Automated GPU Out-of-Bound Access Detection and Prevention in a Managed Environment

Alberto Parravicini, *Graduate Student Member, IEEE,* Davide B. Bartolini, Lukas Stadler, Arnaud Delamare, Marco Arnaboldi, Marco Domenico Santambrogio, *Senior Member, IEEE*

Abstract—GPUs have proven extremely effective at accelerating general-purpose workloads in fields from numerical simulation to deep learning and finance. However, even code written by experienced GPU programmers often offers little robustness, limiting the GPUs' adoption in critical applications' acceleration. Out-of-bounds array accesses are one of the most common sources of errors and vulnerabilities on GPUs and can be hard to detect and prevent due to the architectural characteristics of GPUs.
This work presents an automated technique ensuring detection and protection against out-of-bounds array accesses inside CUDA GPU kernels. We compile kernels ahead-of-time, invoke them at run time using the Graal polyglot Virtual Machine and execute them on the GPU. Our technique is transparent to the user and operates on the LLVM Intermediate Representation. It adds boundary checks for array accesses based on array size knowledge, available at run time thanks to the managed execution environment, and optimizes the resulting code to minimize the impact of our modifications.
We test our technique on 16 different GPU kernels extracted from common GPU workloads and show that we can prevent out-of-bounds array accesses in arbitrary GPU kernels without any statistically significant execution time overhead.

Index Terms—Graphics processors, Compilers, Runtime environments, Arrays

1 Introduction

Graphics Processing Units are beneficial in a multitude of fields (computer graphics, artificial intelligence, engineering, and finance), thanks to the computing power they offer and their ability to process large amounts of data in parallel. However, programming a Graphics Processing Unit (GPU) is inherently more complex than a traditional CPU architecture. It requires the users to understand how the computational kernels are mapped to the underlying GPU processors and memory architecture. This deep level architectural understanding is often not required when working with CPUs, making GPU programming more error-prone and difficult to debug. Despite their success and due to their peculiar architecture, GPUs require a relatively low-level programming model and lack some of the programming tools that are widespread for CPUs. For example, GPU programming is largely done in native languages (e.g., CUDA [1]) even though techniques to integrate GPUs within managed languages and environments [2], [3] have been recently proposed. For these reasons, programming for a GPU still tends to be more complex and error-prone than programming for a CPU. Given how computations are mapped to GPU hardware [4], out-of-bounds (OOB) array accesses, i.e. trying to access a non-existent position in an array, represent a common type of programming errors. While managed languages can protect from OOB accesses on CPUs, limited research exists in the context of GPUs [5].

The ease-of-programmability gap between GPUs and CPUs has seen significant research effort [2], [6], [7]. In this work, we leverage the Graal polyglot Virtual Machine (VM)[1] [8] and, in particular, the recently-published GrCUDA environment[2], which is able to run GPU kernels from high-level languages such as Python, Java and Ruby. The GrCUDA environment also takes care of memory allocation/deallocation using managed memory [9], [10] and maps array types to CUDA, preserving information on types and sizes that we leverage. We use Nvidia hardware and the CUDA platform, but similar considerations can be applied to AMD GPUs and the OpenCL platform [11], given the availability of a managed execution environment.

1.1 Motivation

Out-of-bounds array accesses can pose a variety of problems, such as unexpectedly ending or altering the program execution, and be a major security vulnerability [12], [13]. OOB array accesses are often encountered in GPU programming since the array sizes must be coordinated with offsets determined by the subdivision of GPU threads and other architectural characteristics. For example, when processing an array in CUDA [1], the array indices that each core operates on are determined using architecture-specific parameters such as grid, thread block, and thread identifier for that specific core (Section 2). The programmer is responsible for ensuring that those indices lie within the array's bounds: a wrong choice of parameters, or missing boundary checks, can lead to OOB array accesses as in Figure 1.

- *Alberto Parravicini and Marco D. Santambrogio are with DEIB at Politecnico di Milano, Milan, IT. E-mails: {alberto.parravicini, marco.santambrogio}@polimi.it*
- *Lukas Stadler is with Oracle Labs, Linz, Austria. E-mails: {lukas.stadler}@oracle.com*
- *Arnaud Delamare and Marco Arnaboldi are with Oracle Labs, Zurich, CH. E-mails: {arnaud.d.delamare, marco.arnaboldi}@oracle.com*
- *Davide B. Bartolini mostly contributed while with Oracle Labs; currently with Huawei, Zurich, CH. E-mail: davide.basilio.bartolini@huawei.com*

Manuscript received April 19, 2005; revised August 26, 2015.

1. github.com/oracle/graal
2. github.com/AlbertoParravicini/grcuda

OOB array accesses on GPUs can be challenging to detect because they usually do not cause the program to crash; when using global memory, they are not detected as errors by the CUDA runtime. Moreover, numerical results might be unaffected by these accesses, leaving programmers with a false sense of security over the robustness of their code. However, OOB accesses are still occurring, which introduces vulnerabilities that can be exploited for malicious purposes, as seen in [14], [15], [16]. The lack of OOB protection is a significant deterrent in GPU adoption for applications where data integrity is critical (e.g. in the finance industry). In that context, the best practice is to only execute pre-defined and hardened kernels on the GPU, limiting productivity. Our goal is enabling users to run any custom GPU kernel, as long as the source code is available, without having to worry about the adverse effects of OOB accesses.

1.2 Contributions

We present a technique that combines static program analysis with the power of managed execution environments to identify array accesses in CUDA GPU kernels automatically. We enhance the original code with detection and protection against OOB accesses and provide users with information about problematic array accesses at run time. Our technique operates on the LLVM Intermediate Representation (IR) [17], [18] of the GPU kernel and enables a managed execution environment to transparently provide kernels with the size of input arrays known only at run time, and monitor the occurrence of OOB accesses, with no execution time overhead. Existing techniques focus on the analysis or optimization of this IR, but none can automatically add OOB accesses protections to GPU code.

We validate our technique on several GPU kernels from domains such as Linear Algebra, AI, and Graph Analytics, taken from the Rodinia Benchmark Suite [19], [20], or inspired by open implementations by Nvidia [21]. Some GPU kernels contain OOB array accesses that we identify and fix with our technique. In other cases, we extend our benchmark suite by adding artificial OOB accesses in the original kernels. We process each kernel with our technique and test the result against an equivalent implementation containing hand-written OOB protections, or against the original implementation if it did not contain problematic array accesses. Manually added boundary protections are indicative of the realistic kernel performance if OOB accesses were fixed at source level by an experienced programmer.

In all cases, we confirm that our technique preserves the correctness of numerical results. Our experiments validate that the execution time of the kernels processed with our technique is not statistically worse than the one of the original kernels, making it applicable to practical scenarios. The rest of this document is organized as follows: Section 2 introduces why OOB accesses are commonly encountered in the GPU programming model and why it is critical to prevent them. Section 3 presents the existing research on this topic and how our work differs from previous approaches targeting CPUs or GPUs. Finally, we detail and evaluate our technique, making the following contributions:

- An automated technique that modifies the IR of GPU kernels with detection and protections against OOB accesses, reducing the kernel attack surface and increasing its robustness (Sections 4.1 to 4.3).
- A seamless integration within a managed runtime execution environment that communicates information to GPU kernels at run time (Section 4.4).
- A thorough evaluation of how GPU kernels are affected by our technique, showing that our transformations on average do not introduce statistically significant execution time overhead (Section 5).

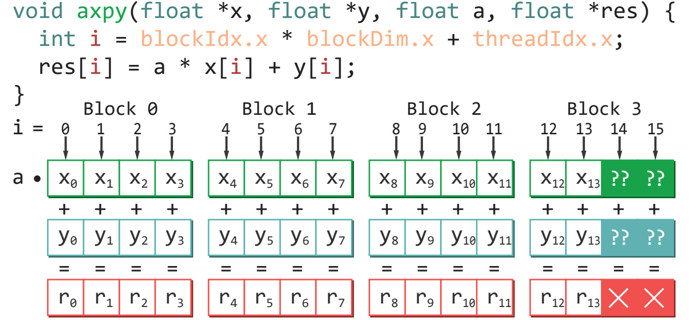


Fig. 1: An Axpy CUDA kernel, which scales an array x by a scalar a and sums it to another array y. Processing an array of size 14, using 4 blocks of size 4, will cause OOB accesses.

2 Context and Problem Statement

GPUs are specialized computer architectures that process large amounts of data in parallel by running computational kernels on each data item (e.g. each pixel in an image or elements of a list) or small groups of data items. The architecture of a GPU is divided into different levels, which can be roughly mapped to the logical levels that divide the computation. In GPUs manufactured by Nvidia, the computation is split across Stream Multiprocessors (SMs), each containing multiple Stream Processors (SPs) (or CUDA cores) that execute the same computational kernel on different data items. At a logical level, the computation is mapped to grids, thread blocks, and threads, with each thread processing a single data item or a small chunk of data items. In this work, an array is a collection of data items processed by the GPU [4].

The axpy kernel in Figure 1 is a simple GPU kernel that we use as a driving example to illustrate our algorithmic pipeline. We assume that the kernel source code is provided, but no assumption is made about its characteristics. To compute the result of this kernel, the CUDA runtime spawns a number of threads at least equal to the number of elements in the input arrays, but that could be larger due to the chosen grid structure[3]. Each block contains an equal amount of threads: in our axpy kernel, if the user decides to have 128 threads per block, and the input array has 1000 elements, it will be necessary to create $\lceil 1000/128 \rceil = 8$ blocks, for a total of $128 \times 8 = 1024$ threads. As the number of threads is larger than the size of the array, OOB array accesses will occur for values of $i \geq 1000$. Coordinating array sizes and grid structure is not trivial: block size is often determined by consideration about performance (and

3. The grid structure defines how many threads each block contains, and how many blocks are employed

A.
```
void axpy_manual_mod(float *x, float *y,
    float a, float *res, int size) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < size)
    res[i] = a * x[i] + y[i];
}
```

B.
```
void axpy_auto_mod(float *x, float *y,
    float a, float *res, uint *array_sizes) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i >= 0 && i < array_sizes[0]) {
    float x_i = x[i];
    if (i >= 0 && i < array_sizes[1]) {
      float y_i = y[i];
      if (i >= 0 && i < array_sizes[2])
        res[i] = a * x_i + y_i;
    }
  }
}
```

C.
```
void axpy_auto_mod_opt(float *x, float *y,
    float a, float *res, uint *array_sizes) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < array_sizes[0] &&
      i < array_sizes[1] &&
      i < array_sizes[2])
    res[i] = a * x[i] + y[i];
}
```

Fig. 2: Axpy Kernel, with manually-added boundary checks Ⓐ, and automatic boundary checks (highlighted), before optimizations Ⓑ and after Ⓒ. The lower bound check `i >= 0` is removed as `i` is guaranteed to be non-negative.

must lie between 32 and 1024), while the number of blocks is either a function of the input size or based on the available SMs [22]. A well-written CUDA program usually provides the user with the flexibility to adjust block size, and possibly the number of blocks, based on the target hardware, instead of forcing values imposed by the code structure or input size. In Figure 1 we schedule more threads than strictly necessary, leading to OOB accesses due to the lack of boundary checks. In this case, the computation result will appear to be correct, as the accesses have been performed on global memory, and the output array values are not directly affected by OOB values. In more complex kernels, OOB accesses can be a security vulnerability [12], [14], [15], [16], and lead to interrupted executions or wrong results (Section 5.1). For these reasons, it is critical to introduce a mechanism that prevents OOB accesses from occurring, instead of just detecting them after they occurred.

Due to this programming model, CUDA makes OOB array accesses more likely than other memory issues (e.g. use-after-free). In addition, memory allocation and deallocation are handled transparently to the programmer by GraalVM and GrCUDA, outside of kernel execution, guaranteeing the temporal validity of the memory exposed to the GPU kernel.

Figure 2 Ⓐ reports the same `axpy` kernel with manually added boundary checks: in practical applications, GPU kernels are significantly more complex, and preventing OOB accesses by hand is not as trivial. Our system operates on the LLVM IR of the code, but to simplify the examples, we present the results of equivalent transformations at source level. Figure 2 Ⓑ shows the result of these transformations.

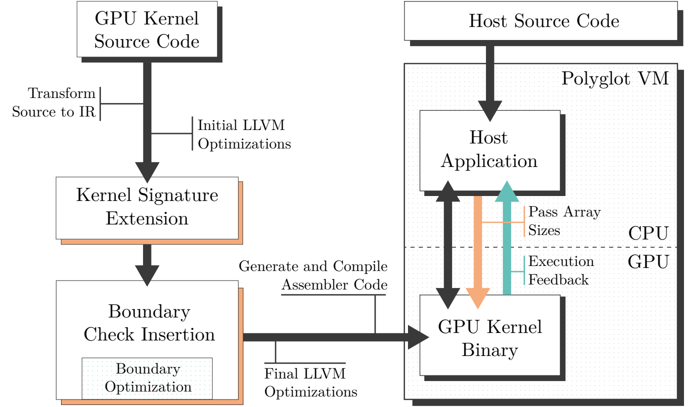


Fig. 3: Our pipeline to add boundary checks to CUDA kernels. We highlight our contributions to the existing pipeline for running pre-compiled GPU kernel in a polyglot VM.

Compared to Ⓐ, we need to be more restrictive in creating boundary checks, as input arrays could have different sizes, and indices might have negative values. Figure 3 represents the main steps of our algorithmic pipeline (Section 4). Automatically generated boundary checks can be further simplified with the techniques in Section 4.3.2, obtaining a result equivalent to Figure 2 Ⓒ. Thanks to these operations' intrinsic simplicity, additional checks do not make the kernels slower than the original unsafe kernel (Section 5).

3 Related Work

GPU OOB accesses are a serious security threat, as presented by Miele [14] and Di [12], and in recent vulnerability disclosures `CVE-2019-5684` and `CVE-2019-5685` [15] from Nvidia, and `CVE-2018-6251` [16]. OOB accesses can cause arbitrary code execution, alter numerical results, and allow denial-of-service attacks (critical in contexts such as autonomous driving, finance, or medical imaging). From Miele's work, even OOB array accesses on global memory constitute a significant security threat, even when they do not noticeably affect the computation and remain unnoticed by less experienced programmers.

3.1 OOB Detection and Prevention on CPUs

Prevention, or mitigation, of OOB array accesses has been investigated for years, focusing on CPU programming. Detecting OOB accesses through static program analysis is known to have limited effect because, in Turing complete programming languages, the problem is equivalent to the halting problem [23], [24]. Existing techniques rely on heuristics that provide limited guarantees on the detection, or the prevention, of OOB array accesses. Tools such as Joern [25] can be used to identify OOB array accesses through static program analysis, but they cannot modify the existing code to offer protection against them.

Another technique consists of running the desired application in a sandboxed environment [26], [27], [28], so that a malicious actor who can exploit OOB accesses would not be able to take control of the machine executing the code. This approach does not prevent OOB accesses but is used to mitigate their effects. During execution OOB accesses still

occur, meaning that the application might provide wrong results or interrupt its execution unexpectedly.

Existing research on array bound checking in managed execution environments on CPUs has mostly focused on removing unnecessary boundary checks, a problem symmetrical to ours [29], [30], [31]. Our work leverages some ideas found in CPU boundary checks optimization, such as leveraging Static Single Assignment (SSA) code representation to simplify dependency and alias analyses and performing boundary check hoisting outside of loops to improve performance. However, these approaches operate on languages such as Java in which array sizes are made directly available by the runtime environment, greatly simplifying analyses. In our case, we need to link this information from the managed execution environment (where sizes are known at run time) to native CUDA code that is transformed statically by augmenting the kernel input with run time information.

3.2 OOB Detection and Prevention on GPUs

OOB accesses prevention on GPUs has been mostly focused on optimizations for code generation, instead of modifying existing CUDA code as in this work. In [5], [32], the Habanero-Java language is extended with a `safe` keyword to denote a section of generated GPU code that will not raise exceptions during execution if it meets preconditions manually specified by the programmer. The computation is performed on the CPU through a slow JVM-based version of the generated code if preconditions are not met. While this approach can prevent unsafe GPU code execution, it relies on the programmer to manually specify preconditions. It forces the execution of a slower alternative if these conditions are unmet. Alternatively, the CPU implementation is run in parallel to the GPU code to detect exceptions that would arise on the GPU. Our approach, instead, is fully automatic and does not rely on code generation (it operates directly on CUDA code) on a slow fallback execution path nor on subtracting computational resources from the CPU to perform additional validations. The technique in [33] also generates OpenCL code starting from a subset of the Java language, with a Just-in-Time (JIT) compiler that hoists boundary checks as in [29]. Information about input data-structures (for example, Java arrays) is directly leveraged to guide code generation, making boundary checks generation and pointer aliasing easier, as the association between arrays and their size is immediately available as a property of the data structure. In this work, we can process existing arbitrary CUDA code and inject instructions for OOB prevention and detection through static LLVM transformations. This approach avoids invasive changes in the front-end compiler and enables the interaction with other LLVM transformations. Table 1 provides a summary of these techniques.

Common techniques used on CPUs might have limited efficacy on GPUs: for example, Erb et al. employ canary values to detect buffer overflows [13]. While this technique works with arbitrary OpenCL code, it is unable to prevent OOB accesses and introduces an execution overhead around 9%. The Futhark language [34] can detect OOB accesses in generated GPU code, but once again cannot prevent them from happening. Indeed, detection of OOB accesses, without preventing their execution, is not sufficient to prevent vulnerabilities and exploits [12], [14]. CUDA only supports a subset of C/C++, with constraints on memory allocation/deallocation and invocation of unsupported arbitrary functions (e.g. functions not compiled for device execution). These constraints restrict the scope of solutions and make code transformation, as we propose, a promising solution. Although adding if-statements may lead to a more complex GPU control flow and exacerbate warp-divergence[4] [35], we introduce optimizations such as boundary check merging and hoisting to mitigate the problem (Section 4.3.2).

TABLE 1: Relevant existing approaches for OOB *detection* (D) and *prevention* (P). *Detection* means that OOB accesses still happen, but the computation is stopped afterwards or errors are reported to the user. *Prevention* means that OOB accesses do not happen in the first place.

Name of technique	Target language	Operation mode	D	P
Joern [25]	C/C++, CPU	Static	✓	
Erb et al. [13]	Arbitrary OpenCL	Run time	✓	
Hayashi et al. [5], [32]	Habanero-Java	Static + Run time	✓	✓
Henriksen [34]	Futhark DSL	Run time	✓	
CUDA-MEMCHECK	Arbitrary CUDA	Run time	✓	
Our work	Arbitrary CUDA	Static + Run time	✓	✓

The debugging tool CUDA-MEMCHECK[5] by Nvidia runs the provided executable in a controlled environment and detects out-of-bound accesses whenever they occur, similarly to Valgrind [36]. CUDA-MEMCHECK cannot transform the source code to prevent OOB accesses, and programmers need to manually modify their code if an OOB access is detected. Moreover, OOB accesses could occur depending on the provided input and grid configuration, meaning that users have to create ad-hoc tests to validate each configuration they intend to run. Compared to existing solutions, our transformations are fully automatic, making the entire process transparent to the user: something that tools such as CUDA-MEMCHECK could not achieve.

4 METHODOLOGY

We explain the main steps required to implement the algorithmic pipeline that automatically adds boundary checks on array accesses in a given GPU kernel, compiles the kernel to a binary file, and executes it through a polyglot VM. A polyglot VM such as GraalVM can run multiple guest languages within the same runtime environment (such as the Java Virtual Machine (JVM)) and provides information about the current execution context across different guest languages. As such, information about the array sizes is made readily available to the GPU kernel. Our work is built on top of the GrCUDA environment, used to compile and invoke CUDA kernels. In the rest of this section, we present in detail the main phases of our technique.

4.1 Intermediate Representation of GPU kernels

The pipeline's input is the source code of a GPU kernel, which is transformed into the LLVM IR through the Clang

4. different threads in the same warp executing different instructions, inefficient on GPUs SIMD architecture

5. https://docs.nvidia.com/cuda/cuda-memcheck/index.html

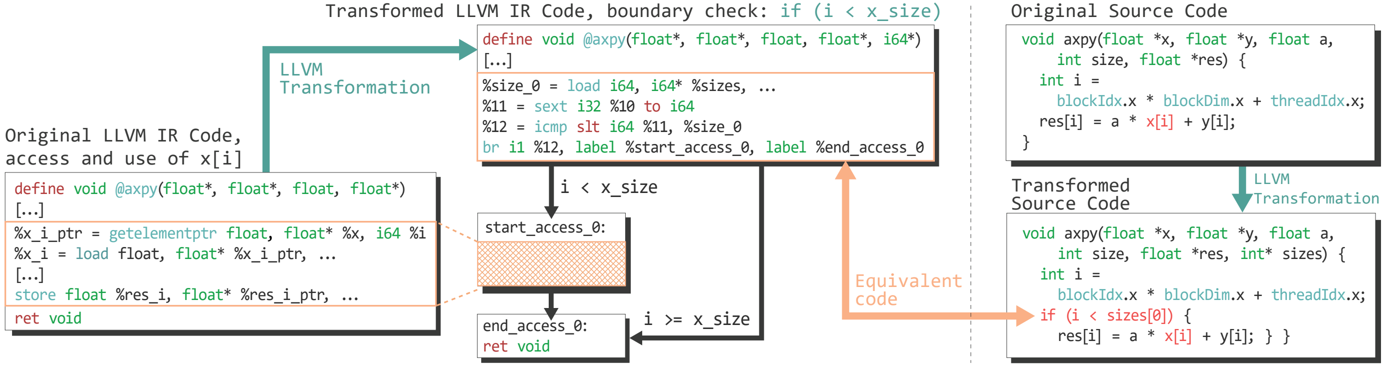


Fig. 4: How our transformations add a boundary check to one array access in the axpy LLVM IR. Relevant portions of the code are highlighted. On the left, we show how our transformations manipulate the LLVM IR and the control flow graph. On the right, equivalent transformations at source code level. Lower bound checks can be avoided as in Section 4.3.2.

compiler. Operating on the SSA LLVM IR simplifies the *liveliness* variable analysis (Section 4.3.1 and the identification of data dependencies. Moreover, operating at source-code level makes generated boundary checks more reliant on further LLVM optimizations: adding boundary checks in the source code introduces a slowdown above 40% if no additional optimization passes are applied to the code (Section 5.2).

We also leverage code-generation simplifications by Clang: for example, different ways to access arrays in CUDA (such as `x[i]` and `*(x + i)`) are translated to the same IR, greatly simplifying our subsequent analyses.

4.2 Extending kernels with array size information

Adding boundary checks to the GPU kernel code and detecting possible OOB accesses requires run time knowledge of the size of each array accessed inside the kernel. While the size of local buffers inside the kernel might be fixed and available at compile-time, we need a way to provide information about the size of input arrays from the managed runtime environment to the CUDA kernel. Moreover, we want a way to track the kernel's execution, monitor array accesses, and return to the managed environment information about the computation (e.g. if any OOB access could have occurred). If an array (or pointer, we use the term interchangeably as GrCUDA only supports pointers that represent arrays, possibly containing a single item) is an input to the GPU kernel, there are no guarantees that the input arguments specify its size. Even if such an argument is present, we cannot rely on the user's value to be correct or that this value is used to perform boundary checks.

Consequently, we need to extend the GPU kernel signature with an additional array: when invoking the kernel from GrCUDA, this array provides the size of each array from the GraalVM run time environment to the actual GPU kernel, in the same order as they appear in the signature. Optionally, we introduce a second array used to track the occurrences of OOB accesses for each input array to provide debugging information and stronger OOB detection.

4.3 Automatically adding GPU boundary checks

The transformation that identifies array accesses and adds boundary checks is applied right after the kernel signature extension. This second transformation directly applies optimizations to the IR to minimize the number of OOB checks.

4.3.1 Identification of problematic array accesses

We initially perform a linear scan of the GPU kernel IR to track aliases of known arrays and to identify *array accesses*. We define as array access all instructions that read or write data on a specified memory address, such as LLVM load/store and CUDA atomic memory operations.

Array accesses require the computation of an address, i.e. a `getelementptr` (GEP) instruction in LLVM. For each GEP, we need to identify which array is accessed starting from the address computed through it. Most importantly, we also need to identify the scope of the values loaded or stored through the address, i.e. understand which instructions the array access will affect. Figure 5 illustrates our algorithm on the LLVM basic block graph of a Sparse matrix-vector multiplication (SpMV) kernel, similar to the BFS and PR benchmarks in Section 5.1 and containing nested array accesses and loops boundaries with array accesses.

The start of the array access scope usually coincides with the GEP instruction, except in the presence of loop boundaries or nodes (instructions used to select a value based on previous basic blocks). In this case, we add the boundary check by introducing a new basic block before the one containing the loop boundary or node ① ② ③. To find where the array access scope terminates, we traverse in a Depth-First-Visit fashion the dependency graph that starts from the GEP instruction and identify all the leaf instructions of this graph. These instructions do not have further unexplored children (i.e. users of its result), and usually represent `store` instructions whose target is not accessed in subsequent parts of the GPU kernel. In the simplest case, all the leaves belong to the same *basic block* as the root of the graph (the GEP instruction): in this situation, there are no branches (such as pre-existing `if` statements) in the control flow that affect the array access. The end of the array access scope is determined by the leaf instruction that appears last in the code. If there exist leaf instructions outside the original basic block in which the GEP appears, we take as the end of the scope the start of the next basic block that is common to all the basic blocks in which the leaves appear (which can be the end of the kernel, if no common block

Function add_boundary_checks(*kernel*):
  array_accesses = []
  - 1. Associate each input array to its size
  array_size_map = parse_parameters(*kernel*)
  - Track other arrays, e.g. local caches
  array_size_map = add_other_arrays(*kernel*, *array_size_map*)
  - If no array was found, return
  if *array_size_map.size()* == *0* **then return**
  - 2. Find array accesses
  for *instruction* ∈ *kernel*.get_instruction() **do**
    - Is the instruction a reference to an array?
    check_if_array_alias(*instruction*)
    - Handle array/pointer accesses, Section 4.3.1
    if is_array_access(*instruction*) **then**
      - For each array access identify the scope of the access and other metadata
      access = handle_access(*instruction*)
      array_accesses.add(access)
  - 3. Filter array access with existing out-of-boundary access protection, Section 4.3.1
  filter_accesses(*array_accesses*)
  - 4. Merge equivalent array accesses, Section 4.3.2
  simplify_array_accesses(*array_accesses*)
  - 5. Add boundary check statements to protect each array accesses, Section 4.3.3
  add_array_access_protection(*array_accesses*)

Algorithm 1: Pseudo-code of the algorithm that adds boundary checks to array accesses in a GPU kernel.

is determined), and introduce a new basic block before it ④. If the result is used in the computation of the loop boundary condition, we modify the condition to include additional boundary checks. If a GEP is used as input to another GEP (for example, in the case of nested array accesses), our algorithm adds boundary checks for both GEPs, with the second boundary check computed only if the first is successful (we *un-nest* the array accesses). We leverage LLVM optimizations to simplify the modified basic block graph: for example, the boundary check `v+1<size(ptr)` is hoisted outside the loop boundary computation as it can be computed only once during the first iteration ⑤.

Given that GrCUDA wraps all pointer inputs through well-defined Java objects, it is not possible to introduce aliasing on the input arrays (e.g. providing both pointers `x` and `x + 10` as inputs). Input arrays are the only valid global memory locations during the kernel execution, limiting the surface that we must cover through boundary checks. Although CUDA allows memory allocations within kernels, this practice is heavily discouraged due to poor performance[6]. Extending our algorithm to support this scenario does not require its in-depth rethinking, as we can deduce the size of dynamically allocated arrays from the `malloc` input parameters. In the case of function calls, we look for usages of the function return value and array arguments and optionally recursively process the function itself.

To guarantee that no OOB occurs, none of the instructions in the identified GEP scope must be executed, and the program should still provide the expected result. While

6. docs.nvidia.com/cuda/cuda-c-programming-guide/index.html\#dynamic-global-memory-allocation-and-operations

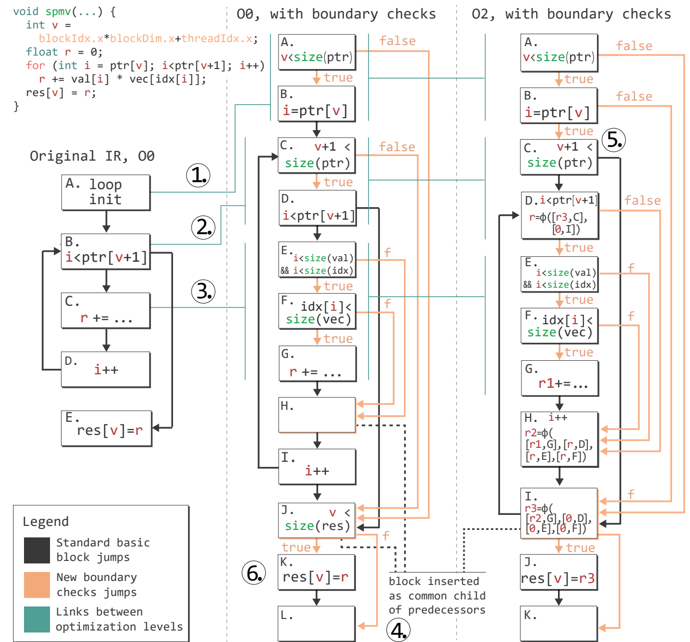


Fig. 5: We add boundary checks to the basic block graph of a SpMV kernel containing loops and indirect array accesses, and leverage LLVM optimizations to minimize overheads.

this procedure could, in principle, alter the program semantic, the instruction that cause OOB accesses should not have been executed in the first place. Moreover, OOB accesses result in undefined behavior, and not executing them cannot affect the program's correctness. The semantics of a program that originally does not present OOB is not affected. For example, in Figure 5 we increment `r` only for valid array indices, and write it to `res` only if the index `v` lies within the array's bounds ⑥. Additional boundary checks do not influence a correct execution, and their impact on performance is negligible (Section 5.2). We also report information to the user about the presence of mitigated OOB accesses, by tracking at run time if the Boolean conditions that represent our additional boundary are false for a given thread or index, and either store the information for the user or interrupt the computation, depending on the user preference. Through this approach, we detect if a program contains problematic array accesses, and we also prevent the execution of instructions that can give rise to vulnerabilities.

4.3.2 Pruning unnecessary candidate boundary checks

We remove any duplicate candidate boundary checks and filter all checks for which we can statically determine that a stronger condition already exists (e.g. boundary checks on arrays with size known statically). In this context, *stronger* entails a stricter Boolean condition, and that the scope protected by the pre-existing boundary check is larger or equal to the scope of our candidate boundary check.

As an optional optimization step, we identify if two boundary checks would protect the same scope and can be concatenated. The boundary checks are hoisted from their original location and concatenated with Boolean AND operations, reducing the number of conditional branches in the code through explicit instruction predication. We also

```
# Allocate arrays on both CPU and GPU
N = 1000; a = 2.0
x = polyglot.eval("grcuda",
  "float[{}]".format(N))
y = polyglot.eval("grcuda",
  "float[{}]".format(N))
res = polyglot.eval("grcuda",
  "float[{}]".format(N))
```

```
# Initialize arrays with sample values
for i in range(size):
  x[i] = i;  y[i] = i; res[i] = 0
```

```
# Load the GPU kernel
K = build_kernel(K_CODE_STR,
  "ptr, ptr, float, ptr")
```

```
# Invoke the kernel
K(NUM_BLOCKS, NUM_THREADS)(x, y, a, res)
```

Fig. 6: Invocation of the Axpy kernel from GrCUDA, with transparent inference of array sizes.

remove 0 boundary checks whose index is a CUDA identifier (e.g. `threadIdx`) or is a non-negative expression computed from CUDA identifiers; by default, this last optimization is disabled to prevent integer overflows. We show these simplification in Figure 2 ⓒ.

4.3.3 Optimized insertion of array boundary checks

We insert boundary checks by embodying the instructions belonging to the identified array access scope inside a conditional block that is executed only if the index used to access the array lies in the array's valid size. We introduce optimized IR sequences that are often faster than a source-code equivalent, especially if few additional LLVM transformations are applied (e.g. `-O0` optimization level, Section 5). Algorithm 1 reports the pseudocode of this algorithm.

4.3.4 Detection of OOB boundary checks

In some cases, users might want to monitor the occurrence of OOB accesses without affecting their program's control flow. Through the same boundary check conditions used to prevent OOB accesses, we can insert instructions that track the occurrence of OOB accesses by (atomically) updating the tracking array provided in the extended kernel signature. These instructions are inserted after each GEP instruction, without affecting the original control flow. We also display which array or index has been involved in the OOB access. After execution, we inspect the tracking information to issue warnings or raise exceptions if any OOB access has occurred. Prevention and detection are not mutually exclusive: our system can prevent OOB accesses and still provide users with information about OOB accesses that might have occurred in their code. In this case, tracking instructions are inserted in an additional *else* basic block that is executed only when a OOB access is prevented. To our knowledge, our technique is the only one offering both prevention and detection of OOB accesses in arbitrary CUDA code.

4.4 GPU code integration with runtime environment

Compilation of the transformed IR can happen in advance or performed Just-in-Time once the run time environment loads the kernel. The GPU kernel's binary code is loaded

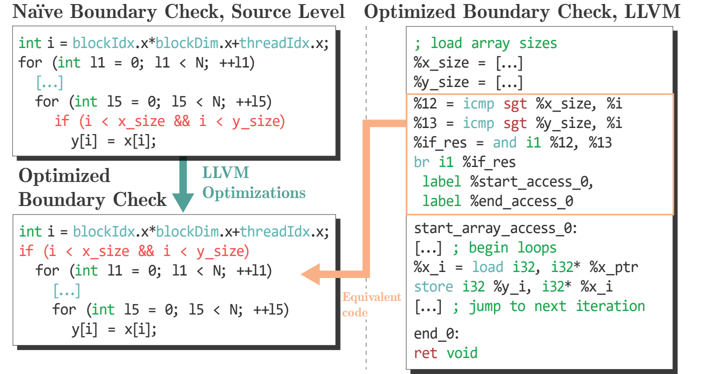


Fig. 7: LLVM code hoisting ensures that boundary checks inside deeply nested loops are moved at the start and executed only once per thread, minimizing the overhead

inside a polyglot VM, allowing users to choose an arbitrary language to interact with the GPU kernel. The VM converts input/output data from their original format to the format suitable for the GPU kernel. In our system, we leverage the GraalVM polyglot VM. The example in Figure 6 shows how a user can load and call a GPU kernel using Python as the host language; the size of input arrays is inferred automatically when the user invokes the kernel. In principle, any system capable of invoking CUDA kernels can leverage our technique, as long as it can perform memory allocation/deallocation and is aware of array sizes at run time. From a practical standpoint, using a polyglot VM makes the process simpler and easy to integrate with multiple high-level languages, without language-specific bindings.

5 Experimental Evaluation

We tested the algorithm on 16 GPU kernels commonly used as building blocks for Linear Algebra, Artificial Intelligence, and Graph Analytics GPU applications. These kernels come from the Rodinia GPU Benchmark Suite [19], [20], or have been adapted from openly available code samples by Nvidia [21]. We divide kernels into two categories:

1) The first group contains kernels with OOB accesses in the original source code. These kernels have been processed with our technique and compared against manually modified kernels to test our approach against hand-optimized boundary checks.
2) The second group contains kernels originally free of OOB array accesses, but presenting complex indexing expressions, or OOB accesses that arise if certain constraints on the input size are not met. We give these kernels additional robustness and measure the overheads over the original code.

Table 2 lists the kernels in our evaluation grouped by testing methodology. We provide information about input data size (the number of elements in each input array) and the types of OOB array accesses in each kernel. Further details about each kernel are provided in Section 5.1.

5.1 Testing Methodology

Tests were performed on an Nvidia Tesla P100 16GB PCIe 3.0 GPU. Kernels have been executed 50 times on randomly

TABLE 2: Summary of kernels in the evaluation. *Input Size* is the number of elements in each input array/matrix.

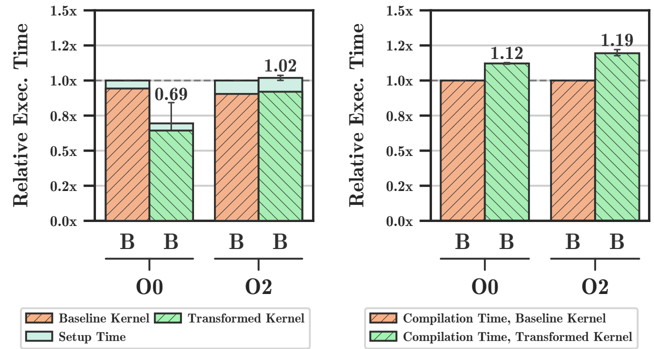
Kernel Group	Kernel Name (Short Name)	Domain	Input Size	OoB Accesses in Kernel	Hard-Coded Sizes in Kernel
vs. Modified Kernel	**Axpy** (AXPY)	Linear Algebra	$4 \cdot 10^6$	Global memory	-
	Dot Product (DP)	Linear Algebra	10^6	Global, shared	Yes
	Convolution 1D (CONV)	Signal Processing	10^6	Global, shared	Yes
	Auto-covariance (ACV)	Signal Processing	10^6	Global, shared	Yes
	Hotspot 3D (HP3D)	PDE Solver	128^3	Global memory	-
	NN - Forward Pass (NN1)	Deep Learning	$2 \cdot 10^5$	Global memory	-
	Breadth First Search (BFS)	Graph Analytics	10^5	Global memory	-
	PageRank (PR)	Graph Analytics	$2 \cdot 10^5$	Global memory	-
	Nested Loops (NEST)	Synthetic Benchmark	10^3	Global memory	-
vs. Original Kernel	**Matrix Multiplication** (MULT)	Linear Algebra	$3 \cdot 10^5$	-	-
	Hotspot (HP)	PDE Solver	600^2	-	Yes
	NN - Backpropagation (NN2)	Deep Learning	$4 \cdot 10^5$	-	-
	Gaussian Elimination (GE)	Linear Algebra	4096^2	-	-
	Histogram (HIST)	Statistics	$2 \cdot 10^6$	-	Yes
	LU Decomposition (LU)	Linear Algebra	2400^2	Global, shared	Yes
	Needleman-Wunsch (NW)	Bioinformatics	2400^2	Global, shared	Yes

generated data. Baseline and automatically modified kernels use copies of the same data to guarantee consistent cache usage and identical data movement between host and GPU.

We choose a list of kernels that highlight different challenges of OOB accesses prevention on GPU: usage of shared memory, thread concurrency and synchronization, multi-dimensional arrays, atomic operations, and more.

The first group contains kernels with OOB accesses in global memory or shared memory. OOB accesses in global memory do not generally cause the kernel execution to terminate but can be a security threat nonetheless [14]. OOB accesses on shared memory cause the computation to stop abruptly. For each of these kernels, we compare the performance of hand-optimized boundary checks added to the source code against boundary checks inserted automatically by our technique. We detail below each kernel.

Axpy: the kernel multiplies a vector by a scalar and sums it with another vector, as in Figure 1.
Dot Product: scalar product between two vectors, implemented as in [21]. It shows how we support atomic operations and shared memory storage.
Convolution 1D: convolution of two vectors, with complex expressions indexing global memory.
Auto-covariance: auto-covariance of a time-series, using 2D shared memory and atomic additions.
Hotspot 3D: from the Rodinia suite, the Hotspot 3D kernel solves a series of differential equations [37]. The original kernel contains OOB accesses on global memory that can produce wrong output values.
Neural Network - Forward Pass: forward pass of a feed-forward neural network [38], from Rodinia.
BFS: breadth-first visit performed on a sparse graph, adapted from the Rodinia suite [39] to store a graph as CSR. Tested on a random sparse graph with 10^5 vertices and maximum degree 10.
PageRank: one iteration of PageRank, implemented starting from the BFS kernel. It contains indirect nested array accesses (e.g. `y[x[i]]`) that are non-trivial to optimize, as the outer index is obtained only after the inner access has been completed. Indirect OOB array accesses can crash the application.
Nested Loops: we created this kernel to test the limitations of our code transformations. The kernel contains array accesses inside a sequence of deeply nested `for` loops, and naïvely generated boundary checks (i.e. inside the loop nest) severely decrease the kernel performance. By leveraging LLVM code hoisting, we compute the boundary check outside of the loop nest, only once per thread (Figure 7).



(a) Relative execution times (b) Relative compilation times

Fig. 8: Geometric mean of the kernel relative execution times ⓐ and compilation times ⓑ, using `O0`/`O2` optimization levels, compared to the respective baselines.

The second group contains kernels that can be considered free of OOB array accesses if certain constraints on the input size are satisfied. There are also boundary checks that cannot be inferred automatically as they depend, for example, on how a 2D matrix is stored in a 1D array. In these cases, we enhance the existing boundary checks with additional controls to prevent OOB accesses in complex indexing expressions. We process each kernel with our technique

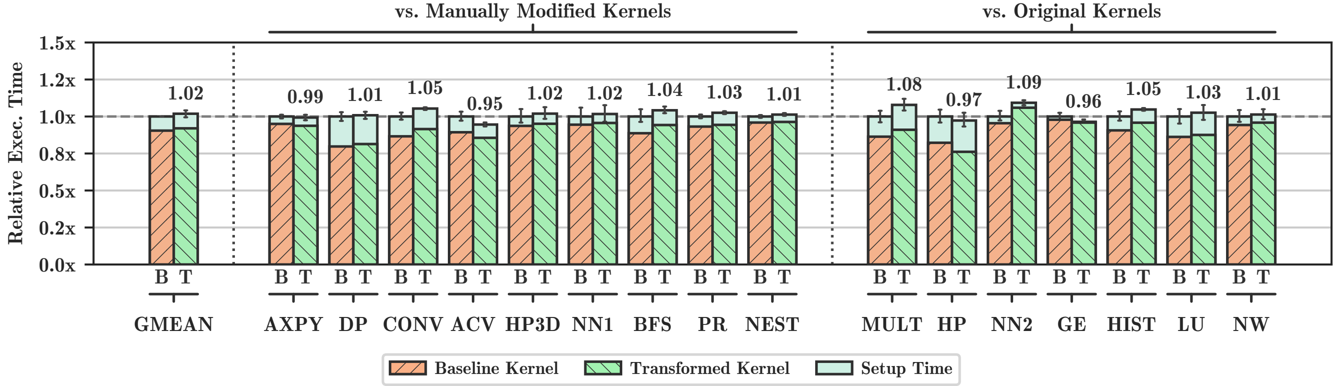


Fig. 9: Relative kernel execution time of automatically modified kernel vs manually modified and unmodified kernels, with -O2 optimization level. We distinguish between kernel execution time and total execution time, which includes GrCUDA overheads such as loading the kernel. On average, modified code is only 6% slower than the original, unprotected, version.

and compare its performance against the original kernel. Programmers can lightheartedly adopt our technique if it ensures higher robustness for little-to-no overhead.

Matrix Multiplication: the product of two row-linearized dense matrices.
Hotspot: from Rodinia, a 2D simulation of the temperature of a processor, using 2D shared memory.
Neural Network - Backpropagation: a Rodinia kernel that computes the weights update phase over a layer of a feed-forward neural network [38].
Gaussian Elimination: taken from the Rodinia suite, this kernel is part of the Gaussian Elimination algorithm used to solve linear systems of equations [40].
Histogram: from the Rodinia suite and used in Hybrid Sort, it counts how many elements of a list fall into equally sized intervals [41]. The original kernel requires a specific block size to work correctly.
LU Decomposition: from the Rodinia suite and part of the LU Decomposition algorithm [42], this kernel makes large use of 2D local caches and nested loops.
Needleman-Wunsch: from Rodinia, it computes the alignment score of two nucleotide sequences, as part of the Needleman-Wunsch algorithm [43]. This kernel uses 2D caches and calls another kernel whose input requires OOB accesses protection. It demands specific input and block sizes to avoid OOB accesses.

Several kernels have hard-coded parameters that define the size of local caches or the value of indexes (see the *Hard-coded Sizes in Kernel* column in Table 2). Users must use precise values for the number of threads per block or the number of blocks to prevent the kernel from crashing. This constraint is not robust enough for real-life applications: our technique avoids OOB accesses even under these circumstances. We selected benchmarks from the Rodinia suite with different characteristics (presence of nested loops, local caches, etc.). Other Rodinia benchmarks are similar to ours or have features currently not supported by GrCUDA, such as using classes/structs as input arguments. Our technique is straightforward to extend to those applications once GrCUDA is updated to supports the missing features.

5.2 Evaluation of compilation and execution overheads

We measure how our transformation affects each kernel's compilation and execution time and the overheads introduced by GrCUDA. Tests are performed using optimization levels -O0 and -O2, to assess the performance difference between manually and automatically generated boundary checks, and the overall overheads of additional boundary checks. We focus on -O2 as it is the most common optimization level in production environments, but other optimization levels (e.g. -O1, -O3) were statistically identical to -O2. Automatically modified kernels have been processed using *boundary check merging* optimizations (Section 4.3.2).

5.2.1 Relative execution time versus baseline kernels

First, we evaluate the execution time of each automatically modified kernel relative to its baseline. Kernels in the first group are compared to kernels where boundary checks have been manually added; kernels in the second group are compared against the original kernels without any modification.

Figure 8 presents the geometric mean of relative kernel execution and compilation times. Code compiled with -O0 is 44% faster than the baseline, showing the benefits of adding boundary checks at IR level instead of modifying the source code. -O2 is only 2% slower than the baseline, showing overall negligible overheads due to our transformations, even after applying equal optimizations to both versions of the kernels. Compilation is also not significantly slower, averaging less than 20% slowdown.

Figure 9 shows the kernel execution time, and the overall execution time comprehensive of overheads added by GrCUDA, normalized with respect to the corresponding baseline execution time. Results are aggregated using geometric mean, and we display the 95% confidence interval. In no case, we observe a large performance gap between the original and automatically modified kernels. The slowdown is only 9% in the worst case (on the NN2 kernel, which has many array accesses and little computation).

Figure 10 compares the execution time distribution of automatically modified kernels with their baselines. Outliers do not explain the results of Figure 9, and the shape of distributions is not affected by our modifications. (e.g. kernels with fat tails like NN1 preserve this characteristic).

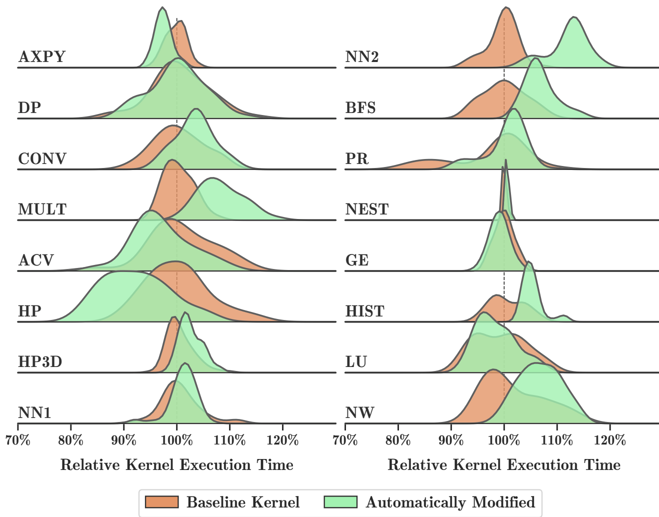


Fig. 10: Relative kernel execution time distribution of automatically modified kernel vs baseline kernels. Vertical axes do not have the same scale, for better readability.

5.2.2 Statistical test on execution times

We validate if the small performance differences in Figure 9 are statistically significant. Each kernel is tested 50 times on random data, with each run using the same data for manually and transformed kernels, making samples dependent and paired. We test statistical difference through the non-parametric Wilcoxon test [44], as executions times are not normally distributed (Figure 10). If the null hypothesis is rejected, one distribution has a higher median than the other. As shown in Table 3, most automatically modified kernels are statistically not slower than their respective baselines. In 9 cases, no statistical difference is observed (given the number of tests, we consider a test significant if its *p-value* is extremely low, e.g. 10 ^{4}), and in 3 cases, automatically modified kernels are even faster than the baselines.

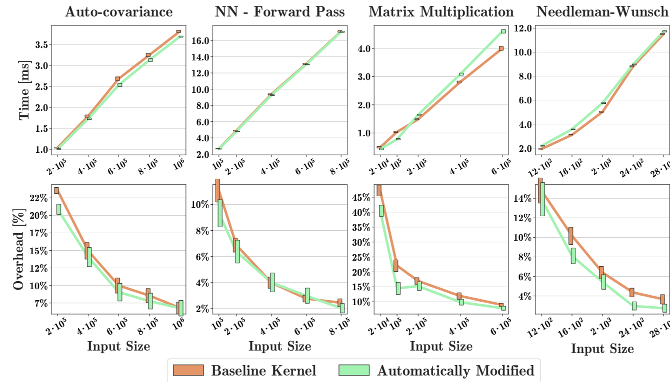


Fig. 11: Scalability and overheads of different kernels compiled with -O2 level optimizations, with 95% confidence intervals. In all cases, the behaviour of the baseline and automatically modified kernels follow the same trend.

5.2.3 Scalability Testing

We measured the execution time with increasing input sizes to test how the performance of each kernel scales, and to observe how the overheads diminish. Figure 11 presents a selection of the results, showing the execution time and the overheads of four kernels compared against their baselines, all compiled with -O2 level optimizations. The first two kernels are compared against manually modified code, the other two against the original, unmodified code. Other kernels display similar trends. The execution time of automatically modified code scales identically to the baselines. We also measure the overheads of calling kernels through GrCUDA, as a percentage of the overall execution time: this overhead, while noticeable for small input sizes with computations ending in about 1 millisecond, becomes irrelevant (less than 5%) for realistic input sizes.

TABLE 3: Wilcoxon test applied to the execution time of each kernel with and without automatic boundary checks. Absence of statistical difference is denoted with a dash (-).

	O2 Optimization Level	
Kernel Name	Wilcoxon p-value	Faster Kernel
vs. Modified Kernel		
AXPY	0.0255	-
DP	0.039	-
CONV	4.58×10^{-5}	Manually Modified
ACV	9.15×10^{-7}	Manually Modified
HP3D	5.61×10^{-6}	Automatically Modified
NN1	4.48×10^{-8}	Automatically Modified
BFS	0.00857	-
PR	0.0267	-
NEST	0.0044	-
vs. Original Kernel		
MULT	7.87×10^{-6}	Original Kernel
HP	4.33×10^{-6}	Automatically Modified
NN2	0.00109	-
GE	0.00442	-
HIST	0.00977	-
LU	0.000146	Original Kernel
NW	0.202	-
Harmonic Mean	0.117	-

5.2.4 Relative execution time versus unmodified kernels

We compare the execution time of automatically modified kernels with the execution time of kernels to which no modification was applied, even if the original computation presents OOB accesses. We selected all kernels where OOB accesses *do not prevent the computation from completing*, even if numerical results are wrong. Namely, the AXPY, HP3D, NN1, BFS and NEST kernels contain OOB accesses on arrays stored in global memory. All kernels are compiled with -O2 level optimizations. On average, we see a slowdown of only 2% compared to the original, unprotected code, with results close to the ones in Figure 9. Our technique does not introduce any practical disadvantage, as it can prevent OOB array accesses while avoiding wasted computation on OOB values at the same time.

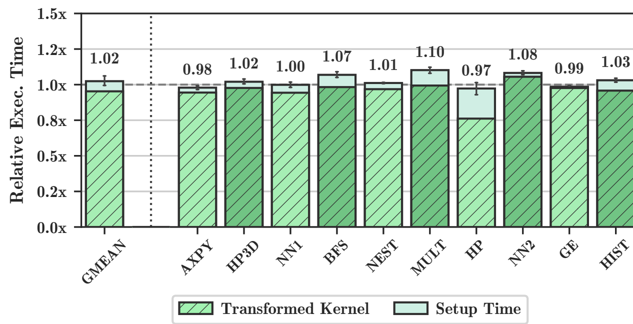


Fig. 12: Relative kernel execution time of automatically modified kernels versus unmodified ones. 5 of these unmodified kernels contain OOB accesses that our technique prevents.

6 Conclusion

We presented a novel technique to detect and prevent out-of-bounds array accesses in CUDA kernels, leveraging the LLVM toolchain to transform CUDA code at compile-time, add information about array sizes, and enhance the code with boundary checks. We introduce no overhead in the modified kernels. The execution time of 16 GPU kernels processed with our technique is statistically equivalent to both unmodified kernels and kernels with hand-tuned boundary checks. As future work, we will extend our technique to cover the CUDA language's features currently not supported by the GrCUDA environment (such as arbitrary input data structures) and dynamic memory allocations in kernels. We will also investigate if our technique effectively protects C/C++ code executed in GraalVM through the Sulong LLVM bitcode interpreter [45].

Our technique seamlessly integrates with the GraalVM polyglot VM: information about array sizes is transparently provided to GPU kernels at run time. This approach greatly extends GPU acceleration's flexibility and makes it readily available to data scientists and engineers who might be unwilling to deploy GPU-based solutions without having the robustness of managed runtime environments.

Acknowledgments

We thank Oracle Labs support and contributions to this work. The authors from Politecnico di Milano are funded in part by a research grant from Oracle. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[2] J. Fumero, M. Steuwer, L. Stadler, and C. Dubach, "Just-in-time gpu compilation for interpreted languages with partial evaluation," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '17. New York, NY, USA: ACM, 2017, pp. 60–73.

[3] C. Kotselidis, J. Clarkson, A. Rodchenko, A. Nisbet, J. Mawer, and M. Luján, "Heterogeneous managed runtime systems: A computer vision case study," *SIGPLAN Not.*, pp. 74–82, Apr. 2017.

[4] Nvidia, "Cuda programming guide, thread hierarchy," docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy, 2020-10-27, retrieved on 2020-12-04.

[5] A. Hayashi, M. Grossman, J. Zhao, J. Shirako, and V. Sarkar, "Speculative execution of parallel programs with precise exception semantics on gpus," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2013, pp. 342–356.

[6] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, A. Sarma, D. Nanongkai, G. Pandurangan, P. Tetali *et al.*, "Pycuda: Gpu run-time code generation for high-performance computing," *Arxiv preprint arXiv*, vol. 911, 2009.

[7] A. Celik, P. Nie, C. J. Rossbach, and M. Gligoric, "Design, implementation, and application of gpu-based java bytecode interpreters," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, 2019.

[8] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck, "Graal ir: An extensible declarative intermediate representation," in *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[9] D. Negrut, R. Serban, A. Li, and A. Seidl, "Unified memory in cuda 6.0: a brief overview of related data access and transfer issues," *Tech. Rep. TR-2014–09, University of Wisconsin-Madison, 2014*, 2014.

[10] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in cuda," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014, pp. 1–6.

[11] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[12] B. Di, J. Sun, and H. Chen, "A study of overflow vulnerabilities on gpus," in *IFIP International Conference on Network and Parallel Computing*. Springer, 2016, pp. 103–115.

[13] C. Erb, M. Collins, and J. L. Greathouse, "Dynamic buffer overflow detection for gpgpus," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 61–73.

[14] A. Miele, "Buffer overflow vulnerabilities in cuda: a preliminary analysis," *Journal of Computer Virology and Hacking Techniques*, vol. 12, no. 2, pp. 113–120, 2016.

[15] Nvidia, "Cve-2019-5684, cve-2019-5685," nvidia.custhelp.com/app/answers/detail/a_id/4841/kw/Security\%20Bulletin, 2019-09-23, retrieved on 2020-12-04.

[16] ——, "Cve-2018-6251," nvidia.custhelp.com/app/answers/detail/a_id/4649/kw/security, 2018-04-26, retrieved on 2020-12-04.

[17] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004.

[18] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, "gpucc: an open-source gpgpu compiler," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 2016.

[19] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*, 2009.

[20] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *IEEE International Symposium on Workload Characterization (IISWC'10)*, 2010.

[21] M. Harris *et al.*, "Optimizing parallel reduction in cuda," 2007.

[22] M. Harris, "Write flexible kernels with grid-stride loops," developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops, 2013-04-13, retrieved on 2020-12-04.

[23] T. V. N. Nguyen and F. Irigoin, "Efficient and effective array bound checking," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 527–570, 2005.

[24] P. Habermehl, R. Iosif, and T. Vojnar, "What else is decidable about integer arrays?" in *International Conference on Foundations of Software Science and Computational Structures*. Springer, 2008.

[25] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[26] Nvidia, "Nvidia virtual gpu technology," www.nvidia.com/en-us/data-center/virtual-gpu-technology/, 2018-06-11, retrieved on 2020-12-04.

[27] Chromium, "Chromium gpu sandboxing," chromium.googlesource.com/chromium/src/+/master/docs/linux/sandboxing.md, 2017-09-08, retrieved on 2020-12-04.