# Luthier: A Dynamic Binary Instrumentation Framework Targeting AMD GPUs

**Matin Raayai-Ardakani**, Norman Rubin (Advisor), and David Kaeli (Advisor)

Department of Electrical and Computer Engineering, Northeastern University, Boston MA USA

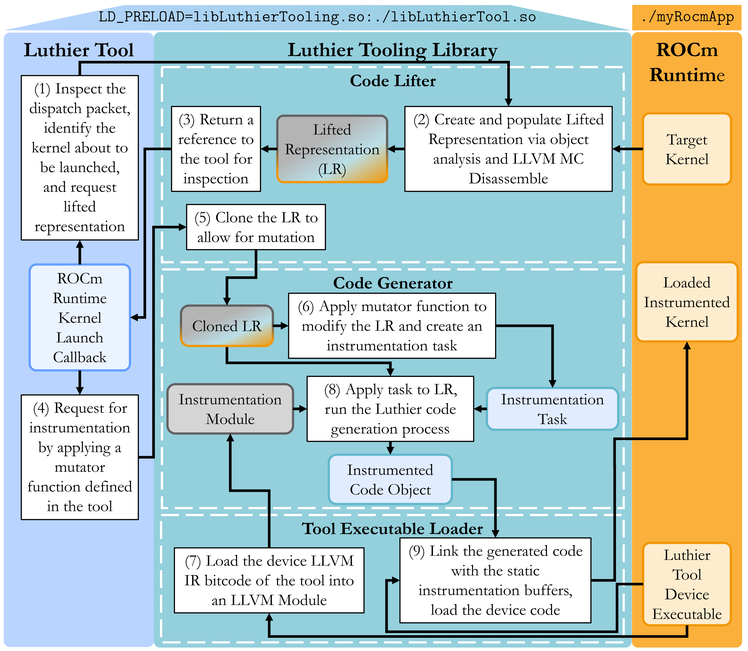
Northeastern University

## Motivation

- AMD GPUs have experienced growing adoption in HPC for accelerating large-scale scientific computing and AI applications
- Current AMD ROCm profiling tools rely on reading hardware counters and Program Counter (PC) sampling provided by ROCm' ROCprofiler-SDK
- Instrumentation of device code can be used to collect significantly more detailed info regarding regions of interest at the cost of increased execution overhead
- Current efforts only focus on offline instrumentation of device code in ROCm
- Dynamic binary instrumentation offers multiple benefits:
  - Does not require application recompilation from source
  - Works with kernels written in assembly
  - Can be selectively applied to manage instrumentation overhead

**We present Luthier, the first open-source dynamic binary instrumentation framework for AMD GPUs on the ROCm stack**

## Luthier's Instrumentation Process




## Key Design Features

- Instrumentation is performed on the relocatable LLVM MIR representation of the target kernel and loaded as a separate copy
  - In-place instrumentation is not feasible due to variable-length instructions and limited range of direct jumps on AMD GPUs.
- Instrumentation code is Just-In-Time (JIT) compiled to:
  - Inline the instrumentation logic in the target kernel's code
  - Reuse available free registers of the target kernel in the instrumentation code
  - Only generate spill code if no free registers are available
  - Leverage additional static analysis on the target kernel to fully optimize instrumentation code
- Instead of using inline assembly, a custom intrinsic binding and lowering mechanism is used (`luthier::sAtomicAdd` in the example code):
  - Allows for more aggressive register usage optimization
  - Can define custom lowering logic for existing intrinsics according to the instrumentation process
- Instrumentation code maintains its own stack and kernel argument values (`$vgpr2` in the example code):
  - Instrumentation stack is recovered/spilled in its custom emitted frame
  - Guarantees correct instrumentation of kernels written in assembly
  - Can be used to fetch thread identifier values and issue calls to `printf`

## Luthier's Code Generation: A Closer Look

**Luthier Tool Logic in HIP**

```
__attribute__((device)) uint64_t Counter;
MARK_LUTHIER_DEVICE_MODULE

LUTHIER_HOOK_ANNOTATE countInstructionsScalar() {
  (void)luthier::sAtomicAdd(&Counter, 1UL);
}
LUTHIER_EXPORT_HOOK_HANDLE(countInstructionsScalar);

llvm::Error instrumentationLoop(
  luthier::InstrumentationTask &IT,
  luthier::LiftedRepresentation &LR) {
  // Instrument the first instruction if it's scalar
  const auto &MI = LR.getKernelMF().begin()->begin();
  if (luthier::isScalar(MI)) {
    LUTHIER_RETURN_ON_ERROR(
      IT.insertHookBefore(MI,
      LUTHIER_GET_HOOK_HANDLE(countInstructionsScalar)));
  }
  return llvm::Error::success();
}
```

**Kernel's Relocatable Lifted Representation (LLVM MIR)**

```
# Machine code for function _Z16vector_incrementPi:
bb.0:
 successors: %bb.1(50.00%), %bb.2(50.00%)
 $sgpr0_sgpr1 = S_LOAD_DWORDX2_IMM $sgpr4_sgpr5, 0, 0
 S_WAITCNT 49279
 $sgpr2 = S_LOAD_DWORD_IMM $sgpr0_sgpr1, 0, 0
 S_WAITCNT 49279
 S_CMP_LT_I32 $sgpr2, 2
 S_CBRANCH_SCC1 %bb.2
bb.1:
; predecessors: %bb.0
 successors: %bb.2(100.00%)
 $sgpr2 = S_ADD_I32 $sgpr2, 1
 $vgpr0 = V_MOV_B32_e32 0
 $vgpr1 = V_MOV_B32_e32 $sgpr2
 GLOBAL_STORE_DWORD_SADDR $vgpr0, $vgpr1, $sgpr0_sgpr1, 0
bb.2:
; predecessors: %bb.1, %bb.0
 S_ENDPGM 0
# End machine code for function _Z16vector_incrementPi.
```

(8.1) Apply task to LR, run LLVM IR Optimization Pipeline

**Optimized Instrumentation Module**

```
; Function Attrs: naked
define void @"$sgpr0_sgpr1 = S_LOAD_DWORDX2_IMM
$sgpr4_sgpr5, 0, 0, MMB ID: 0"() #1 {
 %1 = tail call noundef i64
@"luthier::sAtomicAdd.i64.ptr.i64"(
     ptr noundef addrspacecast (ptr addrspace(1) @Counter
to ptr),
     i64 noundef 1 )
 #2 ret void
}
```

**Instrumentation Module Embedded Inside Tool's Device Code**

```
target triple = "amdgcn-amd-amdhsa"

@Counter = external addrspace(1) externally_initialized
global i64, align 8
; Function Attrs: alwaysinline convergent mustprogress
nounwind
define internal void @countInstructionsScalar() #5 {
  %1 = tail call noundef i64
@"luthier::sAtomicAdd.i64.ptr.i64"(ptr noundef
addrspacecast (ptr addrspace(1) @Counter to ptr), i64
noundef 1) #7
  ret void
}
; Function Attrs: convergent mustprogress noinline
nounwind
declare dso_local noundef i64
@"luthier::sAtomicAdd.i64.ptr.i64"(ptr noundef, i64
noundef) #6
```

(8.2) Select instructions, lower Luthier intrinsics, preserve live registers, allocate "state value array" storage and load positions, and run pre-register allocation passes

**Instrumentation Module's LLVM MIR in SSA Form**

```
# Machine code for function $sgpr0_sgpr1 =
S_LOAD_DWORDX2_IMM $sgpr4_sgpr5, 0, 0, MMB ID: 0: IsSSA,
TracksLiveness
bb.0 (%ir-block.0):
 liveins: $vgpr2, $sgpr4, $sgpr5
 %6:sreg_32 = COPY killed $sgpr4
 %7:sreg_32 = COPY killed $sgpr5
 %3:sreg_64 = SI_PC_ADD_REL_OFFSET target-flags(amdgpu-
gotprel32-lo) @Counter, target-flags(amdgpu-gotprel32-hi)
@Counter, implicit-def dead $scc
 %4:sreg_64_xexec = S_LOAD_DWORDX2_IMM killed %3:sreg_64,
0, 0 :: (dereferenceable invariant load (s64) from got,
addrspace 4)
 %5:sreg_64 = S_MOV_B64 1
 %1:sgpr_64 = COPY %4:sreg_64_xexec
 %2:sgpr_64 = COPY %5:sreg_64
 %0:sgpr_64 = S_ATOMIC_ADD_X2_IMM_RTN %2:sgpr_64(tied-def
0), %1:sgpr_64, 0, 0
 $sgpr4 = COPY killed %6:sreg_32
 $sgpr5 = COPY killed %7:sreg_32
 SI_RETURN implicit $vgpr2, implicit $sgpr4, implicit
$sgpr5
# End machine code for function MI: $sgpr0_sgpr1 =
S_LOAD_DWORDX2_IMM $sgpr4_sgpr5, 0, 0, MMB ID: 0.
```

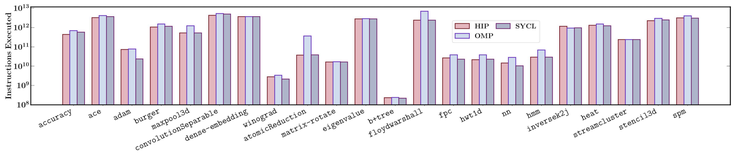
**Instrumented Kernel**

```
_Z16vector_incrementPi:
 s_branch .LBB0_4
.LBB0_0:
 s_load_dwordx2 s[0:1], s[4:5], 0x0
 s_waitcnt lgkmcnt(0)
 s_load_dword s2, s[0:1], 0x0
 s_waitcnt lgkmcnt(0)
 s_cmp_lt_i32 s2, 2
 s_cbranch_scc1 .LBB0_2
 s_add_i32 s2, s2, 1
 v_mov_b32_e32 v0, 0
 v_mov_b32_e32 v1, s2
 global_store_dword v0, v1, s[0:1]
.LBB0_2:
 s_endpgm
.LBB0_4:
 s_waitcnt vmcnt(0) expcnt(0) lgkmcnt(0)
 s_getpc_b64 s[6:7]
 s_add_u32 s6, s6, Counter@gotpcrel32@lo+4
 s_addc_u32 s7, s7, Counter@gotpcrel32@hi+12
 s_load_dwordx2 s[6:7], s[6:7], 0x0
 s_mov_b64 s[8:9], 1
 s_waitcnt lgkmcnt(0)
 s_atomic_add_x2 s[8:9], s[6:7], 0x0
 s_waitcnt lgkmcnt(0)
 buffer_wbinvl1_vol
 s_branch .LBB0_0
```

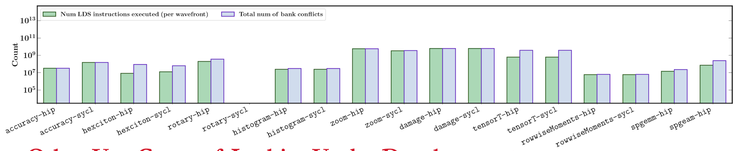
(8.3) Run register allocation passes, Luthier's custom frame lowering, and the rest of the LLVM codegen pipeline. Patch-in the instrumentation code into the lifted representation

## Example Luthier Use Cases

- Luthier can be used to **count the total number of instructions** executed on each GPU throughout the program as well as **a breakdown of the executed instruction opcodes**.
- Luthier's instruction count tool was applied to entries in HeC Bench[1] across HIP, OpenMP, and SYCL running on an AMD Instinct MI100.




- Luthier can also be used to **detect bank conflicts** inside kernels.
- It can point out the **exact instructions the bank conflict occurred**, which can then be correlated back into the source code if debug information is present.
- The Luthier bank conflict tool was applied to entries in HeC Bench[1] across HIP and SYCL and were run on an AMD Instinct MI100.

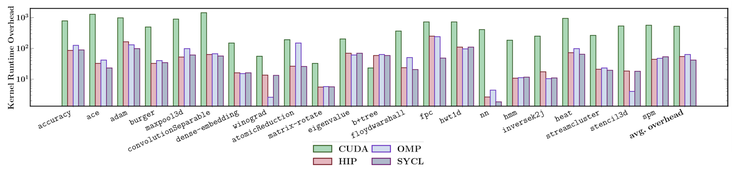



## Other Use Cases of Luthier Under Development

- **Causal Profiling[2]:** Identify most impactful optimization opportunities in exact regions of the source code
- **Call Graph Construction:** Identify functions called and their callers
- **Reliability:** E.g., inject software faults into the device code

## Instrumented Kernel Runtime Overhead

- We compared the overhead of running Luthier's instruction count tool against NVBit's instruction counter tool
- Luthier and AMD benchmarks were run on an AMD MI100, while NVBit and the NVIDIA equivalent benchmarks were run on an NVIDIA V100 attached to the same machine
- On average, Luthier has a 50X runtime overhead, **10 times lower than NVBit**




## Current Focus of Development

- Instrumentation of very large kernels with stripped device function symbols
- Instrumentation of assembly kernels that do not conform to LLVM MIR
- Support for more AMD GPU targets

**For more information, please refer to our paper:**
**M. Raayai-Ardakani et al., "Luthier: A Dynamic Binary Instrumentation Framework Targeting AMD GPUs," 2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Ghent, Belgium, 2025, pp. 137-149, doi: 10.1109/ISPASS64960.2025.00022.**

## Acknowledgements

- Timour Paltashev, Benjamin Welton, Konstantin Zhuravlyov, Mathew Arsenault, Jacob Lambert, Shilei Tian, Tony Tye, Brian Sumner (AMD)
- Rene Van Oostrum (AMD research)
- Omar Shair, Maya De Los Santos, Yuhui Bao, Zlatan Feric, Qishi Wang (Northeastern University)
- Luthier was supported in part by AMD

## References

[1] Z. Jin and J. S. Vetter, "A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model," 2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Raleigh, NC, USA, 2023, pp. 325-327, doi: 10.1109/ISPASS57527.2023.00041. (https://ieeexplore.ieee.org/document/10158214)

[2] 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. (https://dl.acm.org/doi/abs/10.1145/2815400.2815409)

raayaiardakani.m@northeastern.edu

https://github.com/matinraayai/Luthier