Efficient Instrumentation and Tracepoint Insertion for GPU Compute Kernels

Low-overhead Trace Collection on GPU

Sébastien Darche <`sebastien.darche` *at* `polymtl.ca`>

July 8th, 2025

Dorsal - Polytechnique Montréal

Current projects at DORSAL lab

- The Distributed Open Reliable Systems Analysis Lab
- Strong focus on trace collection and performance analysis
- LTTng, Trace Compass

Tooling for HPC

- Score-P traces support through CTF conversion, ROCm runtime instrumentation
- Multiple analyses available
  - Critical path for linux kernel traces
  - Hardware performance counters through Score-P
  - Call stack among ranks, statistics
  - Flame graph
  - Communicators, bandwidth
  - Critical path for MPI
  - ...
- Scalability of Trace Compass through distributed analyses (ongoing work)
- Current work on kernel instrumentation

GPU Tracing with `hip-analyzer`

- Few tools for tracing on GPUs, and often at the cost of very high performance impact (at minima $10\times$ and up to $120\times$) [1] [2]
- GPU Tracing is unwieldy : clumsy memory management, massive parallelism (concurrency control, high throughput)
- Separate buffer allocation and event collection using two kernel runs
- "Online" tracing methods
- LLVM IR (static) instrumentation

What's new ?

- Introduced tracing methods that do not require two executions, but has its own set of challenges
- Requires specific tuning for the hardware
  - Memory locality
  - Allocation granularity
  - Implementation choices
- Last project focuses on reducing instrumentation in the kernel

Results

- Instrumentation tested on the HeCBench [3] benchmark. Overhead is reported as the slowdown factor between the traced kernel execution time and the original, uninstrumented kernel.

	mean	median
`hip-trace`	2.07×	1.50×
4 × padded `hip-trace`	2.18×	1.58×
`hip-global-mem`	3.73×	1.96×
`hip-cu-mem`	2.47×	1.60×
`hip-chunk-allocator`	1.79×	1.33×
`hip-cu-chunk-allocator`	1.77×	1.32×

Reducing the number of tracepoints

- For most kernels, the number of tracepoints could be reduced
  - Reduction in trace size
  - Reduction in run time overhead
- Intuitively, if half of the threads go through an *if* statement, we can deduce the other half goes to the *else* statement
- Can be generalized to switch statements and more complex control flow (more than two outgoing edges)

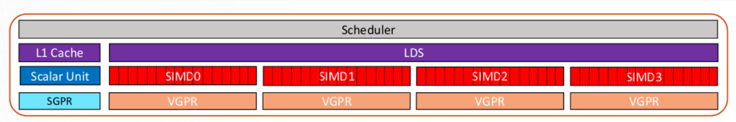


Figure 1: AMD GCN Compute unit [1]

[1] Reproduced from *AMD GPU Hardware Basics*, 2019 Frontier Application Readiness Kick-off Workshop

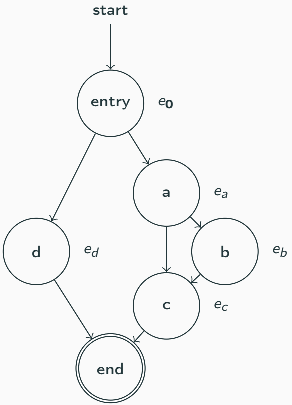
Static analysis

- In an acyclic CFG (simple case), the control flow can be completely computed by instrumenting $n-1$ outgoing edges
- Vertices in the CFG are processed using a variant of Kahn's algorithm

$$\dot{\bigvee_{e_i \in \text{incoming}}} T(e_i) = \dot{\bigvee_{e_i \in \text{outgoing}}} T(e_i) \tag{1}$$

- The algorithm does not terminate for CFGs containing a cycle
- We identify back edges using a depth-first search (DFS), and run the algorithm on the CFG stripped of its cycles. Back edges must be instrumented.

Resulting trace




```
__global__
void simple_kernel() {
    entry();
    if(c0()) {
        a();
        if(c1()) {
            b();
        }
        c();
    } else {
        d();
    }
    end();
    return;
}
```

Static analysis

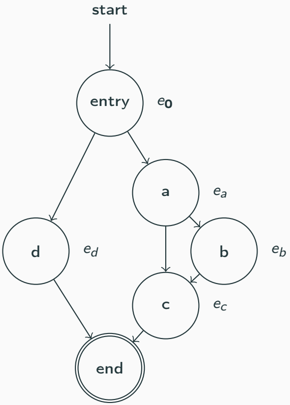
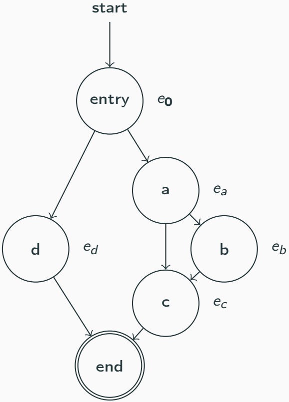


Figure 2: Thread-centric CFG Example

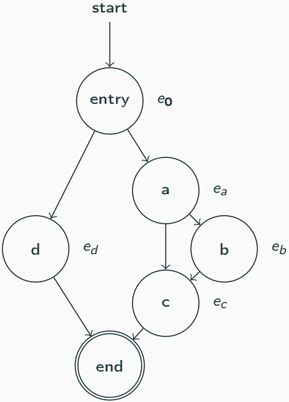
- $e_a = \overline{e_d} \cdot e_0$
- $e_c = e_a + \overline{e_b} \cdot e_a = e_a$
- $e_{end} = e_d + e_c = e_d + e_a = e_d + \overline{e_d} \cdot e_0 = e_0$

Resulting trace



Timestamp	Basic block	Execution mask
t_0	entry	1111_2
t_1	a	0111_2
t_2	b	0010_2
t_3	c	0111_2
t_4	d	1000_2
t_4	end	1111_2

Resulting trace



Timestamp	Basic block	Execution mask
t_0	entry	1111_2
t_1	entry $\rightarrow$ a	0111_2
t_2	a $\rightarrow$ b	0010_2
t_3	(c)	(0111_2)
t_4	(d)	(1000_2)
t_4	(end)	(1111_2)

Trace size reduction

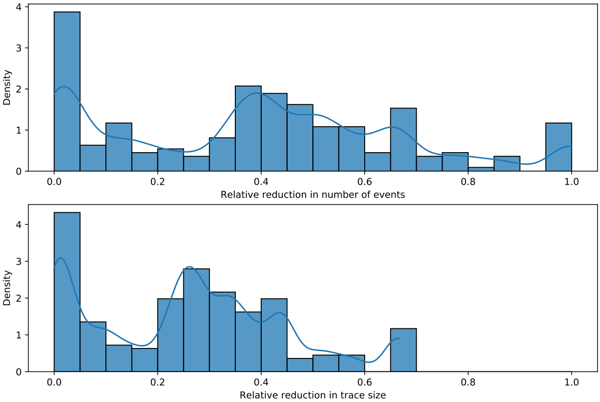


Figure 3: Relative reduction in number of events and total trace size

Run time overhead

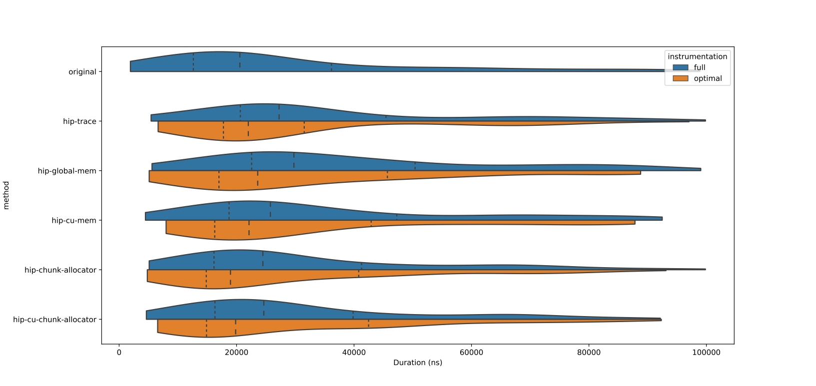


Figure 4: Distribution of kernel run time as a function of collection method and instrumentation

Trade-offs in GPU Tracing – Instrumentation

- Instrumentation methods are intrusive and *will* modify how the kernel runs. Tradeoff between :
  - Increased register pressure (may affect occupancy)
  - Reusing registers (*scavenging*) will probably mean spills
- Tracepoints will incur a runtime overhead

Trade-offs in GPU Tracing – Memory management

- Trace management is a major concern
- Uncertain trace size – may exceed memory
- Synchronization inside kernel bounds is not defined by the memory model
- "Smarter" trace management methods are more costly (*cf* instrumentation)

Trade-offs in GPU Tracing – Trace analysis

- What data are we presenting to the end user?
- Thread-centric (programming language) vs. Vector representation (ISA)
- Large (!) trace files

Tradeoffs in GPU Tracing – Future works

- Better compiler support
    - Scalar / vector registers specifications
    - Scalar / vector instructions
    - Intrinsics
    - Backend plug-ins?
- Better hardware support
    - CU-wide registers and memory access
    - Host / Device interaction
- Finer memory model

Conclusion and future work

- PhD project is nearing its end
- Explored instrumentation methods for tracing compute kernels
- Studied the performance impact of data structures for online tracing
- Improved baseline results by reducing the number of tracepoints
- Interest for the project from partners
- Available freely on Github, feedback and/or use cases are more than welcome

dorsal-lab/hip-analyzer
Compiler plugin for performance analysis of HIP applications
C++ ★ 2 1

dorsal-lab/TraceCompassGpu
Trace Compass GPU plugins
Java

Q&A

References

[1] D. Shen, S. L. Song, A. Li, and X. Liu, **"Cudaadvisor: Llvm-based runtime profiling for modern gpus,"** in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018.

[2] Y. Arafa, A.-H. Badawy, A. ElWazir, *et al.*, **"Hybrid, scalable, trace-driven performance modeling of gpgpus,"** in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[3] Z. Jin and J. S. Vetter, **"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)*, IEEE, 2023, pp. 325–327.