We investigate conditions or principles under which a TLB attack can take effect, including the awareness of GPU TLB microarchitecture, being lightweight, and bypassing existing software and hardware mechanisms. This TLB-based attack can be leveraged to conduct Denial-of-Service (or Degradation-of-Service) attacks. Furthermore, we propose a solution to mitigate TLB attacks. In particular, based on the microarchitecture properties of GPU, we introduce a software-based system, TLB-pilot, that binds thread blocks of different kernels to different groups of streaming multiprocessors by considering hardware isolation of last-level TLBs and the application's resource requirement. TLB-pilot employs lightweight online profiling to collect kernel information before kernel launches. By coordinating software- and hardware-based scheduling and employing a kernel splitting scheme to reduce load imbalance, TLB-pilot effectively mitigates TLB attacks. The result shows that when under TLB attack, TLB-pilot mitigates the attack and provides on average 56.2% and 60.6% improvement in average normalized turnaround times and overall system throughput, respectively, compared to the traditional Multi-Process Service based co-running solution. When under TLB attack, TLB-pilot also provides up to 47.3% and 64.3% improvement (41% and 42.9% on average) in average normalized turnaround times and overall system throughput, respectively, compared to a state-of-the-art co-running solution for efficiently scheduling of thread blocks.

CCS Concepts: • **Computer systems organization → Reliability**; **Single instruction, multiple data;**

Additional Key Words and Phrases: TLB contention, GPU, CUDA, high performance

B. Di and D. Hu contributed equally to this work.

This research was supported by National Natural Science Foundation of China under grants 61972137 and 61772183.

Authors' addresses: B. Di, Hunan University, Changsha, China, and Alibaba Group, China; email: dibang@hnu.edu.cn; D. Hu, J. Sun (corresponding author), and H. Chen (corresponding author), Hunan University, Changsha, China; emails: {daokunhu, jhsun, haochen}@hnu.edu.cn; Z. Xie and D. Li, University of California, Merced, Merced, CA; emails: {zxie10, dli35}@ucmerced.edu; J. Ren, Alibaba Group, Shanghai, China; email: guancheng.rjk@alibaba-inc.com.

1544-3566/2021/12-ART9 $15.00
https://doi.org/10.1145/3491218

ACM Reference format:
Bang Di, Daokun Hu, Zhen Xie, Jianhua Sun, Hao Chen, Jinkui Ren, and Dong Li. 2021. TLB-pilot: Mitigating TLB Contention Attack on GPUs with Microarchitecture-Aware Scheduling. *ACM Trans. Archit. Code Optim.* 19, 1, Article 9 (December 2021), 23 pages.
https://doi.org/10.1145/3491218

1 INTRODUCTION

With the continuous increase of GPU compute density, there is a growing need of sharing GPU among multiple applications to avoid underutilization of GPU resources [6, 24, 31, 39–41, 45, 49, 50, 55]. However, co-running GPU kernels on a single GPU raises concerns on application security. Recent efforts reveal that buffer overflow attack [17], side channel attack [34], and covert channel attack [33] can happen when GPU kernels co-run. Co-running GPU kernels demands strong isolation between different kernels to prevent those potential attacks.

In this article, we reveal that the **translation lookaside buffer (TLB)** attack can happen on GPU when multiple GPU kernels co-run. We show that it is highly possible that a malicious GPU kernel intentionally constructs severe contentions on shared TLB to cause dramatic performance degradation of a co-running kernel. The performance degradation can be leveraged to conduct Denial-of-Service (or Degradation-of-Service) attacks.

Unlike TLB attacks on CPU, TLB attacks on GPU for co-running kernels must meet a couple of conditions or principles, imposed by unique GPU architecture. First, the TLB attack should be based on the awareness of TLB microarchitecture, particularly how TLBs are shared and isolated between **streaming multiprocessors (SMs)** on GPU. Second, the attack kernel has to be lightweight to enable co-running of kernels on GPU. The hardware scheduler on GPU parallels the execution of attack and benign user kernels, only when GPU resource (e.g., registers and thread blocks) consumed by the two kernels is available on GPU. Otherwise, the execution of the two kernels is serialized, which makes the attack invalid. The attack kernel with low requirements on hardware resource has high applicability to attack benign kernels. The requirement of the lightweight attack kernel means that traditional expensive TLB attack approaches [48] cannot work on GPU. Third, the attack kernel must bypass software and hardware mechanisms on GPU to work. These mechanisms include large TLB reach [8], address randomization, and compiler optimization, which was introduced into GPU to enable load balance and high performance.

We show how a TLB attack can happen with a delicate design. In particular, we introduce a small GPU kernel that allocates a number of memory blocks scattered sparsely in the address space, then accesses them to evict **page table entries (PTEs)** in L2 TLBs. Those memory blocks are carefully chosen and widely spread such that accessing them can cover most PTEs in L2 TLB according to TLB microarchitecture, meanwhile beating the effect of address randomization. The attack kernel also considers the impact of hardware-managed scheduling of thread blocks on SMs, and shares L2 TLB between thread blocks of the attack and benign kernels. With such a TLB attack, we observe up to 3.9× performance loss from a benign user kernel.

After demonstrating the potential TLB attacks, we propose a solution to mitigate TLB attacks on GPU. Unlike L1/L2 caches that can be manipulated using specific instructions (e.g., flush and load instructions) to address cache contentions [30], TLB is a hardware component whose details are typically not disclosed, and there is no instruction to directly manipulate TLB. The previous work [23, 32, 52] and our benchmarking results reveal that there are multiple last-level TLBs on GPU. Each last-level TLB is shared between multiple SMs forming an SM group, but across SM groups, there is no TLB sharing. Based on the preceding microarchitecture properties, we introduce a software-based system (named *TLB-pilot*) that binds thread blocks of different kernels to

different SM groups by considering the isolation of last-level TLBs and the application's resource requirements. TLB-pilot avoids or mitigates sharing of last-level TLBs between different GPU kernels and hence mitigates TLB attacks. TLB-pilot is a software-based solution directly deployable on GPU without the need of hardware modification.

However, building the preceding software-based scheduling system on GPU to mitigate TLB attacks is not trivial. First, the software-based scheduling must coordinate with application-agnostic, hardware-based scheduling, which is challenging. We only have primitive knowledge on the proprietary hardware-based scheduling mechanism. Without detailed knowledge on the hardware-based scheduling, the software-based schedule must leverage the scheduling results from hardware to bind thread blocks of a kernel with specific SM groups to enable TLB isolation. Second, mitigating TLB attacks must have minimum impact on the performance of benign GPU kernels. This indicates that we should avoid complicated algorithms or add extra functionality into the GPU kernels that can impair the performance. In addition, we should avoid low SM utilization when applying the software-based scheduling.

Existing works schedule thread blocks of co-running kernels [7, 10, 53, 54, 61], aiming to enhance data locality or enable kernel preemption for prioritized scheduling. Those efforts can be roughly classified into two categories: persistent thread-based approaches [7, 10, 54, 61] and SM-centric scheduling [53]. When applying them to alleviate TLB contentions, we face fundamental limitations.

The persistent thread-based approach lacks hardware information, which is essential to mitigate TLB attacks. In particular, the persistent thread-based approach creates a small number of threads that simultaneously run on GPU. These threads stay alive throughout the execution of co-running kernels and continuously fetch and execute tasks (thread blocks) from one or more task queues. This approach suffers from frequent synchronization between CPU and GPU to transfer tasks, which brings performance overhead. More importantly, this approach binds persistent threads with tasks while leaving the binding between persistent threads and SMs to proprietary hardware and runtime. As a result, persistent threads do not have sufficient knowledge on TLB microarchitecture, which is necessary to enable isolation between thread blocks of attack and benign kernels. The SM-centric scheduling [53] relies on expensive offline analysis (e.g., thread affinity analysis) and online analysis through a slow high-climbing algorithm to decide which thread blocks should go to which SMs, which is difficult to adopt in a production environment.

TLB-pilot addresses the preceding challenges and avoids the limitation in the existing approaches. TLB-pilot employs lightweight online profiling to collect kernel information before kernel launches to decide how the binding should happen. The online profiling can be lightweight, because we leverage offline performance modeling and limited knowledge on hardware scheduling, and request the binding at the granularity of SM groups (instead of SMs as in the existing efforts). The process of deciding the binding considers TLB microarchitecture to isolate TLBs between co-running kernels. To enforce the binding at runtime and avoid incorrect binding because of application-agnostic, hardware scheduling, TLB-pilot manipulates the launching of thread blocks and asks thread blocks to coordinatively terminate themselves to meet the binding goal. In general, our method is lightweight but is microarchitecture-aware without the necessity of using expensive online analysis.

Furthermore, we identify a load imbalance problem encountered when mitigating TLB attack. In particular, when one kernel is terminated while the other co-running kernel is not, the co-running kernel cannot leverage the idling SMs released by the terminated kernel, because of the binding between SM groups and thread blocks to mitigate TLB attacks. To solve this problem, TLB-pilot introduces a kernel splitting mechanism. This mechanism splits the long-running kernel into two small kernels, one that runs in parallel with the short-running kernel with the awareness of

mitigating TLB attack, and another that runs after the termination of the short-running kernel and utilizes all SMs.

This work makes following contributions

- We demonstrate the possibility of mounting TLB attacks when co-running GPU kernels.
- We propose TLB-pilot, a software-based scheduling system to mitigate TLB attack without hardware modification. We make TLB-pilot open source [5].
- We extensively evaluate TLB-pilot with a set of representative benchmarks. We show that when co-running benign user kernels with the TLB attack kernel, TLB-pilot provides an average 56.2% and 60.6% improvement in **average normalized turnaround times (ANTT)** and **overall system throughput (STP)**, respectively, compared to the traditional **Multi-Process Service (MPS)**-based co-running solution. When under TLB attack, TLB-pilot also provides up to 47.3% and 64.3% improvement (41% and 42.9% on average) in ANTT and STP, respectively, compared to a state-of-the-art co-running solution for efficient scheduling of thread blocks.

2 BACKGROUND AND MOTIVATION

2.1 Background

Address translation on GPU. Although the designs of the virtual memory subsystem for commercial GPUs such as NVIDIA, AMD, and Intel are not publicly available, it is widely accepted that contemporary GPUs support TLB-based address translation [23, 32, 52]. The virtual memory subsystem translates virtual addresses at page granularity by storing the virtual-to-physical mappings in a multi-level page table that resides in GPU's global memory. For each virtual address, GPU needs to perform a page table walk, which traverses each level of the page table to locate the physical page frame. For a four-level page table, each address translation involves four global memory accesses. To reduce this nontrivial overhead, TLBs are used to cache PTEs. Due to the lockstep execution model, a single instruction in a warp can generate up to 32 address translation requests. If these requests miss the TLBs, a total of 128 memory accesses is required to complete the address translation of a single SIMT instruction. Therefore, address translation is a first-order performance concern on GPU, especially when multiple kernels are run concurrently [9].

Dissecting TLB structure of GPUs. The existing efforts perform fine-grained benchmarking to unveil the TLB structure of GPUs [23, 32, 52]. They use the following common approach. A single-threaded GPU kernel traverses a continuous memory block with a specified stride and distance, performing pointer-chasing-based data accesses. By timing the cycles of accessing the same memory addresses between two consecutive runs, we can differentiate TLB hits and misses, and consequently identify the number of TLB levels, TLB reach, and page size of each level. To eliminate the impact of data cache misses, the accessed data is minimized so that it can be contained in the L2 cache. In this way, we guarantee that any latency increases during measurements are purely incurred by TLB misses.

In multi-level TLBs, lower-level TLBs are often shared among multiple SMs. We use the approach presented in the work of Karnagel et al. [25] to study the last-level TLBs. First, N pages (the number of pages that fit in one last-level TLB) on SM_i (indicating the i-th SM) are accessed. Second, another set of N pages on SM_j are accessed. Then, the first N pages are accessed again on SM_i, and a low cycle count means no sharing between SM_i and SM_j, whereas a high cycle count indicates TLB sharing (because SM_j evicts the entries loaded by SM_i). By exhaustively testing all possible combinations of SMs, we can obtain the overall TLB sharing structure. Table 1 shows the benchmarking results on GTX 1080 Ti (the Pascal architecture). The 28 SMs on this GPU are

Table 1. TLBs on GTX 1080 Ti

L1 TLB	PTE size	2 MB				
	TLB reach	32 MB				
	Miss cost	9 cycles				
L2 TLB	PTE size	32 MB				
	TLB reach	2048 MB				
	Miss cost	110 cycles				
L2 TLB topology (SM IDs)	Group 1	0	6	12	18	24
	Group 2	1	7	13	19	25
	Group 3	2	8	14	20	26
	Group 4	3	9	15	21	27
	Group 5	4	10	16	22	
	Group 6	5	11	17	23	

divided into six groups, with the first four groups each containing 5 SMs and the last two groups each containing 4 SMs, and each group shares one last-level (L2) TLB.

Co-running GPU kernels. MPS and stream are techniques to co-run GPU kernels on CUDA. MPS was introduced in CUDA 7 and can co-run kernels from different processes; stream was introduced in CUDA 1 and can co-run kernels from the same process. Unless indicated otherwise, we use MPS to co-run user kernel and attack kernel.

2.2 Motivation

Threat model. In a public cloud, cloud vendors share GPU between users to improve resource utilization. For example, the Amazon Cloud provides the Amazon Elastic Graphics service [2] based on kernel co-running to provide low-cost services to users. A user can launch a instance (a virtual machine) to use GPU. The instance leverages "API-forwarding" by intercepting OpenGL calls [3] (CUDA has a similar API-forwarding solution named *rCUDA* [42]) and sending them to remote physical GPU. In this way, each instance has its own virtual GPU. When the adversary and the benign user share a same physical GPU, the adversary can construct an instance that deploys a TLB attack kernel depicted in Section 3 to attack benign kernels. The existing work [25] shows a real incident. Data processing in a large-scale database encounters dramatic performance degradation (13×) when running workloads with irregular hash table accesses and random sampling on the same GPU. In conclusion, the TLB contention attack is feasible and hazardous in real GPU-sharing environments.

TLB attacks based on kernel co-running can become serious and hence deserve more attention for four reasons. First, such an attack is easy to deploy. Launching the attack is as easy as launching a regular instance. The attack easily achieves Denial-of-Service. Second, such an attack can be easily combined with other attacks to corrupt privacy. For example, the overflow detection based on the canary mechanism must rapidly scan all allocated memories to verify canaries [17]. The TLB attack slows down the scan (overflow detection). The slow scan allows the overflow attack to escape detection because the attack has more time to recover canaries between two adjacent scans to disable the canary mechanism. The TLB attack can also degrade application performance to help the adversary build a more stable covert channel to strike attacks [33]. Third, all GPU programs, especially those with a large memory footprint and random memory accesses (e.g., decision tree references), are subject to the threat of TLB attacks, because each program must frequently use TLB for address translation. Fourth, detecting TLB attacks on GPU is difficult.

To detect traditional Denial-of-Service attacks, the detection monitors resource consumption of each program and kills the program when its resource consumption is large [38]. However, the TLB attacks on GPU consume little resources, which makes it harder to detect. Other existing works [14, 22, 57] on CPU introduce extra operations into TLB, which also stalls GPU threads and causes significant performance degradation.

All GPUs (including the most recent ones, e.g., Kepler, Pascal, Volta, Turing, and Ampere architectures) can suffer from those attacks, because each of them has multiple SM groups in a single GPU, and within each group, multiple SMs share L2 TLB. CUDA MPS on recent Post-Volta GPUs only provides isolated virtual address space but still shares TLB between SMs and hence suffers from the TLB attacks as well.

There are a few existing works defending against TLB attacks on CPU [16]. However, they cannot be applied to GPU. The existing work uses static TLB partition and random-fill TLB (i.e., randomly replacing TLB entries). Both techniques can cause significant TLB misses on GPU with massive thread-level parallelism, which loses performance. In conclusion, no existing work can defend against the TLB attack on GPU. We must have a mechanism to effectively address it.

3 TLB ATTACK ON GPU

Threads can be stalled due to TLB misses. Because of the lockstep execution model of GPUs, the stall of one thread can block all other threads in the same warp, causing significant performance degradation. The TLB contention can happen if multiple kernels are co-located on a group of SMs that share the same L2 TLB. Based on this observation, a malicious adversary can leverage the preceding microarchitecture feature to mount performance degradation attacks on benign kernels, which is discussed in the following.

Basic ideas of TLB attack. We present an attack kernel that continuously evicts L2 TLB entries to intentionally interfere with co-running kernels on GPU. The attack kernel allocates a number of small memory blocks that belong to different PTEs and are not consecutive. The attack kernel then launches threads to concurrently access these memory blocks to effectively evict PTEs. When a benign kernel co-runs with the attack kernel, address translation requests of the benign kernel miss TLB. Finally, many of these requests are queued to be serviced by a page table walker, which significantly degrades performance of the benign kernel. L1 TLB can be exploited in the same way, but its performance influence is negligible, because the 9-cycle L1 TLB miss penalty is relatively small compared to the 110-cycle L2 TLB miss penalty [25]. Therefore, we focus on an attack implementation based on L2 TLB contention in the following discussions. Figure 1 depicts the algorithm of the attack.

Implementation details. The GPU used in our experiments is GTX 1080 Ti. The reach of each L2 TLB on this GPU is 2 GB. The reach of each PTE is 32 MB, and each L2 TLB consists of 64 PTEs. The attack kernel allocates a number of memory blocks that are small and scattered sparsely in the virtual address space, which guarantees the effectiveness of evicting PTEs in L2 TLBs (lines 19–27 in Figure 1). In particular, the attack kernel allocates one 2-MB memory block (named the *attack memory block*) and 15 other 2-MB memory blocks (named the *padding memory blocks*) in tandem. These 16 memory blocks (32 MB) form a *group* and use one PTE. By repeating the preceding process and releasing all the padding memory blocks at last, we obtain a set of attack memory blocks, belonging to different pages (`d_arr` in Figure 1).

To cover all PTEs in an L2 TLB (reaching 2-GB address space), we need to allocate 64 groups (2 GB / 32 MB = 64). The size of all attack memory blocks is only 128 MB (i.e., 64 * 2 MB). Note that the size of each attack memory block is the same as that of each padding memory block (2 MB). This allows the attack kernel to avoid the impact of randomized address allocation introduced in recent GPU drivers. With the randomized address allocation, memory blocks of the same size and

```
1  __global__ void tlb_attack(uint32_t **d_arr,
2                  uint32_t steps, uint32_t memBlk) {
3    uint32_t j = 0;
4    uint32_t w = threadIdx.x % memBlk;
5    for (uint32_t k = 0; k < steps; k++) {
6      asm("ld.global.cg.s32 %0, [%1];"
7       : "=r"(j) : "l"(d_arr[w] + j));
8      asm volatile("membar.cta;");
9      w = (w + 1) % memBlk;
10   }
11 }
12
13 void run(uint32_t evict_tlbsize) {
14   uint32_t memBlk = evict_tlbsize / 32;
15   uint32_t *t_arr[memBlk], *tmp[15 * memBlk];
16   uint32_t **d_arr; uint32_t blockSize = 2 * 1024 * 256;
17   uint32_t blockByteNum = sizeof(uint32_t) * blockSize;
18   cudaMalloc((void **)&d_arr, sizeof(uint32_t *) * memBlk);
19   for (uint32_t i = 0; i < memBlk; i++) {
20     cudaMalloc((void **)&(t_arr[i]), blockByteNum);
21     for (uint32_t k = 0; k < 15; k++) {
22       cudaMalloc((void **)&tmp[i * 15 + k],
23       blockByteNum);
24     }
25     cudaMemset(t_arr[i], 0, blockByteNum);
26   }
27   releaseMem(tmp); // free padding memory
28   cudaMemcpy(d_arr, t_arr, sizeof(uint32_t *) * memBlk,
29   cudaMemcpyHostToDevice);
30   uint32_t SMcount = getSMcount();
31   uint32_t threadnum = 128; uint32_t steps = 81920;
32   tlb_attack<<<SMcount, threadnum>>>(d_arr, steps, memBlk);
33   ... // release resource
34 }
```

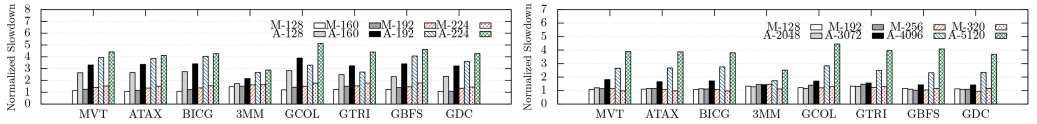
Fig. 1. Code snippet of the attack kernel.

continuous allocation tend to be allocated into the same memory pages. By allocating a specific number (15) of padding memory blocks of the same size between two attack memory blocks, the two attack memory blocks are assigned to two different PTEs. Using this method, we can scatter different attack memory blocks into different PTEs. Figure 3 further depicts the memory allocation.

Using the preceding method, the attack kernel is very lightweight and consumes very few GPU resources. Hence, it is highly likely to be able to co-run the attack kernel with another kernel in GPU-sharing environments.

Besides the preceding method, we use the following techniques to maximize attacks. First, the attack kernel launches n thread blocks to perform TLB eviction, where n is the number of SMs in GPU. The reason is as follows. There is TLB isolation between SM groups. To achieve effective attack, all SM groups should be covered. Since the GPU hardware scheduler uses a loose round-robin strategy [33] to schedule thread blocks to SMs, launching n thread blocks can roughly make each SM own a thread block, and can cover all SM groups to perform TLB attack. Second, PTE evictions are performed concurrently by all threads. Each thread repeatedly accesses all attack memory blocks (lines 1–11 in Figure 1) to maximize attacks. Third, we use special PTX assembly to access memory (lines 6 and 7 in Figure 1). This is used to bypass the L1 cache, which is virtually indexed and does not use TLB. Fourth, we avoid compiler optimization by adding line 8, because the compiler optimization can change the order of memory accesses and degrade the effectiveness of attack.

Performance under attack. We study the performance degradation of benign user kernels when co-running them with the attack kernel. We use those benchmarks listed in Table 3 as user kernels. For each user kernel, we use MPS to co-run it with the attack kernel. We also develop a "malfunctioned" attack kernel. This kernel is the same as the attack kernel, except it evicts much



(a) Performance of user kernels under attack with various numbers of threads per thread block

(b) Performance of user kernels under attack with various memory sizes reached by evicted PTEs

Fig. 2. Normalized execution time of user kernels under the attack of the attack kernel and malfunctioned attack kernel. The numbers of threads per thread block and evicted PTEs vary in this study. The prefixes A- and M- indicate the cases where the attack and malfunctioned kernels co-run with user kernels, respectively. The suffix numbers represent the number of threads per thread blocks or memory size reached by evicted PTEs. For example, in Figure 2(b), the memory size reached by evicted PTEs varies from 128 to 320 MB for the malfunctioned attack kernel and from 2 to 5 GB for the attack kernel. In Figure 2(a), the number of threads for the attack kernel and malfunctioned attack kernel varies from 128 to 224.

fewer PTEs of TLB. In particular, the malfunctioned attack kernel uses the same number of attack memory blocks as the attack kernel, but those attack memory blocks are consecutively allocated. Accessing them in the malfunctioned attack kernel evicts at most 10 PTEs in an L2 TLB. Comparing the performance of user kernels under the attack of the attack kernel and malfunctioned attack kernel, we can highlight the performance degradation caused by L2 TLB contention, not by other resource contention (e.g., the contention on registers). Figure 2 shows the performance of user kernels under the attack.

In our GPU, an L2 TLB has 64 PTEs (reaching 2 GB). To study association between the number of evicted PTEs and performance degradation, we fix the number of threads per thread block for the attack and malfunctioned attack as 128 and increase the number of evicted PTEs (from 2 to 5 GB) in each thread of the attack kernel (Figures 2(b)). We also change the number of threads per thread block but fix the number of evicted PTEs (reaching 4 GB) and study impacts of changing number of threads on the performance of user kernels. Figure 2 shows results (execution time) normalized by the execution time of kernels running alone.

We have two observations from Figure 2. First, the user kernels suffer from serious performance degradation. The performance degradation becomes larger when we increase the number of threads or the reach of evicted PTEs. The largest performance degradation is 3.9× (see the benchmark MVT under the attack of the attack kernel with 5-GB reach of the evicted PTEs (A-5120) in Figure 2), and the average performance degradation with such an attack is 3.4×. In the rest of the article, we use this attack kernel because it causes the largest performance loss. Second, evicting less than 64 continuous PTEs (reaching at most 2 GB) does not have little impact on the performance of user kernels. We speculate that the undisclosed TLB eviction policy may have specific optimizations for such a case, and that GPU may support a large page size [8], which alleviates L2 TLB contention to some extent.

In conclusion, L2 TLBs can be exploited to conduct a Degradation-of-Service attack on co-running kernels. In scenarios where the GPU is shared between kernels, current GPU hardware and runtime systems provide no performance isolation and cannot defend such attacks.

4 TLB-PILOT OVERVIEW

We introduce TLB-pilot to mitigate TLB attacks on GPU. TLB-pilot works for the following scenarios commonly found in data centers. Multiple user processes launch GPU kernels on their own instances and these kernels co-run on the same GPU, to improve hardware utilization and STP. We focus on co-running two kernels, because this is the most common cases [8, 9, 43, 44, 53, 54].

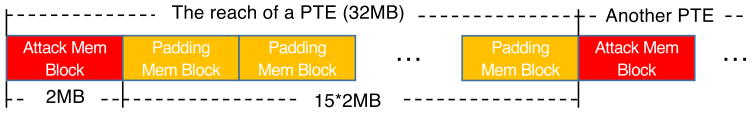


Fig. 3. Scattering attack memory blocks into different PTEs.

A user kernel can be a TLB-attacking kernel or a benign application kernel. However, the system does not have knowledge on whether a co-running user kernel is benign or malign. TLB-pilot is a system component that transforms co-running kernels before their launches (Section 6) and schedules their thread blocks on GPU at runtime (Section 6). The goal of TLB-pilot is tthreefold: (1) achieving TLB isolation to mitigate any potential TLB contention attack, (2) having ignorable runtime overhead during runtime scheduling, and (3) ensuring high STP.

The basic functionality of TLB-pilot is to assign thread blocks of a given kernel to specific SMs to mitigate potential TLB contention. Such assignment of thread blocks for a kernel is named *SM policy* in the following discussion. TLB-pilot consists of two main components: (1) a lightweight online SM dispatcher and (2) an SM binder. TLB-pilot uses a simple code transformation to implement the preceding two components (❶). Figure 4 generally depicts TLB-pilot.

SM dispatcher. The SM dispatcher collects GPU kernel information (❷ in Figure 4), and the information is used to generate an SM policy for each co-running kernel before launching those kernels (❸).

SM binder. The SM binder implements the SM policy (❹). The SM binder coordinates with the default GPU hardware scheduler to delimit on which SMs a kernel should execute. The SM binder requires no offline profiling and does not assume the availability of detailed proprietary hardware scheduling information. The SM binder uses a filling-retreat scheme to assign thread blocks of the user kernel to SMs (❻) according to the policy. The SM binder uses a kernel splitting technique to improve resource utilization (❺). This technique leverages idling SMs after the completion of a short-running kernel among co-running kernels, and removes the side effect of load imbalance introduced by the SM binder. In general, the SM binder ensures execution correctness while providing high performance.

5 TLB-PILOT DESIGN

5.1 SM Dispatcher

The SM dispatcher generates SM policies. An SM policy includes following information: (1) how to assign SMs to co-running kernels and (2) a kernel splitting plan. Using information from (1), we assign different SM groups to different co-running kernels to achieve TLB isolation. The generation of (1) and (2) is based on lightweight online analysis.

Lightweight online analysis. The SM dispatcher collects the kernel information right before kernel launch. The kernel information includes grid size, the number of threads per thread block, input data size, and the size of used shared memory. The kernel information is used to generate information from (1) (discussed in detail in the next paragraph). The SM dispatcher predicts kernel execution time to generate a kernel splitting plan. To predict kernel execution time, we use performance modeling, similar to the existing work [12, 54]. The performance model predicts kernel execution time, using the collected kernel information as the model input. In essence, the performance model builds correlation between kernel execution time and the collected kernel information through linear regression. The performance model is built offline but is employed online to make the performance prediction. It achieves an average prediction accuracy of 93.1% [12, 54], which is high. In this work, we do not aim at building a performance model with 100% accuracy.

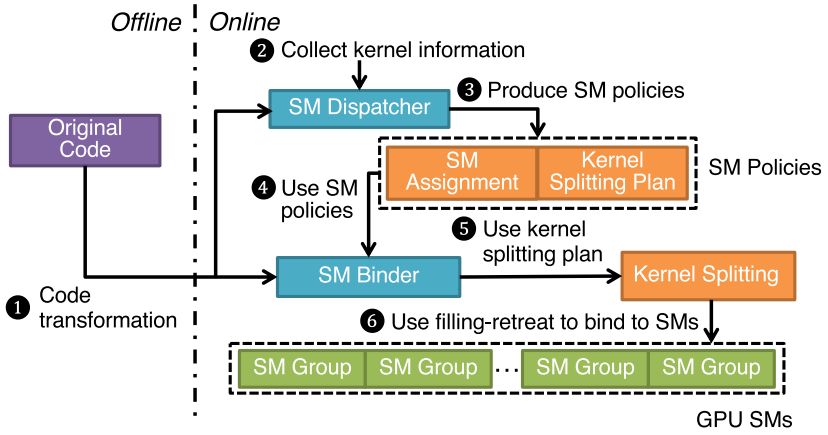


Fig. 4. A high-level overview of TLB-pilot.

Our goal is to use lightweight and reasonable models to assist TLB-pilot for kernel splitting (discussed later). TLB-pilot is highly flexible such that it can easily integrate other performance models. In Section 8.4 (kernel splitting), we show that our performance model helps TLB-pilot substantially improve the performance of co-running benchmarks despite its simplicity.

Generating plans for assigning SMs to co-running kernels. The SM dispatcher decides the assignment of SMs to kernels based on SM groups. This means that one co-running kernel uses a set of SM groups and the other co-running kernel uses another set of SM groups. The two sets have no overlap. TLB-pilot evenly assigns SM groups to co-running kernels so that the two kernels have equal opportunity to get hardware resource (SMs). In our platform (GTX 1080 Ti), for example, this means that each kernel gets two SM groups from the first four SM groups and one SM group from the last two SM groups. It is possible that even assignment of SM groups to co-running kernels causes unbalanced SM exploitation, and as a result, a co-running kernel takes longer time to finish than the other. To avoid the preceding problem and loss of STP, TLB-pilot generates a kernel splitting plan.

Generating the kernel splitting plan. To generate the kernel splitting plan, the SM dispatcher uses performance modeling to predict the execution time of co-running kernels, then picks the long-running kernel to decide its splitting. The kernel splitting plan splits the long-running kernel into parts A and B. The split is implemented by assigning different numbers of thread blocks to different parts (see the implementation details in Section 6). Part A is launched to SMs decided in the SM policy and co-runs with the short-running kernel. Part B does not have any constraint on SMs and begins to run after the short-running kernel is done. We do not constrain where B is launched, because the short-running kernel is finished and there is no TLB contention.

The SM dispatcher uses the following method to decide how to split the long-running kernel. Assume that t_A, t_B, t_sk, and t_lk are the execution time for A, B, short-running kernels, and long-running kernels, respectively. The SM dispatcher uses the following rules to split the kernel:

(1) $t_A \geq t_{sk}$,
(2) $t_A < t_{sk} + t_B$.

If the first rule is violated, then A is shorter than t_{sk}, which indicates that B can co-run with the short-running kernel. Co-running B and the short-running kernel makes the kernel splitting less effective to reduce TLB contention. If the second rule is violated, then A cannot leverage those

ALGORITHM 1: Algorithm for the SM binder.

Require: *block_counter*: the index counter of thread blocks; *SM_policy*: SM IDs allocated to the current kernel; $num\_failures$: the number of failed bindings; $max\_failures$: max number of failed bindings; *max_id*: max block ID defined in the SM policy;

Ensure: *runable*: a Boolean variable indicating the current thread block to execute or not; *block_id*: the id of a thread block;

```
1: smid = the value of SM ID register;
2: if not SM_policy[smid] then:
3:     if atomicAdd(num_failures) < max_failures then:
4:         return false;
5:     end if
6: end if
7: block_id = atomicAdd(block_counter);
8: if block_id > max_id then:
9:     return false;
10: end if
11: return true, block_id;
```

idling SMs after the completion of the short-running kernel or B, because A is constrained to specific SMs, following the SM policy.

The SM dispatcher uses the performance modeling to predict performance of various combinations of A and B, subject to the preceding two rules. Ideally, we want t_A to be close to t_sk as much as possible such that we minimize SM idleness. The SM dispatcher uses a binary search algorithm to decide A and B. In particular, assuming that the total number of thread blocks in the long-running kernel is N, the SM dispatcher assigns $N/2$ to A and then examines if the two rules are respected and A is close to t_sk. If yes, then we find A and B; if not, then we assign half of the prior number of thread blocks (i.e., $N/2$) to A. The SM dispatcher repeats the preceding process by assigning a lesser number of thread blocks to A until it finds good A and B.

5.2 SM Binder

The SM binder implements the binding between thread blocks and SM groups, based on the SM policy. Given the co-running of kernels and round-robin-based hardware scheduling, the SM binder uses an existing filling-retreating scheme [53]. To bind thread blocks with specific SMs, this scheme first increases the number of thread blocks of a kernel, which is the "filling" phase. After the kernel launches, it is up to the hardware scheduler to schedule thread blocks between SMs following the round-robin policy. Some thread blocks are scheduled to some SMs by hardware against software-defined SM policy. Those thread blocks are terminated, which is the "retreating" phase. The filling-retreating scheme is applied to the short-running kernel and part A of the long running kernel. Part B, which runs alone in most cases, does not need this scheme to bind to SM groups.. Next, we discuss how the SM binder applies the filling-retreating scheme.

Filling phase. In the filling phase, the runtime system must decide the number of thread blocks to increase before kernel launching. The number of thread blocks must be sufficiently large such that the SM binder can effectively enforce the SM policies, given the hardware-based scheduling. The number of thread blocks cannot be too large because of the concerns on the overhead of terminating thread blocks and the interference of scheduling those extra thread blocks on the round-robin scheduling policy. The existing work [53] uses an online hill-climbing algorithm to run thread blocks and decide how thread blocks should be co-located in SMs, to determine the number of threads blocks. However, the hill-climbing algorithm can be slow and put the execution

of many thread blocks under the TLB attack. The SM binder uses Equation (1) to decide the number of thread blocks for a kernel:

$$num\_blocks_{new} = SM_{all} \times \lceil num\_blocks_{orig}/SM_{subset} \rceil, \quad (1)$$

where num_block_{new} and num_blocks_{orig} are the number of thread blocks after and before using our filling scheme; SM_{all} is the total number of SMs on GPU; and SM_{subset} is the number of SMs specified in the SM policy.

The rationale behind Equation (1) is as follows. We assume that the GPU hardware scheduler uses a round-robin scheduling policy. *Without co-running with other kernels*, num_blocks_{orig} in a kernel are evenly distributed on SM_{all}. If we want SM_{subset} to have num_blocks_{orig} thread blocks (note that $SM_{subset} < SM_{all}$), we can increase the number of thread blocks to num_blocks_{new} such that the GPU hardware scheduler using the round-robin scheduling policy can put num_blocks_{orig} on SMs_{subset}. The preceding method is lightweight and does not need the execution of any thread block to determine the number of thread blocks for the filling phase.

Retreating phase. We use Algorithm 1 to implement the retreating. Each thread block inquires the SM ID where the thread block is running by reading the SM ID register (lines 1 and 2). If the SM ID belongs to the SM group assigned in the SM policy, the thread block continues running on the current SM; if not, the thread block terminates and releases SM.

Because the hardware-based scheduling does not strictly follow the round-robin policy, we must address two problems during the retreating phase to ensure execution correctness:

(1) The number of thread blocks assigned to the SM group is smaller than that planned in the SM policy.
(2) The number of thread blocks assigned to the SM group is larger than that planned in the SM policy.

To avoid the occurrence of the first problem, we introduce a counter (named $num\_failures$ in Algorithm 1) to count the number of terminated thread blocks because of mis-binding. If the counter value is larger than a threshold $max\_failures$, then we do not terminate thread blocks (lines 5–7). The threshold $max\_failures$ is equal to (num_block_{new} - num_block_{orig}). The threshold defines the maximum number of thread blocks we can terminate without impacting execution correctness.

The preceding method places execution correctness into a higher priority but could assign a thread block to an SM group not specified in the SM policy. However, this mis-assignment does not happen very often. In our evaluation, most benchmarks (five out of eight evaluated benchmarks) and the attack kernel do not have mis-assignment at all. Three benchmarks have mis-assignment in less than 23% of all thread blocks, but the SM policy is effectively enforced to mitigate TLB attack (see Section 8.4 for more discussion). Note that an attacker cannot leverage mis-assignment to bypass TLB-pilot by launching a large number of threads, because if so, it would be difficult to co-run it with the benign kernel (see Section 3).

To address the second problem, we add a counter to count the number of correct assignment of thread blocks. The counter is increased atomically (line 7). When the counter value reaches the user requirement on the number of thread blocks, all other thread blocks are immediately terminated themselves once they are launched by hardware.

6 TLB-PILOT IMPLEMENTATION

TLB-pilot includes a compiler tool for code transformation and a library for runtime control. The library implements the functionality of SM dispatcher, filling phase, retreating phase, and kernel splitting. The compiler tool transforms the user code (including both host-side and GPU-side code) by using the library APIs.

Table 2. APIs in TLB-pilot

	Name	Description
Host Code	*policy* dispatcher(*params x*)	Sends fours parameters (thread_num, block_num, shared_mem_num and kernel_input_data) of kernels to the SM dispatcher and returns the SM policy (*policy*).
	int new_block_num(*int orig_block_num*)	Returns the number of thread blocks for the "filling phase."
	void kernel_splitting_send(*void*)	Sends a message to the long-running kernel.
	void kernel_splitting_receive(*void*)	It blocks execution until it receive a message from a short-running kernel.
Device Code	*bool* run_or_retreat(*policy p*)	Based on the SM policy to run or retreat threads. If it returns *true*, threads continue execution on the SM whose ID is in *p.assignedID*.

Code transformation. The compiler tool is based on the Clang LibTooling library [13]. The code transformation only needs one simple pass to transform both CPU and GPU code. We discuss the implementation details as follows.

At the host side (CPU side), right before each kernel launches, two code blocks are added: one for implementing the SM dispatcher using a TLB-pilot library API, and the other for implementing the kernel splitting and calculating the number of thread blocks to implement the "filling phase" using TLB-pilot library APIs. At the kernel side, at the very beginning of the kernel, a code block implementing Algorithm 1 is added to implement the "retreating" phase. Table 2 summarizes those APIs. Figure 5 gives an example of how to deploy TLB-pilot on a benchmark (MVT) from PolyBench [20], where red lines show where the APIs are inserted.

Implementation of kernel splitting. Given a kernel, the kernel splitting code calculates the numbers of thread blocks for parts A and B (see Section 5.1). The kernel is then launched twice, using the calculated numbers of thread blocks for A and B, respectively. This kernel splitting method changes the thread block indexing but does not impact program correctness, because A and B can be assigned with non-overlapping, contiguous thread block IDs. Part A runs with the other co-running kernel (the short-running kernel) using MPS (instead of CUDA stream), because part A and the co-running kernel come from different processes. Part B immediately runs after the co-running kernel is done. A and B are assigned to two CUDA streams (instead of using MPS), because they come from the same user process. To coordinate the execution of part B and the co-running kernel (the short-running kernel), we employ a named pipe on CPU. In particular, right after the end of the short-running kernel, the user process on CPU that has launched the short-running kernel sends a message to the other process by the pipe to start B. In addition, in the rare case where A and B have to synchronize with each other, an inter-block level synchronization mechanism (e.g., a lock in global memory) can be employed.

Use scenarios. We assume that what kind of user kernels will be run in a data center are known and their source code is available such that we can build performance models and apply code transformation offline. Many use scenarios can meet the preceding assumption. For example, some common services or computation in data centers (e.g., matrix vector product and transpose based on the open source library Eigen [4], and graph traversal based on the open source library [1]) meet the preceding assumption. The existing work makes the same assumption [12, 53, 54]. If the preceding assumption cannot be met, we can use binary translation/instrumentation tools (e.g., NVBit [47] and PANOPTES [28]). These tools allow modifying the assembly code (SASS) of a GPU application without requiring recompilation. We leave this as future work.

Generality of our method. Our method is based on the foundation that the GPU architectures commonly have SM groups and TLB is not shared across SM groups. All GPU architectures (including Kepler, Pascal, Volta, Turing, and Ampere architecture) have this design because of the concerns on TLB scalability.

```
// ========= code snippet of MVT in device code =========
__global__ void MVT(policy p, bool no_binder, ...) {
  // If true, running MVT.
  if (no_binder || run_or_retreat(p)) {
    ... }
}
// ========== code snippet of MVT in host code ==========
void run_MVT(...) {
  ...
  policy p = dispatcher(params);
  uint32_t block_num = new_block_num(orig_block_num);
  //if true, splitting the kernel in MVT.
  if (p.splitting_kernel_name == "MVT") {
  //running splitted kernel MVT_A
  MVT<<<block_num, orig_thread_num, 0, stream1>>>
  (p, false, ...);
  kernel_splitting_receive(void);
  //running splitted kernel MVT_B
  MVT<<<p.MVT_B_block_num, orig_thread_num, 0, stream2>>>
  (p, true, ...); }
  else {
  MVT<<<block_num, orig_thread_num>>>(p, false, ...);
  kernel_splitting_send(void); }
  ...
}
```

Fig. 5. An example of deploying TLB-pilot in the benchmark MVT. TLB-pilot APIs are highlighted in red.

7 DISCUSSION

Defense of covert or side channels on TLB. Attackers can leverage GPU TLB to construct a covert or side channel [19, 36]. These attacks require sharing TLB between benign kernels and malicious kernels. For covert channels, a trojan kernel and a spy kernel need a shared TLB and timing characteristics (TLB miss or not) to communicate bits of information. For side channel, the attacker kernel primes and probes shared TLB to guess which memory page the benign kernel has accessed. TLB-pilot isolates TLBs across kernels, so TLB-pilot prevents these attacks as well.

Comparison between TLB-pilot and existing multitask GPU sharing. Compared with persistent thread-based approaches, TLB-pilot introduces low extra overhead. TLB-pilot assigns kernels to SMs before the kernel launches and does not suffer from synchronization overheads between CPU and GPU after kernels have launched. Compared with SM-centric scheduling, TLB-pilot employs kernel splitting to achieve a better load balance. In addition, TLB-pilot achieves high isolations and securities by assigning kernels into different SM groups.

Comparison between TLB-pilot and multi-instance GPU. The A100 GPU supports **multi-instance GPU (MIG)** capability, and it can divide a single GPU into multiple GPU partitions called *GPU instances*. Each instance has isolated L2 cache banks, memory controllers, and DRAM address busses [37]. Compared with MIG, the software-based TLB-pilot is more flexible because TLB-pilot does not require support of specified hardware (MIG is only available on NVIDIA Ampere GPU Architecture). In addition, TLB-pilot has a higher performance than MIG, because MIG's instances do not release resources (e.g., SMs) even if an instance has been finished and it causes unbalanced SM exploitation (shown in Section 5.1). TLB-pilot employs the kernel splitting to solve unbalanced SM exploitation.

8 EVALUATION

8.1 Experimental Setup

Our experiments are performed on a system with two 2.10-GHz Intel Xeon E5-2683 CPUs and an NVIDIA GeForce GTX 1080 Ti discrete GPU. The system runs Ubuntu 16.04.2 LTS with NVIDIA graphics driver version 418.87 and CUDA runtime 10.1 installed.

Table 3. Description of Benchmarks in Our Evaluation

Suite	Benchmark	Dataset	NumBlocks	NumThreads	ExeTime(ms)	Domain	Description
	Attack	Default	28	3K	73.51	Attack L2 TLB	Evicting L2 TLB.
PolyBench [20]	MVT	Default	32	8k	2.24	Linear Algebra	Matrix vector product and transpose.
	ATAX	Default	32	8K	2.27	Linear Algebra	Matrix transpose and vector multiplication.
	BICG	Default	32	8K	2.26	Linear Algebra	BICG sub kernel of bibgStab linear solver.
	3MM	Default	3K	786K	2.82	Linear Algebra	3 matrix multiplications.
GraphBIG [35]	GCOL	10k	494	491K	7.6	Graph Analytics	Coloring different vertex or edge in a graph.
	GTRI	10k	30	30K	6.43	Graph Analytics	Counting triangles in a graph.
	GBFS	10k	408	408K	7.66	Graph Traverse	Breadth-first search implementation on GPUs.
	GDC	10k	272	272K	2.14	Graph Analytics	Counting the degree in a graph.

To comprehensively assess TLB-pilot, we choose a set of benchmarks with a wide coverage of regular and irregular memory access patterns and consisting of compute-intensive and memory-intensive applications. In particular, we select eight benchmarks for evaluation, shown in Table 3. Those benchmarks are chosen from PolyBench [20] and GraphBig [35] benchmark suites. Those benchmarks are commonly deployed in production and have drawn a lot of attention from the community [43, 44]. The benchmarks from PolyBench are compute intensive; The benchmarks from GraphBIG are memory intensive and represent various graph algorithms widely deployed in the real world. Beside those benchmarks, we use the attack kernel depicted in Section 3.

In the evaluation, we compare the performance of TLB-pilot with the performance of two baselines: (1) we use a state-of-the-art approach [53] to schedule thread blocks of co-running kernels to mitigate TLB attack, and (2) we use MPS to co-run kernels without any mechanism to mitigate TLB attack. The two baselines are labeled as "SMC" and "MPS," respectively in the figures in this section. All results reported in this section are the average of 20 runs.

Besides using execution time as a performance metric, we use the following two metrics to evaluate co-run performance:

(1) *Average normalized turnaround time* [18, 39]: **Normalized turnaround time (NTT)** is defined in Equation (2) and used to quantify how responsive kernel execution is. A smaller NTT indicates a more responsive kernel execution.

$$NTT_i = T_i^{MP}/T_i^{SP}, \tag{2}$$

where T_i^{SP} and T_i^{MP} are the execution time of single-run and co-run, respectively, for the kernel i. NTT is usually greater than 1. ANTT is the average of NTTs of co-running kernels.

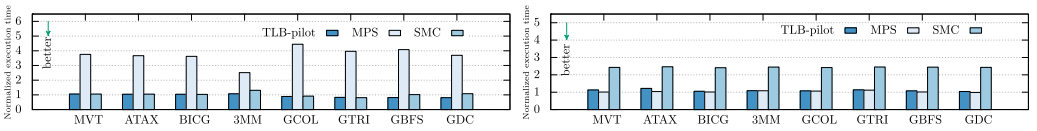
(2) *Overall system throughput* [18, 39]: STP is defined in Equation (3). It varies from 0 to n (where n is the number of co-running kernels), and a larger value of STP indicates better overall throughput.

$$STP = \sum_{i=1}^{n} T_i^{SP} \Big/ T_i^{MP} \tag{3}$$

8.2 Overall Performance with TLB Attack

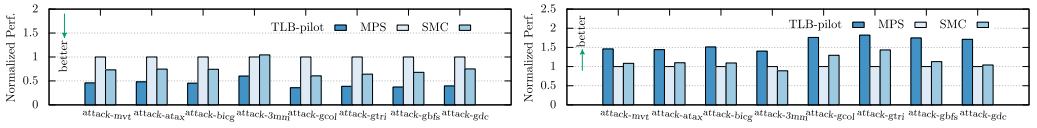
Execution time. Figure 6 shows the execution time of benign and attack kernels under the control of MPS, SMC, and TLB-pilot. The execution time is normalized by that of *stand-alone run*. Figure 6(a) shows that with MPS, the benign kernels suffer from large performance loss, which is up to 4.4× and 3.7× on average. This large performance loss comes from TLB contention introduced by the attack kernel. With SMC, the benign kernels have 3.2% performance degradation on average. With TLB-pilot, the benign kernels have 5.6% performance improvement (no degradation) on average.

We have following conclusions. SMC and TLB-pilot perform better than MPS because of their efforts for TLB isolation. TLB-pilot leads to 8.8% (5.6% + 3.2%) performance improvement over



(a) Normalized execution time of benign kernels (b) Normalized execution time of the attack kernel

Fig. 6. Normalized execution time of benign kernels co-running with the attack kernel.



(a) ANTT (b) STP

Fig. 7. ANTT and overall STP when co-running benign kernels with the attack kernel.

SMC, because TLB-pilot does not fix the number of thread blocks per SM as SMC, which allows the hardware scheduler to schedule thread blocks of benign kernels for better performance. TLB-pilot does not require expensive online profiling as does SMC, which also leads to better performance. TLB-pilot leads to performance improvement (not degradation) in some benchmarks (GCOL, GTRI, GBFS, GDC), because TLB-pilot constrains thread blocks of the benign kernels to a subset of all SMs, which improves data locality and encourages data sharing between thread blocks.

Figure 6(b) shows the performance of the attack kernel. We study the performance of the attack kernel because in a production environment, it is typically unknown whether the co-running kernel is benign or not. TLB-pilot must ensure that the co-running kernel (the attack kernel in this case) does not have performance loss caused by the software-based scheduling mechanism.

Figure 6(b) shows that the performance loss of the attack kernel is 10.1%, 4%, and 143% with TLB-pilot, MPS, and SMC, respectively. TLB-pilot shows great performance advantage over SMC because of the kernel splitting method. The attack kernel is a long-running kernel, and the kernel splitting method allows it to make best use of idling SMs after the benign kernel is done. MPS provides the similar performance advantage over SMC because MPS effectively uses the hardware-based scheduling to use idling SMs. The performance loss of TLB-pilot is slightly larger than that of MPS because of the software overhead in TLB-pilot. Note that although MPS provides better performance in the attack kernel than TLB-pilot, MPS provides much worse performance in the benign kernel than TLB-pilot.

ANTT and STP. Figure 7 shows performance in terms of ANTT and STP. Results are normalized by those with MPS. Compared with MPS, Figure 7(a) shows that SMC and TLB-pilot reduce ANTT by 25.8% and 56.2% on average, respectively, and Figure 7(b) shows that SMC and TLB-pilot improve STP by 13.3% and 60.6% on average, respectively. TLB-pilot performs best in all cases. TLB-pilot performs better than SMC because of kernel splitting to make best use of idling SMs to improve STP. TLB-pilot performs better than MPS because of TLB isolation. The SMC approach has an obvious degradation in both ANTT and STP for benchmarks 3MM, GCOL, GBFS, and GDC. Those benchmarks use relatively large numbers of thread blocks, and using a fixed number of thread blocks per SM in SMC prevents optimization from the hardware scheduler.

8.3 Co-Run of Benign Kernels

We study the performance of co-running benign kernels with MPS, SMC, and TLB-pilot. Given eight benign kernels, there are 28 cases of two co-running kernels. All results are normalized by that of MPS.

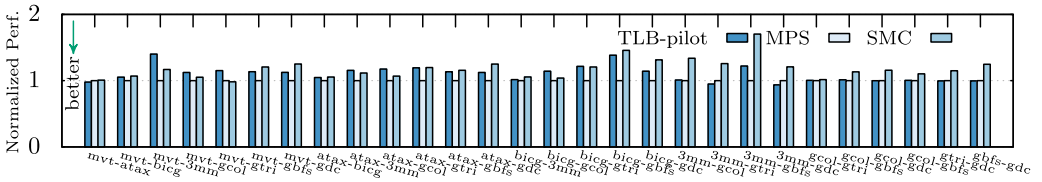


Fig. 8. ANTT of co-running benign kernels.

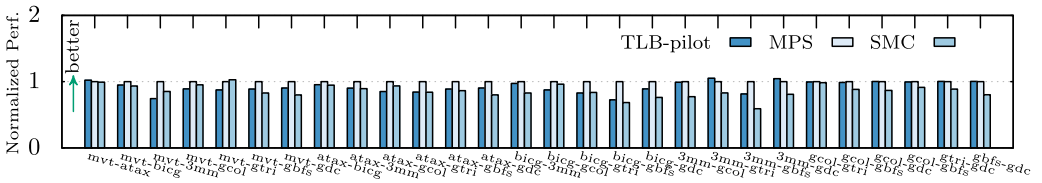


Fig. 9. STP of co-running benign kernels.

Figure 8 shows the ANTT results. Compared with MPS, SMC and TLB-pilot introduce 17.7% and 10.1% degradation on average, respectively. For the benchmark 3mm-gbfs, TLB-pilot has a rather large performance advantage (48.2%) over SMC, because SMC launches massive thread blocks during the filling-phase, which degrades performance. Figure 9 shows the STP results. SMC and TLB-pilot introduce 14% and 7.8% degradation on average, respectively.

In conclusion, TLB-pilot has better performance than SMC but performs slightly worse than MPS. This is because SMC and TLB-pilot are based on MPS to achieve co-running, and the software-based scheduling in SMC and TLB-pilot may impact the effectiveness of hardware-based scheduling, causing the preceding overhead.

8.4 Performance Breakdown Analysis

Effectiveness of kernel splitting. We use the four benchmarks from the PolyBench benchmark suite to evaluate the effectiveness of kernel splitting (the GraphBIG benchmark suite has similar results). We co-run each of them with the attack kernel or co-run any two of them, with and without kernel splitting (in total, we have 10 cases). Figure 10 shows performance in terms of STP. We show STP because kernel splitting is supposed to impact STP. The results in the figure are normalized by those of using MPS. The figure shows that compared with MPS, TLB-pilot with kernel splitting has 8.5% improvement (no degradation) on average and 8% degradation without kernel splitting. Co-running benchmarks attack kernel, and 3 mm with kernel splitting leads to the largest improvement (82%), because the two kernels in the benchmarks have a large difference in the execution time (50×), hence giving more room to use kernel splitting to improve STP.

Occurrence of mis-assigning thread blocks to SMs. Since the hardware-based scheduling does not strictly follow the round-robin policy, thread blocks may be assigned to an SM group not specified in the SM policy to ensure program correctness (see the retreating phase in Section 5.2). We quantify how often this mis-assignment can happen. We use a metric, the mis-assignment ratio, defined as the ratio of number of mis-assigned thread blocks to the total number of thread blocks. Figure 11 shows the mis-assignment ratio for each kernel when it co-runs with the attack kernel.

We have a couple of observations. First, the mis-assignment in most kernels (including the attack kernel) does not happen often (the ratio is 0 in six benchmarks). Second, three benchmarks (3MM, GBFS, and GDC) have a relatively high mis-assignment ratio (21.6%, 21.4%, and 22.6%, respectively). After examining the benchmarks, we find that they use 786K, 408K, and 272K threads, respectively,

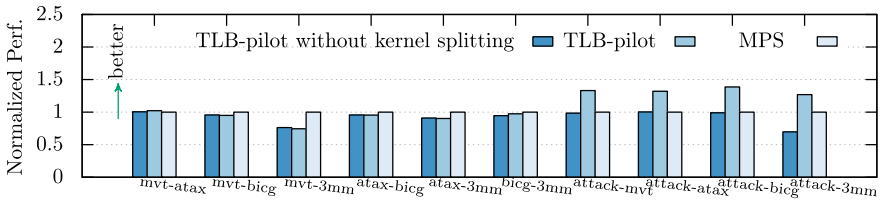


Fig. 10. Normalized STP with and without kernel splitting in TLB-pilot.

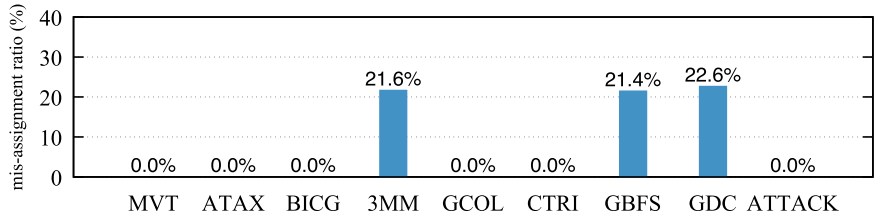


Fig. 11. The mis-assignment ratio for all benchmarks.

much larger than the number of threads our GPU can concurrently execute (56K threads). As a result, it is highly likely that the hardware scheduler runs out of threads in some SMs specified in the policy and uses other SMs to maximize STP. However, even with the mis-assignment, TLB-pilot still effectively mitigates performance impact of TLB attack. Figure 7 shows that for the three benchmarks, compared with MPS, TLB-pilot still causes large performance improvement (42.4%, 45.3%, and 47.3% in ANTT, respectively, and 57.7%, 54.6%, and 64.3% in STP, respectively). Furthermore, the performance difference between the mis-assignment case and stand-alone execution (i.e., the benchmark execution without using co-run) is less than 10% for all three benchmarks, which indicates that the performance impact of mis-assignment is small. This effectiveness is because of two reasons. First, the attack kernel is bound to specific SMs very well without mis-assignment. Second, the mis-assignment does not happen often enough to cause serious performance slowdown in benign kernels.

8.5 Overhead Analysis

TLB-pilot adds extra code to kernels. To quantify runtime overhead due to the extra code, we run each benchmark with TLB-pilot and name its execution time as T_{TLB_pilot}. The runtime overhead is defined as $(T_{TLB\_pilot}\text{-}T_{org})/T_{org}$, where T_{org} is the execution time of the original kernel. Figure 12 provides the overhead results. It shows that TLB-pilot introduces up to 0.8% (0.3% on average) overhead. In general, the runtime overhead introduced by extra code is negligible.

8.6 Security Analysis

The TLB attack causes performance degradation. Therefore, if TLB-pilot guarantees high performance of a benign kernel when co-running with the attack kernels, then TLB-pilot effectively prevents the TLB attack. The high performance means high security in the case of the TLB attack. Figures 6 and 7 show high performance of benign kernels, demonstrating high security provided by TLB-pilot.

The adversary may use the following methods to evade TLB-pilot but cannot succeed. We discuss them next.

Creating multiple attack instances. The adversary can create multiple attack instances such that an attack instance can share the same SM assignment with the benign user to disable TLB-pilot.

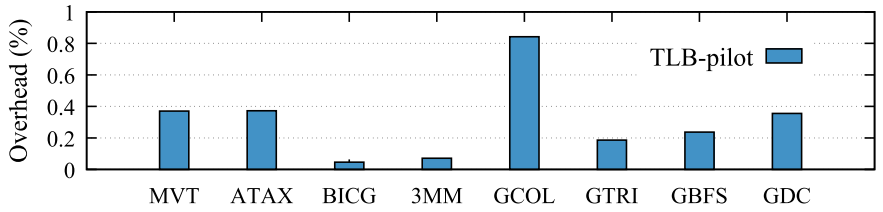


Fig. 12. The runtime overhead of introducing extra code in user kernels by TLB-pilot.

This attack must rely on detailed knowledge on how attack instances are assigned to GPUs. Such knowledge is missing in real production environments because the adversary only sees virtual GPUs and cannot know whether the attack instances are scheduled to the same physical GPU or not.

Leveraging mis-assignment to disable TLB-pilot. Mis-assignment violates the SM policy to assign a thread block to an SM group. Although mis-assignment does not happen often (shown in Figure 11), an attack kernel may leverage the mis-assignment to disable TLB-pilot. However, this attack is not feasible for the following reason. The adversary must launch a large number of threads to cause high mis-assignment to disable TLB-pilot. Figure 11 shows that the adversary must launch at least 272K threads to lead to a 20% mis-assignment ratio but still cannot disfunction TLB-pilot because of high security provided by TLB-pilot. Launching a larger number of threads, the attack kernel has difficulty in co-running with the benign kernel (Section 3) because of hardware scheduling, which makes the attack invalid.

9 RELATED WORK

Multitask GPU sharing. Time multiplexing with application preemption enables GPU sharing by running kernels from difference applications back to back [45, 49]. Elastic kernel [39] is a software approach for spatial multiplexing that requires manual slicing of GPU kernels. Warped-Slicer [55] is a mechanism for partitioning a single SM across multiple kernels in contrast to the coarser granularity of assigning a kernel to a subset of SMs. Preemptive multitasking relies on context switching, which often incurs substantial overhead due to the large context on GPUs. To address this issue, Lin et al. [31] propose a method to reduce the overhead of context switching by compressing the thread block level state. Chimera [40] proposes a collaborative preemption approach that can precisely control the overhead for GPU multitasking. Dai et al. [15] balance memory accesses of concurrent kernels and limit the number of inflight memory instructions to improve performances. Wang et al. [51] propose quality of service mechanisms for a fine-grained form of GPU sharing. However, they do not consider TLB attack.

GPU virtual memory and TLB designs. Vesely et al. [46] present a detailed analysis on virtual memory support in heterogeneous systems, revealing that address translation incurs high latency and hurts performance. Shin et al. [43, 44] address the issues of address translation (especially page table walks) in irregular GPU applications. Virtual caching [58] reduces bandwidth demands for the shared address translation hardware. Most of these works focus on integrated GPUs or a single GPU application, whereas we study TLB contention among concurrent GPU applications on discrete GPUs.

VAST [29] provides the illusion of large memory space for GPU applications using a software TLB design. Karnagel et al. [25] show that random memory accesses in database operations cause TLB contention that results in significant performance degradation. Multitask GPU sharing and TLB contention in real GPU applications inspire this work.

GPU scheduling. TimeGraph [26] is a kernel space real-time GPU scheduler for computer graphics. Gdev [27] provides a GPU scheduling scheme to virtualize GPUs, which enhances the isolation among multiple tasks. These works focus on priority-based scheduling of GPUs in real-time systems. TLB-pilot schedules thread blocks to different SMs on GPU by considering microarchitectural features.

Kernel slicing has been proposed to support preemptive scheduling [60, 62]. It splits long-running kernel into multiple short ones, each of which contains a subset of thread blocks scheduled to run in turn. Our approach is similar to kernel slicing, but the difference is that we consider how to effectively mitigate TLB attack during the kernel splitting. EffiSha [11], FLEP [54], and SMGuard [59] are three systems designed for preemption-based scheduling, based on source code transformation using a compiler-runtime framework. These systems adopt the persistent thread mechanism [21] to enable the scheduling of tasks. The persistent thread-based approach cannot be used to mitigate TLB contention because of the lack of knowledge on TLB microarchitecture.

Side and covert channels on GPUs. Nayak et al. [36] demonstrate that attackers can leverage GPU TLB to construct a covert channel, but this work does not consider how to defend against TLB-based attacks. GPUGuard [56] designs a decision tree based detection and a hierarchical defense framework to close the covert channels. However, GPUGuard's security domains is based on temporal partitioning, so it cannot prevent malicious kernels from evicting PTEs of benign kernels and the TLB-contention attack in our article still works.

10 CONCLUSION

Co-running GPU kernels on a single GPU brings new challenges on application securities. In this article, we reveal how TLB attack can happen on GPU with co-running kernels. This new attack can cause up to 3.9× performance slowdown. We discuss how this attack leveraging GPU microarchitecture and hardware scheduling can happen. We introduce a software-based solution named *TLB-pilot* to mitigate the TLB attack without modification of hardware. TLB-pilot coordinates with application-agnostic, hardware-based scheduling to bind thread blocks of a kernel with specific SMs to enable TLB isolation. The result shows that TLB-pilot mitigates TLB attack. When under TLB attack, TLB-pilot provides large performance improvement over the traditional MPS-based co-running solution and a state-of-the-art co-running solution for efficient scheduling of thread blocks.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers for their helpful feedback.

REFERENCES

[1] Apache. 2019. TinkerPop. Retrieved November 3, 2021 from http://tinkerpop.apache.org.
[2] AWS. 2020. Amazon Elastic Graphics. Retrieved November 3, 2021 from https://docs.aws.amazon.com/AWSEC2/latest/WindowsGuide/elastic-graphics.html.
[3] AWS. 2020. Amazon Elastic Graphics Features. Retrieved November 3, 2021 from https://aws.amazon.com/ec2/elastic-graphics/features/.
[4] GitLab. 2020. Eigen. Retrieved November 3, 2021 from https://gitlab.com/libeigen/eigen.
[5] GitHub. 2020. TLB-pilot. Retrieved November 3, 2021 from https://github.com/qzpm7193/TLB_pilot.
[6] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. 2012. The case for GPGPU spatial multitasking. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA'12)*. IEEE, Los Alamitos, CA, 1–12. https://doi.org/10.1109/HPCA.2012.6168946
[7] Tyler Allen, Xizhou Feng, and Rong Ge. 2019. Slate: Enabling workload-aware efficient multiprocessing for modern GPGPUs. In *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS'19)*. 252–261.

[8] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu. 2017. Mosaic: A GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. ACM, New York, NY, 136–150. https://doi.org/10.1145/3123939.3123975

[9] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu. 2018. MASK: Redesigning the GPU memory hierarchy to support multi-application concurrency. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. ACM, New York, NY, 503–518. https://doi.org/10.1145/3173162.3173169

[10] Can Basaran and Kyoung-Don Kang. 2012. Supporting preemptive task executions and memory copies in GPGPUs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS'12)*. 287–296.

[11] G. Chen, Y. Zhao, X. Shen, and H. Zhou. 2017. EffiSha: A software framework for enabling effficient preemptive scheduling of GPU. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'17)*. ACM, New York, NY, 3–16. https://doi.org/10.1145/3018743.3018748

[12] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. 681–696.

[13] Clang. 2020. Clang: A C Language Family Frontend for LLVM. Retrieved November 3, 2021 from https://clang.llvm.org.

[14] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security'16)*. 857–874.

[15] Hongwen Dai, Zhen Lin, Chao Li, Chen Zhao, Fei Wang, Nanning Zheng, and Huiyang Zhou. 2018. Accelerate GPU concurrent kernel execution by mitigating memory pipeline stalls. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE, Los Alamitos, CA, 208–220. https://doi.org/10.1109/HPCA.2018.00027

[16] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2019. Secure TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*. 346–259.

[17] B. Di, J. Sun, D. Li, H. Chen, and Z. Quan. 2018. GMOD: A dynamic GPU memory overflow detector. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT'18)*. ACM, New York, NY, Article 20, 13 pages. https://doi.org/10.1145/3243176.3243194

[18] Stijn Eyerman and Lieven Eeckhout. 2008. System-level performance metrics for multiprogram workloads. *IEEE Micro* 28, 3 (2008), 42–53.

[19] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. 955–972.

[20] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Proceedings of 2012 Innovative Parallel Computing (InPar'12)*. 1–10.

[21] K. Gupta, J. A. Stuart, and J. D. Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *Proceedings of 2012 Innovative Parallel Computing (InPar'12)*. 1–14. https://doi.org/10.1109/InPar.2012.6339596

[22] Intel. 2020. Intel SGX. Retrieved November 3, 2021 from https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.

[23] Z. Jia, M. Maggioni, B. Staiger, and D. P Scarpazza. 2018. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).

[24] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das. 2015. Anatomy of GPU memory system for multi-application execution. In *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS'15)*. ACM, New York, NY, 223–234. https://doi.org/10.1145/2818950.2818979

[25] T. Karnagel, T. Ben-Nun, M. Werner, D. Habich, and W. Lehner. 2017. Big data causing big (TLB) problems: Taming random memory accesses on the GPU. In *Proceedings of the 13th International Workshop on Data Management on New Hardware (DAMON'17)*. ACM, New York, NY, Article 6, 10 pages. https://doi.org/10.1145/3076113.3076115

[26] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. 2011. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC'11)*.

[27] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. 2012. Gdev: First-class GPU resource management in the operating system. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*.

[28] Chris Kennelly. 2012. Panoptes: A Binary Translation Framework for CUDA. Retrieved November 3, 2021 from http://on-demand.gputechconf.com/gtc/2012/presentations/S0078-Panoptes-Binary-Instrumentation-Framework-for-CUDA.pdf.

[29] J. Lee, M. Samadi, and S. Mahlke. 2014. VAST: The illusion of a large memory space for GPUs. In *Proceedings of the 2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT'14)*. 443–454. https://doi.org/10.1145/2628071.2628075

[30] A. Li, G.-J. van den Braak, A. Kumar, and H. Corporaal. 2015. Adaptive and transparent cache bypassing for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'15)*. ACM, New York, NY, Article 17, 12 pages. https://doi.org/10.1145/2807591.2807606

[31] Z. Lin, L. Nyland, and H. Zhou. 2016. Enabling efficient preemption for SIMT architectures with lightweight context switching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'16)*. IEEE, Los Alamitos, CA, Article 77, 11 pages. http://dl.acm.org/citation.cfm?id=3014904.3015007.

[32] X. Mei and X. Chu. 2017. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (Jan. 2017), 72–86. https://doi.org/10.1109/TPDS.2016.2549523

[33] Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael B. Abu-Ghazaleh. 2017. Constructing and characterizing covert channels on GPGPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. ACM, New York, NY, 354–366. https://doi.org/10.1145/3123939.3124538

[34] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael B. Abu-Ghazaleh. 2018. Rendered insecure: GPU side channel attacks are practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. 2139–2153. https://doi.org/10.1145/3243734.3243831

[35] Lifeng Nai, Yinglong Xia, Ilie Gabriel Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'15)*. ACM, New York, NY, Article 69, 12 pages. https://doi.org/10.1145/2807591.2807626

[36] Ajay Nayak, B. Pratheek, Vinod Ganapathy, and Arkaprava Basu. 2021. (Mis)managed: A novel TLB-based covert channel on GPUs. In *ASIA CCS'21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*, Jiannong Cao, Man Ho Au, Zhiqiang Lin, and Moti Yung (Eds.). ACM, New York, NY, 872–885. https://doi.org/10.1145/3433210.3453077

[37] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. Retrieved November 3, 2021 from https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[38] Oracle. 2020. How to Configure the Linux Out-of-Memory Killer. Retrieved November 3, 2021 from https://www.oracle.com/technical-resources/articles/it-infrastructure/dev-oom-killer.html.

[39] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, NY, 407–418. https://doi.org/10.1145/2451116.2451160

[40] J. J. K. Park, Y. Park, and S. Mahlke. 2015. Chimera: Collaborative preemption for multitasking on a shared GPU. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 593–606. https://doi.org/10.1145/2694344.2694346

[41] J. J. K. Park, Y. Park, and S. Mahlke. 2017. Dynamic resource management for efficient utilization of multitasking GPUs. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. ACM, New York, NY, 527–540. https://doi.org/10.1145/3037697.3037707

[42] rCUDA. 2018. Using Remote GPUs with rCUDA. Retrieved November 3, 2021 from http://www.rcuda.net/pub/rCUDA_isc18.pdf.

[43] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu. 2018. Scheduling page table walks for irregular GPU applications. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE, Los Alamitos, CA, 180–192. https://doi.org/10.1109/ISCA.2018.00025

[44] S. Shin, M. LeBeane, Y. Solihin, and A. Basu. 2018. Neighborhood-aware address translation for irregular GPU applications. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. 352–363. https://doi.org/10.1109/MICRO.2018.00036

[45] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. 2014. Enabling preemptive multiprogramming on GPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. IEEE, Los Alamitos, CA, 193–204. http://dl.acm.org/citation.cfm?id=2665671.2665702.

[46] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee. 2016. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *Proceedings of the 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'16)*. 161–171.

[47] Oreste Villa, Mark Stephenson, David W. Nellans, and Stephen W. Keckler. 2019. NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. 372–383.

[48] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. 2421–2434.

[49] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. 2016. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 358–369. https://doi.org/10.1109/HPCA.2016.7446078

[50] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. 2017. Quality of service support for fine-grained sharing on GPUs. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, NY, 269–281. https://doi.org/10.1145/3079856.3080203

[51] Zhenning Wang, Jun Yang, Rami G. Melhem, Bruce R. Childers, Youtao Zhang, and Minyi Guo. 2017. Quality of service support for fine-grained sharing on GPUs. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. ACM, New York, NY, 269–281. https://doi.org/10.1145/3079856.3080203

[52] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software (ISPASS'10)*. 235–246. https://doi.org/10.1109/ISPASS.2010.5452013

[53] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey S. Vetter. 2015. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the 29th ACM International Conference on Supercomputing (ICS'15)*. 119–130. https://doi.org/10.1145/2751205.2751213

[54] B. Wu, X. Liu, X. Zhou, and C. Jiang. 2017. FLEP: Enabling flexible and efficient preemption on GPUs. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. ACM, New York, NY, 483–496. https://doi.org/10.1145/3037697.3037742

[55] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. 2016. Warped-Slicer: Efficient intra-SM slicing through dynamic resource partitioning for GPU multiprogramming. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. IEEE, Los Alamitos, CA, 230–242. https://doi.org/10.1109/ISCA.2016.29

[56] Qiumin Xu, Hoda Naghibijouybari, Shibo Wang, Nael B. Abu-Ghazaleh, and Murali Annavaram. 2019. GPUGuard: Mitigating contention based side and covert channel attacks on GPUs. In *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019*, Rudolf Eigenmann, Chen Ding, and Sally A. McKee (Eds.). ACM, New York, NY, 497–509. https://doi.org/10.1145/3330345.3330389

[57] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2019. InvisiSpec: Making speculative execution invisible in the cache hierarchy (corrigendum). In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. 1076.

[58] H. Yoon, J. Lowe-Power, and G. S. Sohi. 2018. Filtering translation bandwidth with virtual caching. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. ACM, New York, NY, 113–127. https://doi.org/10.1145/3173162.3173195

[59] C. Yu, Y. Bai, H. Yang, K. Cheng, Y. Gu, Z. Luan, and D. Qian. 2018. SMGuard: A flexible and fine-grained resource management framework for GPUs. *IEEE Transactions on Parallel and Distributed Systems* 29, 12 (Dec. 2018), 2849–2862. https://doi.org/10.1109/TPDS.2018.2848621

[60] J. Zhong and B. He. 2014. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (June 2014), 1522–1532. https://doi.org/10.1109/TPDS.2013.257

[61] Husheng Zhou, Guangmo Tong, and Cong Liu. 2015. GPES: A preemptive execution system for GPGPU computing. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 87–97.

[62] H. Zhou, G. Tong, and C. Liu. 2015. GPES: A preemptive execution system for GPGPU computing. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 87–97. https://doi.org/10.1109/RTAS.2015.7108420

Received April 2021; revised September 2021; accepted October 2021