Gklee$_{pp}$: Parallelizing a Symbolic GPU Race Checker

Mark S. Baranowski
Research Assistant
School of Computing
University of Utah

Ganesh Gopalakrishnan
Professor
School of Computing
University of Utah

***Abstract*—The increased usage of GPGPU programs necessitates tools to detect errors in the program. Existing tools which use hardware instrumentation may miss certain bugs due to limitations in the memory they observe and the requirement that tests must produce the but. Formal methods found in tools such as Gklee$_p$ can find data races without users providing test cases. The primary disadvantage of using such tools is the slowness of execution when compared to the speed at which conventional tools run. We present a parallelized version of Gklee$_p$, christened Gklee$_{pp}$, which better uses computation resources through parallel execution. We can attain speedups between 2 and 8 times the sequential version.**

I. Introduction

The use of General Purpose GPU programs yields tremendous throughput for certain workloads when compared to implementations using conventional CPUs. The increased complexity of GPU kernels allows the introduction of subtle data-race bugs. Additional difficulty in writing GPU kernels is due to differences in scheduling between different chipsets, meaning a program may only exhibit errors when run on a chip different from the chip it was developed on.

We focus on checking data races in GPU kernels written in Nvidia's CUDA C. Data race checking is needed because a race may yield an apparently correct result while races silently corrupt the intermediate calculations. Race checking tools can be used to help users locate races in their programs.

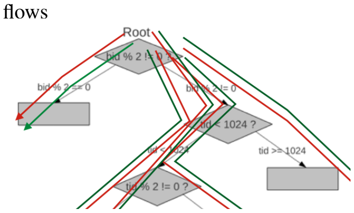
There are two types of tools available to find races: Conventional, hardware based tools such as CUDA-MEMCHECK[1] and GMRace[2] are able to find races in GPU kernels while imposing minimal execution penalty. However, these tools are not wholly sufficient as they are limited to checking shared memory and tests must be generated to exercise the control flows for GPU threads to produce a race. The other type of tool formally checks the GPU kernel to identify races. Examples of formal tool are Gklee[3] and GPUVerify[4]. Gklee executes the GPU program in a virtual machine, recording array accesses and at synchronization points a check is performed using a logic solver to determine if two threads could have raced. Gklee is an extension of the Klee[5] tool with support added for symbolic execution of CUDA programs. This tool can identify races without assistance from the user for program input selection and additionally produces test cases, which can be run to exhibit the error in hardware. The primary drawback to using such tools is they typically take an order of magnitude longer to run than the time for the program to run on hardware.

Gklee models each GPU thread as a symbolic thread; this is a full symbolic execution. The problem with this approach is the verifier cannot scale to the thousands of threads, which is typical of real GPU programs [1]. An enhancement to Gklee, named Gklee$_p$ (Gklee parametric), was developed[6] to overcome this issue where in threads are represented by "parametric flows." Using parametric flows allows the tool to take advantage of the symmetric execution in GPU programs by leaving the thread/block index symbolic and updating memory accesses accordingly. When reaching a conditional when executing a flow if the solver determines from the execution history that the condition can feasibly evaluate to true and false, causing the parametric flow to be split into two flows one executing the 'true' branch, the other executes the 'false' branch. This enhancement allows Gklee$_p$ to scale with the number of threads. An additional tool called SESA[7] (Symbolic Execution Static Analysis) was produced which processes Gklee$_p$'s input executable annotating variables that can be concretized without affecting race checking.

II. Parallelization of Gklee$_p$

While Gklee$_p$'s performance is better than Gklee's [2], it is still too slow to use on moderately sized programs. We investigated several approaches to parallelizing the execution of Gklee$_p$ to take advantage of multicore processors. This new tool is christened Gklee$_{pp}$.

Fig. 1. Representation of parametric flows



Our first attempt added threading to Gklee$_{pp}$ where pairs of memory accesses could be checked in different threads. The data structures used by Gklee perform operations to optimize access by simplifying expressions. This caused numerous data races within the tool, corrupting the data. We attempted

[1] Gklee$_p$ runs 46 times faster when given concrete instead of symbolic input
[2] Up to 1000x in matrix transposition

to lock the data structures, but this caused the tool to deadlock. Next we attempted to implement 'deep' copy operations in order to give each thread data that it could freely modify. We encountered issues implementing these copy operations due to the complex, interconnected nature of the involved structures. We also encountered an issue where the solver used is not amenable to concurrent access.

Another opportunity for parallelization is in parametric execution; here we could execute the parametric flows in parallel. This is the approach we chose in Gklee_{pp}. We implemented parallel flow execution using forked processes (on Linux this is implemented using a copy-on-write clone). Each process is then free to modify memory without corrupting the data used by other processes. Processes are forked (to a defined limit) when encountering a branch where both 'true' and 'false' paths are feasible.

III. Results

We compared the performance of Gklee_{pp} and Gklee_{p} on two GPU kernels: a bitonic sort kernel and the Julia Fractal kernel. We also compared the run time against the SESA annotated executables. All tests were conducted on a dual Intel Xeon CPU E5645 2.40GHz with 48 GB of RAM.

Fig. 2. Results

(a) Bitonic kernel time

Tool	Time(s)
Gkleepp	1166
Gkleep	2649
Gkleepp-sesa	1183
Gkleep-sesa	2659

(b) Julia kernel time

Tool	Time(s)
Gkleepp	51
Gkleep	210
Gkleepp-SESA	50
Gkleep-SESA	217

We witnessed a speed up of approximately 2 when running the bitonic kernel. We observed two active processes during most of the execution. This is due to some forked flows ending execution soon after forking. The Julia kernel operates on distinct memory locations, and branches in the kernel occur at the same line. We witnessed 8 active processes during execution and attained as speed up of about 8. Because of the way processes are forked, we do not currently have a method to communicate changes made by one forked flow back to a controlling process. Despite race checking having only information available within a forked flow, Gklee_{pp} still reported races in the bitonic kernel that were detected by Gklee_{p}.

IV. Conclusion

Gklee_{pp} provides a considerable speed up compared to Gklee_{p}, however the inability to check for inter-flow races limits its race checking abilities. Gklee_{pp} provides a considerable speed up compared to Gklee_{p} In the future we plan to add a facility to encode the updates to data stores to be communicated back to a controlling process so complete race checking can be performed. Adjustments will also need to be done to better utilize resources when using forked processes. We also have plans for a solver server where encoded queries can be dispatched to worker solvers and the Gklee process can issue requests asynchronously. We anticipate this will increase the performance of race checking and be scalable beyond a single machine.

Acknowledgment

Thanks to Professor Gopalakrishnan for supporting this research. Also, Peng Li, the author of Gklee_{p}, for his assistance in learning the Gklee_{p} code base and ideas for parallelizing the tool.

References

[1] NVIDIA. (2014) Cuda-memcheck :: Cuda toolkit documentation. [Online]. Available: http://docs.nvidia.com/cuda/cuda-memcheck/index.html#racecheck-tool

[2] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal, "Gmrace: Detecting data races in gpu programs via a low-overhead scheme," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 1, pp. 104–115, 2014.

[3] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "Gklee: Concolic verification and test generation for gpus," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 215–224.

[4] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "Gpuverify: a verifier for gpu kernels," in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 113–132.

[5] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[6] P. Li, G. Li, and G. Gopalakrishnan, "Parametric flows: automated behavior equivalencing for symbolic analysis of races in cuda programs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 29.

[7] ——, "Practical symbolic checking of gpu programs."