# Making Powerful Enemies on NVIDIA GPUs

Tyler Yandrofski, Jingyuan Chen, Nathan Otterness, James H. Anderson and F. D. Smith
Univerisity of North Carolina, Chapel Hill, United States
tylerdy@cs.unc.edu, leochanj@cs.unc.edu, otternes@cs.unc.edu, anderson@cs.unc.edu, smithfd@cs.unc.edu

***Abstract*—Graphics Processing Units (GPUs) are widely used in safety-critical real-time systems such as autonomous vehicles due to their high performance on artificial intelligence (AI) workloads. As the computing power of recent GPUs keeps growing, it becomes increasingly possible to allow multiple independent programs to access the GPU concurrently. This complicates timing analysis, as contention for shared GPU resources renders execution times less predictable and worst-case execution times (WCETs) difficult to estimate. This paper provides a method for producing *enemy programs* that intentionally contend for GPU resources in order to enable more confident measurement-based WCET estimations. This paper provides an experiment-driven method to design effective enemy programs for several different *interference channels*—specific shared resources within the GPU through which concurrent computations may impact others' execution times. The method is flexible and can be applied to different GPU sharing mechanisms. The enemies are evaluated against a large number of real GPU applications, and the results indicate that these enemies cause higher slowdowns for GPU tasks than other baseline resource-stressing methods.**

***Index Terms*—real-time systems, graphics processing units, measurement-based timing analysis, interference channels**

## I. Introduction

With advances in artificial intelligence (AI) and computer vision (CV), safety-critical real-time systems such as autonomous vehicles increasingly depend on graphics processing units (GPUs) to accelerate highly parallel workloads. As GPUs grow more powerful, more opportunities arise for scheduling multiple computations (called *kernels*) concurrently on one device [1]. However, in real-time systems, concurrent GPU kernels complicate timing analysis due to the presence of *interference channels*.[1] Interference channels are hardware resources such as compute units, cache, and registers that may be shared by more than one concurrent kernel, allowing one kernel to impact the timing properties of another.

Avoiding or accounting for interference channels is a problem for any real-time system running on sufficiently complex hardware. For example, *timing analysis*, which seeks to establish worst-case execution times (WCETs), is incredibly difficult even for multicore CPUs. Establishing a formally "guaranteed" worst-case execution time for real-time tasks running on a multicore system is nearly impossible, with all known methods either being highly pessimistic or requiring adherence to strict restrictions such as disabling processor cores or caches. Instead, there seems to be growing consensus that formal guarantees may need to be abandoned in favor of thorough *measurement-based* approaches, which seek to statistically estimate WCETs by sampling many execution times running on the hardware in question.

**Timing analysis on GPUs.** Unfortunately, problems facing timing analysis on multicore CPUs are often more severe on GPUs, due to even more extensive parallelism and the lack of detailed hardware specifications [3]. Some research proposes *static* timing analysis for GPUs, which attempts to analyze program code in order to compute WCETs offline, but these approaches are often overly pessimistic or limited in the size or complexity of compatible programs [4–7]. Measurement-based analysis has been used to estimate WCETs for individual GPU kernels [3, 8, 9], but these methods do not consider interference from concurrent GPU tasks.

Given modern GPUs' ever-growing capabilities for concurrent usage, we contend that *timing analysis on GPUs must account for interference channels*. Some prior works employ *resource partitioning* to attempt to reduce or remove the impact of interference [10–12], but these works require complicated reverse engineering, or otherwise were not intended to measure the impacts of resource sharing.

In this paper, we explore a different approach: we address the need for accurate measurement-based timing analysis on GPUs by presenting a methodology for stressing a GPU program, following the *enemy program* approach proposed for multicore CPUs [13]. Our focus on interference channels makes our goal with respect to an enemy program clear: it must be tailored to a specific *victim program* to maximize stress on its most sensitive interference channels. A reliable means of creating stress affords greater confidence in any measured WCET estimate for the given victim. To our knowledge, ours is the first work to discuss methods for generating interference between arbitrary, concurrent GPU kernels.

**Contributions.** Our contributions are twofold. First, we introduce an experiment-driven methodology for designing enemy programs that approximate worst-case interference for arbitrary GPU workloads. We identify a wide range of GPU interference channels and define empirical techniques for configuring enemies to stress them. Our study focuses on NVIDIA GPUs, a choice that is justified by their market dominance.

Second, we provide measurements of the execution times of a wide range of real GPU benchmarks when executed concurrently with our enemies. As opposed to simply designating a small number of basic categories such as "compute-bound"

Supported by NSF grants CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

[1]This term is commonly used in certification requirements such as the FAA certification authority document CAST32-A [2] where it means "a platform property that may cause interference between independent applications."

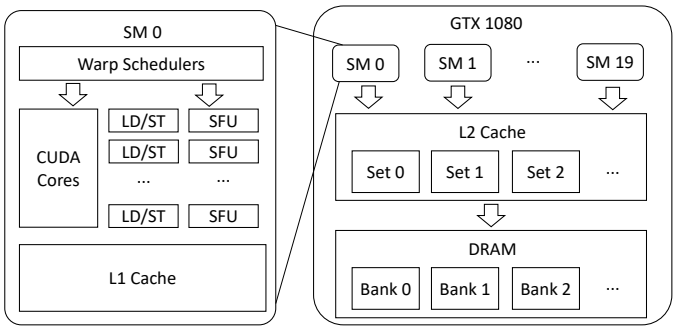


Fig. 1. Organization of computational hardware in the NVIDIA GTX 1080.

or "memory-bound," we identify the sensitivity of each task to a collection of interference channels.

**Organization.** In this rest of this paper, we provide relevant background (Sec. II), describe the methodology with which we design experiments for stressing various interference channels (Sec. III), present our specific experiments and their results (Sec. IV), evaluate and compare our method to existing baselines (Sec. V), discuss limitations and future work (Sec. VI), and conclude (Sec. VII).

## II. Background

In providing the needed background, we begin by providing an overview of basic concepts relevant to NVIDIA GPUs.

**CUDA basics.** *CUDA* is the C/C++ programming interface for applications written for NVIDIA GPUs.[2] As mentioned already, code that executes on the GPU is called a *kernel*—CUDA allows programmers to write kernels, and to specify the number of parallel GPU *threads* used when executing a kernel. GPU threads are grouped into *blocks*, and, in order to launch a kernel, CUDA requires programs to specify the number of threads per block and the total number of blocks. In addition, each kernel is associated with a CUDA *context*, which contains state information needed when using the GPU. Typically, the CUDA runtime creates one such context for each CPU process, and kernels from different contexts will not execute on the GPU concurrently.

**GPU hardware.** Our work uses the NVIDIA GTX 1080 GPU as a test platform. Fig. 1 shows the layout of the GPU's hardware. NVIDIA GPUs, including the GTX 1080, consist of multiple *streaming multiprocessors* (SMs), each capable of executing a large number of GPU threads. Blocks, mentioned above, are important within GPU hardware: in order to execute a kernel, the GPU assigns the kernel's blocks to SMs. Once assigned to an SM, threads within each block will only execute on the same SM until completion—threads within a single block are never divided among SMs or migrate between SMs.

While threads running on separate SMs execute independently, within a single SM threads are organized into groups of 32 threads called *warps*. All threads in the same warp execute identical instructions but on potentially different data: a model known as *single-instruction, multiple-data* (SIMD). Warps are the smallest units of scheduling on an SM. In an SM, warps are managed by one or more *warp schedulers*, as shown in Fig. 1 (our GTX 1080 has four warp schedulers per SM). In each instruction-dispatch cycle, a warp scheduler dispatches one or two instructions from a ready warp onto the SM's computing hardware. If a warp becomes not ready (*e.g.*, if it is waiting for outstanding memory requests or available functional units), it is *stalled*, and cannot be selected for execution.

The GTX 1080 has a multi-level memory hierarchy. Each SM contains its own local L1 (data[3]) cache and 96KB of *shared memory*, and all SMs share a global L2 unified cache and DRAM memory. The L1 and L2 caches contain multiple cache *lines*: consecutive sequences of bytes in memory. When a memory address is requested, the SM first visits its L1 cache. If the address falls into a line currently in the cache, there is a *cache hit*, and data at the requested address can be read or written immediately. Otherwise, there is a *cache miss*, and the line causing the miss must be copied to the L1 cache from the shared L2 cache. This process repeats for the L2 cache, except in this case the GPU will fetch an L2 line from DRAM on a miss. Cache misses have a higher memory latency than hits due to the time required to fetch data from a higher level of the hierarchy, though the GPU attempts to mitigate this via *latency hiding*: while one or more warps are waiting for cache misses to be resolved, instructions from other ready warps can still be dispatched by the warp scheduler. In this way, the GPU's parallelism compensates for the impact of additional latency in some threads by allowing other threads to make progress.

### A. GPU Concurrency and Interference Channels

The term *concurrency* on a GPU may take on more meanings than on a CPU. For example, multiple kernels may be launched "concurrently" from different CUDA contexts. Despite being launched in overlapping segments of time and sharing a single GPU, they may not actually execute at the same time. By default, CUDA enforces *time slicing*, allowing only one context to access GPU hardware at a time. However, threads launched from the same CUDA context can execute truly simultaneously. Two kernels may experience *SM-level concurrency*, running on the same GPU, but independently on different SMs. Kernels may also encounter *warp-level concurrency*, where the warps of one or more independent kernels execute concurrently on the same SM. As the form of concurrency changes from SM-level to warp-level, the number of shared resources, and thus interference channels, increases.

For the rest of this paper, we only consider true, non-time-sliced, GPU concurrency, so we assume that kernels are launched from a single CUDA context. Once again, interference channels are *shared resources* between concurrent kernels, where contention for these resources causes potentially unpredictable increases in kernel execution times.

**Inter-SM interference channels.** When kernels do not share SMs, they will not share SM-local resources, so the

[2]Many frameworks such as PyTorch or TensorFlow expose GPU acceleration to higher-level languages, but still interact with CUDA under the surface.

[3]Each SM also has an instruction cache. We use "L1" to mean the data cache.

primary remaining sources of interference are the L2 cache and DRAM. When sharing the L2 cache, kernels can *evict* cache lines used by other kernels in order to make space for their own data, causing them to experience more cache misses and longer execution times. Another channel associated with the L2 cache is the *miss status holding registers* (MSHRs), a limited number of registers used in non-blocking caches. Each time a cache miss occurs, an MSHR is reserved until the missed line has been fetched from DRAM. If no MSHRs are available, then a pending miss must wait until an ongoing miss has been resolved and an MSHR is freed. Therefore, multiple kernels experiencing frequent cache misses may contend for MSHRs, compounding the issue of increased memory latency.

Beyond the L2 cache, concurrent kernels may also contend for DRAM access. The DRAM of the GTX 1080 consists of multiple *banks*, each containing multiple fixed-size memory elements called *rows*. Each bank contains a *row buffer*, which contains the last row referenced in that bank. If an incoming memory request requires data that is not already contained in the row buffer, then the current contents of the row buffer are evicted and written back to DRAM. Afterwards, the newly requested row is copied into the row buffer. If multiple kernels reference addresses corresponding to different rows in the same bank, they will evict each other's data from the row buffer, causing *row buffer conflicts*. As with cache evictions, this increases memory latency. Still more DRAM-related interference channels exist such as DRAM bus contention, but we choose not to discuss these as their effects are difficult to empirically distinguish on our available NVIDIA GPUs.

**Intra-SM interference channels.** When kernels share an SM, they may contend for the per-SM hardware units shown in Fig. 1. These include the 32-bit *CUDA cores* (which include FPUs: *floating-point units*), *special function units* (SFUs), or *load/store units* (LD/ST) [14]. For example, if all available FPUs are in use, additional floating-point operations will stall, meaning that one kernel's use of FPUs can delay another kernel. In addition to compute hardware, warps on the same SM may also contend for SM-local memory resources including the L1 cache and shared memory (and also the instruction cache). The contention may cause mutual interference similar to the L2 cache and DRAM interference described above.

### B. Prior Work

We now discuss relevant prior work in several topic areas.

**CPU timing analysis.** A wide variety of timing-analysis techniques have been developed over the years for CPUs. Many works, especially for single-core CPUs, make use of *static analysis* to analyze program code in order to compute WCETs [15–17]. Static analysis for multicore CPUs is usually infeasible, as the number of possible interactions between tasks increases explosively with the number of cores and possible ways in which different segments of code may overlap. Therefore, *measurement-based analysis* is frequently employed, generating WCET estimations from observed task execution samples. Under this method, resource-sharing impacts must be accounted for. *Partitioning* is a common approach, dividing platform resources among tasks rather than sharing them, thus preventing interference. Many works (representative examples include [18–22]) enforce isolation on shared CPU resources, including cache, DRAM, and memory bandwidth.

Unfortunately, partitioning techniques may be inflexible and put unnecessary constraints on individual tasks. Instead of partitioning, one may allow tasks to share resources but account for the interference in analysis methods. One work accounts for cache contention by introducing cache-related preemption delays to the schedulability analysis [23]. Other works take an *enemy workload* approach [13, 24–27], the same as ours, albeit for multicore CPUs.

**GPU timing analysis.** Some work has been done on providing robust timing analysis for a single GPU kernel. For example, one work proposes timing models for GPU kernels on NVIDIA's Fermi GPU architecture [28], but the methods apply to limited workloads, and may not extend to other architectures. In two works, statistical theories are utilized to enhance the reliability of measurement-based WCET estimations [3, 8], but these efforts are limited by some less-practical assumptions, such as statistical independence among samples. Other earlier work proposes methods for statically deriving the WCETs of a group of threads on a single SM, but these methods do not extend to multiple SMs [4, 5]. Other works employ a hybrid approach, combining both static and measurement-based analysis [6, 7], but these methods entail using GPU simulators and are limited to small kernels.

**GPU sharing.** We now turn to works on safely sharing the GPU among multiple tasks. One work introduces methods for predictable multi-tasking on NVIDIA GPUs but it too uses simulators [10]. One extensive work implements both cache and DRAM partitioning via page coloring on NVIDIA GPUs [11]. This approach improves predictability for concurrent GPU tasks but required complicated reverse engineering and places some undesirable constraints on concurrent kernels. Our work is largely orthogonal; we allow for concurrent GPU sharing, but aim to account for worst-case interference without partitioning. Other works present management frameworks that enable the sharing of NVIDIA GPUs with varying tradeoffs and limitations [12, 29, 30]. A study has also been presented involving the sharing of embedded GPUs between common workloads [31]. While useful in other contexts, all of these previous works either do not measure resource interference under arbitrary GPU sharing, or were limited in their capacity to handle real applications.

## III. Methodology

Our goal in constructing enemies is to enable robust measurement-based timing analysis for workloads involving the concurrent execution of GPU kernels. As such, we must consider as thoroughly as possible all interference channels that may influence kernel execution times.

We introduce a methodology for generating and measuring the interference from channels influencing a victim[4] that may be concurrently executing with unknown kernels competing for device resources. Our approach, based on prior work on multicore CPU computing, is to execute a victim concurrently with enemies made up of one or more synthetic kernels that contend for GPU resources. The end goal is to produce enemies that are more harmful to the victim's execution time than any actual competing kernels.

GPU kernels are highly parallel, so concurrent kernels in particular have many possible interleavings of instructions, each with a different pattern of resource usage and degree of contention among kernels. It is unreasonable to assume that a kernel in a concurrent workload will experience the worst-possible interference with any observable regularity. However, a carefully designed enemy program can consistently and effectively stress an interference channel, even with relatively simple code. The process defined in this section consists of a framework for setting up a sequence of experiments to maximize the interference effects obtained in an enemy kernel.

**Victim selection.** For our experiments, we selected victims from a set of publicly available benchmark applications. To cover a wide range of applications and types of kernels, we chose to evaluate each kernel in a set of 24 CUDA programs chosen from a variety of sources. These include nearly all programs from the Tango DNN suite [32] and a set of CUDA imaging samples—we exclude only the imaging programs that require graphical user interfaces.

Table I shows the full list of victim benchmark programs. Some of the benchmarks execute several kernels, giving 117 different victim kernels in total. Individual kernels may be invoked multiple times with different inputs. These GPU-using programs cover a variety of domains relevant to real-time GPU usage including image processing, computer vision, and neural networks.

**Methodology overview.** All of the code and data used in our experiments are available online.[5] After selecting a victim, Sec. III-A describes our methodology for defining and implementing the *environment* in which we execute the victim and potential enemies. An environment includes the hardware, CUDA software version, and other constraints on how kernels may compete. After establishing a victim and an environment, Sec. III-B discusses our process for selecting an *enemy template* and *enemy parameters* to form potential enemies. To evaluate these enemies, we use the metrics defined in Sec. III-C, which we obtain using an experiment structure described in Sec. III-D.

### A. Environment Implementation

We now discuss the way we implement our environments, including how we run victims and enemies concurrently, and how we enforce resource constraints.

[4]We use the term *victim* here in a generic sense, to refer to a full GPU-using application. If the distinction between an application or a specific kernel is important, we will clarify using terminology like "victim kernel."

[5]https://github.com/tylerdy/Modified-NVIDIA-Linux-x86_64-460.67/tree/RTSS22

| Benchmark Source | Benchmark | # Kernels |
|---|---|---|
| Basic CUDA Samples | `matrixMult` | 1 |
| | `vectorAdd` | 1 |
| | `scalarProd` | 1 |
| | `fastWalshTransform` | 3 |
| CUDA Imaging Samples | `bilateralFilter` | 1 |
| | `boxFilter` | 2 |
| | `convolutionSeparable` | 2 |
| | `convolutionFFT2D` | 10 |
| | `dct8x8` | 8 |
| | `dwtHaar1D` | 2 |
| | `dxtc` | 1 |
| | `histogram` | 4 |
| | `HSOpticalFlow` | 6 |
| | `imageDenoising` | 1 |
| | `NV12toBGRandResize` | 3 |
| | `recursiveGaussian` | 2 |
| | `stereoDisparity` | 1 |
| FastHOG [33] | `fastHOG` | 7 |
| Tango DNN Suite [32] | `AlexNet` | 7 |
| | `CifarNet` | 8 |
| | `GRU` | 1 |
| | `LSTM` | 1 |
| | `ResNet` | 14 |
| | `SqueezeNet` | 30 |

TABLE I
POTENTIAL VICTIMS.

**Running victims and enemies.** In order to allow victim kernels to run concurrently with enemies, we make use of NVIDIA's *multi-process service* (MPS) [34]. MPS multiplexes kernels launched from separate Linux processes into a single CUDA context, allowing them to execute concurrently on the GPU (see Sec. II). With MPS active, we use basic scripts to *first* launch an enemy and then launch the victim. Furthermore, we ensure that the enemy kernel runs for longer than the victim, so that interference occurs for the victim's entire duration. We shall see later in this subsection that this launch order is crucial to our ability to manage GPU resources.

Even if MPS is in use, some CUDA API functions are unfortunately capable of causing blocking in other processes or being blocked themselves. For example, a victim that calls the `cudaDeviceSynchronize` function may be forced to wait for enemy kernels to complete, even though the victim's code should not logically depend on any enemy activity [35]. To avoid this problem, we modified the source code of some of our benchmarks, to remove or replace calls to such blocking functions. This was the only type of modification we made to victim source code.

**Hardware platform.** Our chosen hardware platform is the NVIDIA GeForce GTX 1080, running CUDA 11.1. We discussed its relevant architectural details in Sec. II.

**Resource constraints.** Resource constraints can take on many forms, *e.g.*, maximum available memory, maximum total concurrent kernels, *etc.* In our work, however, the most important resource constraint is the number of SMs that competing kernels are allowed to run on, or the number of warps per SM they are allowed to occupy.

Constraining computational resources is *essential* in our experiments. For example, if we allow enemies to fully occupy all SMs on the GPU, our measurements would end up being dominated by the time the victims spend *waiting* for SMs to become available, rather than any slowdown associated with

interference channels. Likewise, if we allow a victim to fully occupy all of the available SMs, the enemy would be unable to run, causing the victim to experience no interference at all.

**Environment definitions.** We define two environments, which each uses different tactics to ensure the enemy and victim always have consistent access to the GPU's SMs:

- We use the *SMK*[6] environment to measure intra-SM interference. It limits the enemy to at most 32 warps per SM (half of an SM's capacity), ensuring the remaining warps on each SM are available to the victim.
- We use the *spatial multitasking* environment to measure inter-SM interference. It forces the enemy to fully occupy eight out of the GTX 1080's twenty SMs, allowing the victim to execute only on the remaining twelve SMs.[7]

Two key factors underlie our implementation of both environments. First, *we always start executing enemies before victims*, and second, *blocks execute until completion on whichever SM they are initially assigned* (discussed in Sec. II). In combination, these two properties allow us to control both enemy *and* victim SM assignments: by launching the enemy first (while the GPU is idle), we allow the enemy to "claim" all the resources that we wish to allocate to it. The enemy never relinquishes these resources prior to completion, so as long as the enemy executes for longer than the victim, the victim is forced to execute only on whichever resources have not been allocated to the enemy.

**Implementing the SMK environment.** Following the principles outlined in the previous paragraph, we can construct our SMK environment, in which the victim and enemy are forced to share SMs. We do so by launching an enemy kernel using exactly one block per SM, with each block containing the exact number of warps per SM that we wish the enemy to occupy.[8] We know from prior work that a GPU starting in an idle state will evenly distribute a kernel's blocks across all SMs [37], and we always launch enemies first, while the GPU is idle. Therefore, launching one enemy block per available SM will ensure that exactly one enemy block is assigned to every SM on the GPU. After launching the enemy in this configuration, we launch the victim, which must share each SM with the enemy warps.

Due to device resource limits (*e.g.*, registers per SM), it was not possible to execute all kernels from `AlexNet`, `CifarNet` and `ResNet` concurrently with our enemies under SMK. Therefore, we conducted SMK experiments using only the remaining 88 kernels.

**Implementing the spatial multitasking environment.** Using the same assumptions, we are able to implement the spatial multitasking environment, in which the enemy and victim never share SMs. Once again, our approach depends on the fact that we launch enemies before victims, allowing the enemy kernel to "claim" SMs. Unlike under SMK, for spatial multitasking we configure the enemy to launch 64 warps per SM, fully occupying every SM on the entire GPU. Next, we apply a technique used in prior work (*e.g.*, [11]) to force kernels to use only a specified set of SMs: we instrument the enemy kernel code so that every thread checks its assigned SM ID[9] and immediately exits if running on an SM we wish to allocate to the victim. In other words, the spatial multitasking environment causes the enemy to immediately relinquish any victim SMs, while fully occupying all other SMs.

| Template Type | Parameter | Possible Values |
|---|---|---|
| Compute Enemy | Operation to execute | `ADD`, `MULT`, *etc.* |
| | Operand type | Integer or Floating-point |
| | Operand Size | 16-bit, 32-bit, or 64-bit |
| Memory Enemy | Array size | 960 KB—16 MB* |
| | Stride Length | 8 bytes—128 bytes* |
| | Per-thread or Per-warp Accesses | Per-thread, Per-warp |
| | Memory-Access Operation | Read or Write |

TABLE II
ENEMY TEMPLATES AND PARAMETERS.
*MORE VALUES ARE POSSIBLE; THESE ARE JUST WHAT WE USE IN THE PAPER.

### B. Enemy Implementation

We base all of our enemy kernels on *templates* within which *parameter values* may be varied to cause more or less stress on portions of hardware. This approach is based on similar works for multicore CPUs [13, 24, 26, 27]. We have two templates for two types of enemy kernels—one designed to contend for compute resources, and one designed to contend for memory and cache. Table II lists these two templates along with their associated parameters. We discuss the meaning and utility of each of these parameters next.

**Compute enemy template.** Our compute-oriented enemies are designed to maximize stress on a *single* compute resource.[10] The template for our compute enemies repeatedly executes some chosen instructions to target specific hardware units, *i.e.*, the CUDA cores or SFUs described in Sec. II-A.

The three parameters required by the compute-enemy template are shown in the first three rows of Table II: the *operation executed* (*e.g.*, `ADD`, `MULT`, *etc.*), the *operand type* (*i.e.*, integer or floating point), and the *operand size* (*i.e.*, 16, 32, or 64-bit). We combine these three parameters to determine the instruction the enemy will execute.

**Memory enemy template.** The template for stressing memory and/or cache resources is more complicated, involving the four parameters shown in the latter rows of Table II. All kernels based on the memory-enemy template make strided accesses to an array, repeatedly performing reads or writes to load data into a cache line or a row buffer, attempting to cause hardware to evict the victim's data.

[6]We take the term SMK, *simultaneous multi-kernel*, from prior work that investigated SM sharing [36].

[7]While somewhat arbitrary, this 8-12 split enables some convenient simplifications to our L2-cache-related experiments, discussed in Sec. IV-D.

[8]Our SMK experiments always used 32 enemy warps per SM, which is exactly equal to the maximum number of threads within a single block: (32 warps) $\times$ (32 threads per warp) = 1,024 threads.

[9]In CUDA kernel code, this is accomplished by using inline assembly to read the special `smid` register.

[10]To keep our experiments feasible for this paper, our compute enemies only target a single resource at a time. However, we expect detailed information about single-resource enemies to remain relevant should we attempt to develop multi-compute-resource enemies in the future.

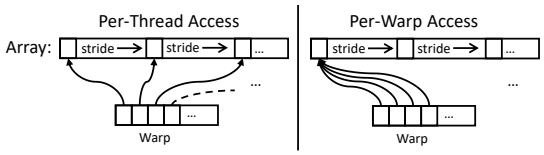


Fig. 2. The difference between per-thread and per-warp memory access.

Naturally, some memory-enemy parameters will take different values depending on the level of the memory hierarchy we intend to stress. For example, when stressing the L2 cache, the *array size* parameter must be at least the size of the L2 cache: 2 MB on the GTX 1080. Next, the *stride length* parameter determines the number of bytes between the enemy's subsequent array accesses. Continuing with the L2-cache example, we expect the most effective enemy will have a stride length matching the cache's line size (128 bytes on the GTX 1080), so that every subsequent array access falls into a different cache line. Granted, these are simply initial assumptions, intended to illustrate potential relations between hardware and memory-enemy parameters—we revisit this topic in Sec. IV's experiments.

The third memory-enemy parameter shown in Table II can take one of two values, *per-thread* or *per-warp* access. A memory enemy makes *per-thread* array accesses if every thread in every warp accesses a different element of the array, and it makes *per-warp* accesses if different warps access different elements while all threads within a given warp access the same element. This distinction is depicted in Fig. 2. Per-warp access is only important in our L1-cache experiments (Sec. IV-B), because the smaller array size used to generate L1 stress is not large enough for every enemy thread to each access an individual array element. All of our other memory enemies (Secs. IV-D, IV-E, and V) use per-thread access.

The final memory-enemy parameter is the *memory-access operation*: whether our accesses to the array consist of reading or writing array data. We evaluate this parameter in both our L1 and L2 experiments, in Secs. IV-B and IV-D.

**GPU driver modifications.** Our GTX 1080 GPU uses *physical addresses* to determine L2 cache lines. This can cause a problem: when we attempt to allocate an array in CUDA, the array's memory is only guaranteed to be contiguous in *virtual* memory—some of the underlying physical pages may map to identical cache lines, causing the array to not utilize the full cache. In order to solve this problem, we modified open-source portions of NVIDIA's Linux driver code. More details about this problem and its solutions are covered in prior work [11], to which we refer interested readers.

### C. Enemy Evaluation

We wish to measure the impact an enemy has on a victim, but we cannot simply compare a victim's performance in isolation (*i.e.*, with access to the entire GPU) to the victim's performance in the presence of an enemy. The problem arises from the mechanism we use to constrain the victim's resource usage. As discussed at length in the preceding section, we limit the victim's SM (or warp) usage indirectly: by allowing the enemy to occupy all remaining warps or SMs. This means that even if a particular enemy causes no interference whatsoever for a given victim, the victim will still execute more slowly than it would in isolation, simply due to having access to fewer of the GPU's SMs or warps.

```
__global__ void OccupyResources(uint64_t iterations) {
    uint64_t i = 0;
    if (threadIdx.x == 0) {
        // Only executed by the first thread in the block
        while (i < iterations) i++;
    }
    // All threads stall here until the first thread finishes
    __syncthreads();
}
```

Fig. 3. Simplified CUDA code for our occupation kernel.

**Occupation kernels.** Rather than attempting to base our evaluation metrics on a victim's performance in isolation, we implemented what we call an *occupation kernel*: a kernel that occupies the same number of SMs and warps as an enemy while otherwise causing as little interference as possible. Fig. 3 shows a simplified version[11] of our occupation kernel's code. Our occupation kernel makes no memory references, and only a single thread per block executes compute instructions in order to waste time. While the one thread is executing trivial computations, all remaining threads in the kernel use CUDA's `__syncthreads` directive to intentionally stall.

**Sensitivity.** With an occupation kernel, we can finally define a metric for enemy interference that factors out the effects of some SMs or warps being unavailable to the victim. Our metric is *sensitivity*: defined as the victim's execution time when run concurrently with the enemy divided by its execution time when run concurrently with the occupation kernel.

When running multiple trials, we define sensitivity slightly differently. Unless otherwise noted, most of the sensitivity measurements in this paper are based on the victim's 90th percentile time vs. the enemy, divided by the victim's median time vs. the occupation kernel. In line with prior work [13], we choose the 90th percentile time for the enemy scenario because we are interested in the highest-stress trials, but wish to ignore extreme outliers due to uncontrollable device operations that are unrelated to our stress methods. We wish to establish a stable "baseline" measurement in the occupation-kernel scenario, which is better served using the median time rather than a sample near the end of the distribution. Recall that some victim benchmarks launch a given kernel multiple times. In these cases, we base the kernel's sensitivity on the invocation for which we observed the highest sensitivity value.

### D. Generic Experiment Structure

All experiments in the following section use the same procedure:

1) Choose a template stress kernel, either compute or memory. Note that some experiments require multiple kernels to combine multiple types of stress.
2) Set all but one enemy parameter to be constant throughout the experiment. The constant parameters are set to

[11]The actual occupation kernel requires additional boilerplate, *e.g.*, to allow for SM assignment using the techniques from Sec. III-A.

either a default value (*e.g.*, the array size for cac experiments), or to a value that produced maximal stre in prior experiments.

3) For the non-constant parameter, select a number values to evaluate. These values define the set of ener kernels we will run concurrently with the victim.
4) Take multiple measurements of the victim's time again both each enemy kernel and the occupation kernel. (V took at least ten samples for the results covered Sec. IV and at least five for Sec. V.)
5) Compute the victim's sensitivity to each enemy kernel based on the previous step's measurements.

In our experiments with large numbers of victims, we occasionally make simplifications about which values to consider in future experiments. Tracking the most effective value for each parameter for each victim can lead to an intractable number of combinations of parameter values. If one particular enemy is only effective against a relatively small number of victim kernels, or the difference between victim sensitivities to two enemies is small, we may omit it from consideration in future experiments. We will explicitly mention any experiments involving such simplifications in Secs. IV and V.

## IV. Stress Experiments

This section presents the experiments we conducted using the methodology discussed in Sec. III. For each experiment, we state the goals, provide the background information needed to understand the issues involved, specify the details of the experimental setup, and then present and explain the results. We organize these experiments based on the interference channel or channels they target. We first present our experiments evaluating intra-SM interference in Secs. IV-A through IV-C, followed by our inter-SM experiments in Secs. IV-D and IV-E.

### A. Intra-SM: Compute Interference

In assessing intra-SM interference channels, we first considered hardware units used for instruction execution.

**Goal.** We designed this experiment to select an instruction to use in the enemy kernel template for compute resources discussed in Sec. III-A. We used this selection in any subsequent experiments that require a compute enemy.

**Background.** This experiment measures interference caused by an enemy's repeated execution of a single computational instruction (we leave experiments with sequences of various instructions for future work). Such interference results from the victim and enemy contending for a limited number of per-SM hardware resources used for instruction execution. The particular hardware resources in our GTX 1080's SMs (*e.g.*, CUDA cores, *etc.*) are discussed in Sec. II.

**Experimental setup.** Our preliminary analysis of source code for the benchmarks listed in Table I found heavy usage of floating-point operations. Thus, in this experiment, we measured the sensitivity of the victim kernels to six enemy kernels that execute the instruction for one of the combinations of floating-point operation (ADD or MULT) and operand size

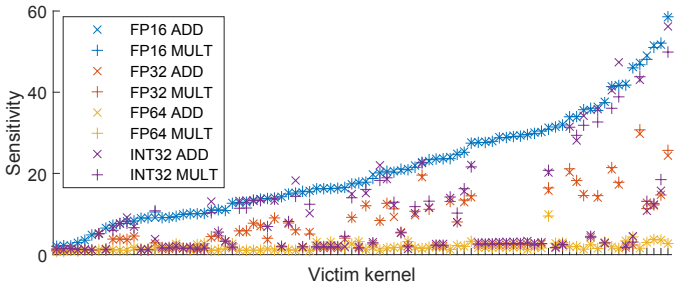


Fig. 4. Victim sensitivities to compute enemies, sorted by sensitivity to the FP16 MULT kernel.

(16, 32, or 64 bits). We also included two enemy kernels that execute ADD and MULT on 32-bit integer operands to provide a comparison with heavily used integer operations. This experiment used the SMK environment, as it focuses on intra-SM interference.

**Results.** Fig. 4 plots the sensitivities of each of the 88 SMK-environment victim kernels to the eight enemy kernels just described. Each victim kernel is represented by a point along the $x$-axis and the corresponding eight markers on the $y$-axis give that victim's sensitivity to each of the eight enemies. The victims are ordered along the $x$-axis by increasing sensitivity to the FP16-ADD enemy.

**Observation 1:** *90% of the victim kernels are most sensitive to enemies executing half-precision floating-point operations.*

**Observation 2:** *There is little distinction between ADD and MULT for each of the operand types and sizes.*

These results for the FP16 enemies are likely due to some unknown artifact of our GTX 1080 platform. None of the victim kernels use half-precision instructions, so there should be no conflicts for the GTX 1080's per-SM half-precision unit. We eliminated the possibility of unknown additional sources of contention using NVIDIA's official profiling tools (`nvprof`) and disassembler (`nvdisasm`). With `nvdisasm`, we verified that compiling the FP16 enemy kernels generated the expected low-level (SASS) GPU instructions: `HADD2` and `HMUL2`. With `nvprof`, we were able to verify that these half-precision instructions were actually executed by the enemy kernels, while the victims executed no half-precision instructions. We have found no affirmative evidence for our FP16 results, so we assume Obs. 1 must be the result of some undocumented property of the GTX 1080's half-precision implementation.

Considering only the non-FP16 sensitivity results, we found that 73% of the victim kernels were more sensitive to INT32 operations than to any floating-point operation. Overall, these results indicate that, in general, INT32 operations would be the most significant source of interference on this intra-SM channel, were it not for the overwhelming interference caused by FP16 operations. Based on the discussion above, we eliminated FP16 operations from all experiments described below to avoid our results becoming overly platform-specific.

**Mixed-operations enemies.** Many victim kernels use some mix of floating-point and integer operations. We designed

an additional experiment to evaluate enemies which exec mixes of floating-point and integer operations. Our experim tal setup remained the same except for the operations execu by the compute enemies. We tested five enemies, with 1 ratios of FP32-ADD to INT32-ADD warps: 32:0, 24:8, 16: 8:24, and 0:32.

**Observation 3:** *84% of victim kernels are more sensitive 32-warp INT32 than to any mixture of INT32 and FP32.*

**Observation 4:** *83% of victim kernels are more sensitive to 32-warp FP32 than to any mixture of FP32 and INT32, other than the one with the most INT32 operations.*

These results indicate that single-operation enemies tended to create greater interference than mixed-operation enemies, and that the interference of INT32 operations remained greater than FP32. In our environment, the greatest stress comes from using all warps to fully saturate either the FP or INT hardware on victims most sensitive to that type of compute operation.

### *B. Intra-SM: L1 Cache*

Having determined the instruction to be used in enemy kernels for intra-SM compute interference, we next turn to the other intra-SM interference channel—the L1 cache.

**Goal.** This experiment was designed to measure interference caused by contention for the L1 cache among concurrent kernels. The enemy creates interference when its data accesses evict cached data belonging to the victim.

**Background.** The L1 cache is the only SM-local memory interference channel that we chose to consider in our experiments.[12] When the threads of a warp execute a load or store instruction, each thread typically references a different address to access thread-dependent data. These addresses may be highly *coalesced* into one or a small number of cache lines, or they may reference as many as 32 different cache lines.

The L1 cache on our GTX 1080 has a size of 48 KB with a line size of 128 bytes. Several other properties are relevant to cache interference, including the write policy (write-through) [39] and potential usage of MSHRs to allow multiple outstanding requests when misses occur [40]. The write-through policy means that writes in the L1 cache are immediately written to the L2 cache, incurring added latency. Since the MSHRs are also relevant for the L2 cache, we defer further consideration of them to Sec. IV-E.

**Experimental setup.** The warps of the enemy kernels evict data from their SM-local L1 caches by continuously iterating through addresses that are 128 bytes (one cache line) apart in a 48 KB (L1 cache size) array of data that is unique for each SM. All threads in a warp reference the same address (per-warp access as described in Sec. III-B) so victim data, if present in the cache, is evicted from one cache line on each memory reference by an enemy warp. Note that it is impossible to generate interference only in the L1 cache since the enemy may also evict victim data from the L2 cache with some of its memory references.

[12] As mentioned in Sec. II, there is also an instruction cache on each SM which is discussed in Sec. VI. CUDA's *shared memory*, used for scratchpad communication between threads within a single block, is another intra-SM memory resource, but we do not consider it in our experiments, since, unlike the L1, it directly limits GPU occupancy, possibly preventing concurrency altogether [37, 38].

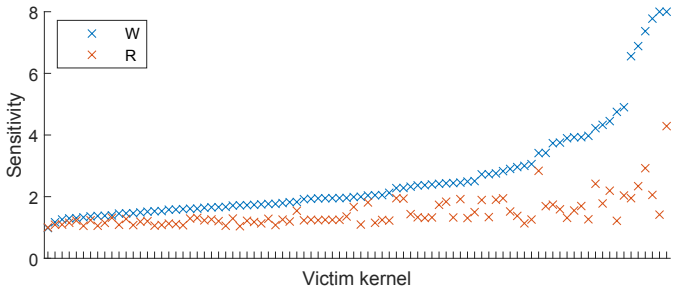


Fig. 5. Victim sensitivities to L1 cache eviction enemies, sorted by sensitivity to enemy W memory references. Note that this plot contains two W outliers with sensitivities of 8.9 and 9.9, which we clamped to $y = 8$.

In this experiment, we measured the sensitivity of the victim kernels to two variants of the L1 cache enemy kernel. These variations use one load (read, R) or store (write, W) instruction to reference memory. Write operations store an arbitrary constant value.

**Results.** Fig. 5 shows the sensitivities to the L1-cache enemy kernels of each of the 88 victim kernels used for the SMK environment mentioned in Sec. III-A. Each victim kernel is represented by a point along the $x$-axis and the corresponding two markers on the $y$-axis gives the sensitivity of that victim to one of the enemy kernels as indicated by the legend. The victims are ordered along the $x$-axis by increasing sensitivity to the write enemy.

**Observation 5:** *Over 98% of victim kernels are more sensitive to cache evictions created by enemy write operations.*

These results indicate that the write-through policy used in the L1 data cache may have a significant effect on interference. The added latency necessary to update the L2 cache likely increases the time that the victim kernel will stall waiting for data in the L1. 75% of the victim kernels are more sensitive to enemy writes than reads by 50% or less. The sensitivity difference is much greater for the remaining kernels, which likely perform a high number of write operations themselves.

Based on Obs. 5, enemy kernels designed for L1 cache interference should use write instructions to access memory.

**Observation 6:** *The sensitivities of the victim kernels to L1 cache interference are much smaller than their sensitivities to compute interference.*

Obs. 6 is supported by comparing Figs. 4 and 5, which show a large difference in sensitivities to these two sources of interference (note the different $y$-axis scales). 100% of kernels are more sensitive to interference caused by some compute operations than they are to L1 interference from enemy memory-write operations. Granted, this may be due to the complete dominance of FP16 enemies in these initial experiments. Our experiments in the following subsection

indicate that intra-SM compute interference remains poter even without relying on FP16 operations.

**Mixed-operations enemies.** Most victim kernels use som mix of read and write operations. We designed an additiona experiment to evaluate enemies which execute mixes of rea and write operations. Our experimental setup remained th same except for the operations executed by the L1 cach enemies. We tested five enemies, with ratios of read (R) t write (W) warps: 32:0, 24:8, 16:16, 8:24, and 0:32.

**Observation 7:** *93% of victim kernels are more sensitive to an enemy with 32 write-only warps than to an enemy configuration involving reads.*

**Observation 8:** *Only 6% of victims are more sensitive to an enemy that mixes reads and writes.*

These results indicate that the added latency of the L1 write-through policy, or other details of the hardware write mechanism, tend to dominate any potential stress-inducing properties unique to reads.

### C. Intra-SM: Combining Intra-SM Interference Channels

We have now established the implementation of enemy kernels to generate interference on the compute and L1 cache intra-SM channels. We next determine if combinations of the compute and memory interference channels have a greater effect than either of them alone. We also reexamine mixtures of compute operations (INT32 and FP32) and memory operations (R and W) in the context of combining compute and memory interference channels.

**Goal.** This experiment is designed to assess the sensitivity of the victim kernels to enemy kernels that create a combination of compute and L1-cache interference using mixtures of compute and memory operations.

**Background.** It is an oversimplification to assume that a real GPU kernel is entirely compute- or memory-intensive or that the compute intensity comes only from integer operations and memory intensity comes only from write operations. Further, compute and memory operations in victim and enemy kernels may be mixed and executed in arbitrary orders with unpredictable interference effects.

**Experimental setup.** We set up two variants in these experiments. The first variant is a simple baseline setup to evaluate a combination of interference channels. We selected the most effective compute and L1-cache enemy for each victim and changed how the warps were allocated in these enemies. As in all of our SMK experiments, we continued to divide each SM's 64-warp capacity evenly between the victim and the enemy. We considered three possible allocations of the 32 warps dedicated to the enemy: *all-compute* (a 32-warp INT32 compute enemy), *all-L1* (a 32-warp write L1 cache enemy), and 16/16 *hybrid* combination (a 16-warp INT32 enemy, and a 16-warp write L1 cache enemy).

The second variant is designed to evaluate combined interference channels each having a mixture of operations. We kept the number of enemy warps per SM the same (32),

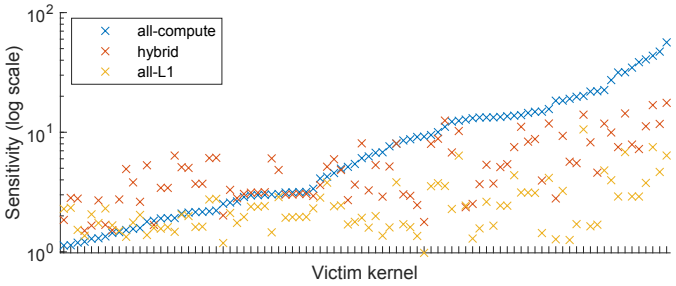


Fig. 6. Victim sensitivities to 32-warp enemies with different combinations of compute and L1 cache warps, sorted by sensitivity to the all-compute combination.

with 16 warps for compute operations and 16 warps for memory operations as in the baseline setup. For describing the combinations and mixtures, we use $CF$ to be the number of warps executing FP32-ADD operations, $CI$ for INT32-ADD warps, $MR$ for memory read warps and $MW$ for memory write warps. For compute warps, we used ratios $CF$:$CI$ of 4:12, 8:8 and 12:4. Similarly, the memory warp ratios $MR$:$MW$ are also 4:12, 8:8 and 12:4. Combining all instances of compute and memory warps gives 9 scenarios in total.

**Baseline combinations: results.** Fig. 6 shows the sensitivities of the 88 victim kernels used for the SMK environment to each of the three combinations. Each victim kernel is represented by a point along the $x$-axis, and the corresponding three markers on the $y$-axis give the sensitivity of that victim to the different enemy warp allocations. The victims are ordered along the $x$-axis by increasing sensitivity to the all-compute combination.

**Observation 9:** *39% of victim kernels are more sensitive to the hybrid 16/16 enemies than to the all-compute enemy.*

**Observation 10:** *94% of victim kernels are more sensitive to the hybrid 16/16 enemy than to the all-L1 enemy.*

These baseline results show that considering interfering kernels to be simply compute-bound or memory-bound is likely to be an oversimplification and that combinations of compute and cache enemy kernels should be considered in many cases for generating interference.

**Combinations and mixtures: results.** We compared the sensitivities of each victim to the 9 mixed-operation enemies and the 3 enemies in the baseline setup. We found that all victims remain the most sensitive to the same enemy as in the baseline setup except for one (`bilateralFilter`). This one victim was the most sensitive to the enemy with parameter 4:12:4:12 (in the form $CF$:$CI$:$MR$:$MW$). Out of the 35 victims that were the most sensitive to the 16/16 hybrid enemy in the baseline setup, only this victim was more sensitive to a variant of the 16/16 hybrid enemy that mixed operations. Comparing with single-channel results, we conclude that most victims remain more sensitive to single-operation enemies, with a few being the most sensitive under a mixed-operation enemy. These results are consistent with previous findings where combining multiple channels may create heavier stress.

**Real vs. idealized enemy configurations.** All of our int SM experiments considered a balanced allocation of the warps per SM: 32 warps each for victim and enemy. When r applications run concurrently, the number of warps assigned the GPU scheduler to concurrent kernels varies dynamica The result is that a kernel will be more or less sensitive interference depending on how and when warps are assigi to it and its competitors. In addition, our intra-SM experime for combinations of interference channels also used a balan allocation of the 32 enemy warps for compute and memory operations (16 for each). Real applications may be more compute-intensive or more memory-intensive, and their type of intensity may vary over the course of execution. Varying the ratio of enemy compute warps to memory warps can alter the sensitivity of a victim kernel to interference depending on the execution intensity of its internal operations.

### D. Inter-SM: L2 Shared Cache

Having completed our discussion of experiments for intra-SM interference channels, we now turn to experiments for inter-SM interference channels. Inter-SM experiments utilize our spatial-partitioning environment, where each SM is either allocated to the enemy or victim as discussed in Sec. III. The interference channels now only include resources that are shared among SMs, including the L2 cache, MSHRs, memory buses and DRAM memory. We first consider the L2 cache.

**Goal.** This experiment is designed to measure interference caused by contention for the L2 cache, which occurs when data referenced by the enemy kernel evicts data belonging to the victim kernel from the cache.

**Background.** The L2 cache is shared by all SMs.[13] The L2 cache on our GTX 1080 is a 2 MB set-associative cache with a line size of 128 bytes [11]. According to prior work [40], the L2 cache on the GTX 1080 implements a write-back policy. Under write-back, writes to the L2 cache are not immediately written to DRAM, but instead are written when the modified cache line is evicted. Thus, if the victim experiences a cache miss and evicts a line written by the enemy, it will incur the extra overhead of writing the dirty line back to DRAM. In addition, MSHRs exist to allow multiple outstanding requests to DRAM on L2 cache misses as discussed in Sec. II-A [11].

**Experimental setup.** Similar to our approach for the L1 cache in Sec. IV-B, we set the array size and stride length of the L2 enemy to the cache size (2 MB) and the line size (128 bytes). Each thread in a warp now has a choice between per-thread access and per-warp access as described in Sec. III-B. Under per-thread access, victim data, if present, can be evicted from up to 32 cache lines on each memory access by an enemy warp. However, per-thread access may lead to higher memory latency and slow down the enemy since the size of data accessed is increased by a factor of 32. With a quick test, we found that 98% of the victim kernels are more sensitive to the enemy with per-thread access than to the enemy with per-warp access given all other parameters identical. We thus used per-thread access for L2 cache and later inter-SM enemies.

[13]The L2 cache also serves as a second-level cache for the instruction caches local to each SM, but we continue to assume this effect is small (see Footnote 12). We also do not consider L2 interference caused by CUDA `memcpy` operations, which also can place data in the L2 cache.

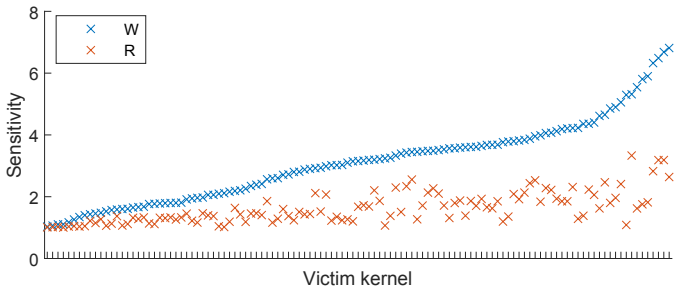


Fig. 7. Victim sensitivities to L2-cache enemies executing different memory access operations, sorted by sensitivity to the W enemy.

We measured the sensitivity of all 117 victim kernels to two enemy kernels intended to cause cache conflicts. One of the two enemies performed a load (read, R) instruction, and the other performed a store (write, W) instruction to access memory. Write operations store an arbitrary constant value.

Note that it is impossible to generate interference only in the L2 cache, since L2 misses may cause interference for the DRAM, the MSHRs and the memory buses. We left discussion of these other channels to Sec. IV-E.

**Results.** Fig. 7 shows the sensitivities of each of the 117 victim kernels to the L2 enemies. Each victim kernel is represented by a point along the $x$-axis, and the corresponding two markers on the $y$-axis gives the victim's sensitivity to the enemy kernels. The victims are ordered along the $x$-axis by increasing sensitivity to write.

**Observation 11:** *All victim kernels are more sensitive to L2 cache evictions created by enemy write operations.*

The cause of higher victim sensitivities under writes may be twofold. First, prior work suggests that the GTX 1080 is less likely to evict L2 lines containing written data than lines containing other data [11]. If this is true, we expect the write enemy to maintain more of its lines in the cache at the victim's expense. Second, due to the write-back policy, victim cache misses may experience extra overhead due to having to write dirty lines back to DRAM.

The clear result from both the L2 and L1 cache experiments is that enemy kernels designed for interference in either cache should use write operations to access memory.

### E. Inter-SM: Beyond the L2 Cache

While the results given in Sec. IV-D focus on L2 cache evictions and the effects of write operations, they also produce interference in other shared components of the memory hierarchy. In Sec. II-A we discussed the DRAM row buffers, MSHRs, and memory buses as additional inter-SM interference channels. In this section we examine whether even more interference can be generated on these channels.

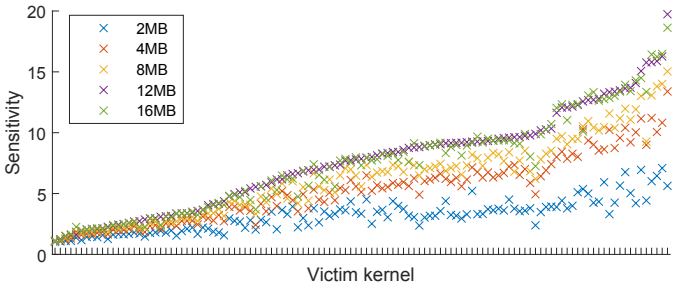


Fig. 8. Victim sensitivities to memory enemies with different array sizes, sorted by sensitivity to the 12 MB array enemy.

**Goal.** This experiment is designed to measure how interference in the memory hierarchy changes when enemies experience higher rates of L2 misses.

**Background.** The enemy kernels for cache evictions described to this point generate interference by probing memory arrays that match the size of the target cache (48 KB for L1, 2 MB for L2). However, if the size of this memory array increases relative to the cache size, the enemy will start to experience misses and evict its own cache data more frequently. While not necessarily evicting more victim data (a cache-sized array is already sufficient to evict all victim data in the cache), these extra misses may generate more interference in other parts of the memory hierarchy. Extra misses cause higher usage of the MSHRs, and extra evictions result in more accesses to the DRAM, each giving rise to more interference.

**Experimental setup.** We only used write operations in this new experiment, but we considered five enemies with different array sizes: 2 MB, 4 MB, 8 MB, 12 MB, and 16 MB. Otherwise, the enemies were identical to those from Sec. IV-D.

**Results.** Fig. 8 shows the sensitivities of each of the 117 victim kernels to the five enemies described above. Each victim kernel is represented by a point along the $x$-axis and the corresponding five markers on the $y$-axis give that victim's sensitivity to an enemy with a different array size. The victims are ordered along the $x$-axis by increasing sensitivity to an array size of 12 MB.

**Observation 12:** *Over 99% of victim kernels are more sensitive to the use of a 12MB array than to 2 MB, 4 MB, or 8 MB arrays.*

**Observation 13:** *35% of victim kernels are more sensitive to the use of a 16 MB array than to a 12 MB array, but only by an average ratio of 1.05:1.*

The results suggest that sensitivities to interference beyond the L2 cache increase with larger sizes of the array referenced by the enemy kernels, reaching a maximum at 12 MB or 16 MB. This shows that increasing the enemy cache miss rate is an effective method to generate interference on inter-SM channels beyond the L2 cache.

## V. Enemy Evaluations

In this section, we evaluate the enemy kernels we constructed for each of our victim kernels in both the SMK and spatial partitioning environments, and compare them to alternative choices of stress kernels from real-world code. We constructed per-victim enemies by applying the results from Sec. IV. Most enemy parameters can be set to values that are highly effective against all victims, but we still tested different values for compute/memory balance, which we found to be slightly more victim-dependent in our discussion of Fig. 6.

As in Sec. IV-C, these experiments involve an important caveat: *we chose to use an INT32-ADD compute enemy even though our experiments found an FP16-ADD enemy was more effective*. We did so to avoid taking advantage of what we consider to be an artifact of our platform. We ran these experiments primarily to test our methodology, so relying heavily on a platform-specific effect would provide little evidence that our techniques could apply to other platforms.

**Alternative stressor kernels.** Ideally, we hope that all victims are more sensitive to one of our constructed enemies than to any other real-world code. We evaluated whether this held for a small experiment using several alternative *stressor kernels*. Stressor kernels are simply real-world kernels that we can use in place of our constructed enemies. We chose six stressor kernels (three for each of our environments), based on two criteria. Each selected stressor is either **1)** a kernel from our own set of victims that we expect to be effective at causing stress, or **2)** a kernel used to cause stress in prior work.[14]

Of our two criteria, **1)** bears more explanation. We assume that a kernel that is *highly sensitive to stress* is one that *contends most strongly for resources*. In other words, we assume that our most-sensitive kernels are also more capable of causing interference themselves. After generating enemies for all of our victims in both environments, we selected the most-sensitive kernel from each environment to serve as an alternative stressor: `scalarProd` in the SMK environment, and the `ExecuteFirstLayer` kernel from `SqueezeNet` in the spatial partitioning environment.

For the remaining alternative stressors, we chose the `matrixMult` and `stereoDisparity` kernels for the SMK environment, along with `fwtBatch2Kernel` from `fastWalshTransform` and `vectorAdd` for the spatial partitioning environment. All of these have been used by prior work [11, 12] to evaluate GPU hardware contention.

**Enemy efficacy relative to alternatives.** Figs. 9 and 10 plot the ratio between each victim's sensitivity to its constructed enemy and its sensitivity to the alternative stressors. For example, if Fig. 9 shows a value of 5 for a given victim and the `stereoDisparity` stressor, that victim's sensitivity to our constructed enemy was five times higher than its sensitivity to `stereoDisparity` under the SMK environment.

Fig. 10 shows that in the spatial partitioning environment, our enemies were more effective than all three alternatives for 98% of victims. 39% of victims were over five times more sensitive to our enemies than any of the three alternatives.

[14]Note that we had to modify the original benchmarks' source code in order to launch these alternate stressor kernels with the block and thread organization required of our enemies.

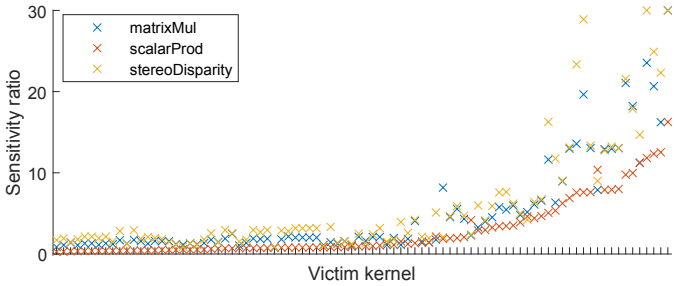


Fig. 9. Ratios between each victim's sensitivity to its constructed enemy and alternative stressors in the SMK environment. Note that the results contained three outliers with sensitivities of 32 (`matrixMul`), 45 and 49 (`stereoDisparity`), clamped to y=30 in this plot.

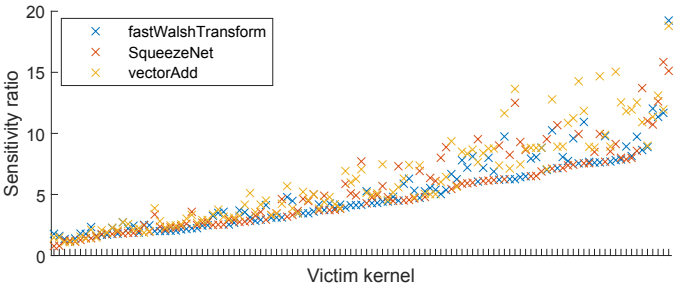


Fig. 10. Ratios between each victim's sensitivity to its constructed enemy and alternative stressors in the spatial partitioning environment.

In the SMK environment, our constructed enemies generated more stress than both matrixMul and stereoDisparity for 98% of victims. However, 52% of victims were more sensitive to `scalarProd`. This is seen towards the left of Fig. 9, where the sensitivity ratio between our constructed enemy and `scalarProd` is less than 1. As discussed in Sec. IV-B, we chose a smaller array size for intra-SM enemies (used in the SMK environment) in order to focus on maximizing L1 interference. This is reasonable when strictly focusing on *intra-SM* interference, but as we acknowledged in Sec. IV-E, a larger array may also cause more stress higher up the memory hierarchy. `scalarProd` has a 4 MB working set, significantly larger than the L1 data cache, so the additional stress it caused for some victims is likely due to this effect. We also note that an all-compute FP16 ADD enemy was able to generate more stress than `scalarProd` for 95% of victims.

## VI. Discussion

**Instruction cache.** Instructions are cached in a multi-level hierarchy with the first two levels private to each SM and a third level in the shared L2 unified data and instruction cache. Instruction caching can be an interference channel in two cases: cache hits causing contention for read cycles on the SM cache levels, and cache evictions causing miss latency on accesses. Only the first of these two sources of interference is present in our experiments in this paper.

Instruction-cache evictions by the enemy kernels are not present in our experiments because the memory footprints of any single benchmark kernel and an enemy fit easily in the per-SM instruction caches. Using NVIDIA's disassembler, `nvdisasm`, we inspected kernels' SASS bytecode and found the codes for eviction in our enemy kernels were all small (less than 1 KB), and victim kernels averaged about 3 KB. Even our largest victim kernels fit with an enemy in the 40 KB of instruction cache in each of the GTX 1080's SMs.

We have left the construction of enemy kernels and experiments to explicitly evaluate instruction caching to future work. Such work can adapt the cache-investigation techniques from prior work [41], creating instruction-cache enemies using no-operation (NOP) instructions to force capacity evictions in the SM caches (taking care that compiler optimization does not eliminate them). Using NOP instructions prevents the introduction of additional compute interference.

**Other limitations and future work.** Our objective was not to produce the most powerful enemy for every victim, but instead to provide a methodology for devising enemy stressors and for raising unique GPU-related issues, filling a crucial gap left in hardware documentation. Nonetheless, there are still limitations to the work presented here. First, we did not experiment with GPU architectures other than Pascal, though our approach should be portable to different GPUs. Second, time constraints prevented us from fine-tuning enemy parameters for each victim over the entire configuration space. We refer interested readers to [13] for methods that can be used to perform heuristic searches for powerful enemy parameters. Third, we did not explore the impact of kernel sensitivity to interference on schedulability analysis. Finally, we did not study the TLB as an interference channel due to the lack of documentation and mechanisms to manipulate it. All of these limitations remain potentially fruitful avenues for future work.

## VII. Conclusion

Our work furthers the development of reliable interference-aware measurement-based timing analysis for concurrent GPU kernels. In this work, we provided an experiment-driven method to construct GPU enemy programs, which attempt to produce worst-case contention on shared GPU resources. Our enemies target both inter-SM resources such as L2 cache and DRAM, and intra-SM resources such as L1 cache and computational units. By evaluating against a large number of real applications, we were able to empirically identify effective parameters for the enemies, including instruction types and memory access patterns. In addition, we independently analyzed the sensitivity of victims to specific interference channels. For the vast majority of victim applications, our constructed enemies produced more stress in comparison to baseline choices of stressing kernels, such as memory-bound or compute-bound kernels used in prior work.

In a thorough timing-analysis process, additional resources we chose not to consider should also be stressed (*e.g.*, per-SM instruction caches), additional resource-allocation choices should be examined (*e.g.*, additional per-SM warp-usage alternatives), and more execution samples would be required. We limited our choices in order for our experiments to remain tractable, but our goal with this work was not to provide definitive measurement-based timing analysis. Instead, the value of this work lies in presenting a *methodology* for examining these stress sources and highlighting which sources we see as being the most impactful.

## REFERENCES


[1] R. A. Cruz, C. Bentes, B. Breder, E. Vasconcellos, E. Clua, P. M. de Carvalho, and L. M. Drummond, "Maximizing the GPU resource usage by reordering concurrent kernels submission," *Concurrency and Computation: Practice and Experience*, 2019.

[2] FAA, "Multi-core processors," Online at https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/#software, 2016.

[3] K. Berezovskyi, F. Guet, L. Santinelli, K. Bletsas, and E. Tovar, "Measurement-based probabilistic timing analysis for graphics processor units," in *Springer International Conference on Architecture of Computing Systems (ARCS)*, 2016.

[4] K. Berezovskyi, K. Bletsas, and S. M. Petters, "Faster makespan estimation for GPU threads on a single streaming multiprocessor," in *IEEE Conference on Emerging Technologies & Factory Automation (ETFA)*, 2013.

[5] K. Berezovskyi, K. Bletsas, and B. Andersson, "Makespan computation for GPU threads running on a single streaming multiprocessor," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[6] V. Hirvisalo, "On static timing analysis of GPU kernels," in *International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2014.

[7] A. Betts and A. Donaldson, "Estimating the WCET of GPU-accelerated applications using hybrid analysis," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[8] K. Berezovskyi, L. Santinelli, K. Bletsas, and E. Tovar, "WCET measurement-based and extreme value theory characterisation of CUDA kernels," in *International Conference on Real-Time Networks and Systems (RTNS)*, 2014.

[9] A. Horga, S. Chattopadhyay, P. Eles, and Z. Peng, "Measurement based execution time analysis of GPGPU programs via SE+GA," in *Euromicro Conference on Digital System Design (DSD)*, 2018.

[10] J. Singh, "Toward predictable execution of real-time workloads on modern GPUs," Master's thesis, University of Illinois at Urbana-Champaign, 2021.

[11] S. Jain, I. Baek, S. Wang, and R. R. Rajkumar, "Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[12] S. K. Saha, "Spatio-temporal GPU management for real-time cyber-physical systems," Master's thesis, UC Riverside, 2018.

[13] D. Iorga, T. Sorensen, J. Wickerson, and A. F. Donaldson, "Slow and steady: Measuring and tuning multicore interference," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[14] NVIDIA Corporation, "NVIDIA Tesla P100 (white paper)," Online at https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[15] S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, "A unified WCET analysis framework for multi-core platforms," *ACM Transactions on Embedded Computing Systems (TECS)*, 2014.

[16] P. Puschner and A. Burns, "A review of worst-case execution-time analysis," *Real-Time Systems*, 1999.

[17] R. Wilhelm, S. Altmeyer, C. Burguière, D. Grund, J. Herter, J. Reineke, B. Wachter, and S. Wilhelm, "Static timing analysis for hard real-time systems," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2010.

[18] Z. Guo, K. Yang, F. Yao, and A. Awad, "Inter-task cache interference aware partitioned real-time scheduling," in *ACM Symposium on Applied Computing (SIGAPP)*, 2020.

[19] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical OS-level cache management in multi-core real-time systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[20] D. B. Kirk, "SMART (strategic memory allocation for real-time) cache design," in *IEEE Real-Time Systems Symposium (RTSS)*, 1989.

[21] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[22] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[23] C.-G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Bounding cache-related preemption delay for real-time systems," *IEEE Transactions on Software Engineering*, 2001.

[24] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla, "On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2012.

[25] M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla, "Assessing the suitability of the NGMP multi-core processor in the space domain," in *ACM International Conference on Embedded software*, 2012.

[26] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *IEEE European Dependable Computing Conference (EDCC)*, 2012.

[27] M. Bechtel and H. Yun, "Denial-of-service attacks on shared cache in multicore: Analysis and prevention," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[28] P. Voudouris and G.-J. van den Braak, "Analysis and modeling of the timing behavior of GPU architectures," Master's thesis, Eindhoven University of Technology, 2014.

[29] H.-E. Zahaf, I. S. Olmedo, J. Singh, N. Capodieci, and S. Faucou, "Contention-aware GPU partitioning and task-to-partition allocation for real-time workloads," in *International Conference on Real-Time Networks and Systems (RTNS)*, 2021.

[30] T. Allen, X. Feng, and R. Ge, "Slate: Enabling workload-aware efficient multiprocessing for modern GPGPUs," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[31] N. Otterness, M. Yang, S. Rust, E. Park, J. Anderson, F. Smith, A. Berg, and S. Wang, "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[32] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and H. Jeon, "Tango: A deep neural network benchmark suite vor various accelerators," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.

[33] V. Prisacariu, I. Reid *et al.*, "fastHOG-a real-time GPU implementation of HOG," *Department of Engineering Science*, 2009.

[34] NVIDIA Corporation, "Multi-process service," Online at https://docs.nvidia.com/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2020.

[35] M. Yang, N. Otterness, T. Amert, J. Bakita, J. H. Anderson, and F. D. Smith, "Avoiding pitfalls when using NVIDIA GPUs for real-time tasks in autonomous systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

[36] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[37] I. S. Olmedo, N. Capodieci, J. L. Martínez, A. Marongiu, and M. Bertogna, "Dissecting the CUDA scheduling hierarchy: A performance and predictability perspective," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[38] NVIDIA Corporation, "Achieved occupancy," Online at https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm, 2015.

[39] NVIDIA Forum, "Pascal l1 cache," Online at https://forums.developer.nvidia.com/t/pascal-l1-cache/49571, 2017.

[40] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2020.

[41] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the NVIDIA Turing T4 GPU via microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.