International Journal of Enhanced Research in Science, Technology & Engineering
ISSN: 2319-7463, Vol. 15 Issue 3, March-2026

Tools and Techniques for CUDA GPGPU Program Analysis

Lakshmi M. Gadhikar[1], Y. S. Rao[2]

[1]Department of Computer Engineering, SPIT, Mumbai, India
[2]Department of Electronics and Telecommunication Engineering, SPIT, Mumbai, India

ABSTRACT

Optimization of CUDA programs for accelerated performance is an intricate task, requiring advanced tools for profiling and analyzing. This paper presents a detailed survey of various tools and techniques used for analysis of CUDA GPGPU programs. This work presents study of different tools for static and dynamic analysis of CUDA programs. We discuss objectives, features and strengths of each tool, that is useful for CUDA programmers and researchers community in selecting an appropriate tool for their specific needs related to analysis of CUDA programs.

Keywords - ***CUDA GPGPU Program Analysis, Static analysis, dynamic analysis, analysis tools, analysis techniques, Profiling, Verification.***

1 INTRODUCTION

The rapid growth in data-intensive and compute-heavy applications such as scientific computations, simulations, machine learning, and real-time rendering has led to an increased demand for high-performance computing solutions. General-Purpose computing on Graphics Processing Units (GPGPU) has emerged as a powerful paradigm to meet these demands by exploiting the massive parallelism offered by modern GPUs. Among the available GPGPU platforms, NVIDIA's CUDA (Compute Unified Device Architecture) has become the most widely adopted due to its maturity, extensive developer support, and integration with NVIDIA hardware. While CUDA enables developers to offload computation-intensive tasks to GPUs and achieve significant speedups, programming for GPUs remains inherently complex. Challenges such as memory hierarchy management, thread synchronization, kernel optimization, and performance bottlenecks make it difficult to write correct and efficient CUDA programs.

A CUDA code may suffer from different performance issues such as race conditions, shared memory bank conflicts, divergent branches. Thus, developers and researchers need tools to analyze the programs to identify performance bottlenecks and optimize their code. Consequently, the need for robust tools that support debugging, profiling, and program analysis has become increasingly important. Tools for CUDA GPGPU program analysis may be categorized as static and dynamic approaches. This work presents details of these static and dynamic approaches used for analysis of CUDA kernels including the process followed in analysis and tools/techniques used for static and dynamic analysis of CUDA programs. A wide range of tools has been developed to assist in CUDA application analysis, including official tools provided by NVIDIA such as Nsight Compute, Nsight Systems, Compute Sanitizer, Nsight Eclipse Edition, Nsight Visual studio Edition, Visual Profiler, and CUDA- GDB as well as third-party tools like Valgrind with Cudagrind, TAU, CUPTI etc.; Static analysis tools such as GPUVerify, GPUDrano, GPUCheck, GPURepair, COoPER; Dynamic analysis tools such as GKLEE, CURD, Simulee, ScoRD, iGUARD, ValueExpert, cuCatch. and hybrid i.e. static cum dynamic approaches such as GPUDrano and others. These tools serve different purposes; some focus on runtime performance profiling, others on memory error detection, race detection, and some on debugging, verification, or optimization. This work covers study of static, dynamic and hybrid approaches for CUDA GPGPU program analysis.

This paper presents a study of these tools used in the analysis of CUDA based GPGPU applications. We aim to provide valuable insights into the selection and effective use of these tools. The goal is to help developers, researchers, and performance engineers make informed decisions when analyzing and optimizing CUDA programs.

The rest of the paper is organized as follows Section 2 provides an overview of the static analysis approach along with details of tools used for static analysis of CUDA programs; Section 3 provides an overview of the dynamic analysis approach along with details of tools used for dynamic analysis of CUDA programs; Section 4 provides an overview of

the hybrid analysis approach along with details of tools used for hybrid analysis of CUDA programs; Section 5 presents related work; Section 6 presents result and discussion and Section 7 concludes the paper with future directions.

Based on the approach used for analysis, tools for CUDA GPGPU program analysis are divided in following categories namely static analysis, dynamic analysis, and hybrid approach.

2. STATIC ANALYSIS OF CUDA PROGRAMS

Static analysis of CUDA programs involves examining the source code without executing it to identify potential issues, ensure correctness, and predict performance characteristics. Static analysis of CUDA GPU programs takes CUDA source code as input. The CUDA source code is converted into a form easy to analyze called an Intermediate Representation (IR). The IR is further analyzed to determine control flow, memory accesses, and variable usage in the program. The massive parallelism of CUDA makes it more intricate. The analysis identifies potential problems in the source code without program execution such as race conditions, shared memory bank conflicts, divergent branches etc. Analysis report is generated, detailing the issues found, their location in the source code, and suggestions for fixes. The programmer analyzes the report, resolves the issues, and repeats the process until the issues are resolved.

Tools and Techniques for Static Analysis of CUDA Programs
Some of the tools developed to perform static analysis of CUDA programs are explained below.

2.1 GPUVerify[1]

GPUVerify is a static verification tool to detect barrier divergence and data races in CUDA and OpenCL kernels by converting the kernel into a sequential form. It uses Clang/LLVM to parse CUDA and OpenCL kernels and converts them into LLVM IR. It then transforms the LLVM IR into a Boogie form using Bugle, a translator to convert LLVM IR into a Boogie form of the kernel. It then uses a Boogie framework algorithm Houdini [2] to compute the conjunctive invariant. The Boogie verification engine then verifies the invariant and the sequential version of the program by applying SMT solvers such as Z3 to prove whether a kernel is free from concurrency errors. Major advantage of GPUVerify is that in addition to detecting failures, it provides proofs of correctness. GPUVerify may be used as a preferred choice in safety-critical, high-assurance scientific applications where correctness guarantees are crucial.

2.3 GPUCheck[3]

GPUCheck is a static analysis framework that detects branch divergence non-coalesceable memory accesses using Thread Dependence Analysis, Arithmetic Control Form Analysis (ACF) and Inter-procedural Arithmetic Control Form Analysis (IACF). GPUCheck works on the LLVM compiler infrastructure and generates ACF on demand from LLVM's SSA (a Static Single Assignment) intermediate representation , which requires a memory dependence. It supports reasoning global memory spaces, making it a comprehensive static checker for CUDA kernel correctness.

2.4. GPURepair[4]

GPURepair is a tool that is used to detect as well as repair and concurrency and synchronization bugs such as barrier divergence and data races. It supports analysis and repair of both CUDA and OpenCL kernels. GPURepair uses the SMT-based model generated by GPUVerify to identify the specific barriers responsible for the error and correct barrier divergence issues and eliminate redundant barriers. This makes it especially valuable in large code-bases where manual debugging is difficult. GPURepair extends GPU debugging beyond detection to automated correction of barrier divergence and inter-block data races for CUDA kernels. In addition to this, GPURepair uniquely provides automated suggestions to resolve inter-block data races by leveraging CUDA Cooperative Groups.

2.5. COpPER[5]

COpPER is a framework that leverages static resource analysis to automatically optimize CUDA GPU kernels without relying on actual execution or specific hardware. By predicting execution costs at compile time, the framework evaluates and selects optimizations that are likely to improve performance, making it especially helpful for programmers who lack deep expertise in GPU optimization. To demonstrate this approach, the authors present COpPER, a tool built on the RaCUDA[ ] static analysis system that explores optimization choices using a guided search driven by a resource- based performance model. COpPER applies hardware-independent transformations and does not require access to a GPU. The prototype implements two optimizations reducing warp divergence through branch distribution and improving memory access efficiency by moving frequently used data into shared memory and achieves performance gains of 2–4% on benchmark kernels. Overall, the study shows that static analysis can serve as a practical and effective basis for detection or identification of potential performance issues related to memory accesses such as uncoalesced memory accesses, out-of-bounds accesses, pointer misuse, shared memory conflicts etc; synchronization and concurrency bugs such as data races, barrier divergence, and illegal synchronization between threads in CUDA programs. Some of the static analysis tools such as GPURepair go beyond detection into the realm of automated correction of data races, barrier divergence including elimination of redundant barriers.

3. DYNAMIC ANALYSIS OF CUDA PROGRAMS

Dynamic analysis refers to techniques that observe a program at runtime in order to detect several issues such as correctness bugs, performance bottlenecks etc. that might be difficult or impossible to discover purely via static ie. compile-time analysis. Dynamic analysis is especially relevant in the GPU programming because many problematic behaviors only manifest when many threads interact and when memory/layout/synchronization conditions get stressed in real execution. Dynamic analysis involves executing the program to observe its real-time behavior, gather performance metrics, and pinpoint bottlenecks. [3, 10].

General process followed for dynamic analysis of a CUDA programs is explained below. The input to a dynamic analysis tool is a CUDA program, written in a C or C++. The code is usually instrumented ie. additional code is inserted into the source or binary to collect runtime data. The instrumented source code is then compiled into a binary that is executed in a profiling environment to collect real- time performance data. Profiling tools, such as NVIDIA Nsight Systems or Nsight Compute are used to collect essential data and parameters from the running program to create a visual and textual representation of the captured data for further analysis. Dynamic analysis tools usually provide both a system-level view and a Kernel-level view. For Example, Nsight Systems provide a High-level view that dipicts both the GPU and of CPU activities that aid in detecting bottlenecks like excessive data transfers between the device and the host. Nsight Compute provides detailed metrics for individual CUDA kernels such as memory throughput, achieved occupancy, and specific stall reasons in a Kernel-level view. The results of analysis are used in an iterative manner to optimize the code that is then re-profiled to ensure that the optimizations provide a significant performance gain.

Dynamic analysis tools can detect misaligned memory operations and out-of-bounds accesses that may get escaped by static analysis tools. Dynamic analysis tools also help to detect concurrency and synchronization bugs such as data races, barrier divergence, and improper use of atomic operations, which can cause nondeterministic behavior in parallel executions. Dynamic analysis tools profile runtime behavior to uncover several performance issues such as non-coalesced memory accesses, warp divergence, and unoptimized use of resources such as high register pressure or low SM occupancy that aids in architecture specific optimization of the CUDA kernel.

Dynamic analysis of CUDA kernels face several challenges due to several factors such as complex internal structure and massive parallelism of modern GPUs makes dynamic analysis of CUDA applications inherently complex. The outcome of dynamic analysis is affected by data input and kernel launch parameters, leading to non-discovery of some defects. Limited access to GPU hardware limits the accuracy of dynamic analysis. Exploration of a large number of complex thread inter-leavings and adding instrumentation at appropriate places in vast CUDA code for gathering the performance metrics imposes an additional overhead. Moreover, fast evolving GPU architectures affect the usability, maintainability, and portability of dynamic analysis tools.

Tools and Techniques for Dynamic Analysis of CUDA Programs
Several tools, APIs and frameworks exist that support dynamic analysis of CUDA programs that gather the performance metrics and detect specific classes of bugs as explained below.

3.1 GKLEE[6]

GKLEE is a (concrete + symbolic) i.e. concolic verification and test generation framework built on KLEE, to analyze CUDA/C++ GPU programs. Many static analysis and debugging tools find it difficult to detect several performance and correctness bugs such as inefficient memory accesses, warp divergence, deadlocks, and data races. These tools often produce false alarms or miss several errors and performance issues due to limited exploration. GKLEE avoids false positives and detects synchronization, concurrency, and performance bugs by using a symbolic virtual machine that accurately models the CUDA execution and memory model. It auto-generates concrete test cases that provide high coverage and can test the kernels on real hardware. It uses path and test-reduction heuristics for scalability. GKLEE demonstrates its effectiveness and practical applicability to real GPU kernels by discovering previously unknown correctness and performance bugs.

3.2 CURD (CUDA Race Detector)[7]

CURD is a dynamic analysis tool tailored for CUDA programs that detects intra-block and inter-block data races. It detects data races in both shared and global memory by instrumenting the kernels to monitor memory accesses at runtime. CURD detects situations where multiple threads access the same memory location with improper synchronization. It also tracks the happens-before relationships between threads and kernels. In contrast to the static tools, CURD operates on actual kernel executions, that facilitates detection of runtime issues that occur due to input-dependent behaviors or specific scheduling scenarios. CURD's optimized instrumentation results in low overhead compared to other dynamic checkers. This makes it a useful tool during development and testing phases where developers need concrete evidence of race conditions.

3.3 Simulee[8]

Simulee is a fully automated, lightweight framework that identifies CUDA synchronization bugs, that are hard to detect due to complex parallel thread interactions in CUDA GPU programs. Simulee works on LLVM bytecode of CUDA kernels. It collects detailed thread-wise memory access information by simulating the execution of CUDA kernels. Simulee comprises of two major components Automatic Input Generation and Bug Detection via a Memory-Access Model. The automatic input generation applies evolutionary programming to generate grid and block dimensions and kernel arguments that induct error and potentially trigger memory-access conflicts. The evolutionary programming approach outperforms coverage-based or random approaches. Using these auto generated inputs, Simulee develops an accurate memory-access model to detect synchronization issues such as barrier divergence, redundant barriers and data races. Thus, Simulee demonstrates higher effectiveness and efficiency than state-of-the-art CUDA synchronization bug detection tools.

3.4. ScoRD (Scope-Based Race Detector) [9]

ScoRD is a hardware-based efficient GPU race detection approach to detect global memory races and scoped races i.e. the races that result from improperly scoped synchronizations. ScoRD uses lockset-based and happens-before techniques with synchronization scope awareness to detect races caused by lock/unlock patterns, fences, and scoped atomic operations in CUDA programs. ScoRD implements minimal hardware state to store race detection logic and information related to synchronization operations. It utilizes a distinct approach of temporal locality in races, lightweight metadata for global memory accesses, and a direct mapped software caching that limits memory overhead to nearly 12.5% without significantly compromising on the race detection accuracy. The authors also created a benchmark suite, ScoR, that consists of 32 microbenchmarks and seven applications that use scoped synchronization to evaluate race detection capabilities. Experimental results prove that ScoRD can detect a wide range of global memory races with no false positives. ScoRD incurs an average runtime overhead of 35%. It can be turned on only during debugging and software testing to avoid production overheads. Overall, ScoRD represents the first hardware-based solution to detect scoped races on GPUs and balances accuracy and performance.

3.5 iGUARD[10]

iGUARD is a runtime race detection tool that is used to detect global memory races that arise due to improper use of modern GPU synchronization features, including Cooperative Groups (CG), scoped synchronization, and Independent Thread Scheduling (ITS). In contrast to the previous approaches that rely on specialized hardware support or CPU involvement for race detection, iGUARD uses binary instrumentation to detect races directly on the GPU. This enables it to lower the performance costs and utilize massive parallelism of GPUs. iGUARD is capable of detecting race conditions that are beyond the reach of existing tools. It expands existing scoped race detection techniques to handle CG and ITS, dynamically identifies lock usage, and combines happens-before and lockset-based analysis. For global memory accesses, it maintains lightweight metadata and minimizes contention to optimize shared metadata access by multiple threads running simultaneously.

Experimental evaluation shows that iGUARD identified 57 races across 21 GPU applications. iGUARD detected these races without any false positives with an average overhead of 5.1× demonstrating a substantial improvement over earlier race detectors. Overall, iGUARD is an efficient race detector for GPU applications.

3.6 ValueExpert[11]

ValueExpert is a profiling and dynamic analysis framework that is used to identify value-related inefficiencies in GPU-accelerated programs by providing detailed and actionable optimization insights. It utilizes two main techniques namely global value flow analysis and microscopic value pattern analysis. ValueExpert is able to discover value patterns at both coarse-grained and fine-grained levels. It uses binary instrumentation to observe GPU memory accesses, tracks the actual values being accessed, and links them to specific data objects such as arrays or tensors. It traces value propagations across CPUs and GPUs through memory allocations, data transfers, usages, updates, and GPU API calls and constructs a detailed value flow graph that identifies inefficiencies beyond API invocations and individual kernels. ValueExpert reduces unnecessary CPU-GPU data transfers, handles massively fine-grained GPU data, minimizes runtime overhead, and efficiently presents massive profiling information using effective visualization techniques for analyzing the profile information. It operates on widely adopted NVIDIA GPU platform, does not require any modifications in the source code and works on fully optimized binaries. ValueExpert revealed performance issues which were unknown previously. It is evaluated on widely used Rodinia benchmark suites, high performance computing applications such as NAMD and LAMMPS, and prominent deep learning frameworks such as PyTorch and Darknet. ValueExpert, achieved geometric mean speedups of 1.58× on RTX 2080 Ti GPUs and 1.39× on A100 GPUs, often with minimal code changes. Authors either verified the correctness of optimizations with developers or upstreamed them in repositories. Authors are able to classify eight GPU value patterns and discuss optimization opportunities for each detected value pattern.

ValueExpert also has some limitations. It uses dynamic approach for profiling and requires binary instrumentation. The detection of value based inefficiences is dependent on representative inputs. It curently works only on NVIDIA GPUs,

emphises on detection of value based inefficiencies and not on fixing them automatically, and relies on programmers to manually apply optimizations based on its findings.

3.7 CuCatch[12]

CuCatch is a software-based debugging tool that detects temporal and spatial memory safety errors in CUDA GPU programs with minimal runtime overhead. It analyzes PTX code and instruments it to access metadata, to perform memory safety checks. It integrates two step optimized compiler instrumentation, an innovative memory safety approach and support from the GPU driver to identify memory violations on standard GPU hardware. It is implemented as an instrumentation pass within NVIDIA's backend compiler. It employs a novel technique, Shadow Tagged Base & Bounds (Shadow TBB), that provides wider error detection coverage than earlier algorithms such as tripwires, canaries, or memory tagging. It uses base pointer analysis to quickly retrieve the metadata required for safety checks. CuCatch includes different optimizations in shared, global and local memory spaces. It incurs an average runtime overhead of 19%, that makes it faster compared to other existing GPU debugging tools. Overall, due to its strong detection capabilities, implementation as an instrumentation pass in NVIDIA's backend compiler and low overhead, cuCatch an efficient and a scalable solution for debugging memory safety issues in modern CUDA applications.

4. Hybrid – Static plus Dynamic Analysis Approach

The hybrid analysis approach combines static and dynamic analysis approach. The static analysis approach analyzes the code in source(CUDA) or intermediate form (PTX for CUDA programs) without executing it. A dynamic analysis approach executes the code to monitor its behavior dynamically at runtime to evaluate the performance of CUDA GPGPU programs more efficiently. It uses static analysis to identify potential issues and generate execution traces and uses dynamic analysis to validate the findings of static analysis by instrumenting the code to collect runtime dependent metrics. The hybrid approach alleviates the limitations of pure static approach that may produce false positives and pure dynamic analysis that incurs a runtime overhead. Overall, the hybrid approach reduces false positives, is faster, and is able to detect both correctness issues such as memory errors, data races and performance bottlenecks such as branch divergence, memory bandwidth etc.

Tools for Hybrid i.e. Static plus Dynamic Analysis of CUDA Programs

GPUDrano[13]

GPUDrano provides a hybrid i.e. static plus dynamic analysis approach to analysis of CUDA kernels. The results of static analysis are compared empirically with the results of a dynamic analysis to validate the correctness of results of static analysis. This proves that the false positives are very rare for most of the programs. GPUDrano is used to determine uncoalesced global memory accesses in CUDA kernels, which may lead to degraded performance on GPUs due to elevated memory latency. GPUDrano uses intra-procedural dataflow analysis, abstract interpretation and uses factors such as set of active threads, data sizes, and index computations to statically analyze memory access patterns. The analysis is implemented in the LLVM-based gpucc compiler and can scale to CUDA applications with thousands of lines of code. Experimental evaluation on the Rodinia benchmark suite shows that GPUDrano accurately detects the majority of uncoalesced accesses, finding 133 out of 143 cases, with corrections resulting in performance improvements of up to 25%. The static analysis approach of GPUDrano avoids the scalability, limited coverage, and high overhead issues of dynamic techniques at the cost of few false positives for some programs. GPUDrano can be used for optimization of memory intensive high-performance computing applications.

5. RELATED WORK

The literature on tools for analysis of GPU-accelerated programs is diverse, covering both performance profiling/trace tools and correctness/debugging tools, often specific to the CUDA platform. In this section we summarise existing literature, highlighting major tool types and their key contributions, then identify gaps that motivate our study.

Michael K. and Bernd M. [14] present two types of tools namely debuggers and performance analysis tools. Debuggers are used to detect bugs in both GPU and CPU code whereas performance analysis tools detect performance bottlenecks that help to improve the execution performance of the code. They present tools for different parallel programming models such as CUDA, OpenCL, OpenACC[15], OpenMP[16], Kokkos[17], RAJA[18] (no profiling interface) etc. They provide overview of debugging tools such as CUDA-MEMCHECK, CUDA-GDB, TotalView, DDT, OMPD and the programming models that support these tools. They also discuss performance analysis tools such as NVIDIA Tools, ARM Tools, Score-P, TAU, HPCToolkit, Extrae/Paraver that support different GPU programming models such as CUDA, OpenCL, OpenACC etc. Overall, this work presents only an overview of these tools and do not discuss these tools in detail.

Bridges et. al.[19] present an overview of of profiling, modeling, and simulation based techniques used to measure, estimate, and predict power consumption in modern GPUs, that play an important role in high-performance computing systems. The paper discusses different approaches such as direct power profiling using external power meters and internal power sensors, counter-based power modeling that predicts energy usage from hardware performance

counters, and GPU power simulation frameworks that estimate power behavior from architectural and code-level models without requiring execution on real hardware.

Al-Mouhamed [20] present a comprehensive review of code restructuring and optimization tools by CUDA-lite, OpenMPC, OpenACC, HMPP, R-Stream, RT-CUDA, CUDA-ChiLL, FastFlow. It also discusses different Basic Architectural Optimizations (BAs) and their functional specifications that should be used by the compiler or the software tool to optimize CUDA programs. It focuses on identifying useful optimizations for improving the performance of Structured Grid Computing (SGC) algorithms such as the iterative linear algebra solvers. It discusses different automatic techniques for reducing the complexity and integration in a framework to simplify the development of optimized code and efficient storage.

Other works[21] exist that present a methodology for detection of data races using dynamic analysis. Eventhough, it provides comparative evaluation of data race detection with two other tools mamely, iGUARD and Compute Sanitizer but does not include the comparison of HiRace with several other race detection tools such as CURD, ScoRD etc.

Anmol P. et.al.[22] present a comparitive study of two verification tools namely a static verification tool GPUVerify and a dynamic concolic verification tool GKLEE. These both tools are used to identify programming bugs such as data races and barrier divergence. G+KLEE additionally reports dynamic aspects such as bank conflicts, inefficient memory accesses, and thread divergence within a warp. The comparison is based on the specific bugs reported by these two tools, their scope, usability, scalability, and learnability aspects.

Cogumbreiro et. al. [23][24] compare several static verification tools including Faial, GPUVerify, and PUG including GKLEE and SESA on real-world CUDA kernels with respect to correctness, scalability and real world usability. The paper claims that their approach Faial detects more bugs, results in fewer false alarms, scales better to larger programs and is more usable on real-world CUDA programs than the existing static verification tools.

While profiling is crucial for performance, debugging for ensuring correctness and elimination of memory, synchronization errors in GPU programs are equally challenging. Although several individual tools exist for profiling, and debugging CUDA programs, very few papers discuss comprehensive comparative analysis and evaluation of these tools with respect to their feature-set, usability, overhead, and limitations. Thus, there is a relative gap in rigorous comparative studies of which tools to use when, how their overheads compare, what feature-tradeoffs exist and what best practices emerge.

6. RESULT AND DISCUSSION

This paper provides a comprehensive overview and classification of various tools designed for CUDA General Purpose GPU (GPGPU) program analysis, profiling, debugging, verification, and optimization focusing on their functionalities. By examining these tools, we aim to guide developers, researchers, and performance engineers to select appropriate solutions for their specific needs in profiling, analysis, debugging, verification and optimization of CUDA application programs.

7. CONCLUSION

The increasing complexity of GPU-accelerated applications necessitates robust tools for performance analysis. In this paper, we present the details of GPU program analysis techniques and tools in response to increasing computational demands. This paper provides a comprehensive overview and classification of various tools designed for CUDA General Purpose GPU (GPGPU) program analysis, profiling, debugging, verification, and optimization. This comprehensive survey of tools, will aid the developers, researchers, and performance engineers to select appropriate solutions for their specific needs in profiling, analyzing, debugging, verification and optimization of CUDA application programs. This work is limited to the study of third-party tools for the static, dynamic and hybrid approaches to the analysis of CUDA GPGPU programs. Details of official tools provided by NVIDIA for analyzing and profiling CUDA kernels will be included in the extension of this work.

REFERENCES

[1] Bardsley, E., Betts, A., Chong, N., Collingbourne, P., Deligiannis, P., Donaldson, A.F., Ketema, J., Liew, D., Qadeer, S.: Engineering a static verification tool for gpu kernels. In: Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26, pp. 226–242 (2014). Springer.
[2] Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: FME. pp. 500–517 (2001).
[3] Lloyd, T., Ali, K., Amaral, J.N.: Gpucheck: Detecting cuda thread divergence with static analysis (2019) https://doi.org/10.7939/R3W669R4S.

[4] Joshi, Saurabh, and Gautam Muduganti. "GPURepair: automated repair of GPU kernels." In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pp. 401-414. Cham: Springer International Publishing, 2021.

[5] Lou, Mark, and Stefan K. Muller. "Automatic Static Analysis-Guided Optimization of CUDA Kernels." In *Proceedings of the 15th International Workshop on Programming Models and Applications for Multicores and Manycores*, pp. 11-21. 2024. **COpPER**

[6] Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: Gklee: concolic verification and test generation for gpus. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 215–224 (2012). https://doi.org/10.1145/2370036.2145844.

[7] Peng, Y., Grover, V., Devietti, J.: Curd: A dynamic cuda race detector. ACM SIGPLAN Notices 53(4), 390–403 (2018) https://doi.org/10.1145/3192366.3192368.

[8] Mingyuan Wu, Yicheng Ouyang, Husheng Zhou, Lingming Zhang, Cong Liu, and Yuqun Zhang. 2020. Simulee: detecting CUDA synchronization bugs via memory-access modeling. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20). Association for Computing Machinery, New York, NY, USA, 937–948. https://doi.org/10.1145/3377811.3380358.

[9] A. K. Kamath, A. A. George and A. Basu, "ScoRD: A Scoped Race Detector for GPUs," 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 2020, pp. 1036-1049, doi: 10.1109/ISCA45697.2020.00088.

[10] Aditya K. Kamath and Arkaprava Basu. 2021. IGUARD: In-GPU Advanced Race Detection. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21). Association for Computing Machinery, New York, NY, USA, 49–65. https://doi.org/10.1145/3477132.3483545

[11] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. 2022. ValueExpert: exploring value patterns in GPU-accelerated applications. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22). Association for Computing Machinery, New York, NY, USA, 171–185. https://doi.org/10.1145/3503222.3507708.

[12] Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W. Keckler, and Mark Stephenson. 2023. CuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications. Proc. ACM Program. Lang. 7, PLDI, Article 111 (June 2023), 24 pages. https://doi.org/10.1145/3591225.

[13] Alur R., Devietti, J., Navarro Leija, O.S., Singhania, N.: Gpudrano: Detecting uncoalesced accesses in gpu programs. In: Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30, pp. 507–525 (2017). Springer.

[14] Knobloch, Michael, and Bernd Mohr. "Tools for gpu computing–debugging and performance analysis of heterogenous hpc applications." *Supercomputing Frontiers and Innovations* 7, no. 1 (2020): 91-111.

[15] Dietrich, R., Juckeland, G., Wolfe, M.: OpenACC programs examined: a performance analysis approach. In: 2015 44th International Conference on Parallel Processing, 1-4 Sept. 2015, Beijing, China. pp. 310–319. IEEE (2015), DOI: 10.1109/ICPP.2015.40

[16] Dagum L, Menon R (1998) OpenMP: an industry-standard API for shared-memory programming.IEEE Comput Sci Eng 5(1):46–55. https://doi.org/10.1109/99.660313

[17] Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing 74(12), 3202–3216 (2014), DOI: 10.1016/j.jpdc.2014.07.003

[18] Beckingsale, D.A., Burmark, J., Hornung, R., et al.: RAJA: Portable Performance for LargeScale Scientific Applications. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC, P3HPC, 22-22 Nov. 2019, Denver, CO, USA. pp. 71– 81. IEEE (2019), DOI: 10.1109/P3HPC49587.2019.00012

[19] Bridges, R. A., Imam, N., & Mintz, T. M. (2016). Understanding GPU power: A survey of profiling, modeling, and simulation methods. *ACM Computing Surveys (CSUR)*, *49*(3), 1-27.

[20] Al-Mouhamed, Mayez A., Ayaz H. Khan, and Nazeeruddin Mohammad. "A review of CUDA optimization techniques and tools for structured grid computing." *Computing* 102, no. 4 (2020): 977-1003.

[21] HiRace: Accurate and Fast Data Race Checking for GPU Programs Jacobson, John, Martin Burtscher, and Ganesh Gopalakrishnan. "Hirace: Accurate and fast data race checking for gpu programs." In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-14. IEEE, 2024.

[22] Panda, Anmol, Philipp Rümmer, and Neena Goveas. "A comparative study of GPU verify and GKLEE." In *2016 Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, pp. 112-117. IEEE, 2016.

[23] Cogumbreiro, Tiago, Julien Lange, Dennis Liew Zhen Rong, and Hannah Zicarelli. "Checking data-race freedom of GPU kernels, compositionally.", In International Conference on Computer Aided Verification, pp. 403-426. Cham: Springer International Publishing, 2021.

[24] Cogumbreiro, Tiago, Julien Lange, Dennis Liew, and Hannah Zicarelli. "Memory access protocols: certified data-race freedom for GPU kernels." *Formal Methods in System Design* 63, no. 1 (2024): 134-171.