Towards Concurrency Repair in GPU Kernels with Resource Cost Analysis

Gregory Blike Tiago Cogumbreiro

GPU programming has gained immense popularity for filling needs in applications for high performance computing and artificial intelligence. The incredible parallelism available in GPUs comes with penalties of being notably difficult to program and analyze. Several program analysis techniques can aid programmers in ensuring that their code is free from certain classes of errors [2, 3, 5, 6]. Such approaches can pinpoint potential bugs, and even explain the conditions that trigger the error, but offer no actionable feedback.

Concurrency repair extends program analysis with code fixes to mitigate bugs. AutoSync [1], AuCS [7], and GPURepair [4] remove errors such as data-races (leads to unexpected behavior) and barrier divergences (leads to deadlocks) by placing synchronization barriers. Correct placement of these barriers ensures error-free execution, but also impact performance, the primary benefit of GPU programming. Repair uses safety-detection as an oracle to decide when a program is bug-free. In case the program is potentially buggy, the repair algorithm selectively places barriers in the GPU program. Repair algorithms account for the performance impact of adding barriers, by attaching weights to each placement location, minimizing the overall weight of all barrier placements. The limitations of these approaches is that the weights are selected empirically and do not take into consideration important parameters, such as the number of loop iterations. Additionally, since the state of the art lacks correctness results, such techniques cannot guarantee performance bounds on the repaired program.

Our approach. In order to develop a repair algorithm that relies on a sound metric, our goal is to introduce a quantitative program logic for GPU programs that allows the repair algorithm to reason about the program the resource usage (here number of synchronizations) based on [6]. Our resource cost analysis is automated and derives worst-case bounds that are polynomials on inputs of a GPU program, which the repair algorithm can use to minimize the total synchronization cost. By relying on a theory that *soundly* establishes bounds on a resource metric, our goal is to develop a repair algorithm which provably fixes a bug while soundly minimizing performance overhead.

In this talk, we present a calculus to measure the impact of synchronization repair in GPU programs. Our approach is to apply an automated amortized resource analysis (AARA) [6] to an abstraction of GPU programs based on behavioral type theory, Memory Access Protocols [3]. We show how our approach can find optimal placement of synchronization barriers.

References

[1] Sourav Anand and Nadia Polikarpova. Automatic synchronization for GPU kernels. In *Proceedings of FMCAD*, pages 1–9, Piscataway, NJ, USA, 2018. IEEE. doi: 10.23919/FMCAD.2018.8602999.

[2] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify: a verifier for GPU kernels. In *Proceedings of OOPSLA*, pages 113–132, New York, NY, USA, 2012. ACM. doi: 10.1145/2384616.2384625.

[3] Tiago Cogumbreiro, Julien Lange, Dennis Liew, and Hannah Zicarelli. Memory access protocols: certified data-race freedom for GPU kernels. *Formal Methods in System Design*, pages 1–38, 2023.

[4] Saurabh Joshi and Gautam Muduganti. GPURepair: Automated repair of GPU kernels. In *Proceedings of VMCAI*, pages 401–414, Cham, 2021. Springer. ISBN 978-3-030-67067-2.

[5] Guodong Li and Ganesh Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of FSE*, pages 187–196, New York, NY, USA, 2010. ACM. doi: 10.1145/1882291.1882320.

[6] Stefan K. Muller and Jan Hoffmann. Modeling and analyzing evaluation cost of CUDA kernels. *Proceedings of the ACM on Programming Languages*, 5(POPL), 2021. doi: 10.1145/3434306.

[7] Mingyuan Wu, Lingming Zhang, Cong Liu, Shin Hwei Tan, and Yuqun Zhang. Automating CUDA synchronization via program transformation. In *Proceedings of ASE*, pages 748–759, Piscataway, NJ, USA, 2019. IEEE.