GASan: Evaluating the Overheads of Modifying ASan to Run on GPUs

Ammar Ratnani

Abstract—Graphics Processing Units (GPUs) have become indispensable for high-performance computing. However, memory safety on GPUs remains an unsolved problem — and a particularly important one given the sensitive data they now find themselves handling. To solve it, this paper proposes GPU Address Sanitizer (GASan), a modification of Google's Address Sanitizer to run on GPUs. It notes that this approach would be practical, not requiring any major infrastructure changes. It measures the performance impact and memory footprint of the modification. It finds a 31% average runtime overhead — much better than with ASan on CPUs. It also finds a neglibible footprint outside of what is strictly needed for protection.

1 Introduction

Graphics Processing Units (GPUs) have become ubiquitous as a general-purpose computation platform. Their massively parallel nature is used to accelerate cryptocurrency mining, scientific applications, and machine learning. However, almost all GPU code is written in memory-unsafe languages for performance. It is therefore possible to mount memory corruption attacks on GPUs [2], [12], [15] — a particularly alarming fact given the sensitive data they now find themselves handling. Fortunately, memory safety on CPUs is a well-understood problem with many proposed solutions both at the hardware and software level [9], [10], [17]. It is therefore worth investigating whether these same solutions are practical on GPUs, either as is or with modifications.

Address Sanitizer (ASan) [17] is one such solution. It is certainly the most popular one, having been incorporated into both GCC and LLVM under `-fsanitize=address` [5], [16]. From a theoretical standpoint, unlike many other memory safety techniques, ASan is not probabilistic. It does not rely on canaries to protect out-of-bounds accesses, instead retaining metadata for the entire virtual address space. More importantly for practical adoption, ASan also operates entirely at the compiler and runtime level, requiring no changes to the operating system or the hardware. Particularly because of the proprietary nature of CUDA's infrastructure, that fact makes ASan a particularly enticing technique to ensure memory safety on GPUs.

Therefore, this paper proposes GPU Address Sanitizer (GASan). It details the changes required to the original ASan framework to make it work on GPUs, and it evaluates the memory and performance impact of those changes via simulation. Surprisingly, it finds that the average runtime overhead of GASan is much less than that of ASan — just 31% on average. It additionally finds that the technique has negligible overheads outside of what is necessary for protection — about 2x. This paper concludes that GASan would be a viable approach to GPU memory safety, especially since it can be implemented with current open-source software.

2 Background

As this paper presents a port of ASan [17], it assumes familiarity with the original work. This section briefly summarizes it, focusing on the components adopted by GASan.

2.1 ASan

ASan [17] builds its higher-level protections on its primitive of shadow memory. It has one shadow byte encode for one quadword of normal memory. It constructs this compression by assuming that the start of any single allocation is aligned to eight bytes. Thus, any single quadword always contains some number k of bytes the program is allowed to access followed by exactly $8 - k$ inaccessible bytes. ASan stores this k value for each block of eight bytes.

To store this metadata, ASan requires one-eighth of the virtual address space. For example, a common 2^{47}B virtual address space would require $2^{44}\text{B} = 16\text{TiB}$ of shadow memory. However, note that this much space is only *reserved* on program startup. Shadow memory is only *committed* with physically-backed pages once the application requests to use the corresponding region of memory.[1] Thus, the memory footprint of ASan is approximately 12.5% over what the application actually uses.

From the primitive of shadow memory, ASan constructs encodings for stack content, global variables, and heap variables. Heap variables are the focus of GASan, so ASan's approach to them is summarized here. It allocates blocks of memory as accessible quadword aligned blocks surrounded on both sides by inaccessible "redzones". The allocator's metadata is stored in the redzones. That way, normal application code cannot corrupt it, since it has shadow memory checks inserted into it. The size of the redzones can be varied depending on the level of protection required. The defauklt was originally 128B [17], but this was later changed to the greater of 32B and 25% of the requested size.

3 GASan Implementation

GASan follows the same high-level approach as ASan. For every load or store to global memory, the compiler prepends instrumentation code to consult shadow memory and check that the access is valid. However, GASan necessarily differs from the original in implementation. Specifically, it modifies both how the shadow memory encodes bounds information and how memory is managed.

3.1 Shadow Memory Encoding

GASan makes two major changes to the bounds encoding. The first is out of necessity. ASan uses bytes with negative

1. On Linux, pages are reserved by marking them as `PROT_NONE`, and they are committed transparently via lazy zero-page allocation [16].

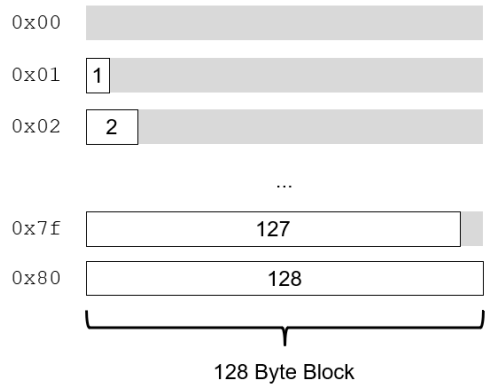

Fig. 1. Mapping of shadow byte values to the number of accessible memory locations in a block for GASan-128.

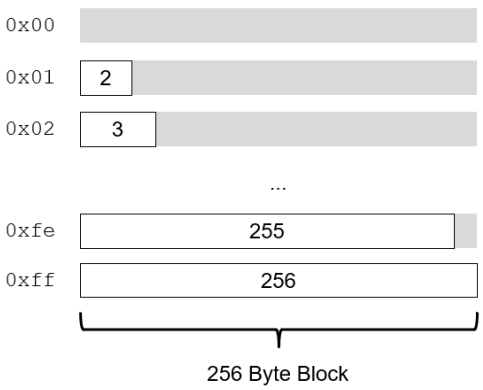

Fig. 2. Same as Fig. 1, but for GASan-256.

values to signal inaccessible blocks. However, CUDA's SASS does not appear to have an instruction to load a sign-extended byte from memory, and as such there is no simple way to test whether a shadow memory byte is negative. Therefore, the encoding is modified so that a value of `0x00` encodes an inaccessible block instead.

The other change is for optimization. Recall that ASan assumes pointers returned by `malloc` are aligned to eight bytes, and uses that insight to construct its eight-to-one mapping for shadow memory. That observation can be further exploited on GPUs, since device memory allocated by CUDA is aligned to 256B [14]. Ideally, this means the overhead of shadow memory only needs to be 0.4% instead of the usual 12.5%. Unfortunately, there is barely not enough space in the original encoding for 256-byte aligned data.

To meet these new constraints, the authors propose and evaluate two different encodings.

3.1.1 GASan-128

One method is to have each byte of shadow memory code for a 128-byte block. A shadow memory value n means the first n bytes of the block are accessible, for example with $n = \texttt{0x00}$ encoding an inaccessible block. In other words, a load or store of width `w` to address `a` is instrumented with the code below.[2]

```
size_t  s_off = (a >> 7) & ShadowMask;
size_t  b_off = (a & 0x7f) + w-1;
uint8_t s_val = *(ShadowBase + s_off);
if (s_val <= b_off)
    Crash();
// Continue
```

The authors term this approach GASan-128. The primary downside to this method is that it takes twice the amount of physical memory as would be required in the ideal case.

3.1.2 GASan-256

Alternatively, one can assume that no 256-byte block will ever have just a single byte allocated in it. With that case eliminated, each byte of shadow memory can code for 256B. A shadow value of $n \neq \texttt{0x00}$ means the first $n + 1$ bytes of the block are accessible, while $n = \texttt{0x00}$ denotes an inaccessible block. In code, a memory operation of width `w` at address `a` is prefixed with the listing below.[2]

```
size_t  s_off = (a >> 8) & ShadowMask;
size_t  b_off = (a & 0xff) + w-1;
uint8_t s_val = *(ShadowBase + s_off);
if (s_val == 0 || s_val < b_off)
    Crash();
// Continue
```

The authors term this approach GASan-256. It has half the memory footprint of GASan-128, but it requires the runtime library to ensure that no allocations are made with size congruent to $1 \bmod 256$. That scenario is unlikely in practice, and the runtime can just allocate an extra byte if it is ever encountered. However, this extra byte will be treated as accessible even though it was never asked for.

3.2 Memory Management

3.2.1 Challenges

Recall from Sec. 2.1 that ASan heavily uses the virtual memory facilities offered by the CPU to construct its shadow address space. However, many of those facilities are not offered by the GPU. For example, CUDA does not expose a way to mark allocations as read-only or inaccessible. More critically, the API gives no control over the placement of buffers in the GPU's virtual address space. This makes it impossible to reserve a section of the virtual address space for shadow memory, as would be done on the CPU. In a similar vein, GPUs seemingly cannot use lazy zero-page allocation.[3] This means it would not be possible to simply allocate the entire shadow memory space in one call — doing so would return `cudaErrorMemoryAllocation`.

Another constraint worth mentioning is that CUDA's allocation functions do not clear the device-side buffer they return. This is in contrast to `mmap` on the CPU, which guaranteed to return a buffer of zeros. This means initializing shadow memory incurs a `cudaMemset`, but the time that takes was measured to be negligible on human scales. Still, the authors try to minimize the number of calls required.

3.2.2 Proposal

In order to meet these harsh constraints, the authors propose allocating a pool on the device, and having the runtime service all allocations from that pool. This way, allocations are guaranteed to be in a small section of the virtual address space, so shadow memory only needs to code for that section. This in turn means shadow memory can be small enough to be physically backed.

However, the pool size cannot be fixed. If it were, its size would have to be very large to service memory-heavy

2. See Sec. 3.2.2 for definitions of `ShadowBase` and `ShadowMask`

3. This appears to be the case even on Pascal GPUs and later, which allow device memory oversubscription.

applications, which in turn would cause a dramatic increase in the footprint of applications that are memory-light. Thus, the authors also propose dynamically resizing the pool.

The pool is transparent to the application, so resizing it cannot invalidate any device pointers that are in use. However, CUDA provides no way to extend an allocation while keeping it at the same virtual address. To sidestep this, the authors propose adding an indirection layer between the "device pointers" returned to the application and the actual device pointers. Specifically, handles the application receives have their k most-significant bits replaced with a unique identifier.[4] That identifier is fed into a hash-table to retrieve the base address of the allocated buffer. The least-significant bits are treated as the offset into the buffer. The runtime intercepts all the library functions and kernel launches to perform this translation.

With the pool in place, constructing shadow memory is simple. It resides in its own allocation of size

$$\texttt{ShadowSize} = \text{NextPow2}\left(\frac{s}{2^{\texttt{ShadowScale}}}\right)$$

where s is the size of the pool, `ShadowScale` is either 7 or 8 for GASan-128 and GASan-256 respectively, and

$$\text{NextPow2}(x) = 2^{\lceil \log_2 x \rceil}$$

rounds up to the next power of two[5]. Using that, the runtime computes $\texttt{ShadowMask} = \texttt{ShadowBase} - 1$. Finally, it populates `ShadowBase` with the address returned by `cudaMalloc` for the shadow memory buffer. Note that the first address of shadow memory does not necessarily correspond to the first address of the pool. Consequently, the pool's image in shadow memory may wrap around, but it will never overlap itself.

3.3 Redzone Calculation

Aside from the above major changes, GASan slightly modifies how redzones are calculated. ASan originally had a constant 128B redzone one each side of every allocated buffer, but that might be insufficient when the average allocation size is very large. Therefore, GASan opts to take ASan's later approach and scale to size of the redzones with the size of the buffer. Specifically, it exposes two user-controlled parameters: L and B. The former is the proportionality constant between the amount of data housed in an allocation and the size of the redzones on each side of the buffer. The latter is a lower bound on the redzones' size to account for small allocations. For an allocated buffer of size s, GASan creates redzones on each side with size at least $\max\{Ls, B\}$. By default, $L = 50\%$ and $B = 256$B.

4 Limitations

4.1 Unhandled Behavior

The approach outlined here does not address many aspects of memory management on GPUs. For instance, it does not bounds-check accesses to shared memory. However, there is no reason the approach could not work in principle. The shadow memory would have to be allocated in shared memory and be populated by the kernel on launch, but the instrumentation would remain the same. The same approach could be used for static variables in local memory. However, it would not work for the stack since CUDA's API does not expose direct control over the stack pointer. Thankfully, the stack is only used to spill registers to memory or to support recursion, both of which are unlikely in practice.

There is also the issue of the device dynamically allocating memory inside the kernel. The approach proposed here cannot be adapted to support that since `malloc` on the device returns pointers aligned to just 16B instead of 256B. That breaks the encoding discussed in Sec. 3.1, with substantial performance penalties as shown in Sec. 5. It should be noted, however, that using `malloc` in kernel code itself incurs severe penalties, and as such it should never be used in performant code.

Additionally, this approach does not handle CUDA's Unified Virtual Memory (UVM) [14]. On newer GPUs, pages of memory can be logically shared between the host and the device. Physically, the page is resident on either the host-side or the device-side — whichever was last using it. The page is migrated on demand when the other side tries to access it. Crucially, the host can treat data allocated by `cudaMallocManaged` just like any other buffer in CPU memory. This obviously conflicts with Sec. 3.2.2's proposal for expanding memory, in which CPU code treats device pointers as opaque handles.

Finally, the authors are aware of ways to substantially reduce the overhead of ASan with the compiler [20]. In fact, that approach might confer more benefits on the GPU compared to CPU code. Given the authors' method however, they find it more practical to simply instrument every memory access. Thus, they compare their results to the original ASan [17] and leave evaluating these compiler optimizations to future work.

4.2 Correctness

In rare cases, GASan can cause previously correct programs to crash. Recall that the runtime has to translate the handles seen by the application to actual device pointers for the kernel. However, it is possible to bypass the runtime and feed pointers directly to the device. Being more concrete, the code in Fig. 4 will crash with GASan enabled.

5 Evaluation

5.1 Methodology

5.1.1 Performance Overhead

To estimate the runtime overhead, the authors choose to simulate using the trace-based `accel-sim` [8] and the Turing TU106 RTX 2060 traces provided with it. For every load and store to global memory encountered in the traces, the authors prepend a trace fragment with instructions corresponding to the required instrumentation code. The fragments used for GASan-128 and GASan-256 are shown in Fig. 5 and 6 respectively. In this way, the authors measure GASan's performance overhead.

Additionally, the authors simulate other configurations of GASan to gain insight into its performance characteristics. They simulate GASanNoMem-128 and GasanNoMem-256, which are the same as their normal counterparts except for omitting the instruction to load from shadow memory. These variants are used to measure how much of the runtime overhead comes from memory access and how much comes from just executing more instructions. They also simulate GASan-8, which uses the original shadow memory encoding from ASan[6]. It uses the same instrumentation code

4. The value of k can vary depending on the application. This system supports 2^k concurrent allocations, each of size at most 2^{64-k}.

5. Shadow memory must have a power of two size so that instrumentation can just AND with `ShadowMask` to compute `b_off`.

6. However, GASan-8's instrumentation code is *not* equivalent to ASan's. For 64-bit loads, ASan optimizes its instrumentation by only checking that the entire block is accessible. GASan-8 avoids this optimization so as to best avoid the confounding variable of non-memory execution overhead per load.

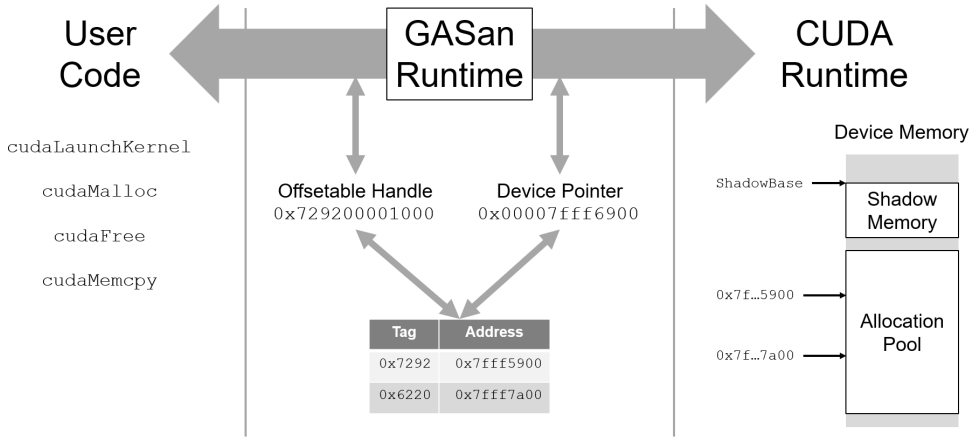

Fig. 3. Block diagram of the memory management scheme proposed by GASan. User code communicates with the CUDA runtime only via the GASan runtime. The GASan runtime is responsible for translating device pointers to user-facing handles and vice versa.

```
int host_x = 0xC57292;

__global__ void kernel(int **ppx) {
    printf("%x\n", **ppx);
}

int main(void) {
    void *devp_x;
    void *devp_px;
    cudaMalloc(&devp_x,  sizeof(int));
    cudaMalloc(&devp_px, sizeof(int *));
    cudaMemcpy(devp_x,  &host_x, sizeof(int),   cudaMemcpyHostToDevice);
    cudaMemcpy(devp_px, &devp_x, sizeof(int *), cudaMemcpyHostToDevice);

    kernel<<<1,1>>>((int **)devp_px);
    cudaDeviceSynchronize();
}
```

Fig. 4. An example of correct CUDA code that will crash with GASan enabled. It should print `c57292`. However, it copies a pointer to device memory into device memory, bypassing the runtime's ability to translate the pointers. This leads to an illegal access error on the second dereference of `ppx`.

as GASan-128, but only scales by three bits instead of seven[7]. This variant is used to observe the effect of the compact encodings proposed in Sec. 3.1.

5.1.2 Memory Footprint

To estimate the memory footprint of GASan, the authors also simulate, this time using a custom simulator. The traces were generated by using `LD_PRELOAD` to log all the calls to `cudaMalloc` and `cudaFree` along with their parameters. The authors verified that those were in fact the only allocation functions used by the benchmarks[8]. Additionally, they reason that the hardware used cannot affect the traces since only function-level information is retained.

The simulator models an unbounded pool of memory from which contiguous chunks can be allocated. The allocator scans the free list left-to-right and simply places the data into the first chunk that can contain it, with buffers on each side for the redzones. If the chunk is much larger than the requested allocation size, only the left part is used and the right part remains in the free list. When a buffer is freed, its chunk is returned to the free list and merged with its immediate neighbors. Additionally, the simulator attempts to merge redzones wherever possible, so that the right redzone of one buffer may overlap with the left redzone of the subsequent one.

After the trace is done, the simulator reports M the maximum amount of memory that had to be requested from the pool. That figure includes heap fragmentation, as well as redzones if they were enabled. Additionally, it is rounded up to the next multiple of the 4KiB page size. From M, the simulator calculates the required size of shadow memory[9] and rounds it up to the next power of two[5].

5.1.3 Assumptions

Importantly, the performance model does not capture the actions of the host nor the considerations from Sec. 3.2. Most importantly, it does not capture calls to `cudaMalloc` and its related functions, which in GASan incur the runtime overhead of updating shadow memory and migrating the pool if needed. However, the authors reason that this overhead is negligible compared to the cost of running the kernel.

7. For this reason, GASanNoMem-8 is not simulated, since the generated code would be identical to GASanNoMem-128, at least from the simulation's perspective.

8. In particular, none of the becnhmarks used the array allocation, nor did they use managed allocation. GASan can handle the former case, but it cannot handle the latter.

9. This is calculated as either $\frac{1}{128}M$ or $\frac{1}{256}M$ for GASan-128 and GASan-256 respectively.

```
      IMAD.SHR     Rb_val, Ra,     7, RZ
      LOP.AND      Rb_val, Rb_val, 0, RShadowMask
      IMAD.IADD    Rb_val, Rb_val, 0, RShadowBase
      LDG.E.U8.SYS Rb_val, [Rb_val]
      LOP.AND      Rs_off, Ra,     0x7f,   RZ
      IMAD.IADD    Rs_off, Rs_off, =(w-1), RZ
      ISETP.GT.AND P0, P0, Rb_val, Rs_off, PT
@!P0 BRA Crash
```

Fig. 5. Instructions inserted before a global memory operation with GASan-128. All the general-purpose registers (except `RZ`) correspond to variables in the code snippet in Sec. 3.1.1. As-is this code snippet clobbers the predicate register `P0`, but the authors reason this will not break program correctness since `ISETP` always immediately preceeds a `BRA` with no intervening memory operation.

```
      IMAD.SHR     Rb_val, Ra,     8, RZ
      LOP.AND      Rb_val, Rb_val, 0, RShadowMask
      IMAD.IADD    Rb_val, Rb_val, 0, RShadowBase
      LDG.E.U8.SYS Rb_val, [Rb_val]
      LOP.AND      Rs_off, Ra,     0xff,   RZ
      IMAD.IADD    Rs_off, Rs_off, =(w-1), RZ
      ISETP.NE.AND P0, PT, Rb_val, RZ,     PT
      ISETP.GE.AND P0, P0, Rb_val, Rs_off, PT
@!P0 BRA Crash
```

Fig. 6. Same as Fig. 5, but for GASan-256. Here, the registers correspond to variables in Sec. 3.1.2.

TABLE 1
Registers Used Compared to CUDA's Limit on Rodinia 3.1 in Increasing Order

Benchmark	Registers	Usage
nn	12	5%
gaussian	16	6%
pathfinder	18	7%
bfs	20	8%
b+tree	24	9%
backprop	27	11%
kmeans	28	11%
hotspot	35	14%
hybridsort	41	16%
nw	52	20%
dwt2d	54	21%
srad_v1	61	24%
lud	64	25%
myocyte	117	46%

No performant program's runtime should be dominated by memory allocation.

Additionally, the trace segments shown in Fig. 5 and 6 assume the existence of registers holding the `ShadowBase` and `ShadowMask` variables, as well as temporary registers to hold `s_off`, `b_off`, and `s_val`. The assembly code requires four extra registers in total, and the authors do not simulate the effects of spilling them to memory. However, none of the benchmarks required this. CUDA kernels can use up to 255 registers [14], and all the Rodinia 3.1 benchmarks fall well within that limit as shown in Table 1.

Turning to memory footprint, the allocator's simulator does not model the overhead for metadata. The authors reason that this overhead will be negligible compared to the other factors being measured, especially since device-side allocations typically store large buffers, and especially since redzones are a much larger per-allocation overhead as shown in Sec. 5.3. Additionally, metadata can be stored inside redzones if the allocation pool is backed with UVM, so it would not substantially increase the memory footprint in that case, though it could decrease performance.

As for the allocator, its model is relatively simple. It does have some sophistication — it tries to merge redzones wherever possible, for instance. However, it does not try to bin allocations to improve performance. It also does not try to detect or mitigate fragmentation, instead simply assigning data to the first sufficiently large chunk. Nonetheless, the authors believe this model suffices for the benchmarks being tested, especially given how few allocations they make. For larger and longer running applications, these considerations would become more important.

5.2 Performance Overhead

The simulated cycle count for both variants of GASan on the Rodinia 3.1 benchmark suite is shown in Fig. 7. They show an average slowdown of approximately 31%. The worst slowdown seen for GASan-128 was 88% and for GASan-256 was 78%, both on `bfs`. Overall, GASan-256 performs better, though usually by only one percentage point.

These results are much better than those of ASan, which reports a 73% slowdown on average and a 160% slowdown in the worst case [17]. Additionally, both variants of GASan (somehow) show a 35% speedup on `kmeans`. (See Sec. 5.2.3.)

5.2.1 Effect of Additional Instructions

In the original paper describing ASan, the authors note that reducing the number of instrumentation instructions had only a minor effect on performance, though that result is confounded by requiring `-fPIE` [17]. GASan, on the other hand, sees substantial performance degredation from just the arithmetic instructions (math, logic, conditionals) used for instrumentation. On Rodinia 3.1, Fig. 8 shows that the overhead of these non-memory instructions is 27% of the overall instrumentation cost on average, and it shows that number can be as high as 51% — greater than the cost of consulting shadow memory. Similarly, Fig. 9 shows that arithmetic instructions account for 50% of the instrumentation overhead on memory microbenchmarks, and can account for 96% of it in the worst case. Additionally, many of the microbenchmarks that show particularly egregious

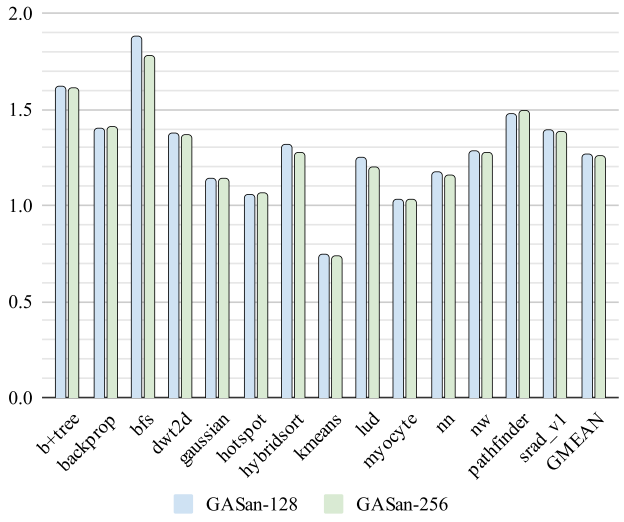

Fig. 7. Normalized total cycle count on Rodinia 3.1 for GASan-128 and GASan-256. The geometric mean over all benchmarks, excluding `kmeans`, is a 31.9% and 30.6% slowdown respectively.

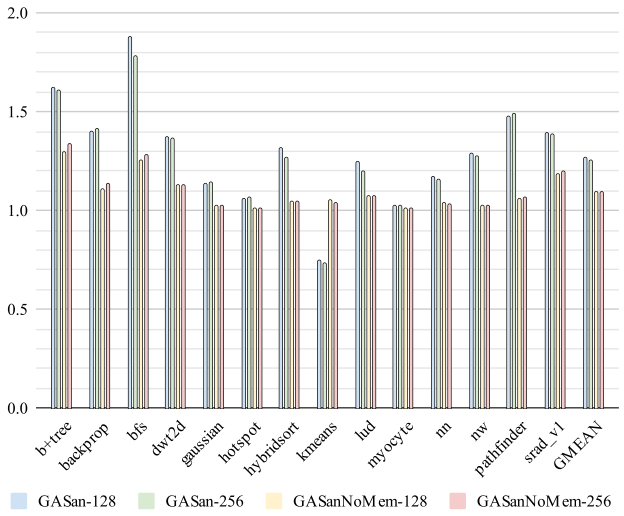

Fig. 8. Normalized total cycle count on Rodinia 3.1 for all the variants of GASan and GASanNoMem.

performance have their performance overhead dominated by arithmetic instructions.

While there is a high degree of nonuniformity in the performance impact of instrumentation over memory microbenchmarks, all of them see between a 2x to 4x increase in dynamic instruction count. This suggests that the difference in overhead is driven by other factors, namely:

- the dynamic instruction frequency of instrumented memory accesses, and
- the average latency of those memory accesses.

For the first point, it makes intuitive sense that more accesses imply more instrumentation code which implies a greater runtime overhead. However, the overhead does not scale linearly with the density of memory instructions. For

11. GASan-128 failed to run on `l1_lat`. The performance overhead given here is an estimate based on its behavior on other benchmarks.

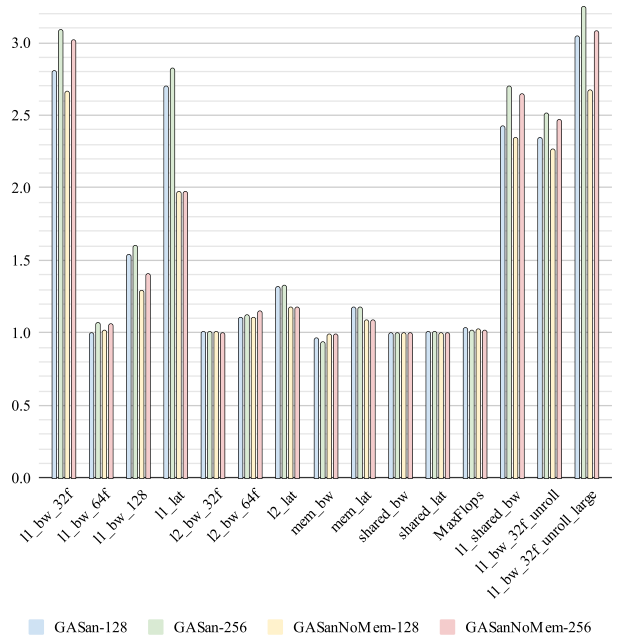

Fig. 9. Normalized total cycle count on memory microbenchmarks for all the variants of GASan and and GASanNoMem.[11]

example, consider the microbenchmarks `l1_bw_32f` and `lw_bw_64f`. The former performs four loads in a loop while the latter only does two, but `32f` sees a 3x slowdown while `64f` sees almost none. It appears the extra instructions can be hidden in the shadow of a load. Recall that GPUs try to hide access latency by switching to a different warp when the current one makes a request. If the density of memory operations is low enough, this would allow the GPU to handle instrumentation when it would otherwise just be waiting. If the density is too high though, it would prevent the GPU from coalescing accesses. The exact crossover point would be determined by the access latency.

As for the second point, notice that most of the microbenchmarks in Fig. 9 that show severe performance degradation also focus on the L1 cache. In that case, the runtime of the instrumentation code would not be amortized by the runtime of the memory access itself. Furthermore, microbenchmarks like `l1_bw_128` and `l2_bw_64` also show cache pollution with lines from shadow memory. That would increase the average latency of normal memory accesses and further contribute to runtime overhead.

5.2.2 *Effect of Compact Encoding*

It is highly variable whether a particular benchmark benefits from the compact encodings discussed in Sec. 3.1. On Rodinia 3.1 shown in Fig. 10, GASan-8's instrumentation code sometimes shows the same or even slightly better performance than those of both usual variants of GASan, while other times it exhibits over a 2x slowdown. The variance is even more pronounced on memory microbenchmarks in Fig. 11. On average for Rodinia though, GASan-8's instrumentation takes 44% longer to run that those of GASan-128 and GASan-256.

One explanation for both the performance and the variability has to do with caching. On the GPU, one 128B cache line encodes bounds information for either 16KiB or 32KiB of contiguous memory, the former in GASan-128 and the

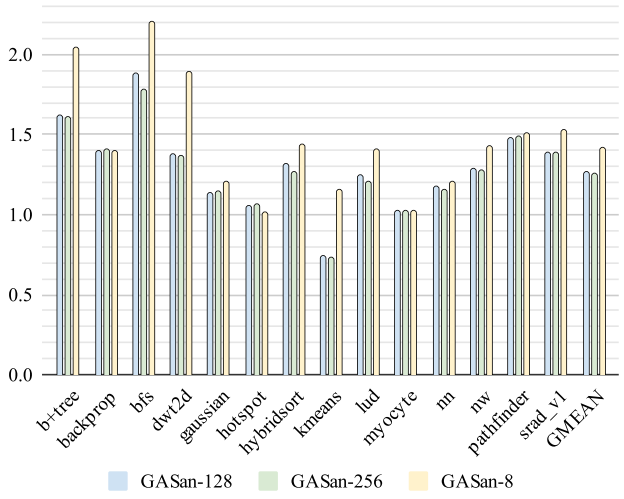

Fig. 10. Normalized total cycle count on Rodinia 3.1 for both variants of GASan and GASan-8.

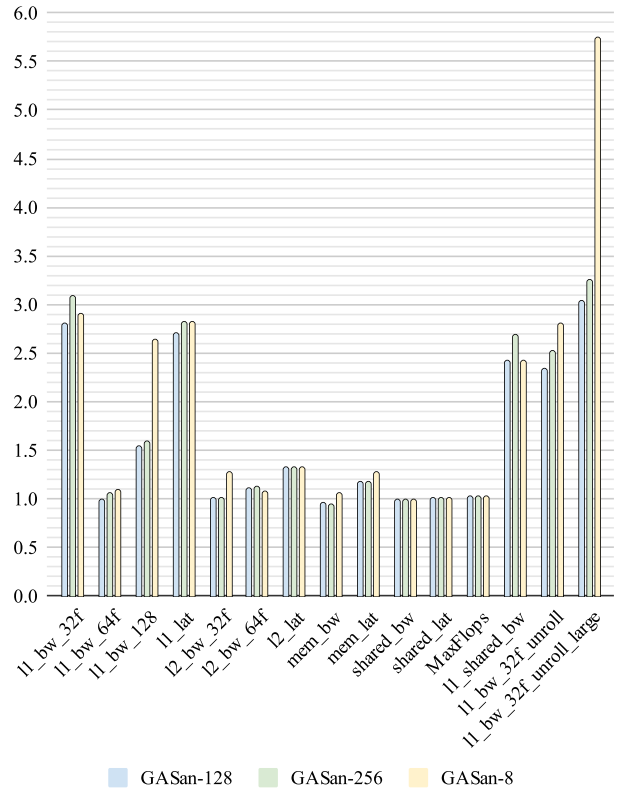

Fig. 11. Normalized total cycle count on memory microbenchmarks for both variants of GASan and GASan-8.[11]

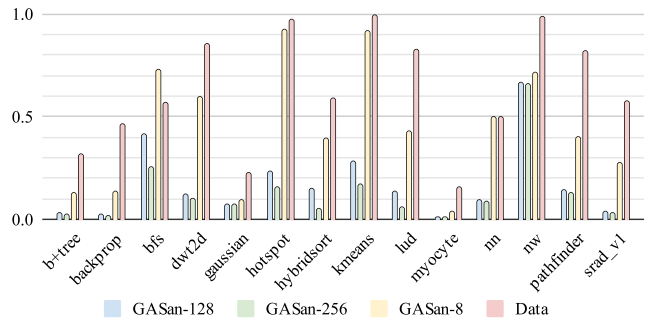

Fig. 12. L1 cache miss rate on Rodinia 3.1 for shadow memory data in both variants of GASan and GASan-8. Also shows the miss rate for data normally accessed by the application.

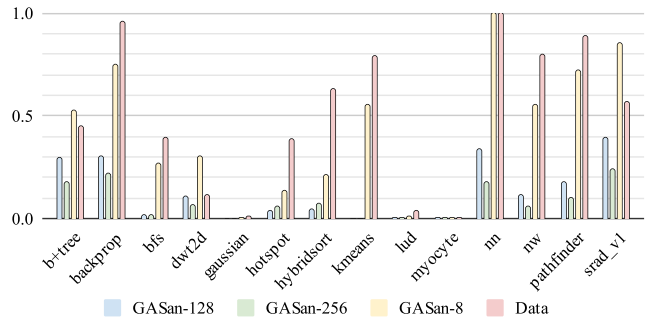

Fig. 13. Same as Fig. 12, but for the L2 cache.

latter in GASan-256. That stands in contrast to ASan on the CPU, where one line encodes for only 512B of memory. This larger chunk size means less cache space is devoted to shadow memory compared to useful data. Additionally, larger chunks are accessed more often, so the corresponding shadow lines have an easier time remaining in the cache, being brought to the most recently used position more often. Of course, the magnitude of these benefits vary depending on the workload, with memory-intensive ones profiting the most.

This theory is corroborated by Fig. 12 and 13. In almost all cases, the L1 cache miss rate of shadow data for both GASan-128 and GASan-256 is overshadowed by that of the data normally accessed by the application. For any cases where it is not, namely `bfs` and `nw`, that statement holds for the L2 cache miss rate. The same cannot be said for the shadow data for GASan-8, which displays a miss rate on par with and sometimes worse than application data. Overall, the compact encodings from Sec. 3.1 improve performance by making shadow memory more cache friendly.

5.2.3 *Super-Unity Performance*

Strangely, some benchmarks appear to run faster with GASan enabled. The most prominent example is `kmeans` from Rodinia 3.1, but `mem_bw` from the memory microbenchmark suite also exhibits this behavior, albeit on a much smaller scale. As shown in Fig. 8 and 9, this speedup vanishes when the loads consulting shadow memory are removed. However, the speedup remains when all the "arithmetic" instructions are removed to leave only the loads from shadow memory. In that case, the authors see a 2% speedup on `kmeans` and a 3% speedup on `mem_bw`.

Admittedly, it is possible that these results stem from a bug in AccelSim. The statistics given below are hard to make sense of given the nature of the instrumentation. For the sake of argument however, the authors assume the results are correct. Under that assumption, the culprit likely lies in the memory hierarchy. That makes sense given the contents of the offending benchmarks. The relationship is clear in the case of `mem_bw`. For `kmeans`, almost all of the performance gains are realized on the kernel shown in Fig. 14. It is memory intensive, with a tight loop consisting entirely of a load, a quick addition, and a store.

For `kmeans`, the component contributing to this speedup is likely the L2 cache. The original code sees a miss rate of 82%, while both variants of GASan see miss rates of 62%. The extra hits are not for shadow memory either, since GASan sees 23% fewer cache misses in total. The same characteristics are present when the arithmetic instructions are removed, though to a much lesser extent. In that case, GASan sees a miss rate of 78% and negligibly fewer total misses. Another interesting statistic for `kmeans` is the cache's port utilization. Baseline has a 15% fill port usage[12], but a data port utilization of only 4%. The data port usage slightly increases to 5% when shadow memory loads are present, but using GASan proper yields a utilization of 12%. This observation suggests that the original benchmark pulls a lot of data into the cache that it cannot take advantage of. In fact, that corroborates the single-use strided access pattern shown in Fig. 14[13].

For `mem_bw`, the mechanism appears to lie in main memory. On average, baseline sees 4.0 row buffer accesses per activation. That number increases to 6.0 for GASan-128 and to 9.8 for GASan-256. Even when the arithmetic instructions are removed, the effect is still strong — 5.5 and 6.0 for the modified GASan-128 and modified GASan-256 respectively. Additionally, enabling GASan seems to decrease both the average and the maximum latencies observed for memory accesses by a factor of 2.

Unfortunately, it is unclear how GASan is causing these benchmarks to become cache friendly. The authors make a few observations though. Note that the rows of the `input` matrix in `kmeans` are short, which means over half of a row's data is in the same cache line as the row just above or below it. Also note that the access pattern for `mem_bw` has consecutive threads accessing consecutive memory locations, which means they share DRAM rows. Furthermore, trimming the trace for `kmeans` to use fewer threads causes the speedup to vanish. For those reasons, the authors suspect shared data might be involved, though it is not obvious how. If it were in the straightforward way, one would expect a lot of hits on reserved elements[14] in the original benchmark, with that number dropping with GASan active. However, the opposite behavior is observed.

5.3 Memory Footprint

Table 2 shows the simulated memory footprint of GASan on the Rodinia 3.1 benchmark suite. The results are given as a percentage over the application's footprint without GASan. As for the different configurations, recall from Sec. 3.3 that the size of the redzones is configurable. It is computed as $\max\{Ls, B\}$ for an allocation of size s.

Because of the compact encodings introduced in Sec. 3.1, shadow memory contributes very little to the programs' memory footprint. It is true that its size has to be a power of two, meaning half of it can go unused in the worst case. But even with that considered (as modeled by the simulator), the overhead of shadow memory was never observed to be more than 2%.

Instead, the vast majority of the extra footprint comes from redzones. For a particular value of L, note that the redzones' relative footprint is guaranteed to be at least L and can be as high as $2L$ — there is one redzone on each side of an allocation. Depending on how well the allocation pattern lends itself toward merging rezones, different workloads will fall in different places on that spectrum. On average for the benchmarks, the relative additional footprint is $1.35L$, but values as high as $1.92L$ were observed. Importantly though, this memory footprint can be controlled by the user, with lower values for L consuming less memory but offering proportionally less protection.

Finally, special attention should be drawn to the outlier: `myocyte`. It sees relative memory footprints far lower than the average because of how little memory it allocates. The application uses only 812B in total, meaning it fits entirely inside one 4KiB page, even with the overheads of redzones and alignment. Since the minimum allocation size is one page, `myocyte` effectively incurs no overhead outside of shadow memory. However, this situation is highly atypical. Most allocations on the GPU are for large buffers which can't take advantage of the phenomenon discussed here.

6 Related Work

6.1 CUDA Memcheck

To the authors' knowledge, the only production-ready tool addressing memory safety on GPUs is `cuda-memcheck` [13], which appears to be based on Valgrind's `memcheck` tool. It catches both out-of-bounds and unaligned accesses executed on the accelerator itself, unlike other tools[15] which only check the host's interaction with the GPU. The overhead of `cuda-memcheck` varies heavily depending on the workload. It can be as high as 200x or as low as 1.2x, but the slowdown is 70x on average [1], [11]. In either case, the overhead makes the tool only useful for debugging, and even then just for small programs.

6.2 Canary-Based Approaches

Other software solutions exist, primarily based on canaries. For instance, clARMOR [4] uses `LD_PRELOAD` to wrap OpenCL API calls that allocate memory. It inserts a randomly generated value at the start and end of each allocated buffer, and it checks that all the canaries still have the correct value when each kernel finishes. The overhead of clARMOR increases with the number of allocations, but is 14% on average. GMOD [3] uses a similar design, but instead of performing canary checks at the end of each kernel run, it runs a guard kernel on the GPU throughout the lifetime of the application. That kernel is made aware of allocations in other threads, and it continuously iterates over the known allocations while checking the canaries. GMOD's overhead is just 3% on average and 10% in the worst case.

12. That number goes up to 17% with GASan, though curiously only when the arithmetic instructions are present. Without the arithmetic instructions, the fill port utilization remains 15%.

13. No software prefetch instructions were emitted in this case. There is no hardware prefetching for the data cache either — only for the texture cache.

14. That's `HIT_RESERVED` in AccelSim.

15. For example: `cudagrind` [1]

```
int point_id = blockDim.x*blockIdx.x + threadIdx.x;
if (point_id < npoints)
    for (int i = 0; i < nfeatures; i++)
        output[npoints*i + point_id] = input[nfeatures*point_id + i];
```

Fig. 14. The first kernel executed by the `kmeans` benchmark: `invert_mapping`. The `input` is an `npoints` × `nfeatures` matrix, and similarly the `output` is `nfeatures` × `npoints`. The benchmark was run with `npoints` = 819200 and `nfeatures` = 34. This code is clearly performing a matrix transpose, with each thread responsible for one row of the input.

TABLE 2
Memory Footprint Over Baseline on Rodinia 3.1 for Different Redzone Sizes on Both GASan Variants[a]

Benchmark	GASan-128[b]			GASan-256[b]		
	$L = 10\%$	$L = 50\%$	$L = 100\%$	$L = 10\%$	$L = 50\%$	$L = 100\%$
b+tree	19.2%	92.1%	181.7%	18.6%	90.8%	180.4%
backprop	16.4%	76.4%	152.7%	15.7%	75.7%	151.3%
bfs	18.3%	87.3%	171.9%	17.6%	86.0%	170.6%
dwt2d	11.8%	53.0%	106.1%	11.1%	52.3%	104.7%
gaussian	16.3%	76.0%	151.9%	15.5%	75.2%	150.4%
hotspot	11.1%	52.1%	102.1%	10.5%	51.0%	101.0%
kmeans	15.9%	76.5%	153.0%	15.4%	75.6%	151.1%
lud	11.7%	51.6%	101.6%	10.9%	50.8%	100.8%
myocyte	0.8%	0.8%	0.8%	0.4%	0.4%	0.4%
nn	15.1%	67.5%	135.7%	14.3%	66.7%	134.1%
nw	11.6%	51.6%	103.1%	10.8%	50.8%	101.5%
srad_v1	12.1%	58.0%	114.2%	11.7%	57.1%	113.3%
Average[c]	14.4%	66.8%	132.3%	13.8%	65.9%	130.9%

[a] Statically linked benchmarks could not be measured with `LD_PRELOAD`
[b] $B = 256$B
[c] Excluding `myocyte`; see Sec. 5.3

6.3 GPUShield

In terms of hardware solutions, there is GPUShield [11]. It proposes changes across the entire tech-stack to perform bounds checking on host-allocated buffers, heap variables, and local variables. First, it adds compiler passes to deduce which memory accesses could be out-of-bounds at runtime. From that information, it produces a Bounds Analysis Table (BAT). Then, it modifies the GPU's kernel driver to populate a hardware Runtime Bounds Table (RBT) from the BAT. Finally, during execution it has a hardware Bounds Check Unit (BCU) consult the RBT and raise an exception if necessary. By adding this complexity, it achieves an average overhead of just 0.8%.

6.4 AMDGPU ASan

Somewhat similar to this work, AMD has recently started implementing an ASan runtime library for their GPUs [19]. They note that modern accelerators share a virtual address space with the CPU, which means they can share via page migration a single shadow memory space residing on the host [18]. This sidesteps the main drawback of this paper's proposal — that it has to do a lot of work to compensate for the lack of virtual memory facilities. It also naturally supports the CPU and GPU sharing memory via UVM. Unfortunately, this method requires the runtime to support Heterogeneous Memory Management (HMM) [6], which NVIDIA does not appear to have plans to implement [7]. Moreover, this approach does not support the compression schemes discussed in 3.1 since here the CPU and GPU share the same shadow memory.

7 Conclusion

Many of the solutions in Sec. 6 either have a prohibitively high runtime cost, require changes to (often proprietary) software and hardware, or are canary-based and thus probabilistic. In contrast, this paper provides a more principled approach to memory safety on GPUs. In simulation, it incurs a memory overhead of at most 2x for $L = 50\%$, as well as an average runtime overhead of just 31%. While those figures are not feasible for an always-on solution, they are viable for large-scale debug builds. More importantly for practical adoption, the changes it proposes can be implemented on top of the CUDA API, so it requires no changes to the lower parts of the tech stack. Ultimately, GASan would be a viable approach for furthering memory safety on GPUs.

Nonetheless, it is worth noting the system-level changes that would benefit GASan. The main point of improvement would be memory management. An actual implementation of GASan would spend a lot of effort working around the virtual addresses returned by the CUDA API since it has no direct control over them. Here, AMD's approach would be most helpful. HMM already allows modern GPUs to share via page migration a virtual address space with the CPU. Implementing HMM in CUDA would not only enable GASan but also make GPU programming easier. Barring that, it might be possible to extend UVM to support arbitrary sections of the host's virtual address space, much like `cudaHostRegister` does for UVA. This way, shadow memory can again be allocated on the host and transparently migrated to the GPU on demand, thus sidestepping many of this paper's limitations.

Availability

The code used to analyze, generate, and simulate the traces is available on GitHub at `https://github.com/ammrat13/accel-sim-framework/tree/gasan-sim`.

References

[1] T. M. Baumann and J. Gracia, "Cudagrind: Memory-usage checking for cuda," in *Tools for High Performance Computing 2013*, A. Knüpfer, J. Gracia, W. E. Nagel, and M. M. Resch, Eds. Cham: Springer International Publishing, 2014, pp. 67–78.

[2] B. Di, J. Sun, and H. Chen, "A study of overflow vulnerabilities on GPUs," in *Network and Parallel Computing*, G. R. Gao, D. Qian, X. Gao, B. Chapman, and W. Chen, Eds. Cham: Springer International Publishing, 2016, pp. 103–115.

[3] B. Di, J. Sun, D. Li, H. Chen, and Z. Quan, "GMOD: A dynamic GPU memory overflow detector," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243176.3243194

[4] C. Erb, M. Collins, and J. L. Greathouse, "Dynamic buffer overflow detection for GPGPUs," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. IEEE Press, 2017, p. 61–73.

[5] "Using the GNU Compiler Collection (GCC)," Free Software Foundation. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

[6] J. Hubbard and J. Glisse, RedHat, May 2017. [Online]. Available: https://www.redhat.com/files/summit/session-assets/2017/S104078-hubbard.pdf

[7] hwinkler and stonecat, NVIDIA Developer Forums, Jun. 2020. [Online]. Available: https://forums.developer.nvidia.com/t/hmm-support-in-linux-driver/50996

[8] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated GPU modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[9] Y. Kim, J. Lee, and H. Kim, "Hardware-based always-on heap memory safety," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 1153–1166.

[10] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, and A. DeHon, "Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 721–732. [Online]. Available: https://doi.org/10.1145/2508859.2516713

[11] J. Lee, Y. Kim, J. Cao, E. Kim, J. Lee, and H. Kim, "Securing GPU via region-based bounds checking," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 27–41.

[12] A. Miele, "Buffer overflow vulnerabilities in cuda: a preliminary analysis," *Journal of Computer Virology and Hacking Techniques*, vol. 12, no. 2, pp. 113–120, 05 2016.

[13] NVidia, *`cuda-memcheck`: User Manual*, Aug. 2022. [Online]. Available: https://docs.nvidia.com/cuda/archive/11.7.1/pdf/CUDA_Memcheck.pdf

[14] NVIDIA, *CUDA C++ Programming Guide*, Feb. 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[15] S.-O. Park, O. Kwon, Y. Kim, S. K. Cha, and H. Yoon, "Mind control attack: Undermining deep learning with GPU memory exploitation," *Computers & Security*, vol. 102, p. 102115, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404820303886

[16] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer implementation," LLVM. [Online]. Available: https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Instrumentation/AddressSanitizer.cpp

[17] ——, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[18] B. Sumner, Personal Communication, AMD, Apr. 2023.

[19] B. Sumner, P. Velliengiri, and K. Zhuravlyov. (2023, Feb.) ROCm device libraries. AMD. [Online]. Available: https://github.com/RadeonOpenCompute/ROCm-Device-Libs

[20] Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu, "Debloating address sanitizer," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 4345–4363. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen