POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Low-overhead trace collection and profiling on GPU compute kernels

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

December 5th, 2024

Dorsal - Polytechnique Montréal

Introduction

- GPUs have become ubiquitous in many fields, notably HPC and machine learning
- Multiple programming models have been developed, both low and high level
  - CUDA, HIP, OpenCL
  - SYCL, OpenMP, OpenACC
- GPU programming remains a difficult task

Motivation

- Tooling is maturing, mostly for profiling from the host point of view
  - ROC-profiler
  - Intel VTune
  - HPCToolkit [1], ...
- Most tools rely on hardware performance counters and/or PC sampling
- Current work on device instrumentation
- Little consideration for instrumentation noise (runtime overhead, register pressure, ...)

1. K. Zhou, L. Adhianto, J. Anderson et al., "Measurement and analysis of GPU-accelerated applications with HPCToolkit", *Parallel Computing*, t. 108, p. 102 837, 2021.

Shortcomings of current work

- CUDAAdvisor [2] proposes LLVM-based instrumentation of compute kernels. PPT-GPU [3] is similar, with dynamic instrumentation.
  - little consideration for overhead (costly kernel-wide atomic operations)
  - Overhead ranging from $\sim 10\times$ to $120\times$
- CUDA Flux [4] introduces Control-Flow Graph (CFG) instrumentation combined with static analysis
  - only one thread is instrumented, does not support divergence
  - Overhead ranging from $\sim 1\times$ to $151\times$ (avg. $13.2\times$)

---

2. D. Shen, S. L. Song, A. Li et al., "CUDAAdvisor: LLVM-Based Runtime Profiling for Modern GPUs", in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018.
3. 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, p. 1-15.
4. L. Braun et H. Fröning, "CUDA Flux: A Lightweight Instruction Profiler for CUDA Applications", in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, 2019.

Baseline method

We propose a method for instrumenting kernel execution on the GPU with a minimal runtime overhead.

- Relies on a set of LLVM passes for the host and device Intermediate Representation (IR)
- Multi-stage performance analysis
  - Control-flow counters to retrieve the control flow of the program
  - Event collection for precise analysis
  - Optionally, original kernel for timing data
- Knowledge of the control flow allows for pre-allocation of the buffers
- Deterministic execution is ensured by reverting memory
- Article published in the ACM Transactions on Parallel Computing [5]

5. S. Darche et M. R. Dagenais, "Low-Overhead Trace Collection and Profiling on GPU Compute Kernels", *ACM Trans. Parallel Comput.*, fév. 2024, Just Accepted, issn : 2329-4949. doi : 10.1145/3649510. adresse : https://doi.org/10.1145/3649510.

Baseline Results

- Instrumentation tested on the Rodinia [6] benchmark

	Average overhead	Median overhead
Counters instr. (kernel)	2.00×	1.67×
Tracing instr. (kernel)	1.50×	1.29×
Program execution time	1.60×	1.26×

- Good improvements over state of the art
- Correlation between kernel complexity and overhead

6. S. Che, M. Boyer, J. Meng et al., "Rodinia: A benchmark suite for heterogeneous computing", in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, p. 44-54.

Runtime Trace Collection

- First approach works well, but is unweildy in many ways
  - Two kernel runs require saving context & input data
  - Limited by non-deterministic kernels (parallelism ?)
- "Regular" tracing is possible but has its own set of challenges
- Requires specific tuning for the hardware
  - Memory locality
  - Allocation granularity
- Many GPU allocation algorithms to explore !

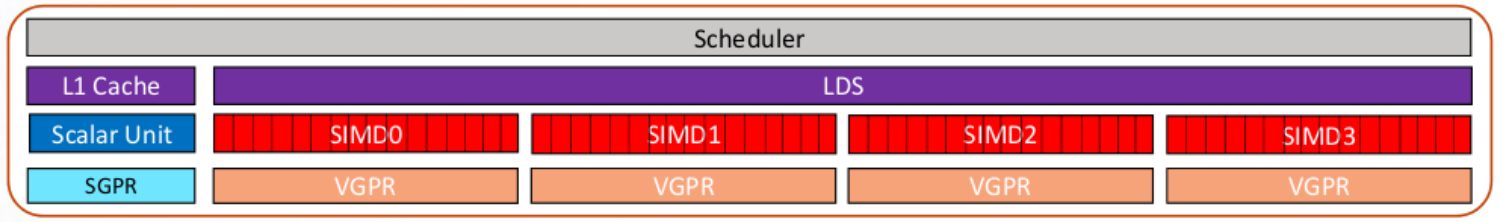


Figure 1 – AMD GCN Compute unit [7]

7. Reproduced from *AMD GPU Hardware Basics*, 2019 Frontier Application Readiness Kick-off Workshop

Challenging Scale

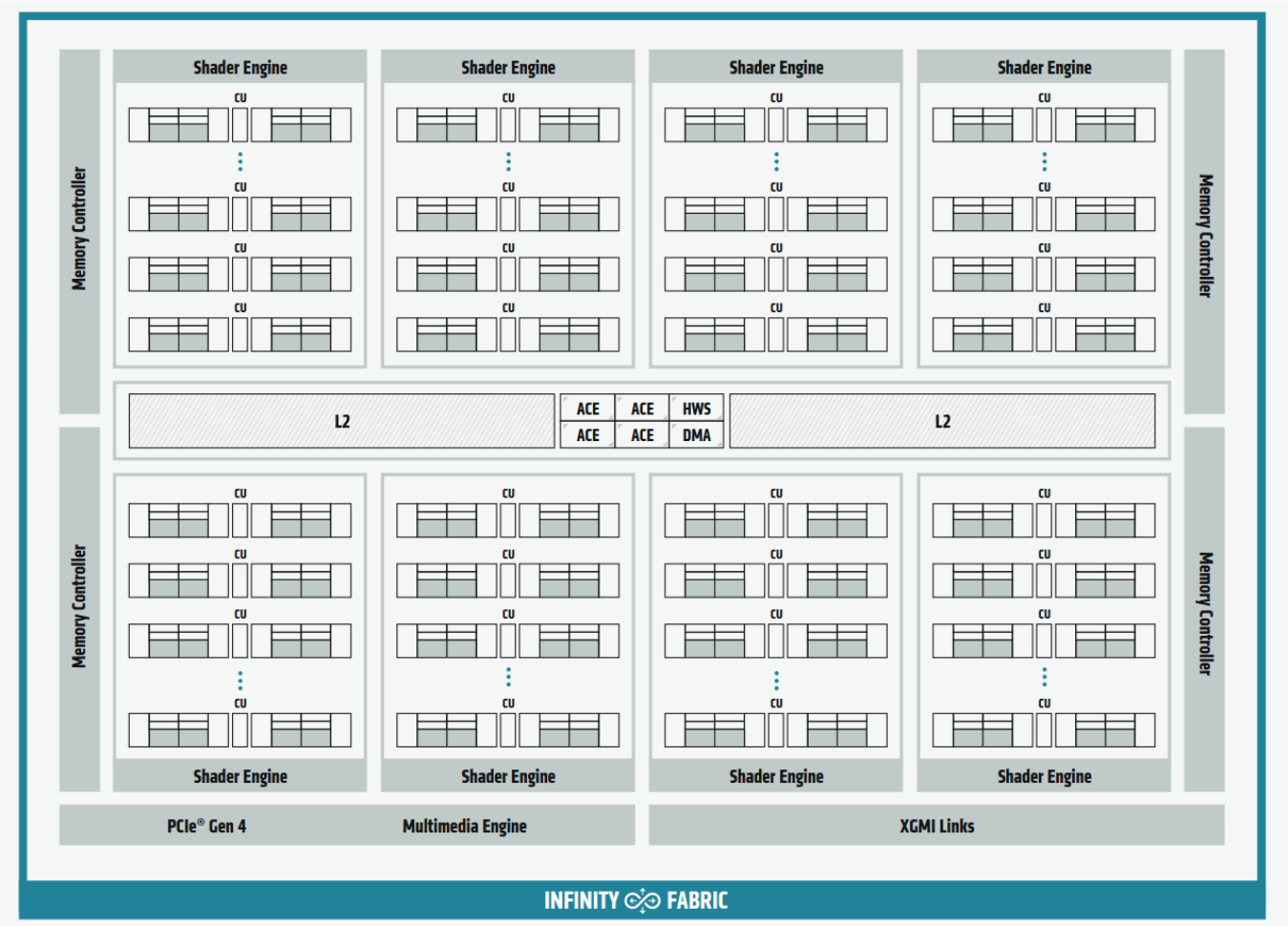


Figure 2 – AMD CDNA1 Architecture block diagram [8]

8. Reproduced from *Introducing AMD CDNA Architecture*, 2020 AMD Whitepaper

Implementation, Results

- Multiple approaches implemented :
  - Single shared buffer in global device memory – no resizing possible, enqueuing single events relies on heavy use of atomics
  - Shared buffer, per CU for improved memory locality
  - Global circular buffer but allocated by fixed-size chunks
  - Per CU circular buffer, fixed-size allocations
- All with handwritten (GPU) assembly !
- Interesting results, article submitted

Data structures

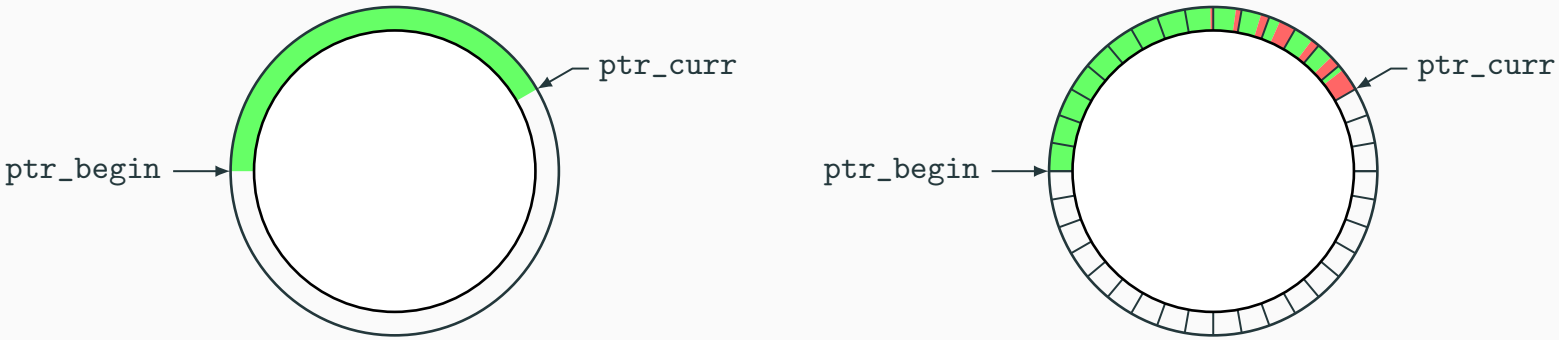


Figure 3 – `hip-global-mem` and `hip-chunk-allocator` data stuctures representation. Chunk allocation reduces the number of interactions with other producers

Performance

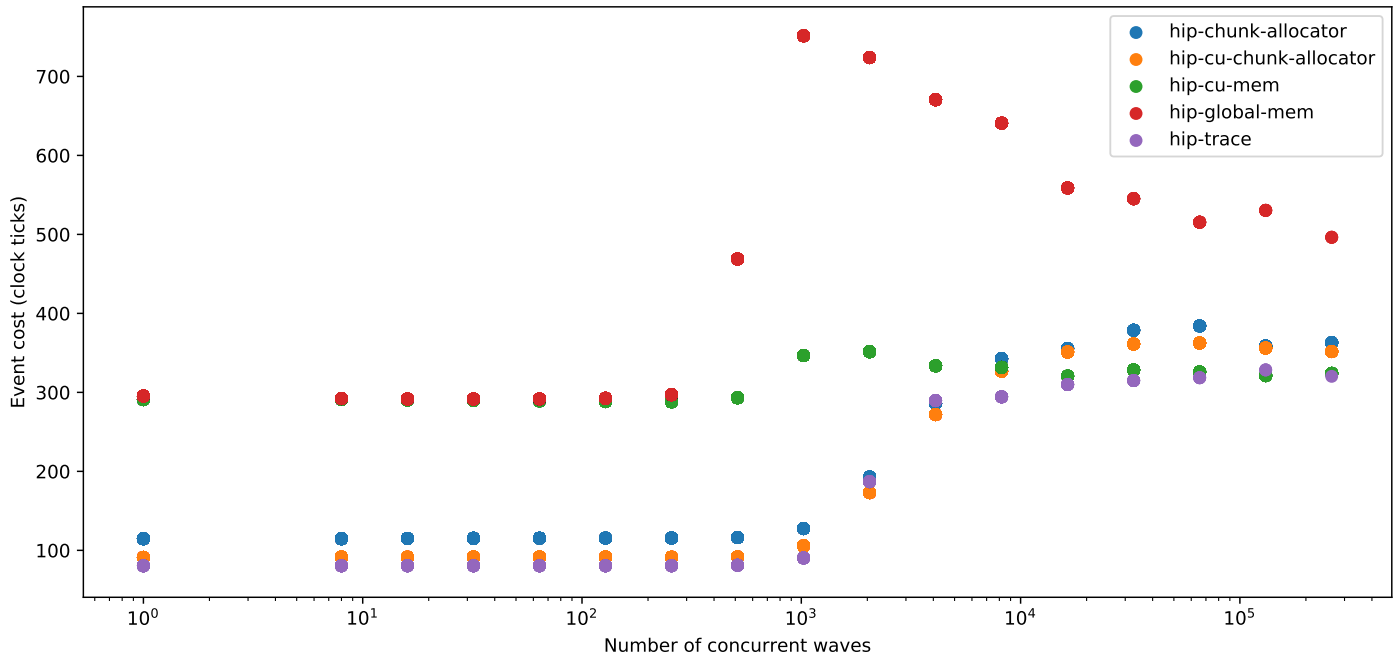


Figure 4 – Event cost (time per event) on a microbenchmark. Performance is highly dependent on the number of concurrent wavefronts

Results

- Instrumentation tested on the HeCBench [9] 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×

9. Z. Jin et 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, p. 325-327.

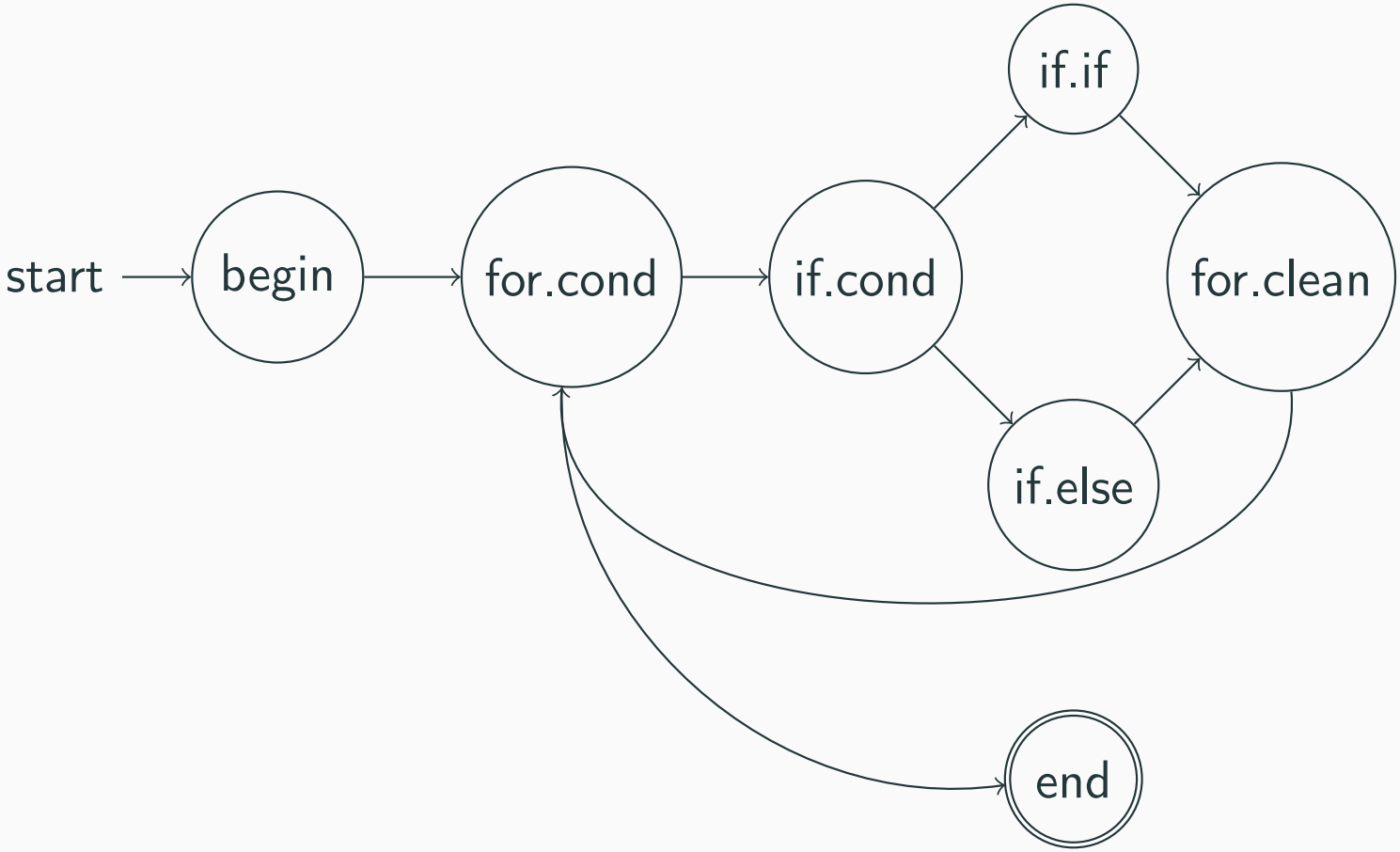
Hardware limitations

- Memory synchronization between host and device is only allowed at kernel boundaries (officially)
  - It could work consistently for a page, but scalar cache would have to be flushed from the kernel (costly)
- Synchronized memory (PCIe atomics) is slow, but could be used to mark a buffer as complete
- APU shared memory is promising (MI300A)
- Event output is still very high

Future work

- In the process of migrating integration passes to the compiler backend to reduce optimization (and register allocation) interference
- A lot of tracing data is redundant - this could be improved through better static analysis
- Intrinsics to allow the programmer to insert custom tracepoints

Scalar vs Vector CFG



Conclusion and future work

- Good results for online tracing
- Picking up interest from partners
- Currently exploring better integration within the compiler and improving static analysis
- Available freely on Github, feedback and/or use cases are more than welcome