Dynamic Buffer Overflow Detection for GPGPUs

Christopher Erb
AMD Research
Advanced Micro Devices, Inc.
Austin, TX, USA
Christopher.Erb@amd.com

Mike Collins
AMD Research
Advanced Micro Devices, Inc.
Austin, TX, USA
Mike.Collins@amd.com

Joseph L. Greathouse
AMD Research
Advanced Micro Devices, Inc.
Austin, TX, USA
Joseph.Greathouse@amd.com

Abstract

Buffer overflows are a common source of program crashes, data corruption, and security problems. In this work, we demonstrate that GPU-based workloads can also cause buffer overflows, a problem that was traditionally ignored because CPUs and GPUs had separate memory spaces. Modern GPUs share virtual, and sometimes physical, memory with CPUs, meaning that GPU-based buffer overflows are capable of producing the same program crashes, data corruption, and security problems as CPU-based overflows. While there are many tools to find buffer overflows in CPU-based applications, the shift towards GPU-enhanced programs has expanded the problem beyond their capabilities.

This paper describes a tool that uses canaries to detect buffer overflows caused by GPGPU kernels. It wraps OpenCL™ API calls and alerts users to any kernel that writes outside of a memory buffer. We study a variety of optimizations, including using the GPU to perform the canary checks, which allow our tool to run at near application speeds. The resulting runtime overhead, which scales with the number of buffers used by the kernel, is 14% across 175 applications in 16 GPU benchmark suites. In these same suites, we found 13 buffer overflows in 7 benchmarks.

1. Introduction

Buffer overflows are a common software problem with a long history [7]; famous security attacks such as the Morris Worm, Code Red, and Slammer were all predicated on this error. By allowing accesses outside of the "correct" region of memory, buffer overflows can lead to program crashes, data corruption, and security breaches [53]. Owing to this long history in CPU-based applications, numerous tools have been built to find and stop buffer overflows [10, 12, 29, 42, 45, 58–60, 65, 68, 74].

In contrast, little attention has been paid to buffer overflows on GPUs. While GPU programs are just as susceptible to memory bugs as CPU programs, the following differences have led developers to incorrectly ignore the problem:

1. GPU and CPU memory has traditionally been separated, making it difficult for GPUs to corrupt the CPU memory.
2. GPU programs rarely use pointers or make function calls, making visible crashes from buffer overflows less likely.
3. GPU memory is often allocated in a less dense manner than CPU memory, so overflows from one buffer will not necessarily corrupt useful data in another.

Unfortunately, these neither prevent overflows nor protect against their effects. Miele, for instance, recently demonstrated that GPU-based buffer overflows can lead to remote GPU code execution [54], as did Di et al. [26].

Additionally, GPUs can access CPU memory over interconnects such as PCIe®, and standards such as Heterogeneous System Architecture (HSA) allow virtual memory sharing between CPUs and GPUs [13, 40]. Such tight integration allows GPU overflows to easily corrupt CPU data.

There currently exist few tools to help catch GPU buffer overflows. Oclgrind can instrument OpenCL™ programs with checks to find such errors, but it causes runtime overheads of up to 300× [61]. Techniques from fast CPU-based detectors like Electric Fence [60] and StackGuard [21] are difficult to add to closed-source vendor runtimes; they require changes to virtual memory managers or compilers.

Towards this end, this paper describes the design of a runtime buffer overflow detector for OpenCL GPU programs. By catching function calls that allocate OpenCL memory buffers, we can add canary regions outside of the buffers. After an OpenCL kernel completes, our tool checks these canary regions to see if the kernel wrote beyond its buffers' limits. If so, our tool alerts the user to this problem.

We limit the overhead of these analyses by tuning our canary checks and switching them between the CPU and GPU. Our tool causes a mean slowdown of only 14% across 175 programs in 16 OpenCL benchmark suites. We use the same benchmarks to demonstrate its efficacy by finding a total of 13 buffer overflows in 7 of the programs.

In total, this paper makes the following contributions:

- We describe the design of the first canary-based OpenCL buffer overflow detector. It is also the first to work with OpenCL 2.0 shared virtual memory buffers.
- We detail techniques, such as using GPU kernels to check canary values for overflows, which limit our tool to an average slowdown of only 14%.
- We show that our tool finds real problems by detecting and fixing 13 buffer overflows in 7 real benchmarks, many of which have not been observed by other tools.

2. Background

This section includes background information for this work. Section 2.1 discusses buffer overflows and tools to find them. Section 2.2 describes GPU memories, and Section 2.3 discusses how OpenCL™ 2.0 presents this memory to software.

2.1 Buffer Overflows

Buffer overflows are software errors that result from accessing data beyond the limits of a buffer. An example of a buffer overflow (perhaps most famously described by Aleph One [6]) is illustrated in Figures 1(a) and 1(b). Here, an array and the function's return address are placed next to one another. Copying too many values into the array will overwrite the address, allowing an attacker to take control of the application. Overflows can also cause silent data corruption, memory leaks, and crashes, among other problems.

Numerous tools have been built to catch buffer overflows in CPU programs. Heavyweight tools like Valgrind [58] add extra checks to validate each memory access, but their overheads mean that they are used sparingly by developers.

This work therefore focuses on more lightweight techniques. In particular, as illustrated in Figure 1(c), we place *canary values* outside of buffers that are susceptible to overflows. The values are later checked and, if they have been overwritten, the tool reports a buffer overflow.

Previous tools have used canaries to check for overflows in CPU programs. StackGuard [21] uses canaries to protect the stack, while ContraPolice can protect heap structures [45]. Both of these tools add canary checks at various points in the application. Along similar lines, Electric Fence [60] adds canary pages around heap structures and protects the canaries using the virtual memory system; a page fault therefore indicates a buffer overflow. While these tools are useful for finding CPU-based buffer overflows, they do not work for GPU memory.

2.2 GPU Memory

CPUs and GPUs have traditionally had separate memory spaces, so overflows on the GPU would not corrupt CPU data. In addition, because GPUs rarely performed tasks like dereferencing pointers, corrupted GPU buffers rarely caused crashes. GPU memory managers often pack data less densely than CPU managers, making it harder for an overflow to corrupt other buffers [50]. As such, GPU buffer overflows were often ignored or left undetected, and there were few tools built to find them.

While this may imply that buffer overflows are a minor problem on GPUs, this is not the case. Miele recently demonstrated that buffer overflows on the GPU could be used to inject code that could allow attackers to take control of the GPU's operation [54]. Beyond hijacking GPU control flow, the move towards tightly integrated heterogeneous systems means that GPUs can also corrupt CPU memory by directly accessing CPU buffers over interconnects like PCIe®.

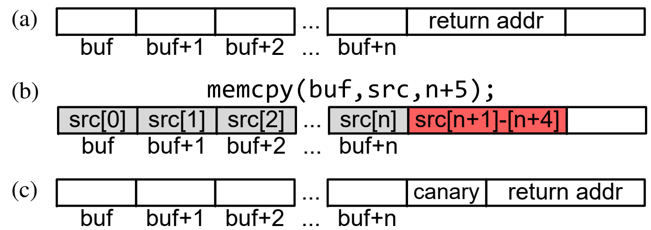


Figure 1. Example of a buffer overflow. (a) and (b) show how copying too much data into a buffer can corrupt neighboring variables. (c) shows a canary value after the buffer; if this canary changes, a buffer overflow has occurred.

This can happen in OpenCL applications that allocate buffers with flags such as CL_MEM_ALLOC_HOST_PTR or CL_MEM_USE_HOST_PTR [3]. Similarly, fine-grained shared virtual memory buffers are stored in this manner in AMD's OpenCL 2.0 implementation [8]. Margiolas and O'Boyle took advantage of these techniques to reduce DMA transfers between the CPU and GPU memories [52]. They only moved buffers into the GPU's memory if they would be accessed frequently enough to justify the transfer costs. As such, they designed an automated system that may leave some CPU buffers susceptible to GPU buffer overflows.

New chip designs bring advancements that will exacerbate this problem. Single-chip heterogeneous SoCs are now available from companies such as AMD [46], Intel [28], and Nvidia [27] that allow CPUs and GPUs to share the same physical memory. Both AMD and Intel allow these devices to share virtual memory, as well. Similarly, HSA compliant devices share virtual memory between the CPU and accelerators [13, 70]. This means that GPU buffer overflows will become more problematic in the future.

2.3 OpenCL Memory Buffers

This work focuses on buffer overflows caused by OpenCL™ kernels running on GPUs. As such, this section details the type of buffers used in OpenCL kernels.

Stack Values Many OpenCL implementations do not put stack variables into memory, instead preferring to allocate them entirely in registers. In addition, analyzing these variables requires modifying the OpenCL compiler, which is often a proprietary part of a vendor's software stack. As such, our buffer overflow detector does not analyze stack values.

Local Memory Local memory is often allocated into on-chip scratchpad memories at kernel invocation time. Because this memory is not shared with the DRAM buffers, attempting to access values outside of the allocated region often causes GPU kernels to crash immediately. As such, our tool does not search for local memory overflows.

Global cl_mem Buffers These memory buffers are allocated by the host using functions such as `clCreateBuffer`. By default, these buffers are allocated into the GPU's memory and cannot contain pointers. However, as mentioned in

Section 2.2, they can be forced into the CPU's memory. Our tool watches for buffer overflows in these regions by wrapping calls to functions that create `cl_mem` buffers and expanding the requested size to include our canary regions.

Global cl_mem Images Images are multi-dimensional buffers created using functions such as `clCreateImage2D` (before OpenCL 1.2) and `clCreateImage` (OpenCL 1.2+). Images are like cl_mem buffers, except that it is possible for an application to overflow one dimension of the image without writing past the "end" of the buffer (the final dimension). To enable discovery of these overflows, our tool expands each dimension of an image with canary regions.

Sub-Buffers A sub-buffer is created by calling the function `clCreateSubBuffer`, which takes a reference to an existing cl_mem buffer and returns a cl_mem object that points into the middle of the original buffer. Because this sub-buffer is within a memory region that has already been allocated, our tool cannot expand this buffer with canary regions at sub-buffer creation time. Instead, our tool creates a shadow copy of this buffer, as further described in Section 3.

Coarse-grained SVM Shared virtual memory (SVM) is a feature of OpenCL 2.0 that allows buffers that reside in the GPU's memory to contain pointers into their buffer and into other SVM buffers. These regions are allocated with the `clSVMAlloc` function. Coarse-grained SVM buffers must be mapped into the CPU's memory in order to access them on the CPU, and this generally copies the data. Once the buffer is mapped on the CPU, the pointers it contains are still valid.

Our tool watches for buffer overflows in these regions by wrapping calls to `clSVMAlloc` and expanding the requested allocation size to include our canary regions.

Fine-grained SVM Like coarse-grained SVM, these buffers can contain pointers that are valid on both the CPU and the GPU. However, fine-grained SVM buffers need not be mapped and copied in order to access them from the CPU. AMD's OpenCL runtime enables this by storing them in host memory and allowing the GPU to access this CPU memory region [8]. Our tool watches for buffer overflows in these regions by wrapping calls to `clSVMAlloc` and expanding the requested allocation size to include our canary regions.

Fine-grained System SVM These are pointers to traditional CPU memory, such as heap data returned from `malloc`. Because the allocation of these buffers does not go through any OpenCL APIs, and because most modern discrete GPUs do not support fine-grained SVM, our detector does not analyze this type of memory.

GPUs that *do* support these regions (such as the integrated systems discussed in Section 2.2) do so by sharing virtual memory between the CPU and GPU. As such, CPU-based buffer overflow detection mechanisms such as Electric Fence [60] would work for fine-grained system SVM. A GPU buffer overflow to such a region would cause a GPU page fault [73], meaning that our tool would not be required.

3. Design of a Buffer Overflow Detector

Our buffer overflow detector uses *canary values* to detect whether a GPU kernel has written past the end of any global memory regions. This is akin to tools like StackGuard [21], ContraPolice [45], and Electric Fence [60]. In such systems, the canary values beyond the end of a buffer are periodically checked to ensure that no buffer overflow has happened.

StackGuard and ContraPolice add the canary checks into the application, requiring it or the system libraries to be recompiled. Electric Fence protects canary pages using the virtual memory system, and writing into the canary region will cause a page fault. This is done by dynamically linking against Electric Fence and replacing calls to functions like `malloc`, which does not require recompilation.

Like Electric Fence, our tool works on unmodified programs. We accomplish this using the Unix LD_PRELOAD mechanism to create an OpenCL™ wrapper [62] (though similar library injection techniques can also be performed on the Windows® operating system [75]).

Our wrapper catches OpenCL API calls that allocate global memory and expands the requested sizes to include canary regions that are initialized with known patterns. We then wrap calls that set GPU kernel arguments in order to keep track of which buffers to check. Finally, we wrap the function call that launches GPU kernels, and, after the kernel completes, we check the canary values from any buffer it could access. If a canary value has changed, a bug in the kernel has caused a buffer overflow.

We check the canary values after the kernel completes because we cannot catch canary writes in the kernel (like StackGuard) or with memory protection (like Electric Fence). This limits the types of errors our tool finds, since there is a time when a corrupted value could be used before the canaries are checked. An attacker could avoid our checks by taking control of the GPU before we get a chance to see the canary values and could even reset the canary values to avoid detection. As such, our buffer overflow detector offers no security guarantees. Nonetheless, like the lightweight bounds checking technique by Hasabnis et al., this tool can still find useful problems [39]. As we demonstrate in Section 6, our technique is useful as a debugging tool.

Our tool's three wrapper mechanisms are illustrated in Figure 2. The following sections detail a simple version of our system to make the explanation easier. Section 4 discusses performance optimizations.

3.1 Buffer Creation APIs

As Figure 2(a) shows, we wrap buffer creation APIs such as `clSVMAlloc`, `clCreateBuffer`, and `clCreateImage`. We then extend the buffers created by these functions and record meta-data about them. For all buffers whose creation does not use an existing allocation as a base, the size of the buffer is increased by the length of a canary region (8 KB in our studies), which is initialized with a static data pattern.

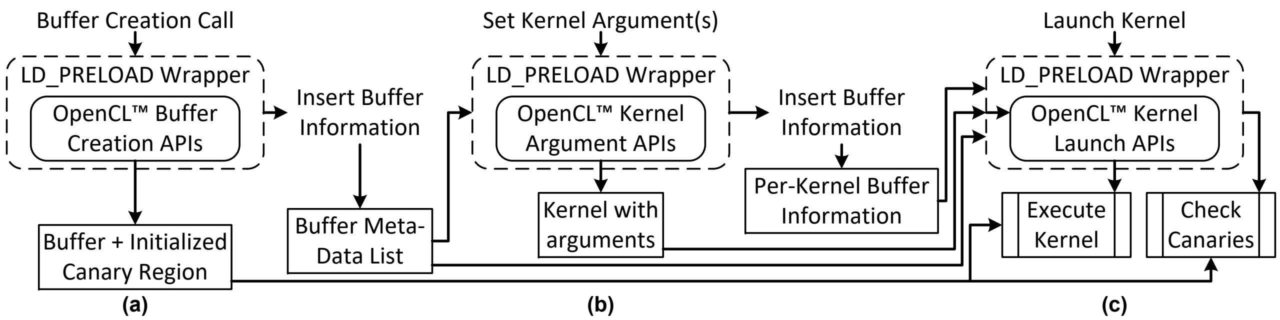


Figure 2. Architecture of our GPU Buffer Overflow Detector. (a) shows the wrapper for OpenCL buffer creation, which allows us to extend the allocations and add canary regions. (b) illustrates the wrapper for the functions that set kernel arguments. These allow our tool to know which buffers must be checked when running a kernel. (c) depicts work that takes place upon launching a kernel, which includes checking the canary values after the kernel completes to detect any overflows.

The flag `CL_MEM_USE_HOST_PTR` adds some difficulty to this scheme, since it allows previously allocated CPU memory to be used as an OpenCL buffer. We are unable to resize this region because we cannot update all of the user's pointers to it. To work around this problem, we create an extended *shadow copy* of the buffer whenever we are about to execute a kernel. The copy is extended with canaries, and, upon completion of the kernel, the useful data in the shadow buffer is copied back to the original host memory. This is valid because the OpenCL standard allows implementations "to cache the buffer contents" of host pointer regions. Our tool caches them until after our canary check completes.

We use shadow copies more generally to solve problems arising from buffer creation using pointers to existing allocations. Similar to using the `CL_MEM_USE_HOST_PTR` flag, sub-buffers are references to previously allocated buffers. Images may also be created with a previously existing buffer or image. In these cases, because our tool cannot update all references to the original data, a shadow copy is used.

3.2 Setting Kernel Arguments

Figure 2(b) shows how we catch the calls that are used to assign arguments to OpenCL kernels, `clSetKernelArg` and `clSetKernelArgSVMPointer`. Wrapping these allows us to maintain a list of the buffers that each kernel can access.

This list is used when the kernel is launched to know which buffers we should check for overflows. For each global memory buffer argument, we keep a list of buffer sizes, canary values, and pointers to the buffers' meta-data. In addition, this bookkeeping allows us to know if any SVM buffers are accessible from this kernel. As an optimization, we check for identical arguments; if two arguments use the same buffer, we only check the canary values once.

3.3 Kernel Enqueues

Figure 2(c) illustrates how we wrap the kernel launch function, `clEnqueueNDRangeKernel`. This is where buffer overflow detection takes place. The detector first analyzes the kernel's arguments. Kernels with no global buffers cannot cause overflows and are run like normal.

If, however, there are `cl_mem` buffers passed to the kernel, we must verify that these buffers' canary regions were not perturbed by the kernel. If any of the buffers were allocated without a canary region (e.g. `CL_MEM_USE_HOST_PTR` was used), the wrapper makes temporary shadow copies that contain enough space for our canary region and assign them as kernel arguments. We then launch the kernel.

While this kernel is executing, we enqueue a checker that will execute immediately after the original kernel finishes. This checker will verify the canaries of all the buffers that the kernel could have accessed.

SVM buffers add extra complexity, because we are unable to tell which will be accessed solely by looking at the kernel's arguments. SVM regions can contain pointers to other SVM regions, so if any argument to a kernel is to an SVM buffer, it is possible to access all other SVM buffers in the application. As such, if any of the kernel's arguments give it access to an SVM buffer, the canary regions for all SVM buffers in the application must be verified.

Checking image canaries also adds complexity. In order to detect overflows in any dimension (e.g. writing past the end of a row in a 2D image), we allocate a canary region per dimension. As such, the number of canaries depends on both the number of dimensions and their size. We therefore read the canaries hierarchically from the image into a one-dimensional array. This flattened collection of all image canaries is fed into a checker kernel along with a buffer that contains the end point for each image.

Should the verification function find an overflow, a debug message is printed to the screen and, optionally, execution is halted. The debug message, shown later in Figure 11, shows the kernel name, the argument name, and where in the canary region the first corruption happened. We are able to obtain the function argument's name by using the `clGetKernelArgInfo` function, since we know the argument index of the overflowed buffer. This does not work for SVM regions that are not passed as kernel arguments.

Finally, any shadow copy buffers are copied back to their CPU memory regions and the application continues.

3.4 API Checking

We also perform simple checks for functions that operate directly on cl_mem objects (e.g. `clEnqueueWriteBuffer`). These quickly compare the inputs for the operation with the cl_mem's instantiated size. This identifies overflows not caused by kernels and prevents us from later finding them with a checker kernel and misattributing the error.

4. Accelerating Buffer Overflow Detection

Section 3 described our buffer overflow detector, while this section describes techniques to increase its performance.

We use a GPU microbenchmark with a variable number of buffers to test these techniques. Our detector will check each buffer's canary values after the work kernel (which does no real work) ends. We then record time taken to perform the canary checks, allowing us to test the overheads of our tool at a variety of configurations. More buffers will lead to more checker overhead, because there are more checks to perform.

CPU vs GPU Checkers Intuitively, GPUs should excel at the parallel task of checking canary values. However, GPUs must amortize kernel launch overheads and need a significant amount of parallel work to fill their hardware resources. As such, we compared the overheads of checking canary values on both the CPU and the GPU as we vary the number of buffers (and thus the number of canaries).

For the CPU checker, our `clEnqueueNDRangeKernel` wrapper asynchronously enqueues a command to read the canary regions back to the host after the kernel finishes. A call to `clWaitForEvents` allows us to wait until this read completes, and a single CPU thread then checks the canaries.

For the GPU checker, our wrapper launches dependent checker kernel(s) immediately after the work kernel. The GPU checker will begin execution after the work kernel ends, and it will check the canary values in parallel.

Checking multiple buffers per kernel The simplest GPU canary checker uses one kernel per buffer, where the kernel's arguments are the buffer to check and the offset to the canary region. This results in poor GPU utilization since each buffer only has a few thousand canary values to check.

A slightly more complicated solution uses a single kernel that takes a variable number of buffers in a fixed number of kernel arguments. We accomplish this by copying the canary regions from all of the buffers into a single buffer. Afterwards a single parallel kernel can check the entries from many buffers at once, leading to better GPU utilization.

Utilizing SVM Pointers Like cl_mem buffers, the canaries in SVM regions can also be checked using either one kernel per buffer or a single kernel that checks copies of the canaries for all buffers. Alternatively, it is possible to create a buffer of SVM pointers, each of which points to the beginning of a canary regions in the original SVM buffer. This allows the checker kernel to directly read the SVM regions' canary values without requiring any extra copies.

Cleaning Modified Canaries We must reset any corrupted canary values before subsequent kernel iterations to avoid falsely declaring more buffer overflows. This is faster for the checkers that directly check the original buffer (like the SVM-pointer method), since they can immediately write over the modified canaries. For checkers that use canary copies, we use an asynchronous `clEnqueueFill<X>` to reset the canary regions after they have been copied.

Asynchronous Checking Stalling the CPUs until the work kernel and canary checks complete can cause significant overhead. First, the application itself may have CPU work that can take place while the work kernel is running; adding synchronous canary checks will eliminate this parallelism. In addition, launching GPU checker kernels synchronously can expose dozens of microseconds of launch overhead.

To prevent this, we asynchronously launch the GPU checker kernels and use OpenCL™ events to force them to wait on the work kernels. We then launch a thread that waits on the checker's completion event in order to print any debug messages. The checker's event is returned to the application so that waiting on the work kernel will also wait on the checker. Calls to `clGetEventProfilingInfo`, however, return the profiling information for the worker kernel.

Overheads of Checking Canaries The overheads of these techniques across various numbers of buffers are compared in Figures 3-5. The overhead of the GPU checkers are split into two parts: the added time spent on the host arranging canary regions and launching kernels, and the added time spent in the checker kernels. We note that applications which asynchronously launch their work kernels could perform other useful work in parallel to these checks.

Figure 3 shows the overheads of checking cl_mem buffers. For small numbers of buffers, and consequently small data sets and transfers, checking on the CPU results in less overhead. In these situations, the GPU has little work to do, and amortizing the launch overheads is more difficult.

As more buffers are added, using a single GPU checker for all of the buffers results in less overhead. The time spent marshaling the canary values increases along with the number of buffers, but the checker kernel time increase more slowly, since the GPU can check many canaries in parallel. The difference in kernel times between using one kernel per buffer and using one kernel for all buffers demonstrates the benefit of running many canary checks in parallel.

Figure 4 shows a similar test for SVM buffers. The CPU checker is always slower here because, it must first copy canaries into a smaller SVM before mapping that to the host. A GPU check can instead use SVM copies or pointers and remain relatively unaffected by data movement and mapping.

Passing an array of SVM pointers to the checker kernel (rather than marshaling the canaries themselves) leads to lower host-side overheads. The most efficient way to check SVM buffers is therefore using one kernel for all canaries and passing it an array of pointers to the canary regions.

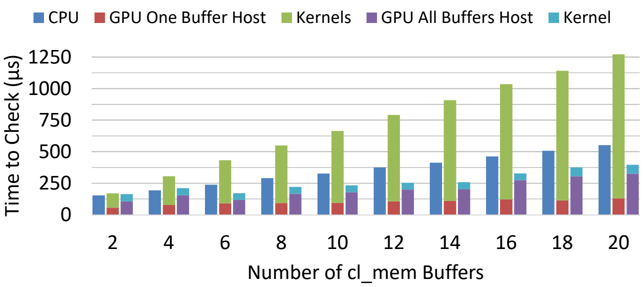


Figure 3. Time to check cl_mem buffers. Using one checker kernel per buffer on the GPU suffers from underutilization, so the time spent in the kernels quickly increases.

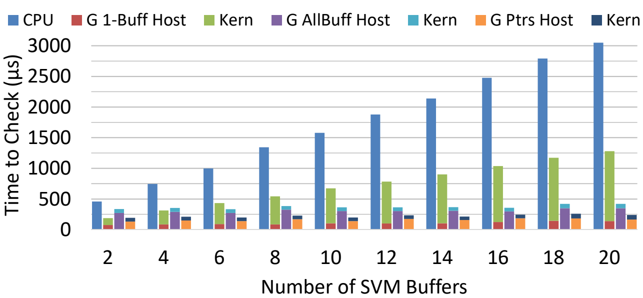


Figure 4. Time to check SVM buffers. The CPU overhead is higher here because we must copy a subsection of the SVM to a shorter SVM to map it back to the host. Single-buffer GPU checks still suffers from underutilization. Using SVM pointers speeds up the multi-buffer GPU check.

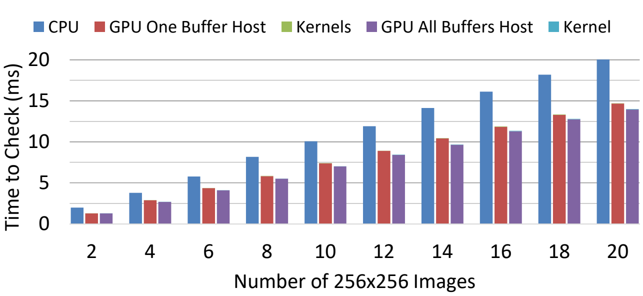


Figure 5. Time to check images. Images have many canaries due to their multi-dimensional nature, so their total check time is high. GPU checks eliminate bus transfer times.

Figure 5 shows this test for images. These multi-dimensional buffers have many canary regions. For instance, a 2D image with 256 rows has 257 canary regions – one at the end of each row, and one beyond the final column. Checkers therefore spend a great deal of time on the host enqueueing data transfers. Performing the checks on the GPU saves time by avoiding the use of the PCIe® bus for most data transfers.

Amortizing Kernel Compilation The previous sections showed that GPU canary checks can increase performance through parallelism and reduced bus transfers. These tests, however, did not include the cost of compiling the GPU checker kernel, which must be paid once each time the application is run. Figures 6-8 show the checker costs, including compilation time, as we repeatedly call the work kernel.

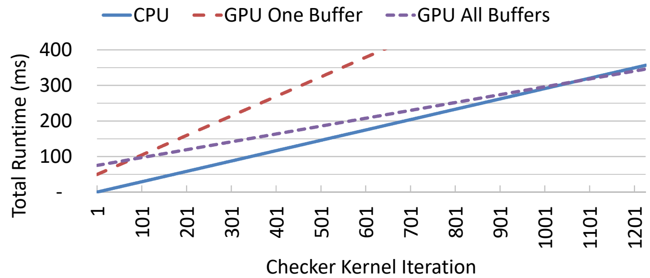


Figure 6. Total checker runtime across iterations checking 8 cl_mem buffers. The large kernel compilation time for GPU checkers takes many iterations to amortize.

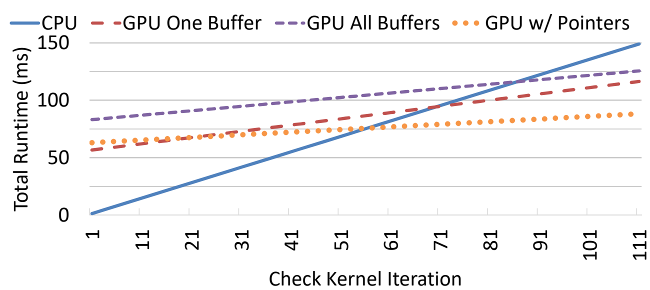


Figure 7. Total checker runtime across iterations checking 8 SVM buffers. The GPU-based SVM checks are much faster, so it takes less time to amortize the checker kernel compilation overheads.

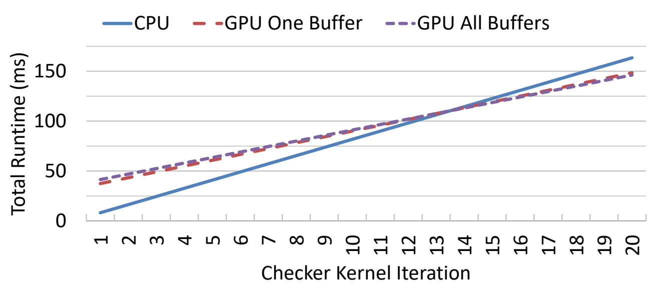


Figure 8. Total checker runtime across iterations checking 8 256×256 images. The host overheads make it easy for GPU kernels to amortize their initial compilation time.

Figure 6 shows that it takes more than 1000 kernel invocations before the compilation overhead is amortized when checking cl_mem buffers on the GPU. Because of this, we fall back to performing all canary checks for cl_mem buffers on the CPU. While some programs may run thousands of iterations, the benefits of GPU checks would only slowly reach those implied by Figure 3. Dynamically switching between CPU and GPU checks based on the number of observed iterations is an interesting direction for future study.

Figures 7 and 8 show how many iterations it takes to amortized kernel compilation time for SVM and image buffers, respectively. For these buffer types, GPU checks are much faster for each iteration. As such, the difference is great enough as to make up for the compilation time of the kernel within a few iterations. As such, for these buffer types, we always perform the checks on the GPU.

5. Benchmark Analysis

This section reports the overheads caused by running an application under our buffer overflow detector. We describe our experimental setup and benchmarks in Sections 5.1, detail the memory overheads caused by our tool in Section 5.2, and show performance overheads in Section 5.3.

5.1 Experimental Setup

All of our experiments were performed on a system with a 3.7 GHz AMD A10-7850K CPU, 32 GB of DDR3-1866, and an AMD FirePro™ W9100 discrete GPU. The GPU's core runs at 930 MHz, it has 320 GB/s of memory bandwidth to its 16 GB of GDDR5 memory, and it is connected to the CPU over a 3rd Generation PCIe® x8 connection.

The system ran Ubuntu 14.04.4 LTS and version 15.30.3 of the AMD Catalyst™ graphics drivers (`fglrx`). We used the AMD APP SDK v3.0 as our OpenCL™ runtime.

We ran 175 benchmarks from 16 open source OpenCL benchmark suites: the AMD APP SDK, FinanceBench [36], GPU-STREAM [23], Hetero-Mark [55], CloverLeaf and TeaLeaf from Mantevo [51], NAS Parallel Benchmarks [67], OpenDwarfs [32], Pannotia [16], Parboil [72], Phoronix [47], PolyBench/ACC [35], Rodinia [15], SHOC [22], StreamMR [30], ViennaCL [66], and the exascale proxy applications CoMD, LULESH, SNAP, and XSBench [1].

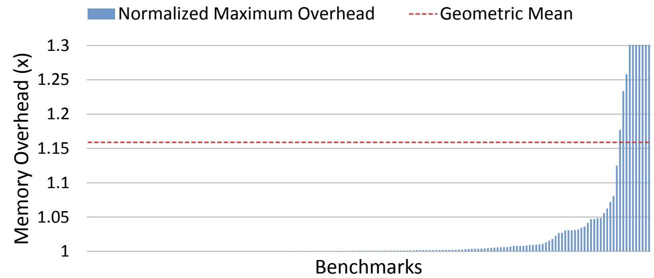
We ran two series of experiments. The first measured the memory overhead caused by extending OpenCL buffer allocations with 8 KB of canaries. We use this to show that the memory overhead of our tool is manageable and will not break most applications. The second set of experiments measured the change in wall-clock time between running these applications alone and with our buffer overflow detector. Because of the large number of benchmarks, these results are shown as the geometric mean of all benchmarks within each suite, as well as the geometric mean across all benchmarks.

5.2 Application Memory Overheads

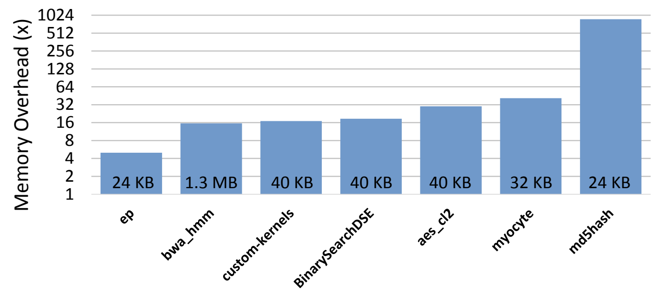
Figure 9 shows the maximum amount of OpenCL™ buffer space added by using our tool. Figure 9(a) sorts all of the benchmarks from lowest to highest overhead, while Figure 9(b) focuses on those with overheads higher than 30%.

The geometric mean of this overhead across all 175 benchmarks is 16%, though it is often much less than 1%. The median overhead is 0.1%. Nevertheless, some of the benchmarks see very high overheads because the canary regions are a fixed 8 KB, while buffer allocation may be small. Our detector can also use some additional internal buffers to store things like arrays of SVM canary pointers. As shown in Figure 9(b), these overheads can reach almost 1000×.

For example, the SHOC benchmark `md5hash` allocates three cl_mem buffers that are 4, 8, and 16 bytes, respectively. Each of these buffers is then extended with 8 KB of canaries, increasing the aggregate buffer size by 24 KB, or 878× the initial 28-byte allocation.



(a) The memory overhead of our tool across 175 benchmarks.



(b) A detailed view of the highest 7 memory overheads from (a).

Figure 9. OpenCL memory overhead when using our detector. 9(b) details the 7 benchmarks with overheads higher than 30% and lists the absolute amount of added storage.

In addition to the relative overheads illustrated with bars, Figure 9(b) lists the absolute amount of OpenCL buffer space that our detector adds. In general, the programs that have large relative overheads have small absolute overheads because the buffer space used by the application is small. This implies that our tool will rarely cause major memory pressure issues that will prevent applications from running.

Nevertheless, our 8 KB canary size was somewhat arbitrarily chosen based on our GPU's maximum workgroup size (256), the length of a double (8 bytes), and width vector width recommended for older AMD GPUs (4). We believe there is future research in sizing canary regions to maximize error coverage while minimizing memory overheads.

5.3 Application Performance Overheads

Our detector checks the canaries for each buffer that a kernel can access, and the canary regions are a fixed size. As such, the runtime of our checker should scale linearly with the number of buffers and kernel invocations In contrast, the relative overhead of our detector depends on the runtime of the kernels, since short kernels will make the checker time more prominent. As such, it is useful to analyze the performance overheads on real programs.

Figure 10 shows the runtime overhead of our tool on the 175 OpenCL™ enhanced applications described in the previous sections. We divide the benchmarks into their 16 respective suites in order to improve legibility. The blue bars show the geometric mean of the runtime overheads within a suite, while the upper and lower bars show the highest and lowest runtime overheads within that suite, respectively.

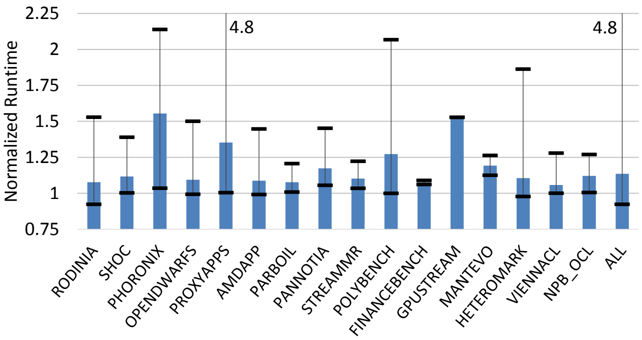


Figure 10. Normalized runtime when running buffer overflow detection. Each blue bar is the geometric mean of the overheads within a suite. The upper and lower bars represent the maximum and minimum overheads within a suite, respectively. Our detector causes an average 14% overhead.

The worst overhead caused by our tool is a 4.8× slowdown in the proxy application SNAP_MPI [24]. This overhead is caused by a significant number of very small kernels that are launched synchronously. The kernels `zero_buffer` and `sweep_plane` take an average of 400 µs and 230 µs, respectively, and are called repeatedly. `sweep_plane` also uses 15 cl_mem buffers. As such, the execution time of the checkers is greater than the runtime of the real kernels. In addition, both kernels are executed synchronously; very soon after they launch, all other CPU work stops to wait for the kernel to complete. The execution time of the check is therefore almost fully exposed as overhead.

Despite this, our detector rarely caused overheads greater than 50%; only 10 applications saw more than 50% slowdown. This typically occurred because, like SNAP_MPI, the program used short kernels or frequent synchronization. Short kernels do not last long enough to amortize the cost of canary checking, and synchronization prevents our detector from overlapping checks with unrelated CPU work.

In rare situations, our tool caused the application to run slightly faster. This was due to secondary effects on the host, such as link order and memory layout changes [56]. For instance, Rodinia's `hotspot3D` consistently wrote its output files faster when using our tool, yielding a 7% speedup.

The final category, "ALL," shows the geometric mean, maximum, and minimum overheads across all 175 benchmarks. Our tool causes an average slowdown of 14% across these benchmarks, with a median overhead of just 6%. We believe that this level of performance will allow our tool to be used in continuous integration systems, nightly and regression tests, and other development situations that preclude more heavyweight tools.

Looking forward, GPU applications are moving towards more asynchronous operation [2] and launching further work from within kernels [37, 43, 64]. Both of these would help amortize or hide the costs of our canary checker. As such, we believe that the overheads seen by our detector will decrease on future workloads.

```
*******Error, buffer overflow detected*******
Kernel: splitRearrange
Buffer: keys_o
First wrote 1 word(s) past the end
```

Figure 11. Example buffer overflow detector output. It lists the kernel that caused the error, which buffer was affected, and where the first corrupting write happened.

6. Buffer Overflows Detected

An important test of software analysis tools is whether they can find real problems, so this section details the overflows that we found in our benchmarks. When our tool detects an overflow, it emits information about the kernel that caused the overflow and how far past the end of the buffer it wrote, as shown in Figure 11. This can help pinpoint the problem, but finding and fixing the root cause is still a manual process.

In aggregate, our tool found buffer overflows in 13 kernels across 7 programs and returned no false positives. These errors are shown in Table 1, which explains what problems we found when writing patches or workarounds.

The error in Parboil's `mri-gridding` occurs because the sizes of two output buffers, `keys_o` and `values_o`, are based on the number of input elements, `n`. The `splitRearrange` kernel can cause an overflow because it assumes their lengths are evenly divisible by four. A fix for this is to allocate the buffers based on $(((n + 3)/4) * 4)$. This error may have also been found by the authors of Oclgrind when they claimed that their detector "has been used to identify bugs in real OpenCL applications ... including ... Parboil" [61].

In the StreamMR benchmarks `kmeans` and `wordcount`, some output buffers are allocated based on previous kernel results. The `copyerHashToArray` kernels write contiguous data to these buffers from all 64 work items in each workgroup, but the host does not ensure their sizes are evenly divisible by 64. In addition, these kernels sometimes use an uninitialized variable to access their hash tables. Both of these can result in buffer overflows and are easily fixed.

Hetero-Mark's `kmeans` allocates the `features_swap` buffer based on `npoints`. The number of work items for the `kmeans_swap` kernel is set to be ≥`npoints` and evenly divide by 256. The kernel does not check the length of the buffer, so work items beyond `npoints` write past the end of the array based on their ID. This error is derived from a known problem in Rodinia's `kmeans`, and it can be fixed (as in Rodinia 3.1) by adding a length check into the kernel.

Hetero-Mark's `sw` allocates multiple buffers based on `sizeInBytes`, a product of `m_len` and `n_len`. The kernels `sw_init_velocities`, `sw_compute0`, `sw_compute1`, and `sw_update0` access these buffers using a variety of bad offset calculations. For instance `sw_compute0` will attempt to write to `z[n_len * m_len + m_len]`, which will result in a buffer overflow. We could not verify our fixes to each kernel's miscalculations, since we are not the application's authors. A workaround is to increase the size of `sizeInBytes`.

Suite	Benchmark	Kernel	Problem
Parboil [72]	mri-gridding	splitRearrange	keys_o and values_o are not allocated enough space.
StreamMR [30]	kmeans	copyerHashToArray	outputKeys, outputVals, and keyValOffsets are too small.
	wordcount	copyerHashToArray	outputKeys, outputVals, and keyValOffsets are too small.
Hetero-Mark [55]	OpenCL 1.2 kmeans	kmeans_swap	Incorrect range check in threads writing to feature_swap.
	OpenCL 2.0 kmeans	kmeans_swap	Incorrect range check in threads writing to feature_swap.
	OpenCL 1.2 sw	sw_compute0	Bad SizeInBytes causes cu, cv, and z to be too small.
		sw_compute1	Bad SizeInBytes causes u_next and v_next to be too small.
		sw_init_velocities	Bad SizeInBytes causes u and v to be too small.
		sw_update0	Bad SizeInBytes causes cu, cv, and z to be too small.
	OpenCL 2.0 sw	sw_compute0	Bad SizeInBytes causes cu, cv, and z to be too small.
		sw_compute1	Bad SizeInBytes causes u_next and v_next to be too small.
		sw_init_velocities	Bad SizeInBytes causes u and v to be too small.
		sw_update0	Bad SizeInBytes causes cu, cv, and z to be too small.

Table 1. Overview of the errors found by our tool. We found buffer overflows in 13 separate kernels across 7 benchmarks.

The source of the errors in the OpenCL 1.2 and 2.0 Hetero-Mark programs are very similar. However, the 2.0 benchmarks used SVM rather than cl_mem buffers, so tools that do not support SVM would not have been able to find the errors. As such, we categorize these as different errors.

The errors found by our tool remained hidden until now primarily because they rarely crash today's GPUs. However, we found that using a system to dynamically hold buffers in CPU memory (as demonstrated by Margiolas and O'Boyle [52]) resulted in observable problems. For example, the GPU buffer overflow would corrupt metadata in neighboring heap objects, leading to crashes or double-free errors [33]. This was especially difficult to debug without our tool, since common mechanisms like CPU-based watchpoints will not catch the offending writes from the GPU.

7. Related Work

This section discusses works related to detecting buffer overflows. Section 7.1 describes CPU-based detection tools, and Section 7.2 describes other analysis tools for GPUs.

7.1 CPU Buffer Overflow Detection

Perhaps the most popular memory analysis tool is the dynamic binary instrumentation engine Valgrind [58]. Its Memcheck tool searches for memory errors such as buffer overflows [69]. While this can find many problems, its runtime overhead (dozens of times slowdown) limits the situations where it can be used. Similar open source and commercial tools have roughly the same limitations [14, 42].

Compile-time instrumentation tools like AddressSanitizer [68], Baggy Bounds Checking [9], and SoftBound [57] can perform checks with overheads of roughly 10%-2× [68]. More traditionally, StackGuard (which inspired techniques used in GCC) inserts canaries before critical stack values and adds checks to verify them before they can cause security problems [21]. Our tool also uses canary values, but we do not require recompilation. Additionally, because our checks take place after the kernel completes, we cannot offer the same security guarantees as these inline checks.

The desire to further reduce overheads led to hardware-supported bounds checking, as in HardBound [25] and Intel's MPX [41]. Our GPU buffer overflow detector has the benefit of requiring no added hardware support.

Electric Fence [60], and related tools like DUMA [10], catch calls to malloc and add protected canary pages around the allocated memory. Writing to one of these canary pages will result in a page fault that will eventually crash the program. Our tool similarly wraps allocation calls to create canary regions, but it does not utilize virtual memory to catch overflows. Systems that allow the GPU to share a coherent virtual memory with the CPU will be able to use such techniques, but many current GPUs do not share the full virtual memory space with the CPU and do not allow the canary regions to be protected in this way.

7.2 GPU Analysis Tools

Ours is not the first GPU debugging tool, nor is it the first to search for GPU buffer overflows. This section compares other GPU analysis tools to our buffer overflow detector.

Oclgrind Oclgrind is an OpenCL™ device simulator that, like Valgrind for CPU applications, can be used to build analysis tools for OpenCL kernels [61]. Like Valgrind, one of the tools that comes prepackaged with Oclgrind is a memory access checker. Oclgrind presents itself to the kernel as a CPU device, however, which limits its ability to be automatically run on some applications; the majority of our benchmarks would need manual modifications to run in Oclgrind.

While requiring manual intervention adds some difficulty, the primary limitation of Oclgrind is its execution overhead. Because it simulates OpenCL devices on the CPU, it adds extra analysis overheads and also runs the original kernel much slower. We tested a subset of our benchmarks and found that Oclgrind ran them up to 300× slower than native execution. This aligns with the authors' claim of running "typically a couple of orders of magnitude slower than a regular CPU implementation." Compared to our tool's 14%, 300× slowdowns severely limit what Oclgrind can test.

Nonetheless, the authors used Oclgrind to find numerous real buffer overflows. Based on their descriptions, it is likely that they found the same overflow in `mri-gridding` that our tool found. In addition, Oclgrind can perform more analyses than our specialized buffer overflow detector.

GPU Binary Instrumentation SASSI [71] and GPU Ocelot [31] are tools that can dynamically instrument GPU kernels. SASSI instruments Nvidia's GPU assembly, while GPU Ocelot instruments the CUDA intermediate language.

Like Oclgrind, these tools are more general than our overflow detector. In addition, because they allow instrumented code to run on the GPU, these tools are much faster than Oclgrind (though they are still slower than our technique). For example, while the SASSI authors did not directly test a buffer overflow detector, similar tools (memory divergence testing and value profiling) average between 60% and 140% slowdowns at the application level. In addition, SASSI only works on Nvidia GPUs and GPU Ocelot's AMD GPU support is experimental; our tool works at the OpenCL API level and could be used on any compliant GPU.

WebCL Validator Khronos's WebCL Validator is a source-to-source compiler that adds dynamic memory checks into WebCL programs [44]. Unlike our tool, the WebCL validator can work on private, local, and global memory because it adds checks during a compiler pass. Essentially, it modifies kernel memory accesses so that they will always be contained within verified memory regions. This prevents accesses outside of their targets, but adds runtime overheads. The authors measured GPU overheads of $3\times$ unless the end-user makes careful code modifications [48].

According to the authors, one of the major features of WebCL that helps their tool is that they "know start and end addresses of all the memory which is meant to be accessed." This is possible in WebCL because it passes these limits as kernel arguments; this is not the case for most OpenCL kernels. In addition, SVM buffers add a layer of difficulty; pointers can lead to accesses to any other SVM buffer, making it difficult to quickly constrain accesses to these buffers.

CUDA-MEMCHECK CUDA-MEMCHECK is a memory checking tool from Nvidia for CUDA kernels [34]. It is closed source, so we cannot say for certain how our technique compares. However, because it can identify overflows when they happen and associate them with the line of kernel code that caused them, CUDA-MEMCHECK likely uses the compiler to add checks into the kernel. This would lead to runtime slowdowns; the CUDA-MEMCHECK manual claims that "applications run much slower under CUDA-MEMCHECK tools," and the authors of Cudagrind measured this slowdown to be roughly 120% [11]. In addition, as mentioned for the WebCL Validator, this tool would need to instrument the CUDA APIs in some way in order to know the limits of the allocated buffers. Finally, unlike our tool, CUDA-MEMCHECK only works on GPUs from Nvidia.

8. Conclusions and Future Work

This work introduced a GPU buffer overflow detector for OpenCL™ kernels. We demonstrated that buffer overflows can happen in GPU kernels and detailed the design of a canary-based tool to automatically find these problems.

Our tool wraps OpenCL API calls in order to catch buffer allocations, expand each buffer, and insert canary values after it. We then catch kernel argument assignments in order to know which buffers are vulnerable to overflow in a kernel. By catching kernel invocations, we can check the buffers' canary values after the kernel finishes to detect overflows.

We demonstrated how to accelerate these tests by asynchronously checking the canary values on the GPU. This technique scales with the number of buffers, and it leads to an average overhead of only 14% across 175 applications.

Finally, we showed that our tool can find real errors. In the applications we tested, we found 13 separate buffer overflows across 7 of the programs. These ranged from missing range checks in the kernels to incorrect host-side allocations.

Looking forward, there are a number of future directions to take this work. Our tool does not work on fine-grained system SVM, which allows GPUs to coherently access data using pointers allocated from CPU functions like `malloc` [49]. Such systems could detect overflows using tools like Electric Fence [60], since they rely on virtual memory protection mechanisms or page migration to work [38].

It would also be useful to add compiler-focused features to our tool. Our buffer meta-data could be passed as arguments to the kernel and, like the WebCL Validator [44] or CUDA-MEMCHECK [34], the compiler could add checks within the kernel to detect buffer overflows. This could pinpoint problems within the kernel and detect overflows in local and shared memory at the expense of vendor neutrality.

Finally, while this work focused on GPUs, more accelerators are being introduced into heterogeneous systems [4, 5, 17–20, 63, 76]. Building broadly useful tools that work across these accelerators will be an important area of work.

The tool described in this work is available at: `https://github.com/GPUOpen-ProfessionalCompute-Tools/clARMOR`

Acknowledgments

We would like to thank Arkaprava Basu, Leonardo Piga, and the anonymous reviewers for providing comments and suggestions that improved the final version of this paper. Thanks also to Michael LeBeane for his naming recommendations.

References

[1] Advanced Micro Devices, Inc. AMD ComputeApps. `https://github.com/AMDComputeLibraries/ComputeApps`.

[2] Advanced Micro Devices, Inc. Asynchronous Shaders: Unlocking the Full Potential of the GPU. White Paper, 2015.

[3] Advanced Micro Devices, Inc. AMD APP SDK OpenCL™ Optimization Guide, Rev. 1.0, August 2015.

[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2015.

[5] K. Aingaran, S. Jairath, and D. Lutz. Software in Silicon in the Oracle SPARC M7 Processor. Presented at Hot Chips, 2016.

[6] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 49(14), 1996.

[7] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, Oct. 1972.

[8] G. Andryeyev. OpenCL with AMD FirePro W9100. Presented at Toronto SIGGRAPH. `http://toronto.siggraph.org/wp-content/uploads/2015/05/uoft-ocl.pdf`, May 2015. Accessed: 2016-12-02.

[9] P. Arkitidis, M. Costa, M. Castro, and S. Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *Proc. of the USENIX Security Symp.*, 2009.

[10] H. Aygün. Speicher-Debugging mit DUMA. *Programmieren unter Linux*, 1(1):74–78, 2005.

[11] T. M. Baumann and J. Gracia. Cudagrind: Memory-Usage Checking for CUDA. In *Proc. of the Int'l Workshop on Parallel Tools for High Performance Computing*, 2014.

[12] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, 2007.

[13] N. Brookwood. Everything You Always Wanted to Know About HSA But Were Afraid to Ask. White Paper, October 2013.

[14] D. Bruening and Q. Zhao. Practical Memory Checking with Dr. Memory. In *Proc. of the Int'l Symp. on Code Generation and Optimization (CGO)*, 2011.

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)*, 2009.

[16] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding Irregular GPGPU Graph Applications. In *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)*, 2013.

[17] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[18] J. Clemons, A. Pellegrini, S. Savarese, and T. Austin. EVA: An Efficient Vision Architecture for Mobile Systems. In *Proc. of the Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2013.

[19] L. Codrescu. Qualcomm Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications. Presented at Hot Chips, 2013.

[20] J. Coombs and R. Prabhu. OpenCV on TI's DSP+ARM® Platforms: Mitigating the Challenges of Porting OpenCV to Embedded Platforms. Texas Instruments White Paper, 2011.

[21] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. of the USENIX Security Symposium*, 1998.

[22] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proc. of the Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2010.

[23] T. Deakin and S. McIntosh-Smith. GPU-STREAM: Benchmarking the Achievable Memory Bandwidth of Graphics Processing Units. In *Poster Session at the Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[24] T. Deakin, S. McIntosh-Smith, and W. Gaudin. Expressing Parallelism on Many-Core for Deterministic Discrete Ordinates Transport. In *Proc. of the Int'l Conf. on Cluster Computing (CLUSTER)*, 2015.

[25] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. HardBound: Architectural Support for Spatial Safety of the C Programming Language. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[26] B. Di, J. Sun, and H. Chen. A Study of Overflow Vulnerabilities on GPUs. In *Proc. of the Int'l Conf. on Network and Parallel Computing (NPC)*, 2016.

[27] M. Ditty, J. Montrym, and C. Wittenbrink. Nvidia's Tegra K1 System-on-Chip. Presented at Hot Chips, 2014.

[28] J. Doweck and W. Kao. Inside 6th Generation Intel® Core™: New Microarchitecture Code Named Skylake. Presented at Hot Chips, 2016.

[29] F. C. Eigler. Mudflap: Pointer Use Checking for C/C++. In *GCC Developers Summit*, 2003.

[30] M. Elteir, H. Lin, W. Feng, and T. Scogland. StreamMR: An Optimized MapReduce Framework for AMD GPUs. In *Proc. of the Int'l. Conf. on Parallel and Distributed Systems (ICPADS)*, 2011.

[31] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan. A Framework for Dynamically Instrumenting GPU Compute Applications within GPU Ocelot. In *Proc. of the Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2011.

[32] W. Feng, H. Lin, T. Scogland, and J. Zhang. OpenCL and the 13 Dwarfs: A Work in Progress. In *Proc. of the Int'l Conf. on Performance Engineering (ICPE)*, 2012.

[33] J. N. Ferguson. Understanding the Heap By Breaking It. In *Black Hat USA*, 2007.

[34] G. Gerfin and V. Venkataraman. Debugging Experience with CUDA-GDB and CUDA-MEMCHECK. Presented at GPU Technology Conference (GTC), 2012.

[35] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a High-Level Language to GPU Codes. In *Proc. of Innovative Parallel Computing (InPar)*, 2012.

[36] S. Grauer-Gray, W. Killian, R. Searles, and J. Cavazos. Accelerating Financial Applications on the GPU. In *Proc. of the Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2013.

[37] I. E. Hajj, J. Gómez-Luna, C. Li, L.-W. Chang, D. Milojicic, and W. Hwu. KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism. In *Proc. of the Int'l. Symp. on Microarchitecture (MICRO)*, 2016.

[38] M. Harris. CUDA 8 and Beyond. Presented at the GPU Technology Conf., 2016.

[39] N. Hasabnis, A. Misra, and R. Sekar. Light-weight Bounds Checking. In *Proc. of the Int'l Symp. on Code Generation and Optimization (CGO)*, 2012.

[40] HSA Foundation. Heterogeneous System Architecture. `http://www.hsafoundation.com/`. Accessed: 2016-12-02.

[41] Intel Corp. Introduction to Intel® Memory Protection Extensions. Technical report, 2013. Accessed: 2016-12-02.

[42] Intel Corp. Intel® Inspector 2017. `https://software.intel.com/en-us/intel-inspector-xe`, 2016. Accessed: 2016-12-02.

[43] S. Jones. Introduction to Dynamic Parallelism. Presented at GPU Technology Conference (GTC), 2012.

[44] Khronos Group. WebCL Validator. `https://github.com/KhronosGroup/webcl-validator`, 2014. Accessed: 2016-12-02.

[45] A. Krennmair. ContraPolice: a libc Extension for Protecting Applications from Heap-Smashing Attacks. `http://synflood.at/tmp/website/doc/cp.pdf`, 2003.

[46] G. Krishnan, D. Bouvier, L. Zhang, and P. Dongara. Energy Efficient Graphics and Multimedia in 28nm Carrizo APU. Presented at Hot Chips, 2015.

[47] M. Larabel and M. Tippett. Phoronix Test Suite. `http://www.phoronix-test-suite.com/`. Accessed: 2016-12-02.

[48] M. Lepistö and R. Ylimäki. WebCL Memory Protection: Source-to-Source Instrumentation. `http://learningwebcl.com/wp-content/uploads/2013/11/WebCLMemoryProtection.pdf`, 2013. Accessed: 2016-12-02.

[49] B. T. Lewis. Performance and Programmability Trade-offs in the OpenCL 2.0 SVM and Memory Model. Presented at the Workshop on Determinism and Correctness in Parallel Programming (WoDET), 2014.

[50] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: Trading Address Space for Reliability and Security. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[51] A. C. Mallinson, D. A. Beckingsale, W. P. Gaudin, J. A. Herdman, J. M. Levesque, and S. A. Jarvis. Cloverleaf: Preparing hydrodynamics codes for exascale. The Cray User Group, 2013.

[52] C. Margiolas and M. F. P. O'Boyle. Portable and Transparent Host-Device Communication Optimization for GPGPU Environments. In *Proc. of the Int'l Symp. on Code Generation and Optimization (CGO)*, 2014.

[53] H. Meer. Memory Corruption Attacks: The (almost) Complete History. In *Black Hat USA*, 2010.

[54] A. Miele. Buffer Overflow Vulnerabilities in CUDA: A Preliminary Analysis. *Journal of Computer Virology and Hacking Techniques*, 12(2):113–120, May 2016.

[55] S. Mukherjee, X. Gong, L. Yu, C. McCardwell, Y. Ukidave, T. Dao, F. N. Paravecino, , and D. Kaeli. Exploring the Features of OpenCL 2.0. In *Proc. of the Int'l Workshop on OpenCL (IWOCL)*, 2015.

[56] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[57] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, 2009.

[58] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, 2007.

[59] K. Pattabiraman, V. Grover, and B. G. Zorn. Samurai: Protecting Critical Data in Unsafe Languages. In *Proc. of the European Conf. on Computer Systems (EuroSys)*, 2008.

[60] B. Perens. Electric Fence. `http://linux.die.net/man/3/efence`, 1987. Accessed: 2016-12-02.

[61] J. Price and S. McIntosh-Smith. Oclgrind: An Extensible OpenCL Device Simulator. In *Proc. of the Int'l Workshop on OpenCL (IWOCL)*, 2015.

[62] K. Pulo. Fun with LD_PRELOAD. Presented at linux.conf.au, 2009.

[63] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2014.

[64] P. Raghavendra. OpenCL™ 2.0: Device Enqueue and Workgroup Built-in Functions. `http://developer.amd.com/community/blog/2014/11/17/opencl-2-0-device-enqueue/`. Accessed: 2016-12-02.

[65] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time Detection of Heap-based Overflows. In *Proc. of the Large Installation Systems Administration Conf. (LISA)*, 2003.

[66] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. In *Int'l Workshop on GPUs and Scientific Applications (GPUScA)*, 2010.

[67] S. Seo, G. Jo, and J. Lee. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)*, 2011.

[68] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proc. of the USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[69] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proc. of the USENIX Annual Technical Conference (USENIX ATC)*, 2005.

[70] R. Smith. HSA Foundation Update: More HSA Hardware Coming Soon. `http://www.anandtech.com/show/9690/`, Oct. 2015. Accessed: 2016-12-02.

[71] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler. Flexible Software Profiling of GPU Architectures. In *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2015.

[72] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, March 2012.

[73] J. Vesely, A. Basu, M. Oskin, G. Loh, and A. Bhattacharjee. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *Proc. of the Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2016.

[74] G. Watson. Dmalloc - Debug Malloc Library. `http://dmalloc.com/`, 1992. Accessed: 2016-12-02.

[75] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, March-April 2007.

[76] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The Architecture and Design of a Database Processing Unit. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.