Luthier: A Dynamic Binary Instrumentation Framework Targeting AMD GPUs

Matin Raayai-Ardakani
Northeastern University
Boston, Massachusetts, USA
raayaiardakani.m@northeastern.edu

Norman Rubin
Northeastern University
Boston, Massachusetts, USA
n.rubin@northeastern.edu

David Kaeli
Northeastern University
Boston, Massachusetts, USA
kaeli@ece.neu.edu

Abstract

In this poster we present Luthier [7], the first open-source dynamic binary instrumentation framework targeting AMD GPUs. We highlight key features of our framework, including example use cases and runtime overhead comparison with NVIDIA's NVBit. We also go over some major enhancements under development in the latest version of Luthier that support more of the growing family of AMD GPUs and additional instrumentation scenarios.

Keywords

Dynamic Binary Instrumentation, AMD GPU, AMD ROCm

ACM Reference Format:
Matin Raayai-Ardakani, Norman Rubin, and David Kaeli. 2025. Luthier: A Dynamic Binary Instrumentation Framework Targeting AMD GPUs. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage, and Analysis ('SC).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In recent years, AMD GPUs have experienced rapidly growing adoption in data centers and supercomputers for accelerating large-scale scientific computing and AI applications. This trend is highlighted by the emergence of Frontier and El Capitan at the top of the TOP500 supercomputers list [9]. To ensure these powerful GPUs are utilized to their full potential, effective profiling tools for ROCm applications are needed.

Currently, ROCm profiling tools rely primarily on services provided by ROCm's `ropcrofiler-sdk`, such as reading GPU hardware counters and Program Counter (PC) sampling [1]. Recent efforts have focused on introducing *binary instrumentation* capabilities targeting the device code of ROCm applications [4, 7, 8, 10]. Instrumentation is a valuable profiling technique. Compared to services provided by `rocprofiler-sdk`, instrumentation can pinpoint the exact location of hotspots inside the kernel code, though at the cost of patching and running additional code alongside the kernel [2].

In this poster, we present Luthier [7], the first open-source dynamic binary instrumentation framework for instrumenting the device code of ROCm applications. We also provide details on ongoing development work on the framework.

'SC, Saint Louis, MO

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 Luthier's Instrumentation Process

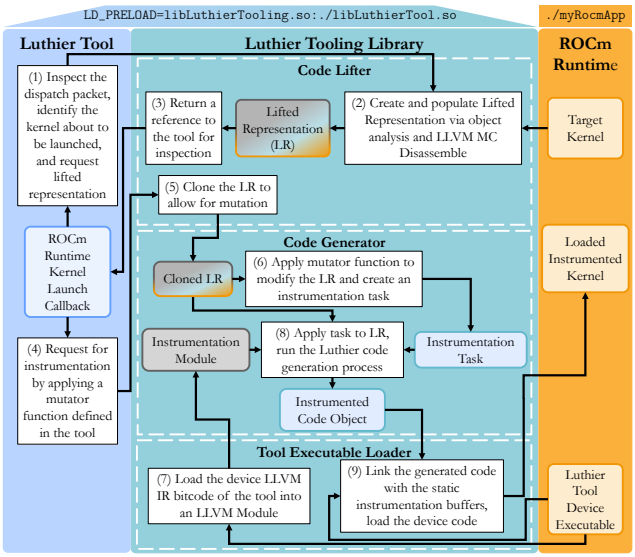

Figure 1: High-level overview of Luthier's instrumentation process.

Figures 1 and 2 show a high-level overview of Luthier's instrumentation and code generation process. Luthier tools are shared object libraries written in HIP loaded via `LD_PRELOAD`. When a kernel is launched, the tool can request the kernel's relocatable LLVM Machine IR (MIR) view called a *lifted representation* (figure 2-b). The lifted representation can be inspected by the tool and modified via a mutator function. Direct modifications of the target instructions, and injection of calls to instrumentation functions are supported by the mutator (figure 2-a).

In step 8.1 of figure 2, the lifted representation (figure 1-b), the mutator function (figure 1-a), and the tool's embedded device bitcode (figure 1-d) are used to create and optimize the instrumentation IR logic that will be injected before target instructions (figure 1-d).

In steps 8.2 and 8.3, the LLVM instruction selection and backend passes, plus a set of custom Luthier passes are applied to obtain the MIR of the instrumentation logic. It is then patched into the lifted representation to obtain the instrumented kernel (figure 2-f).

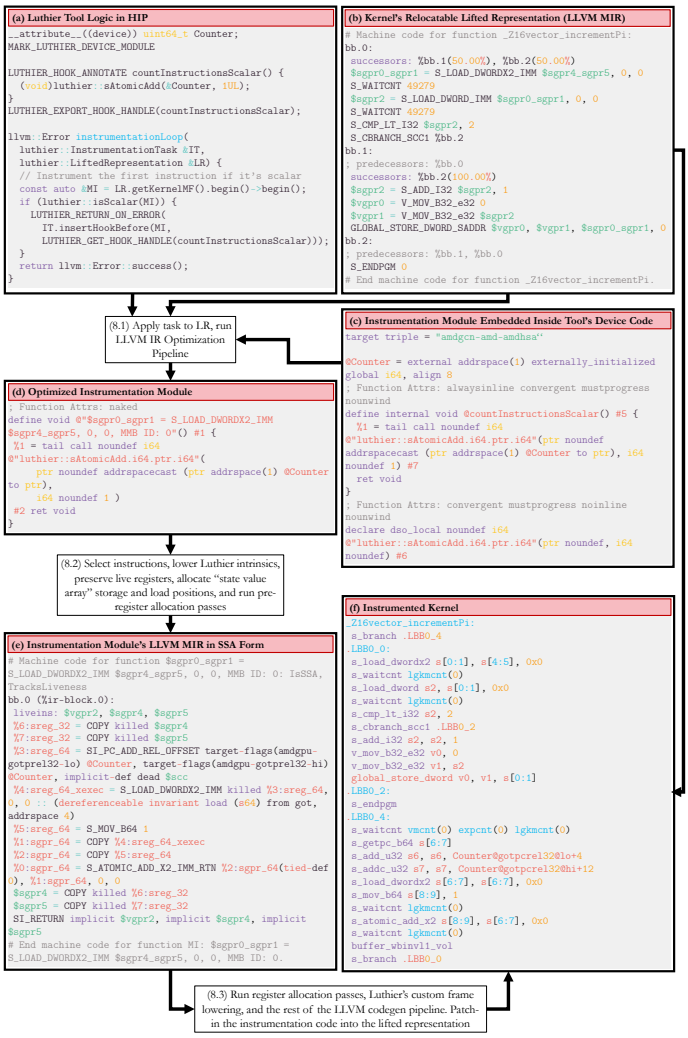

Figure 2: Luthier's code generation process (step 8 of figure 1).

The instrumented kernel is then loaded on the device after being linked.

3 Key Design Features

Instead of instrumenting the loaded kernel code in-place, Luthier instruments and loads a copy. This design choice is made due to the variable-length encoding of the AMD GPU instructions and the limited range of the direct unconditional jump instruction [7].

Instead of inserting calls to a pre-compiled instrumentation function, Luthier uses *Just-In-Time* (JIT) compilation to generate and patch in instrumentation code. JIT compilation unlocks plenty of opportunities to optimize the register usage of the instrumented kernel via static analysis on the target kernel. It can also make sure that instrumentation functions are inlined for reduced overhead.

Luthier does not support inline assembly inside its tool device logic. Instead, it provides a custom intrinsics binding and lowering mechanism to express low-level logic, allowing for more aggressive register usage optimization. Luthier does not rely on the presence of a stack pointer inside the target kernel and instead uses the *state-value array* (SVA) mechanism to maintain its own stack and the kernel's arguments [7]. The SVA ensures kernels not adhering to calling conventions can be instrumented. SVA and Luthier's intrinsic lowering are also used to support calling complicated routines including `printf` inside the instrumentation logic.

4 Example Luthier Use Cases

Luthier can be used to count the number of device instructions executed throughout the program, as well as report an opcode breakdown. Luthier's instruction count tool was applied to benchmarks from HeC Bench [5], which includes HIP, OpenMP, and SYCL programming models, and run on an AMD Instinct MI100 [7]. Figure 3-a shows the results of the experiments, indicating that, on average, OpenMP benchmarks execute more instructions compared to their HIP and SYCL counterparts.

Luthier can also be used to detect bank conflicts inside kernels. This tool can point out the exact instructions where the bank conflict occurred, which can then be correlated back to the source code. The Luthier bank conflict tool was applied to entries in HeC Bench [5] across HIP and SYCL and run on an AMD Instinct MI100 [7]. Figure 3-b shows the number of bank conflicts detected in select HeC benchmarks.

5 Other Use Cases of Luthier Under Development

We plan on developing additional use cases for Luthier. Some examples include Causal profiling [3] to identify most impactful optimization opportunities in the source code, Call Graph reconstruction to identify functions called and their callers, and injecting software faults into kernel code to study device code resilience.

6 Instrumented Kernel Runtime Overhead

We compared the overhead of running Luthier's instruction count tool (section 4) against its NVBit's counterpart on the same benchmark entries as seen in figure 3-c [7]. The NVIDIA benchmarks were run on an NVIDIA V100 attached to the same machine. On average, Luthier incurs a 50X runtime overhead, which is 10 times lower than NVBit, demonstrating the power of JIT compilation in Luthier's instrumentation process.

7 Current Focus of Development

We are expanding Luthier to instrument very large kernels with calls to stripped device functions. Lack of symbol information makes it difficult to identify functions due to their alignment. Usage of function pointers makes this more challenging. We are also working on instrumenting kernels written in assembly that might not conform to LLVM MIR conventions (e.g., overlapping function logic), as outlined by [6]. Support for additional AMD CDNA and RDNA GPUs is also being actively worked on.

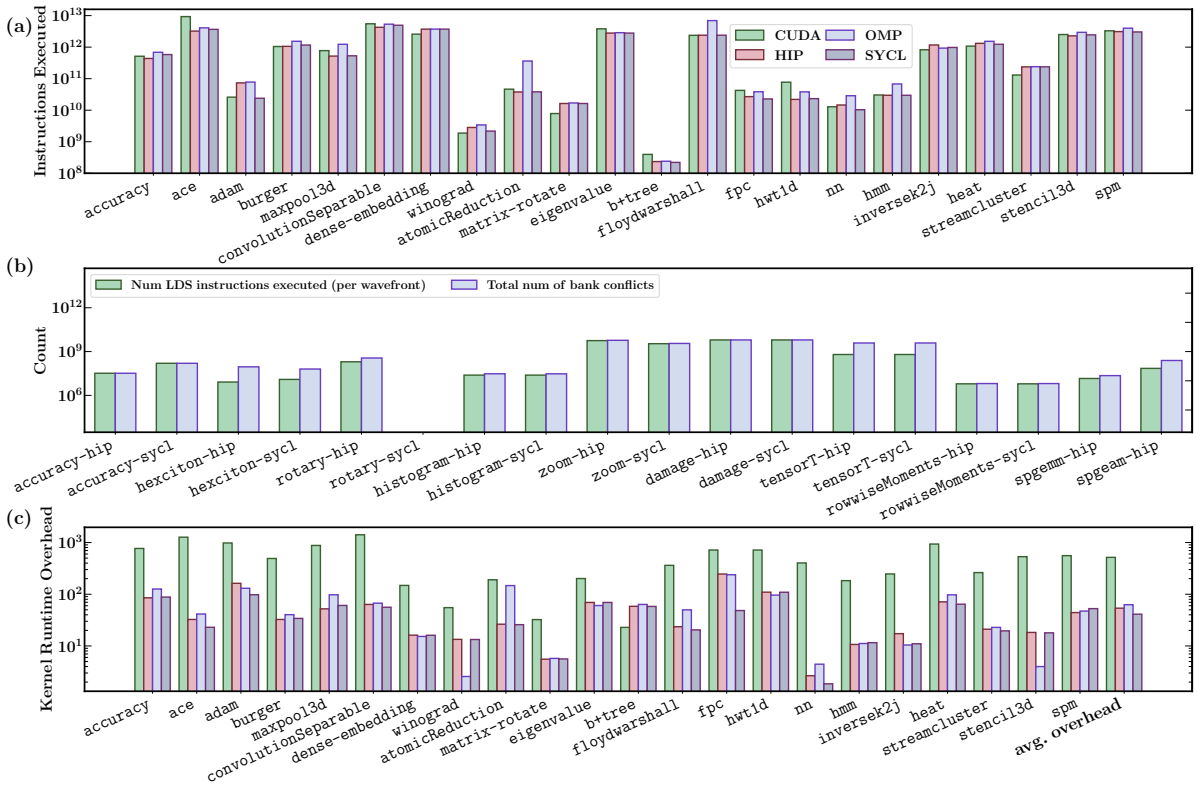

Figure 3: (a) Number of instructions executed by each thread for select HeC [5] benchmarks in HIP, OpenMP, and SYCL. (b) Number of LDS instructions executed per wavefront and the number of bank conflicts, for select HeC [5] benchmarks. (c) Kernel runtime overhead incurred when running the instrumented versions of benchmarks from the first use-case, calculated as instrumented kernel runtime divided by un-instrumented kernel runtime.

8 Acknowledgments

The authors would like to once again thank the individuals involved in helping with preparing the contents of the original Luthier paper [7].

References

[1] Advanced Micro Devices. 2024. The ROCm Profiler SDK Library. https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/amd-mainline/index.html.

[2] E.R. Altman, D. Kaeli, and Y. Sheffer. 2000. Welcome to the opportunities of binary translation. *Computer* 33, 3 (2000), 40–45. doi:10.1109/2.825694

[3] Charlie Curtsinger and Emery D Berger. 2015. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 184–197.

[4] Sébastien Darche and Michel R. Dagenais. 2024. Low-Overhead Trace Collection and Profiling on GPU Compute Kernels. *ACM Trans. Parallel Comput.* 11, 2, Article 9 (jun 2024), 24 pages. doi:10.1145/3649510

[5] Zheming Jin and Jeffrey S. Vetter. 2023. A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 325–327. doi:10.1109/ISPASS57527.2023.00041

[6] Xiaozhu Meng and Barton P Miller. 2016. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 24–35.

[7] Matin Raayai-Ardakani, Andrew Nguyen, Ivan Rosales, Daoxuan Xu, Yuwei Sun, Yifan Sun, David Kaeli, and Norman Rubin. 2025. Luthier: A Dynamic Binary Instrumentation Framework Targeting AMD GPUs. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, IEEE, Piscataway, NJ, 137–149.

[8] Corbin Robeck, Keith Lowery, Rene Van Oostrom, and Cole Ramos. 2024. LLVM/MLIR Based Instrumentation of AMDGPU Kernels. https://github.com/CRobeck/instrument-amdgpu-kernels

[9] Erich Strohmaier. 2006. TOP500 - TOP500 supercomputer.. In *SC*. ACM Press, New York, NY, 18. http://dblp.uni-trier.de/db/conf/sc/sc2006.html#Strohmaier06

[10] Shuen-Heng Wu. 2024. Register Analysis for General Instrumentation of AMDGPU Kernels. https://dyninst.github.io/scalable_tools_workshop/petascale2024/assets/slides/AMDGPU-Reg-Analysis-STW-2024.pdf