CS 7292 A: Project Proposal

Ammar Ratnani

Abstract—I propose evaluating the performance of a potential port of Address Sanitizer to Graphics Processing Units.

1 Problem

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 [1], [9], [12], which could leak the sensitive data they now find themselves handling.

Memory safety on CPUs is a well-understood problem with many proposed solutions [13]. Clearly, it would be worth investigating whether these same solutions can work on GPUs, and what changes are required if they don't function out-of-the-box.

2 Related Work

In general, it appears that defenses for memory safety on GPUs specifically has not received much attention in the literature. To my knowledge, the only production-ready tool addressing memory safety on GPUs is `cuda-memcheck` [10]. It appears to be based off Valgrind's `memcheck` tool, and it has a similar runtime overhead — on the order of 100x [7].

Aside from that, there is GPUShield [8]. It proposes changes across the entire tech-stack to perform bounds checking on host-allocated buffers, heap variables, and local variables. Specifically, it requires:

- compiler passes to deduce which memory accesses need bounds checking and thus generate the Bounds Analysis Table,
- modifying the GPU's kernel driver to populate the Runtime Bounds Table, and
- adding a Bounds Check Unit to the hardware.

It also only checks that heap accesses fall within the heap, not that they fall onto the right variable.

Additionally, there are canary-based approaches. For instance, clARMOR [3] wraps OpenCL API calls that allocate memory, inserting a randomly generated value at the start and end of each buffer. At the end of each kernel, they check that all the canaries still have the correct value. Importantly it doesn't require modifying the binary, instead relying on `LD_PRELOAD`. There is also GMOD [2], which delegates checking the canaries to a dedicated GPU thread.

These existing solutions either probabilistic, have high overhead, or require extensive infrastructure changes. I seek a solution that has significantly less overhead than Valgrind, and which could be implemented entirely in software without any changes to proprietary code.

3 Proposed Solution

I propose evaluating the possibility of converting Google's Address Sanitizer (ASan) [15] for use on GPUs, which I'll call GASan. It can be implemented entirely within a compiler like LLVM, requiring no changes to the hardware or proprietary runtime [14]. It is expected to have a performance overhead of at most 2x, which is much lower than that of `cuda-memcheck`. Even if that makes it impractical for release builds, it would still have value in debugging, making it easier to find and squash memory safety bugs.

It should be noted that there are some important architectural differences between GPUs and CPUs that will affect GASan's implementation and effectiveness. Most importantly, memory allocated through the CUDA runtime is always aligned to 256 bytes [11, Section 5.3.2], which is much greater than the 8-byte alignment assumed by ASan. Using the same approach as the original paper, this would give GASan a much lower memory overhead — $\frac{1}{256}$-th of main memory to be exact. This also means shadow memory would take less space in the cache and would better be able to use coalescing [4].

Another important concern for an actual implementation of GASan is that shadow memory has to be located on the device instead of on the host. This means updates to shadow memory may be much more expensive than with ASan, though one could argue they shouldn't happen often anyway. It might be possible to put shadow memory in Unified Virtual Memory [5] to improve performance.

4 Evaluation Methodology

I plan to use `macsim` [6] to evaluate how much performance would degrade in the presence of GASan. Specifically, I plan to modify existing traces by inserting instructions before every memory operation to simulate checking shadow memory. With this change, I am expecting a slowdown of 2x or lower. If it's much more than that, GASan would not be viable to actually build.

Importantly, I do not plan to model the actions of the host when populating shadow memory. In other words, my simulation assumes the overhead of functions like `cudaMalloc` is negligible compared to the time spent in the GPU kernel. This should be a sensible assumption to make — one has bigger problems if they are spending most of their time allocating and freeing memory.

Less sensibly, I don't plan to model overhead in functions like `cudaMemcpy`, reasoning that the time spent in the copy operation will dominate the time spent checking shadow memory, especially if the copy is large.

5 Plan for Milestones

I will spend the first milestone setting up `macsim` — getting it to compile and run a trace. I will also read through the

documentation and the code to understand traces' formats, as well as what exactly `macsim` models. For the second milestone, I will write code to inject the appropriate instructions before every load and store. I will use my code to modify existing traces, and I will observe the performance overhead in simulation. Finally, for the third milestone, I will do experiments and data collection. For example, I will observe what happens if only writes are instrumented, or if shadow memory is made less dense.

Given the short time frame, I am trying to not make my goals so ambitious. I can always do more work if I have the time, but I don't want to over-promise.

REFERENCES

[1] 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.

[2] 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

[3] 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.

[4] M. Harris, "How to access global memory efficiently in CUDA C/C++ kernels," NVIDIA, 01 2013. [Online]. Available: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

[5] ——, "Unified memory for CUDA beginners," NVIDIA, 06 2017. [Online]. Available: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

[6] H. Kim, "`macsim`," Georgia Tech HPArch. [Online]. Available: https://github.com/gthparch/macsim

[7] J. Lee, Personal communciation, student in this class and former research advisor.

[8] 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.

[9] 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.

[10] NVidia. (2010, 08) `cuda-memcheck`: User manual. [Online]. Available: https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/cuda-memcheck.pdf

[11] NVIDIA, *CUDA C++ Programming Guide*, 02 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[12] 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

[13] M. Qureshi, Personal communication.

[14] 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

[15] ——, "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