Guardian: Data Isolation for Multi-Tenant GPU Sharing

Anonymous Author(s)
Submission Id: <PAPER ID>

Abstract

Modern applications, such as machine learning frameworks, can only partially utilize beefy GPUs, leading to GPU underutilization in cloud environments. Sharing GPUs across multiple applications from different users can improve resource utilization and consequently cost, energy, and power efficiency. However, GPU sharing creates memory safety concerns because kernels need to share a single GPU address space (common GPU context). Previous GPU memory protection approaches have limited deployability because they require specialized hardware extensions or source code, which is not available in GPU-accelerated libraries heavily used from ML frameworks.

In this paper, we present Guardian, a PTX-level bound checking approach for GPUs that limits GPU kernels of each application to stay within the memory partition allocated to them. Guardian relies on three mechanisms: (1) It divides the common GPU address space into separate partitions for different applications. (2) It intercepts and checks data transfers, fencing erroneous operations. (3) It instruments all GPU kernels at the PTX level –available in closed GPU libraries– fencing all kernel memory accesses outside application memory bounds. We implement Guardian as an external, dynamically linked library that can be pre-loaded at application startup time. Guardian's approach is transparent to applications and can support real-life, complex frameworks, such as Caffe and PyTorch, that issue billions of GPU kernels. Our evaluation show that the overhead of Guardian for such frameworks is between 4% and 12% (on average 9%), compared to native.

1 Introduction

Graphic Processing Units (GPUs) have become necessary for accelerating various applications, including machine learning (ML) and deep learning [17, 21, 45, 58]. Over the years, the compute and memory resources per GPU have been increasing to meet increasing processing demands [5, 6, 63, 64]. As a result of this trend towards "beefier" GPUs, individual applications and, even more so, individual GPU kernels often fail to fully utilize the available resources, leading to GPU underutilization [5, 29, 37, 38, 41, 53, 64, 65]. Sharing GPUs among applications of different users can improve resource utilization. GPUs today already support time-sharing, which allows switching from one application to another. Time-sharing involves costly context switches at the GPU level [29, 42, 61] and is not effective in cases where the application executing cannot utilize fully or adequately accelerators [5, 41].

To address these limitations, several approaches propose spatial sharing mechanisms [10, 20, 38, 46, 56, 57, 62, 65]. Spatial sharing requires a single GPU context, i.e., a common GPU address space, to execute kernels from different applications (and users) concurrently. However, this approach introduces a significant concern: GPU kernels execute in the same GPU address space, hence they can modify (either inadvertently or deliberately) memory locations that belong to other applications [5, 11, 27, 28, 31, 43, 47] leading to memory violations. This lack of memory protection makes spatial sharing impractical for multi-user environments executing various GPU applications.

There are two types of memory protection mechanisms for GPU sharing: hardware- [5, 37] and software-based [65]. Hardware-based approaches do not require source-code due to the use of a special unit dedicated to perform security checks. However, this extra/special hardware unit limits the applicability/deployability of such approaches. This problem is further exacerbated by the fact that GPU drivers in the operating system are closed-source. On the other hand, current software-based approaches [65] are easier to deploy but require source code, making them impractical in real-life GPU applications that do not (always) provide host and kernel source code. The source code is required to port applications to customized abstractions [65] or to perform static analysis [44]. Static analysis is crucial to minimize the number of conditional checks in the GPU kernel, thus reducing the cost of *address checking*. An alternative method is *address fencing* (sandboxing) [1, 14, 24, 34, 40] that applies bit-masking instructions before every `load` or `store`. Unlike address checking, fencing enforces memory references within specified bounds, resulting in minimal overhead. However, it lacks the capability to identify which references exceed these bounds. Software-based approaches have the potential to provide a viable protection solution if they overcome the requirement for source-code.

The majority of real-life GPU applications and frameworks [21, 45] rely heavily on domain-specific GPU libraries, such as cuBLAS, rocBLAS, cuDNN, oneDNN, and cuFFT [54]. A significant portion of these libraries do not provide source code (closed-source) limiting the applicability of software based approaches. GPU closed-source libraries include host code (e.g., allocating memory and invoking kernels) and device code (GPU kernels). The code of GPU kernels is only available in virtual assembly (PTX [39]) or in binary files (cuBIN [54])). Furthermore, each call to a high-level function of a GPU closed-source library involves host calls that are hidden from the developer. For instance, `cuBLAS` contains about 4000 distinct GPU kernels in PTX, while a single function call, such as `cublasIsamax()`, performs more than fifteen CUDA calls, including memory operations (`cudaMalloc()`, `cudaMemcpy()`), and kernel invocations (`cudaLaunchKernel()`). The existence of closed-source libraries makes protected GPU sharing more challenging.

In this paper, we propose Guardian, a PTX-based bound checking approach that provides transparent memory protection for spatial GPU sharing. Guardian divides the GPU memory into partitions assigned to different applications. It fences the GPU kernels, even if the source code is unavailable, using address fencing or address checking instructions at the PTX level. It is implemented as a dynamically loadable library; thus, it is completely transparent to ML frameworks, such as PyTorch and Caffe, without requiring source code modifications, compilation, or extra hardware. Guardian effectively addresses two main challenges, as follows.

Support GPU applications without requiring source code or extra hardware. High-level function calls (e.g., `cublasIsamax()`) of closed-source NVIDIA libraries perform CUDA runtime API calls that are not visible to the developer. Guardian intercepts allocation, copy, and kernel launch related calls, transparently at the CUDA runtime library level. To achieve that, Guardian uses a dynamically linked runtime library that is preloaded during application execution. Our approach requires the application to be dynamically linked with CUDA runtime/driver and statically linked with closed-source GPU libraries, however, this is usually not a significant concern. The intercepted calls are forwarded to the GPU manager, which runs as a separate trusted process (and address space) and is the only entity with access to the GPU. Furthermore, closed-source GPU libraries contain the GPU kernel code in PTX format. During an offline phase, Guardian extracts and instruments – add bound-checking instructions before every load/store– the kernel code at the PTX level of ML frameworks and closed-source GPU libraries. At runtime, Guardian replaces all kernel invocations with their instrumented counterparts. For each kernel of an application Guardian passes the base address and the mask of the partition assigned to that particular application. To divide the GPU memory into logical partitions, Guardian uses a custom allocator. As a result, all allocation calls are served from the application's partition. Finally, Guardian checks every host-initiated transfer against the application's assigned partition boundaries.

Lightweight bound checking. Address checking (conditional checks) is more powerful because it can detect illegal accesses, however according to our findings, it is much more expensive than address fencing (bitwise OR - AND). As a result, Guardian uses address fencing, which requires power-of-two block size partitions. Popular frameworks such as TensorFlow and PyTorch use similar allocation policies; thus, we choose to optimize the common case. However, Guardian GPU manager can utilize both mechanisms transparently to serve different needs. Performing just-in-time compilation at runtime to convert a sandboxed PTX to a binary is expensive, thus the GPU manager, during its initialization, compiles the sandboxed kernels. Finally, address fencing requires fewer extra registers than address checking, reducing the resource consumption implied by our sandboxing mechanism.

We implement Guardian for NVIDIA GPUs and evaluate it with several micro-benchmarks and real-life ML applications (Caffe and PyTorch) that link with closed-source GPU libraries. For Caffe and PyTorch, which invoke billions of GPU kernels, Guardian address fencing has on average 9% overhead compared to native unprotected execution, whereas address checking is 1.7× worse than native. Guardian (protected) spatial sharing is 4.84% slower than MPS (unprotected). At the same time, it improves the total execution time of co-located applications by 37% compared to time-sharing the alternative sharing and protection mechanism used from other systems [25, 59, 64]. Finally, Guardian imposes minimal increase in register usage, and thus register spilling occurs only in 0.9% of PyTorch kernels.

The main contributions of this paper are:

- We design, implement, and evaluate Guardian, a novel system that offers *transparent* memory protection for applications executing concurrently on a GPU without relying on hardware support nor the existence of source code. We demonstrate its effectiveness using a broad range of kernels and complex, real-life ML frameworks (Caffe and PyTorch) that extensively use closed-source GPU libraries.
- We present a mechanism to intercept all GPU-related calls at the CUDA runtime and driver library level. This allows transparently tracing and monitoring any GPU application or closed-source GPU library.
- We present PTX-level address fencing for memory bounds checking under sharing, and we find that it is highly efficient for GPU kernels compared to other software- or hardware-based approaches. Compared to CPUs, GPU address checking has lower overhead because GPU kernels have simpler access patterns.

2 Background

In this section, we discuss four aspects of NVIDIA GPUs that are related to our work. We believe that AMD and Intel GPUs have similar architectural characteristics [27].

2.1 GPU Context

All actions performed from the CUDA runtime and driver interfaces, including memory allocation, data transfers, and

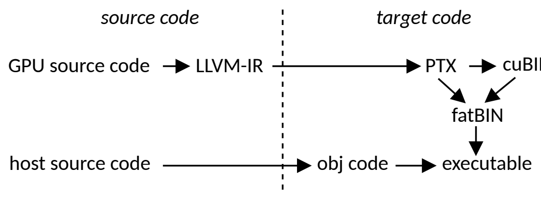


Figure 1. Compilation flow of CUDA applications.

kernel launch, are encapsulated in a CUDA context on the GPU itself. A GPU context is similar to a CPU process. As such, each CUDA application creates its own context during the first CUDA runtime call. The context contains all the information regarding the resources used by an application, such as GPU memory, streams, cores, page table, and the GPU kernels to be used. An application that executes in a context cannot access memory locations used by an application in a different context. At any point in time, the GPU can execute multiple kernels on different streams, however, all kernels must belong to a single context. The GPU allows different contexts to time-share resources using a context switching mechanism. GPUs support preemption, in which case the state of a GPU context is swapped to GPU DRAM so that another context can be swapped in and run. However, context switching does not allow different applications to spatially share the same GPU.

2.2 GPU Compilation Workflow

CUDA applications consist of host-level (`.cpp`) and GPU-level source code (`.cu`). The host source code is compiled with `clang` or `gcc`, while GPU (device) code with `nvcc`. As Figure 1 shows, the device source code is converted to LLVM-IR format, which is then compiled, via `cicc`, to Parallel Thread eXecution (PTX) [39] assembly.

PTX is a virtual assembly specification supported by the NVIDIA toolchain in all NVIDIA GPU architectures, past and future. The PTX code can be just-in-time compiled by the CUDA driver and executed on the target device, at runtime. This allows the generation of forward-compatible optimized machine code that runs on the target device. The latter can be explicitly enforced by setting the environment variable `CUDA_FORCE_PTX_JIT`: the device driver ignores any `cuBIN` files embedded in an application or CUDA library and just-in-time compiles the embedded PTX code instead.

The `nvcc` compiler embeds the PTX representation of the device code in the target applications or libraries. Besides that, the compiler also generates the machine code for specific GPU architectures (using the `ptxas` assembler), which also embeds in the target application, in the form of `cuBIN` files. The developer can specify during compilation (`sm` flag) the target architectures for which `cuBIN` files should be generated and included in the target application. The generated PTX code and the `cuBIN` files are merged in a `fatBIN` file.

CUDA version	**NVIDIA GPU architecture**		
	Turing (7.5)	**Ampere (8.0-8.7)**	**Hopper (9.0)**
10.0-10.2	PTX		
11.0-11.7.1	cuBIN	PTX	
11.8-12.0	cuBIN	cuBIN	PTX

Table 1. cuBIN and PTX kernel code included in CUDA-accelerated libs for different CUDA versions and GPUs.

CUDA closed-source libraries also contain the GPU kernels in PTX and `cuBIN` files. As shown in Table 1, a CUDA library of a particular CUDA version contains the kernels in PTX format for the most recent GPU architecture and for all previous architectures the kernel code in cuBIN files. For instance, CUDA 11.7.1 is the most recent CUDA SDK version for Ampere architecture, hence, CUDA libraries of this CUDA version contain the cuBIN files for all previous architectures (Turing) and PTX to support Ampere and Hopper.

2.3 GPU Memory Sharing Scope

GPU memory is divided into *on-chip* and *off-chip* [6, 27, 52]. On-chip consists of the register files and the shared memory. However, only off-chip memories can be accessed from co-running kernels [11, 27, 65]. For NVIDIA GPUs, Off-chip DRAM is divided into local, heap, global, constant, and texture memory. The local memory (stack) is located off-chip and is mainly used if the variables used from a kernel exceed the number of available registers (i.e., register spilling). However, GPU programs and compilers aim to minimize register usage, avoiding register spilling since it degrades performance. Heap memory is allocated and deallocated by kernels using `malloc` and `free` and is not accessible through host-side CUDA calls, e.g., `cudaMemcpy`. However, heap memory is rarely used because in-kernel allocations imply large overheads, up to 63× [27] compared to allocations in global memory using `cudaMalloc`.

Global memory is managed dynamically from the host (e.g., using `cudaMalloc` and `cudaFree` functions), or statically from the device (using the `.global` keyword). A CUDA kernel uses load and store instructions to access data in global memory. Constant and texture memory is a read-only part of the global memory space. According to our findings, the use of these memories is extremely rare in ML applications, hence we can ignore it for protection purposes.

Unified memory is introduced to reduce the programmer's burden to transfer data from/to the host memory explicitly. Instead, data is transferred automatically from the host memory in page granularity by the page fault handler in the GPU driver [27] and the IOMMU [5]. CUDA kernels access such data using the same load and store instructions as if data were in global memory.

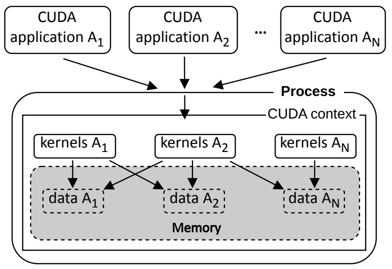


Figure 2. Multi-tenant spatial GPU sharing, without Guardian. The common GPU context that is required for spatial sharing allows different applications to access each others memory.

3 Threat Model

Our work considers memory safety across kernels from different applications or users that share a GPU spatially in cloud or shared environments. In particular, Guardian prohibits applications from different users to read or modify each other's data in the host or device memory. Within the realm of GPU security concerns, our primary emphasis is on memory safety, as it represents a prominent threat to the integrity of GPU spatial sharing.

We consider all GPU kernels unsafe, provided by individual users or from GPU libraries. As a result, any instruction that performs loads or stores from a base address fetched from a destination register is considered unsafe and should be protected via bounds checking. Branch instructions are considered safe because they jump to labels defined inside a PTX file. The assembler will report errors if the labels are absent from the PTX file or they are incorrect. On the other hand, *indirect* branch instructions (`brx.idx`) are unsafe because they use a register to index an statically defined array of labels. The register employed for indexing cannot be validated at compile time, potentially leading to out-of-bounds accesses.

Our threat model assumes that the GPU driver and the GPU device are trustworthy and reliable. Consequently, security issues related to exploiting GPU resource contention or side-channels [61] or denial-of-service [32] are outside the scope of this paper. Finally, we do not consider data confidentiality and integrity issues and cases where an adversary (1) can control the entire software system (e.g., hypervisor) and (2) has physical access to all server hardware.

4 Guardian Design

The goal of Guardian is to prevent applications of different users from reading or modifying each other's data when executing concurrently on the same GPU. Spatial GPU sharing requires a common CUDA context to execute kernels from different applications concurrently. Previous work [18, 38, 46, 65] uses a separate process that creates that single context. Applications issue all their GPU tasks to this process,

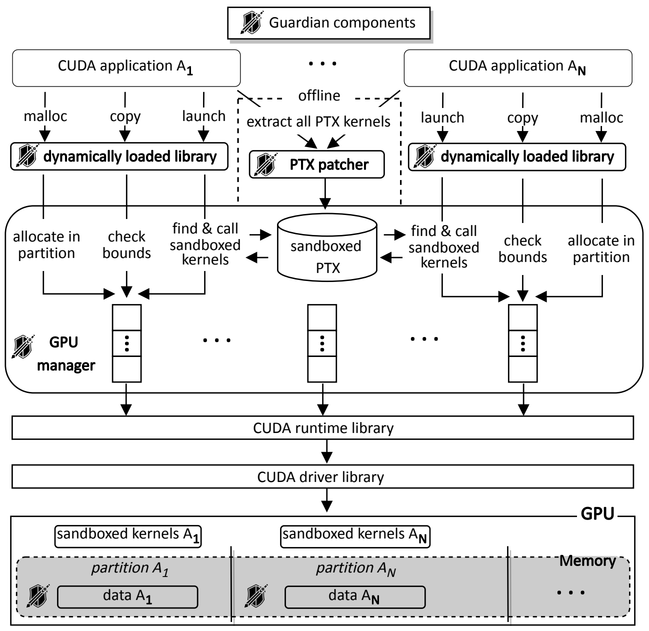


Figure 3. Guardian online and offline (dashed annotated) mechanisms to allow protected spatial GPU sharing. Guardian intercepts the CUDA runtime interface used from applications and perform the necessary checks at memory allocations, transfers, and kernel executions. This allows kernels from different applications to execute concurrently on different memory partitions, eliminating illegal accesses.

which enqueues to different streams, thus kernels can be executed concurrently. However, without proper protections, this approach allows GPU kernels to modify (either inadvertently or deliberately) memory locations belonging to other applications, as shown in Figure 2. Guardian uses a trusted process, namely GPU manager, responsible for performing allocations, transfers, and kernel invocations on behalf of the applications. GPU manager enables concurrent execution of these operations with protection guarantees.

To support applications that do not contain source code, Guardian uses three main mechanisms shown in Figure 3 and described in more detail below: (a) The dynamically loadable library (§4.1), (b) the GPU manager (§4.2), and (c) the PTX patcher (§4.3). We use address fencing to provide a lightweight bound checking mechanism (§4.4).

4.1 Dynamically Loadable Library

Guardian ensures that all GPU calls related to allocations, copies, and kernel launches are intercepted by a dynamically linked library. Guardian's dynamically loaded library is preloaded to the application and replaces the default CUDA runtime. The CUDA runtime and driver interfaces are the lowest levels that provide generic CUDA calls for managing GPU resources, such as allocating memory and launching kernels. CUDA applications, frameworks, and CUDA-accelerated libraries use mainly [13, 15] the CUDA runtime interface and, to a lesser extent, the CUDA driver interface. The CUDA driver interface is used by applications only for specific, lower-level operations, such as explicit PTX (un)loading and context management.

We find that shared CUDA-accelerated libraries include a statically linked version of the CUDA runtime library. Therefore, it is impossible to intercept CUDA runtime calls when applications use the shared version of CUDA-accelerated libraries. On the other hand, the static version of CUDA-accelerated libraries (e.g., `libcudblas_static.a`) uses the shared version of the CUDA runtime library. Consequently, Guardian requires applications to be linked with the static version of CUDA-accelerated libraries to be able to intercept CUDA runtime calls using its dynamically loaded library.

By intercepting the CUDA runtime, Guardian is able to handle CUDA-accelerated libraries transparently. Such libraries provide high-level domain-specific functions, that use the CUDA runtime library implicitly. They also include kernels that they invoked for execution in the GPU. According to our findings high-level functions invoke several CUDA runtime calls, including allocations, transfers, and kernel launches. For example, a single cuBLAS function, such as `cublasIsamax()`, can invoke implicit several CUDA runtime calls, such as `cudaMalloc()`, `cudaMemcpy()`, and kernel launches via `cudaLaunchKernel()`. These high-level function calls cannot be verified that they access the correct partitions without intercepting any implicit CUDA call. The intercepted CUDA calls are transferred to the GPU manager, that performs the appropriate operations as we discuss next.

4.2 GPU manager

The GPU manager allocates GPU a memory partition per application (§ 4.2.1), fences memory transfers (§ 4.2.2), calls the sandboxed GPU kernels (§ 4.2.3), and allows multi-tenant GPU sharing (§ 4.2.4) using the mechanisms described below.

4.2.1 GPU Memory Partitioning. The virtual memory of the GPU is managed through the `cudaMalloc()`-family functions, which return arbitrary addresses upon each call. To perform GPU memory isolation across different applications, Guardian uses a custom allocator that initially reserves all available GPU memory and splits it into partitions. Each partition is a contiguous memory block assigned exclusively to an application (or tenant). Having contiguous memory for each application enables Guardian to protect memory by simply checking that memory accesses are always within this partition's boundaries.

Guardian intercepts `cudaMalloc()` (malloc in Figure 3) and returns a pointer in each application's partition. As a result, this procedure is transparent to the application. Similarly, our allocator marks the region in a partition as free by intercepting the `cudaFree()` function. For each application, we keep the application id, the base address, and the partition size in a *partition bounds table* used in the online phase. Currently, Guardian partitions GPU memory statically; hence, each application needs to specify its maximum memory requirements at initialization. Though this is sufficient for the applications and workloads we examine, it is interesting for future work to explore dynamic partition resizing.

4.2.2 Data Transfers. Data transfers include operations that move data between the host and the GPU memory (e.g., `cudaMemcpyH2D()`) or within the GPU memory (e.g., `cudaMemcpyD2D()`). Even though these calls are initiated from the applications, they still refer to the same GPU address space; hence applications could still perform memory operations to partitions of other applications. Guardian dynamically loadable library intercepts the memory management CUDA runtime calls (copy in Figure 3) and uses the *partition bounds table*, maintained by Guardian allocator, to verify that the memory ranges are within the correct partition. Guardian allows a transfer to complete if it satisfies two conditions: (a) The transfer's pointer is between the partition base pointer and the partition base pointer plus the partition size. (b) The destination pointer plus the transfer size does not exceed the partitions ba se pointer plus the partition size. For `cudaMemcpyH2D()` our mechanism checks the destination pointer, for `cudaMemcpyD2H()` it checks the source pointer, and for `cudaMemcpyD2D()` it checks both.

4.2.3 GPU Kernel Invocation. As shown in Figure 3, Guardian intercepts each kernel invocation via `cudaLaunchKernel()` and executes the corresponding sandboxed kernel instead. To achieve that, Guardian interposes `cudaRegisterFunction()` that is called implicitly by the CUDA driver during application initialization. This function is used to register and make every kernel from a CUDA application callable. The GPU manager creates a new `CUmodule` for each PTX exported and patched at the offline phase (§4.3). A `CUmodule` is a CUDA code (PTX or cuBIN) unit that can be dynamically loaded and executed on the GPU. The `CUmodules` are then loaded into the current context using `cuModuleLoadData()`. A `CUmodule` can contain more than one kernel, hence we use `cuModuleGetFunction()` to create a `CUfunction` handle for each kernel. The `CUfunction` handles are stored in a map, called *pointerToSymbol*, which is used at every kernel invocation to find the kernel needed for execution. In particular, every time a kernel is invoked for execution (through `cudaLaunchKernel()`), Guardian performs a lookup at the *pointerToSymbol* table to find the `CUfunction` handle of the corresponding sandboxed kernel. Then, it adjusts the number of parameters accordingly; for address fencing, it passes the mask and the base partition address (§4.3), whereas for address checking the partition base and ending addresses.

```
__global__ void kernel(int *A, int j){
  int tid = threadIdx.x;
  A[i] = j;
}
```

Listing 1. Sample CUDA kernel source code.

Each partition's information (base address, mask or end address) is retrieved through the *partition bound table*. Finally, the GPU manager issues the sandboxed kernel using `cuLaunchKernel()`. When the GPU manager detects that an application runs standalone, it selects to issue a native kernel, avoiding the overhead implied by the extra instructions.

4.2.4 Spatial Multiplexing.
To enable spatial sharing, GPUs require a single context and CUDA streams provided by the GPU manager as in previous works [38, 46, 65]. As a result, applications do not create their own context; instead, they funnel their work to the GPU through the context of the GPU manager. All CUDA kernels and data transfers originating from a single application will be executed in-order from the GPU manager. In contrast, kernels and data transfers from different applications will be executed concurrently using different streams. Applications and the GPU manager run in different address spaces. As a result, we use an IPC channel and a separate shared memory segment between applications and the GPU manager. The channel is used to transfer all operations that are intercepted, and the shared memory segment is used to exchange the data used in `cudaMemcpy()`-family functions, similar to other API remoting approaches [18, 33, 46, 65].

4.2.5 Distributed Machine Learning.
In cases where ML applications span to multiple GPUs across different servers, the Guardian GPU manager runs in each server. As a result, the overhead of each protected and shared GPU is the same as in the single node.

4.3 Offline Kernel Sandboxing

During the offline phase, the Guardian patcher uses `cuobjdump` [35] to extract any embedded PTX kernel from the application executable and the CUDA libraries (offline in Figure 3). The extracted PTX kernels are then sandboxed to ensure they do not access data outside the correct partition boundaries. Listing 2 shows the sandboxed PTX code of the original kernel shown in Listing 1. The original PTX (without sandboxing) consists of a kernel function definition that includes a list of parameters –lines 2 and 3. These parameters are addressable, read-only variables declared in the `.param` state space. Parameters are loaded to registers using `ld.param` instructions – lines 11 and 12. Each kernel allocates the minimum number of registers used throughout the execution –lines 9 and 10. Then the kernel uses these registers to load and store the values generated in each execution step –lines 20-23 and 30-31.

```
1   .visible .entry kernel(
2   .param .u64 kernel_param_0,
3   .param .u32 kernel_param_1,
4   // Base address
5   .param .u64 kernel_base,
6   // Mask parameter
7   .param .u64 kernel_mask)
8   {
9     .reg .b32       %r<3>;
10    .reg .b64       %rd<5>;
11    ld.param.u64    %rd1, [kernel_param_0];
12    ld.param.u32    %r1, [kernel_param_1];
13
14    // Extra registers for base and mask
15    .reg .b64       %grdreg<3>;
16    // Load extra parameters to registers
17    ld.param.u64    %grdreg1, [kernel_base];
18    ld.param.u64    %grdreg2, [kernel_mask];
19
20    cvta.to.global.u64      %rd2, %rd1;
21    mov.u32         %r2, %tid.x;
22    mul.wide.s32    %rd3, %r1, 4;
23    add.s64         %rd4, %rd2, %rd3;
24
25    // Bit-wise And with mask
26    and.b64         %rd4, %rd4, %grdreg2;
27    // Bit-wise OR with base addr.
28    or.b64         %rd4, %rd4, %grdreg1;
29
30    st.global.u32   [%rd4], %r2;
31    ret;
32  }
```

Listing 2. Sample sandboxed PTX CUDA kernel. Guardian address fencing instructions are explained with comments.

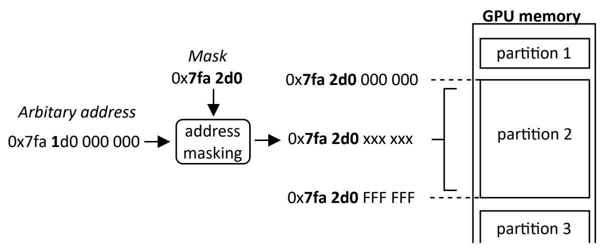


Figure 4. Bitwise instructions mask addresses that fall outside a partition.

Our patcher (1) adds two extra parameters in each kernel –lines 5 and 7, (2) defines two extra registers to load the mask and the base partition address parameter –line 15, (3) loads the extra parameters in the registers –lines 17-18, and (4) appends two `bitwise` instructions –lines 26 and 28– before every load/store. The `bitwise AND` operation is performed between the load/store address and the mask. The mask for each partition is calculated using the highest address and the partition size. For instance, if the partition starting address is 0x**7fa2d0**000000 and the partition size is 16 MB: the ending address is 0x**7fa2d0**FFFFFF and the mask is 0x**000000**FFFFFF (partition 2 in Figure 4). In any case, the number of zeros in the mask depends on the partition size. Then we use a `bitwise OR` between the address and the base address of a partition. The `bitwise AND` with

```
//A. Full virtual address (similar for store)
ld.global/shared/local %val, [base_addr];

//B. Base address+offset (similar for store)
ld.global/shared/local %val, [base_addr+offset];
```

Listing 3. GPU memory addressing modes.

the mask and the `bitwise OR` with the partition base address make an address outside the partition to start from the begging of the partition, i.e., wrap around, as shown in Figure 4. The illegal address that points to partition 1 (assigned to another application) due to the bitwise operations with the masking address, will finally point to partition 2. With this approach, only invalid or malicious kernels will wrap around and potentially corrupt their data. If this data corruption leads to infinite execution (e.g., no convergence in ML applications), our GPU manager can kill the application. Alternatively, Guardian offers address checking (§4.4) to detect invalid accesses, but at a higher cost (§6.2).

Intel, AMD, NVIDIA GPUs have two addressing modes for loading or storing data to memory [27, 39], as shown in Listing 3. In the first case, the base address is loaded into the destination register, while in the second, an offset is first added to the base address, and the result is loaded into the destination register. The same modes apply to stores and all off-chip memories. The patcher applies the bit-masking instructions directly to the base address for the first mode. For the second mode, the patcher calculates the new address by adding the offset to the base address and stores this in a new temporary register. Then, it applies the masking instructions to this new address, preventing out-of-bound access. Our patcher instruments `.func` in the same way as kernels (`.entry`). The `.func` directive denotes a function callable from host code and other kernels using `call` instruction.

Indirect branch instructions are unsafe, but according to our findings, they do not exist in PyTorch kernels. Even if indirect branches are rare, Guardian can protect those instructions by applying a modulo operation to the index relative to the array's size, causing the index to wrap around.

4.4 Sandboxing Optimizations

In our initial design, we use *conditional checks* (address checking) to verify if a load/store address is inside a partition's lower and upper address. Address checking, different from address fencing, does not require the partition to be aligned in the power of two and detects an out-of-bound access. However, branches in GPUs are executed by the Address Divergence Unit (ADU), which requires more than 80 cycles. To reduce this overhead, we use modulo: $fenced\_addr = partition\_base + ((arbitary\_addr - partition\_base)\%partition\_size)$. CUDA ISA does not provide an instruction for modulo (between 64-bit numbers); instead, it is emulated via a function call, which implies more than 2x overhead compared to native. We implement the modulo (inline) with three

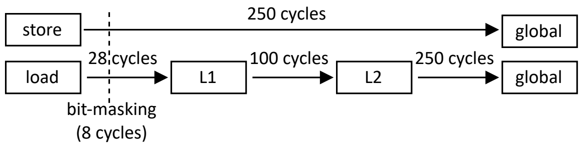


Figure 5. Bit-masking latency (8-cycles) compared to latency of different memories.

instructions and an extra parameter holding the $\frac{1}{parttition\_size}$. The extra parameter is used to avoid the division's high overhead since it is also emulated via a function call. This approach requires 28 cycles for the seven extra instructions.

Address fencing requires almost 8 cycles –4 cycles per bitwise operation [2]. As shown in Figure 5, a load/store instruction requires 28 cycles if the data reside in L1-cache, whereas if the data are in global memory, it requires 220-350 cycles [6, 22]. In the rare case that all the data are in L1-cache (100% cache hit ratio), our approach implies 30% overhead, whereas in the typical case (data in global memory), we add on average 3.5% (§6.5). Our approach requires the partitions to be in power of two to cut down the extra instructions required to check a partition's upper and lower bound. The power-of-two block size allocators restrict the number of concurrent applications, however PyTorch and TensorFlow use this type of allocator as default. Consequently, we choose to optimize the common case and leave the allocation issue as future work. Furthermore, Guardian can dynamically change to the address checking mode that overcomes this issue either using conditional checks or inline modulo.

Guardian passes the mask and the base partition address to every kernel using two extra parameters. Using these parameters inside the kernel requires two extra registers. The extra registers that our approach requires do not lead to register spilling (§6.3), because GPU kernels use the minimum number of registers and the nvcc compiler optimizes even more the register usage [19, 30, 47]. We have examined two other possible solutions before using the two additional parameters. The first was a global map stored in GPU memory, but updating the map is prohibitively expensive. The second was to generate a different kernel binary for every partition with the mask hard-coded. This approach does not scale when multiple applications use thousands of kernels. Using just-in-time compilation to avoid pre-compiling kernels induces considerable overhead. Guardian GPU manager compiles at its initialization the sandboxed PTX with the extra parameters avoiding JIT.

5 Experimental Methodology

Our evaluation tries to answer the following questions:

- What is the impact of Guardian on GPU spatial sharing compared to other sharing approaches (§6.1)?

Specifications	**RTX A4000**	**RTX 3080 Ti**
Compute Capability	8.6	8.6
#SMs	48	80
#CUDA cores	6144	10240
L1 (KB)	128	128
L2 (KB)	4096	6144
Global memory (GB)	16	12
#Registers / Thread	255	255
PCIe	v4 x16	v4 x16
L1 hit latency (cycles)	28 [6, 22]	28 [6, 22]
L2 hit latency (cycles)	193 [6, 22]	193 [6, 22]
Global memory BW (GB/s)	448	912
Error Correction Code	Yes	No

Table 2. GPU specifications we use for the evaluation.

- What is the overhead of Guardian on real-life applications –running standalone– compared to native execution and other protection approaches (§6.2)?
- What is the impact of address fencing on GPU register usage (§6.3)?
- What is the performance of CUDA call interception (§6.4), and address fencing at high cache hit ratios (§6.5) using different GPUs and access patterns (§6.6)?

Server infrastructure: To evaluate Guardian we use two GPU models (Table 2), that are installed on two different servers. The first server is equipped with a Quadro RTX A4000 GPU, four AMD EPYC 7551P NUMA CPUs with 8 physical cores each (running at 3.0 GHz, hyper-threaded), and 128 GB of DRAM. To avoid passes over QPI/UPI, we pin applications to the cores closer to the GPU. The second server contains a GeForce RTX 3080 Ti, an Intel(R) Core i7-8700K CPU with 6 cores running at 3.70 GHz, and 32 GB of DRAM. Both servers have NVIDIA CUDA v11.7 with NVIDIA driver v.515 installed. All the experiments, except §6.6, are performed in the Quadro RTX A4000. Regarding GPU kernel scheduling, we use NVIDIA's default policy (leftover).

Workloads and datasets: To evaluate the overheads of Guardian under *real-world* scenarios, we use multiple neural networks from Caffe [21] and PyTorch [45] frameworks with large data sets that invoke billions of kernels and execute for hours and applications from the Rodinia benchmark suite [8]. Regarding ML applications, we run lenet, siamese, computer vision, and rnn neural networks with the mnist dataset [26], while for cifar10, the cifar dataset [23]. Both mnist and cifar datasets contain hundreds MBs of images. All the above neural networks are executed with 100 up to 500 epochs and invoke up to 142 million CUDA kernels. We also run experiments with imagenet dataset [48], which consists of 256 GB of images, using googlenet, alexnet, caffenet, vgg11, mobilenetv2, and resnet50 as neural networks. These networks invoke billions of kernels, and we run them for ten epochs leading to 99% accuracy. Table 3 shows the total number of kernels, functions, and the load/store (`ld`/`st`) instructions contained in the libraries and frameworks that we use in our evaluation. Regarding Rodinia [9], we increase the default dataset size by 10× and kernel execution time by 8×, compared to previous work, because the default values are small for executing on a real systems.

Libraries/ Frameworks	**#kernels**	**#func**	**#total loads**	**#total stores**
cuBlas (v11)	4115	0	341249	106399
cuFFT (v10)	5173	4	175256	371932
cuRAND (v10)	204	0	4949	3610
cuSPARSE (v11)	4335	0	334694	101792
cuDNN (v7)	11713	5	1032688	551610
Rodinia	23	7	544	285
Caffe	1294	4	87267	32946
PyTorch	27987	319	2083978	857987

Table 3. Load and store instructions in CUDA-accelerated libraries and frameworks we use.

Workloads with **same** apps			Workloads with **different** apps		
ID	**Name**	**Epochs per app**	**ID**	**Name**	**Epochs per app**
A	2xlenet	500	I	lenet-siamese	500-50
B	4xlenet	500	J	siamese-cifar10	30-100
C	2xcifar10	100	K	2xlenet-siamese-2xcifar10	500-30-100
D	4xcifar10	100	L	3xlenet-siamese-2xcifar10	500-30-100
E	2xgaussian	-	M	hotspot-guassian	-
F	4xgaussian	-	N	gaussian-lavamd	-
G	2xlavamd	-	O	particle-hotspot	-
H	4xlavamd	-	P	gaussian-hotspot-lavamd-particle	-

Table 4. Mixes of workloads used for assessing the performance of Guardian under GPU sharing.

From Caffe, PyTorch, and Rodinia, we create a set of workloads shown in Table 4, to evaluate Guardian under concurrently running applications. Each workload is a mix of compute- and data-intensive applications and covers scenarios in which applications compete and stress the GPU resources. As in previous works [10, 29, 38, 50], we create workloads with 2-6 concurrent clients. The workloads A-H use multiple instances of the same application, while I-P includes different applications. To ensure that application executions overlap, we appropriately modify the number of epochs of each application, affecting the total execution time. We also vary the batch size to increase memory usage in each application from 500 MB to 2 GBs. To assess the applicability and coverage of Guardian, we use CUDA library samples [36] for `cuBLAS`, `cuFFT`, and `cuSPARSE` libraries. These examples include more than 30 library calls that are not in the real-world frameworks we use.

Performance measurements: We use Nsight to profile GPU kernel execution and collect metrics, such as cache

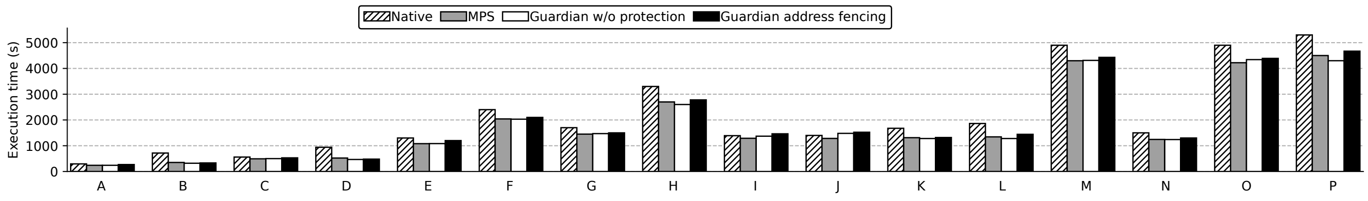


Figure 6. Multi-tenant GPU sharing using native CUDA time sharing (protected), MPS spatial sharing (unprotected), Guardian spatial sharing without protection, and Guardian spatial sharing with address fencing.

hits, GPU calls latencies, and kernel invocations. We use the Xptxas=-v compiler flag to measure register and constant memory use by the sandboxed kernels in Guardian. For measuring the duration of host calls, we use the rdtsc instruction and we set the CPU frequency to maximum.

Baseline and Guardian Deployments: We purposely use four deployments for ***GPU sharing***. *Native* uses the default CUDA runtime environment, which offers time sharing with protection and represents the baseline performance. The other three setups provide GPU spatial sharing: *MPS* uses the NVIDIA Multi-Process Service (MPS) [38] that allows concurrent execution of multiple kernels but without strong protection guarantees. Guardian without protection is our open-source implementation of spatial sharing, analogous to MPS. Guardian with address fencing is our main approach for protection using bit-masking. Although we use an in-house GPU manager for spatial sharing, Guardian can be integrated into other managers as well [20, 38, 65] if their source code is available. Regarding G-NET [65] that uses network functions for its evaluation, we extrapolate its protection mechanism for ML applications using address checking. Mask [5] uses Mosaic simulator [4] for its evaluation; thus, the comparison is infeasible. Finally, MIG [37] statically partitions high-end NVIDIA GPUs, leaving GPU resources underutilized, making comparison irrelevant.

Guardian GPU manager and applications run in different address spaces, and thus, they require IPC mechanisms for exchanging data and tasks. Such mechanisms increase the execution time of GPU calls. Additionally, spatial sharing increases resource contention, which may increase the latency of loads and stores in GPU memory, and Guardian overheads are amortized. To isolate Guardian protection overheads, we use a setup that Guardian GPU manager and applications run in the same address space, and we run ***standalone neural networks***. Regarding this scenario, we use: (a) *Native* CUDA as a baseline. (b) Guardian *without protection* that just intercepts GPU calls but does not perform any checks nor instrumentation. This setup models the overhead of intercepting and forwarding the CUDA runtime functions to a separate GPU manager process. (c) Guardian with *address checking*, to evaluate the overhead of control flow instead of bitwise instructions. This setup aims to show the extra overhead required to detect if an address is out of bounds instead of only preventing it. (d) Guardian with *address fencing* that contains all the mechanisms of our approach.

6 Experimental Evaluation

6.1 Impact of Guardian at GPU Sharing

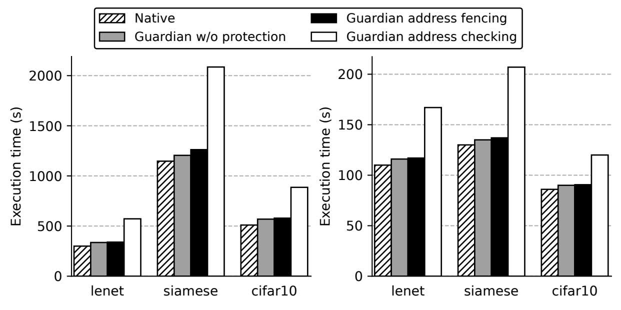
Figure 6 shows the execution times of *Native*, *MPS*, *Guardian without protection* and *Guardian address fencing* for the workloads of Table 4. Comparing Guardian address fencing to MPS, our approach is, on average, 4.84% slower due to the extra checks enforced to prevent out-of-bound accesses. When we turn off these checks in Guardian (no protection), the execution times achieved are 0.05% worst than MPS. In high resource contention, as in workloads I-P, the overheads of Guardian address fencing are lower on average 3.2% since our overheads are amortized. Guardian without protection performs better than MPS in workload with thousands of pending kernels (D, P, K, H), because the MPS server becomes a bottleneck according to previous works [46].

Finally, we compare spatial and temporal sharing, which is the default mechanism used from many previous works [7, 25, 59, 64] because it ensures protection. Guardian address fencing is, on average 23% faster than native, while in some cases, it is up to 2× faster due to parallel kernel execution. We note that the performance improvements of spatial sharing are primarily affected by the resources required by the concurrently executing workloads. In cases where the resources needed are low, as in workloads B and D, the benefits are more prominent, i.e., 2×, while the performance gap is reduced for more resource-intensive workloads.

6.2 Guardian Overheads Compared to Other Approaches Without Sharing

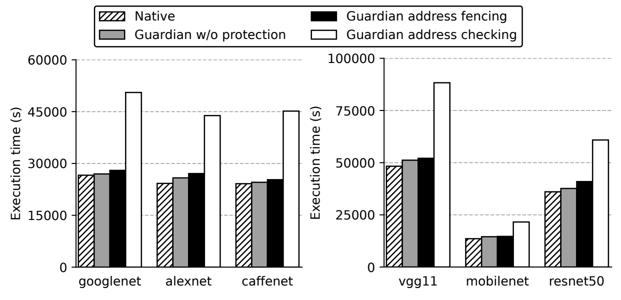
Figures 7 and 8 plot the times of the individual execution for several ML frameworks and CUDA-accelerated libraries (Table 3), using *Native*, *Guardian without protection*, *Guardian address fencing*, and *Guardian (address checking)*. We note that the training phase for lenet, siamese, cifar10 issues up to 142 million kernels, whereas googlenet, alexenet, caffenet, vgg11, mobilenetv2, and resnet50 issue billions of kernels. The inference phase issues up to 8 million kernels.

Figure 7(a) shows lenet, siamese, and cifar10 training, while Figure 7(b) shows the inference phase of the same neural networks. Guardian has between 5.9% up to 12% overhead



(a) Training (b) Inference

Figure 7. Comparison of address fencing with other approaches, using Caffe with *mnist and cifar* dataset.

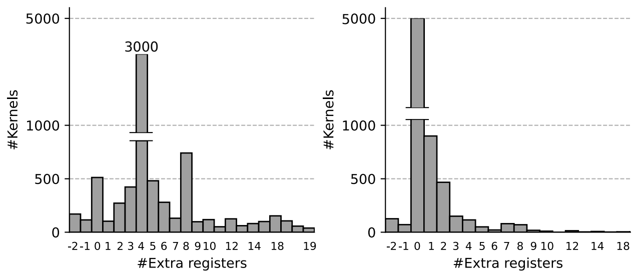


(a) Caffe Training (b) PyTorch Training and Inference

Figure 8. Comparison of address fencing with other approaches, using Caffe and PyTorch with the *imagenet* dataset.

compared to the unprotected native CUDA. The Guardian without protection approach includes the interception of CUDA calls and the search in the *pointerToSymbol* table to find the appropriate sandboxed kernel. The kernel issued in the GPU does not contain the bit-masking instructions, while transfer instructions do not contain the out-of-bounds checks added from Guardian address fencing. The Guardian without protection approach has an overhead from 3.7% to 10% compared to native. By comparing Guardian address fencing and Guardian without protection, the overhead of Guardian address fencing is between 1.05% up to 4.3%. As a result, the overhead added by bound checking (in transfers and PTX kernels) is 2.9% on average.

Figure 8(a) shows googlenet, alexenet, and caffenet training. Guardian address fencing has between 4.5% up to 10% overhead compared to the unprotected native CUDA. The Guardian without protection approach has an overhead from 1.36% to 6% compared to native. By comparing Guardian address fencing and Guardian without protection, the overhead of Guardian is between 2.9% up to 4.3%. Figure 8(b) shows vgg11, mobilenet, and resnet50 training and inference



(a) No Optimizations (-G) (b) Optimization level 3 (-O3)

Figure 9. Guardian's per thread register usage vs native.

using PyTorch. The overhead of Guardian for call interception is, on average 5.5% (native vs. Guardian without protection). The overhead of Guardian address fencing compared to Guardian without protection is on average 7.6%.

Finally, conditional checks increase execution time by 1.7× on average compared to native. This is because the branch instructions are more expensive compared to bitwise. Additionally, in the addressing mode that uses `address+offset` we add up to eight instructions (32cycles) to check the low and upper bounds of each memory partition.

6.3 Impact of Address Fencing on Register Usage

Figure 9 shows the number of extra registers that are eventually added for storing the address mask and the base address. Figure 9(a) shows the extra registers used from our approach when compiling the PTX without any optimization flag, whereas Figure 9(b) with full optimizations. The lack of optimization flag results in kernels (from cuBINs) using up to 4 extra registers in 62% of the total kernels. However, when we use full optimizations in the compilation (03), 71% of kernels use no extra registers, 13% use up to one extra register, and 7% use up to two extra registers. In some rare cases, the number of registers is smaller than the default because the compiler spills some registers in the global memory. Regarding the constant memory affected by the extra parameters Guardian adds in 99% of kernels 16 bytes.

6.4 Performance of CUDA calls Interception

The interception of kernel invocations in Guardian requires between 214 and 900 CPU cycles ("Lookup kernel ptr" in Table 5) for the lookup operation to locate the sandboxed kernel (stored in a `c++ unordered map`). Regarding the extra arguments passed in the kernel, we require between 300 and 600 CPU cycles to allocate a new parameter array and copy the new and old parameters in this array ("Augment kernel params" in Table 5). Guardian adds on average 957 CPU cycles per `cudaLaunchKernel`. We perform each experiment ten times, excluding the minimum and maximum values.

The `cudaLaunchKernel` NVIDIA system call is measured using the Nsight profiler. The average execution time (for

	Lookup kernel ptr	Augment kernel params	Launch kernel to GPU
Native	0	0	~9000
Guardian	557	400	~9000

Table 5. Guardian average latency in CPU cycles for the main operations performed when a kernel launch is intercepted and replaced with a sandboxed kernel.

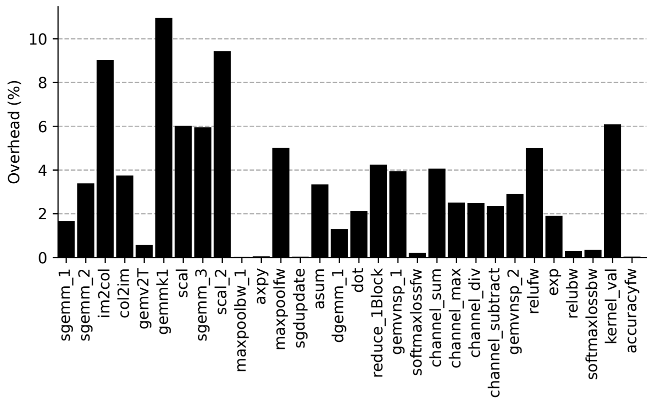


Figure 10. Performance overhead of sandboxed kernels against native execution.

more than one thousand kernels) in CPU cycles is approximately 9000 CPU cycles ("Launch kernel to GPU" in Table 5). So our overhead *without* the kernel execution is 10% on average. We have profiled lenet and cv applications (executing millions of kernels) and found out that the kernel execution time *without the cudaLaunchKernel* is, on average 18000 CPU cycles. Consequently, the overhead of Guardian, including the kernel execution time, is 3% per kernel, on average.

Memory allocation and data transfer. We use a microbenchmark that uses memory allocations and data transfers of different sizes to evaluate Guardian's memory management operations. The results suggest that (a) our allocator does not imply extra overhead compared to native CUDA, and (b) the extra protection checks used on every data transfer over the PCIe bus imply negligible overhead.

6.5 Performance of Address Fencing at High Cache Hit Ratio

Figure 10 shows the overhead of Guardian address fencing normalized to native for 890000 kernels used in lenet. The overhead of Guardian is, on average 3.2%. We have performed the same breakdown for computer vision and observed similar results. The overheads of Guardian bit-masking instructions depend on the latency of the load and store instructions. A load instruction that retrieves data from global memory is 220-350 cycles [6, 22], while if data are in L1-cache is 28 cycles. Our approach adds two (bitwise AND, OR) up to four instructions (for cases that include address+offset) per load

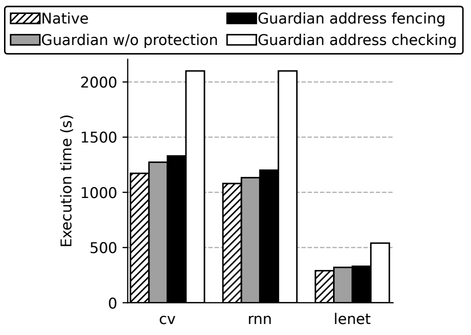


Figure 11. Guardian overhead with PyTorch and Caffe on GeForce GPU, compared to native execution.

and store instruction. Each of these instructions is executed in almost 4 cycles. As a result, if all data are in the L1 cache –not common–, our overhead is from 28% up to 57%. If all data are only in global memory, our overhead (with global memory latency 285 cycles) is from 2% up to 5%. We have profiled all kernels of lenet and observed that the average L1 cache hit rate is 37%, while for L2 is 72%. L2 latency is 180 cycles, only 1.4× better than global's. Overall, address fencing in Guardian incurs small additional overhead due to two main reasons: (1) As we show, ML kernels exhibit a low cache hit ratio. (2) As shown from previous works [3], cache hits result in a lower load/store instruction latency in the rear case that every thread in the warp hits in the cache.

6.6 Performance of Guardian on Different GPUs and Access Patterns

Figure 11 shows three neural networks from PyTorch and Caffe executed in the GeForce GPU. In computer vision (cv) and rnn Guardian (address fencing) incurs 12% and 10% overhead compared to native, respectively. Lenet with Guardian incurs 13% overhead compared to native. Conditional checks exhibit on average $1.8 time$ worst execution time compared to native. Overall, we note that Guardian has similar overhead across different GPU types.

Figure 12, shows the performance of Guardian over CUDA-accelerated library calls that are not contained in the ML frameworks used previously. Guardian successfully intercepts these calls and adds 4% overhead, on average, which is similar to the results observed with the Quadro GPU.

7 Related work

Table 6 summarizes the main characteristics of GPU-sharing approaches that offer memory protection and closer to Guardian.

7.1 Protect GPU Memory under GPU sharing

Time-sharing offers memory protection since it allows only one context to be active in the GPU at any time; thus, it is mainly used from previous works [13, 25, 59, 60, 64]. The

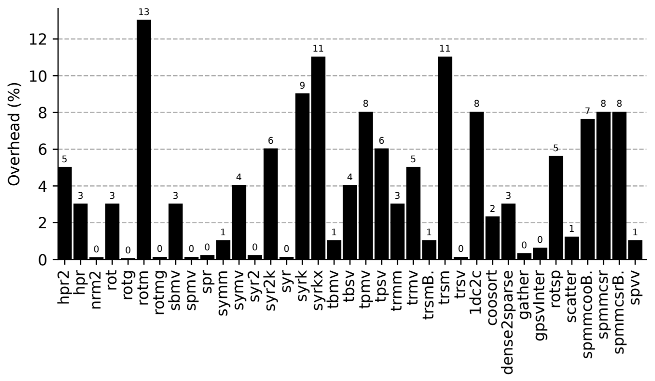


Figure 12. Guardian overhead (%) for 37 kernels from CUDA-accelerated libraries compared to native execution of each kernel on GeForce GPU.

Approach	No src code mod.	CUDA lib support	No extra /special HW	Spatial sharing
Time-sharing [38]	✓	✓	✓	-
Mask [5]	✓	✓	-	✓
MIG [37]	✓	✓	-	✓
G-NET [65]	-	-	✓	✓
Guardian	✓	✓	✓	✓

Table 6. Comparing Guardian with state-of-the-art memory protection approaches for GPU sharing.

device driver is responsible for allocating and managing resources belonging to a context. Upon a context switch, its resources are freed, and the translation lookaside buffer (TLB) is invalidated. Consequently, application data are protected at the cost of GPU utilization because context switching is expensive [5, 61, 65]. On the other hand, Guardian eliminates the expensive context switching and improves GPU utilization by offering protected spatial GPU sharing.

Mask [5] is a hardware-based approach that allows applications to share spatially and securely a GPU. Mask extends the GPU TLB to hold information about warps and the memory they can use, and as a result, it supports protected spatial sharing. Mask supports closed-source GPU libraries but with limited applicability due to the special hardware required. Guardian does not need extra or special hardware, protects closed-source libraries, and implies comparable overhead, making it more practical, powerful, and generic.

NVIDIA's Multi-Instance GPU (MIG) [37] partitions statically high-end NVIDIA GPUs (i.e., A100 and H100) in completely isolated parts. Besides the limited applicability (requires special hardware), recent works [5, 29] showed that MIG static partitioning leads to under-utilization and that changing from one partition scheme is not flexible. AMD and Intel GPUs do not offer any protection and, by default, allow applications to share a GPU [46] spatially. Guardian uses a more dynamic GPU sharing scheme, similar to MPS [38], with protection guarantees. Regarding compute resource isolation (i.e., CUDA cores), Guardian can use existing approaches [41, 65] or MPS resource provisioning [10].

G-NET [65] is a software-based approach that overcomes the limited applicability of hardware-based ones. G-NET deploys a custom type of pointer [49], namely `isoPointer`, that checks if the accessed memory address belongs to the correct partition. However, leveraging these pointers requires manual effort to port the kernel source code. The source code requirement is a serious limitation leading to weaker protection because most CUDA-accelerated applications rely heavily on closed-source GPU libraries, e.g., cuBLAS and cuDNN. Guardian operates in the kernel code's virtual assembly (PTX) available in closed-source GPU libraries.

7.2 Detect Buffer Overflows for a Single Application

clArmor [16] and GMOD [12] protect against overflows by adding canary values around the allocated buffers. However, such approaches have limited security coverage because they cannot capture non-adjacent accesses that jump over canaries. Parravicini et al. [44] add conditional checks inside the kernel LLVM-IR to preserve Java memory safety semantics in NVIDIA GPUs. They use static analysis to minimize the significant overhead implied by conditional checks, which require the application and kernel source code (or LLVM-IR), limiting its applicability. GPUShield [27] overcomes the limitation of source code using an extra hardware unit that performs the address checking. CUDA-MEMCHECK and cuCatch [51] are debugging tools that operate in the PTX [39] level and detect out-of-bound accesses without requiring extra/specific hardware. All these approaches focus on detecting overflows of a single application and are considered orthogonal to Guardian.

7.3 Ensure Privacy and Data Confidentiality

Graviton [55] is a trusted execution environment (TEE) providing privacy and data confidentiality guarantees. Graviton requires minimal hardware modifications only in the GPU command processor. Honeycomb relies on address checking to avoid the need for hardware modifications. It uses static analysis (i.e., source code) to minimize the need for runtime checks and its overhead. Although, TEEs tackle a significantly different problem, Guardian can be combined with Honeycomb to provide a TEE for GPUs with low overhead and support for closed-source GPU libraries.

8 Conclusions

In this paper, we present Guardian, a GPU memory sandboxing approach that makes GPU sharing from real-life applications, belonging to different users, practical. The benefits of Guardian are threefold: (1) It is transparent to applications, even when applications use closed-source GPU-accelerated libraries that include host-level and GPU kernel code. (2) It fences all memory accesses of GPU kernels, including closed-source kernels, by instrumenting kernels at the PTX level. (3) It incurs low overhead using bit-masking to fence addresses. Our evaluation using real-world ML frameworks shows that Guardian can support ML frameworks and GPU-accelerated libraries transparently introducing on average 9% overhead compared to native unprotected execution.

References

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *USENIX Security '09*.

[2] Yehia Arafa, Abdel-Hameed A Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. 2019. Low overhead instruction latency characterization for nvidia gpgpus. In *HPEC'19*.

[3] Rachata Ausavarungnirun, Saugata Ghose, Onur Kayiran, Gabriel H Loh, Chita R Das, Mahmut T Kandemir, and Onur Mutlu. 2015. Exploiting inter-warp heterogeneity to improve GPGPU performance. In *PACT '15*.

[4] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. 2017. Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *MICRO '17*.

[5] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. 2018. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *ASPLOS '18*.

[6] M Bari, L Stoltzfus, P Lin, C Liao, M Emani, and B Chapman. 2018. Is Data Placement Optimization Still Relevant On Newer GPUs?. In *OSTI.GOV*.

[7] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, N. Kwatra, and S. Viswanatha. 2020. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning. In *EuroSys '20*.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC '09*.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2022. Rodinia dataset. Retrieved May 2023 from https://www.dropbox.com/s/cc6cozpboht3mtu/rodinia-3.1-data.tar.gz

[10] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In *SoCC '20*.

[11] Bang Di, Jianhua Sun, and Hao Chen. 2016. A Study of Overflow Vulnerabilities on GPUs. In *NPC' 16*.

[12] Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. 2018. GMOD: A Dynamic GPU Memory Overflow Detector. In *PACT '18*.

[13] Jose Duato, Antonio J. Pena, Federico Silla, Juan C. Fernandez, Rafael Mayo, and Enrique S. Quintana-Orti. 2011. Enabling CUDA acceleration within virtual machines using rCUDA. In *HiPC '11*.

[14] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *NDSS '17*.

[15] Niklas Eiling, Jonas Baude, Stefan Lankes, and Antonello Monti. 2022. Cricket: A virtualization layer for distributed execution of CUDA applications with checkpoint/restart support. In *Concurrency and Computation: Practice and Experience*.

[16] Christopher Erb, Mike Collins, and Joseph L. Greathouse. 2017. Dynamic buffer overflow detection for GPGPUs. In *CGO '17*.

[17] Martín Abadi et. al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[18] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J. Rossbach. 2022. DGSF: Disaggregated GPUs for Serverless Functions. In *IPDPS '22*.

[19] Mark Gebhart, Stephen W. Keckler, Brucek Khailany, Ronny Krashinsky, and William J. Dally. 2012. Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor. In *MICRO '12*.

[20] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Reddy Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, et al. 2020. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. In *MICRO'20*.

[21] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, S. Guadarrama, and T. Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *ArXiv*.

[22] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. In *ArXiv*.

[23] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[24] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta pointers: Buffer overflow checks without the checks. In *EuroSys '18*.

[25] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. AlloX: Compute Allocation in Hybrid Clusters. In *EuroSys '20*.

[26] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*.

[27] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. 2022. Securing GPU via Region-Based Bounds Checking. In *ISCA '22*.

[28] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *S&P '14*.

[29] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2022. MISO: Exploiting Multi-Instance GPU Capability on Multi-Tenant GPU Clusters. In *SoCC '22*.

[30] Chao Li, Yi Yang, Zhen Lin, and Huiyang Zhou. 2015. Automatic data placement into GPU on-chip memory resources. In *CGO '15*.

[31] Andrea Miele. 2015. Buffer overflow vulnerabilities in CUDA: a preliminary analysis. In *Journal of Computer Virology and Hacking Techniques*.

[32] Thomas Moscibroda and Onur Mutlu. 2007. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX Security '07*.

[33] Diana M. Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. 2020. Accelerated serverless computing based on GPU virtualization. *J. Parallel and Distrib. Comput.*

[34] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, et al. 2023. Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI. In *ASPLOS '23*.

[35] NVIDIA. 2021. CUDA Binary Utilities. Retrieved May 2023 from https://docs.nvidia.com/cuda/pdf/CUDA_Binary_Utilities.pdf

[36] NVIDIA. 2022. CUDALibrarySample. Retrieved April 2023 from https://github.com/NVIDIA/CUDALibrarySamples/tree/master/

[37] NVIDIA. 2022. Multi-Instance GPU. Retrieved April 2023 from https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_MIG_User_Guide.pdf

[38] NVIDIA. 2022. Multi-Process Service. Retrieved May 2023 from https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[39] NVIDIA. 2022. Parallel Thread Execution ISA. Retrieved May 2023 from https://docs.nvidia.com/cuda/pdf/ptx_isa_8.1.pdf

[40] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. In *POMACS '18*.

[41] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU Concurrency with Elastic Kernels. In *ASPLOS '13*.

[42] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2017. Dynamic Resource Management for Efficient Utilization of Multitasking GPUs. In *ASPLOS '17*.

[43] Sang-Ok Park, Ohmin Kwon, Yonggon Kim, Sang Kil Cha, and Hyunsoo Yoon. 2021. Mind Control Attack: Undermining Deep Learning with GPU Memory Exploitation. In *Computers and Security*.

[44] Alberto Parravicini, Davide B. Bartolini, Lukas Stadler, Arnaud Delamare, Marco Arnaboldi, and Marco Domenico Santambrogio. 2015. Automated GPU Out-of-Bound Access Detection and Prevention in a Managed Environment. In *ArXiv*.

[45] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NIPs '19*.

[46] Manos Pavlidakis, Stelios Mavridis, Antony Chazapis, Giorgos Vasiliadis, and Angelos Bilas. 2022. Arax: A Runtime Framework for Decoupling Applications from Heterogeneous Accelerators. In *SoCC '22*.

[47] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. 2016. CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix. In *TECS '16*.

[48] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. In *IJCV '15*.

[49] Sagi Shahar, Shai Bergman, and Mark Silberstein. 2016. ActivePointers: A Case for Software Address Translation on GPUs. In *ISCA '*.

[50] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. 2013. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *HPCA '13*.

[51] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W. Keckler, and Mark Stephenson. 2023. CuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications. (2023).

[52] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B. Gibbons, and Onur Mutlu. 2016. Zorua: A Holistic Approach to Resource Virtualization in GPUs. In *MICRO '16*.

[53] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Saugata Ghose, Abhishek Bhowmick, Rachata Ausavarangnirun, Chita Das, Mahmut Kandemir, Todd C Mowry, and Onur Mutlu. 2016. A Framework for Accelerating Bottlenecks in GPU Execution with Assist Warps. *arXiv preprint arXiv:1602.01348* (2016).

[54] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *MICRO '19*.

[55] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted Execution Environments on GPUs. In *1OSDI '18*.

[56] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous multikernel GPU: Multitasking throughput processors via fine-grained sharing. In *HPCA '16*.

[57] Florian Wende, Thomas Steinke, and Frank Cordes. 2014. Multithreaded kernel offloading to gpgpu using hyper-q on kepler architecture. In *ArXiv*.

[58] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *NSDI '22*.

[59] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, F. Yang, and L. Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI '18*.

[60] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *OSDI '20*.

[61] Qiumin Xu, Hoda Naghibijouybari, Shibo Wang, Nael Abu-Ghazaleh, and Murali Annavaram. 2019. GPUGuard: Mitigating Contention Based Side and Covert Channel Attacks on GPUs. In *ICS'19*.

[62] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G Rogers. 2017. Pagoda: Fine-grained gpu resource virtualization for narrow tasks. In *PPoPP '17*.

[63] Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Chenchen Liu, and Xiang Chen. 2022. A survey of multi-tenant deep learning inference on GPU. In *ArXiv*.

[64] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. 2020. AvA: Accelerated Virtualization of Accelerators. In *ASPLOS '20*.

[65] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. 2018. G-Net: Effective GPU Sharing in NFV Systems. In *NSDI'18*.