Experiences with Top-down Performance Analysis of GPUs

John Mellor-Crummey

Scalable Tools Workshop
July 27, 2026
Rice University

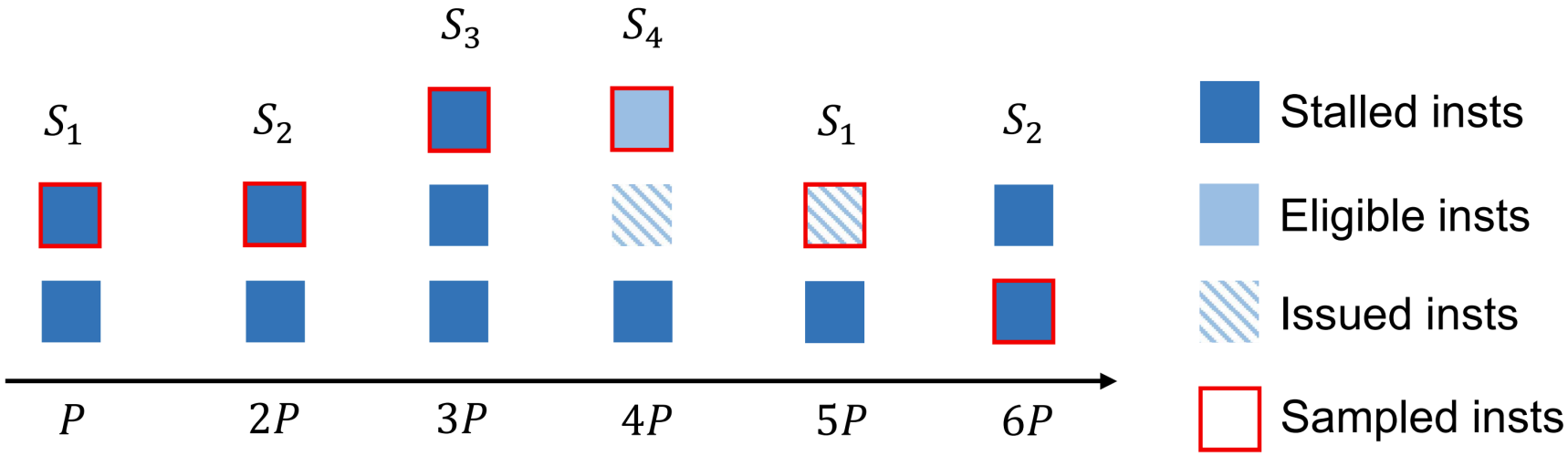
RICE

PC Sampling on GPU Architectures

- GPUs employ a set of processing units for massive parallelism
  - NVIDIA: Streaming Multiprocessors (SM) [Blackwell: 160 SM]
  - AMD: Compute Units (CU) [MI300A: 228 CU]
  - Intel: Xe Core [Data Center Max (PVC): 128 Xe Core]
- Measure kernels at the instruction level as they execute
- Periodically sample an instruction on each SM/CU/Xe Core
  - record GPU instruction addresses as program executes
  - map instructions back to source code using post-mortem analysis

PC Sampling on an NVIDIA GPU's SM

- Every P cycles, one warp scheduler of each SM samples next instruction of each active warp
- Sampling rotates round robin across each of the four warp schedulers



Keren Zhou, Mark W. Krentel, and John Mellor-Crummey. ICS '20. ACM, New York, NY, USA, Article 26, 1–12. https://doi.org/10.1145/3392717.3392752

Common GPU Instruction Stall Reasons

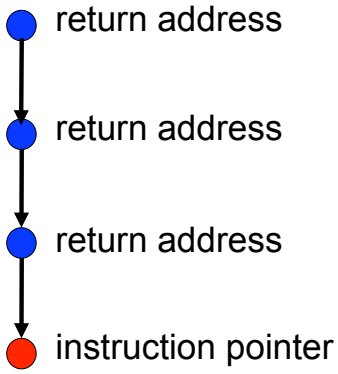
- Memory: waiting to issue requests or the arrival of values
- Branch: waiting for branch target to be computed
- Instruction fetch: waiting for instruction fetch at a target address to complete
- Synchronization: waiting for sibling threads at barrier
- Memory throttle: waiting for L/S queue entry to become available
- Pipeline busy: waiting for an oversubscribed pipeline to become available
- Not selected: available to execute, but not selected
- Sleeping

- GPUs use a high degree of multithreading to hide stalls
- An instruction stall is uninteresting unless execution stalls

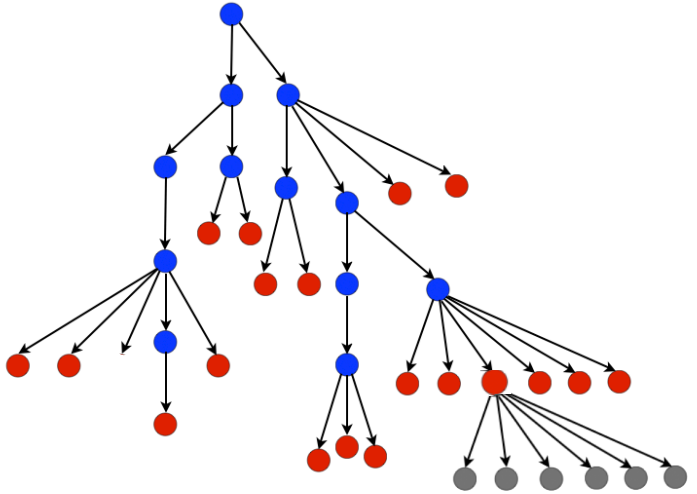
Call Stack Unwinding to Attribute Costs in Context

- Unwind when timer or hardware counter overflows
  - measurement cost ∝ to sampling frequency rather than call frequency
- Unwind to capture context for GPU kernel launches

Call path sample



Calling context tree



GPU PC samples

Information in AMD MI300A PC Samples - Part 1

- Where was the sample collected?
  - (chiplet, compute unit, SIMD unit) (are all chipsets and compute units being used?)
- How many wavefront slots are full?
  - waves_active (1..32) (occupancy, for hiding latency)
- How many threads were active?
  - execution mask (64 bits) (is the program exploiting vector/SIMD parallelism)
- What instruction was sampled
  - machine code address
- What kinds of instruction was it?
  - scalar, vector, dual vector, matrix, memory operation
- Did the instruction stall? Why?

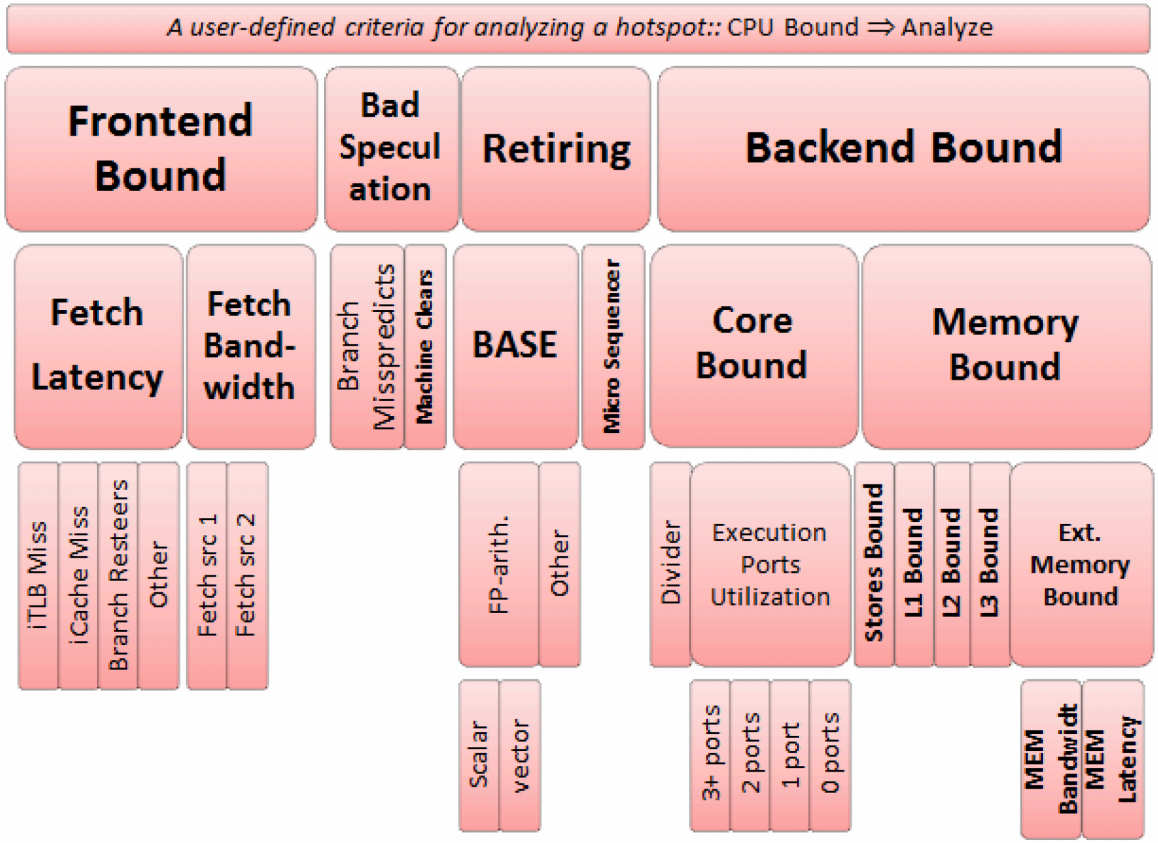
Information in AMD MI300A PC Samples - Part 2

Pipeline information

- What pipelines are active?
  - –matrix, vector, scalar, flat, local data store, texture, branch/msg, …

- For each pipeline
  - –was an issue attempted?
  - –was the issue successful or did it stall because of a lack of resources?
  - –was the pipeline idle?

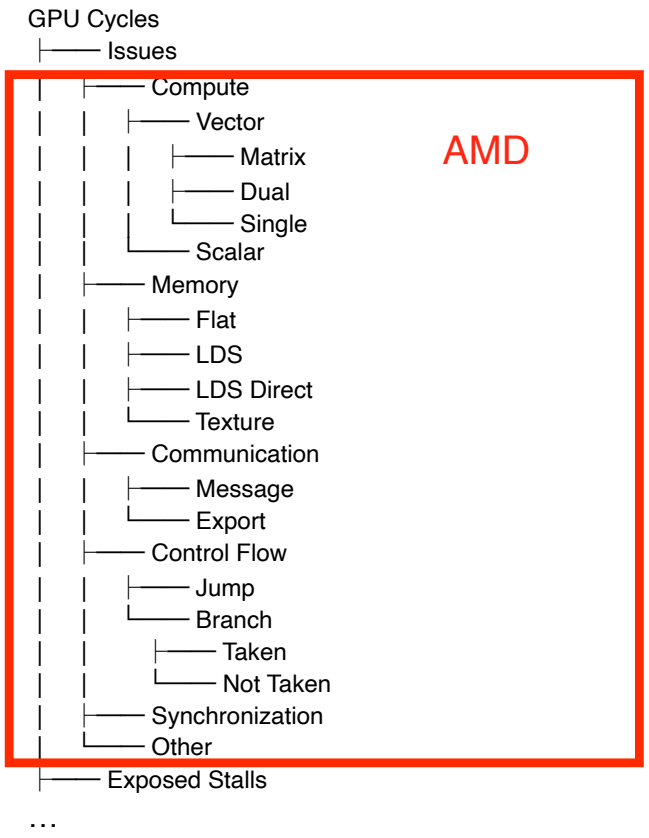
Inspiration from Intel's "Top-down" Performance Model for Sapphire Rapids

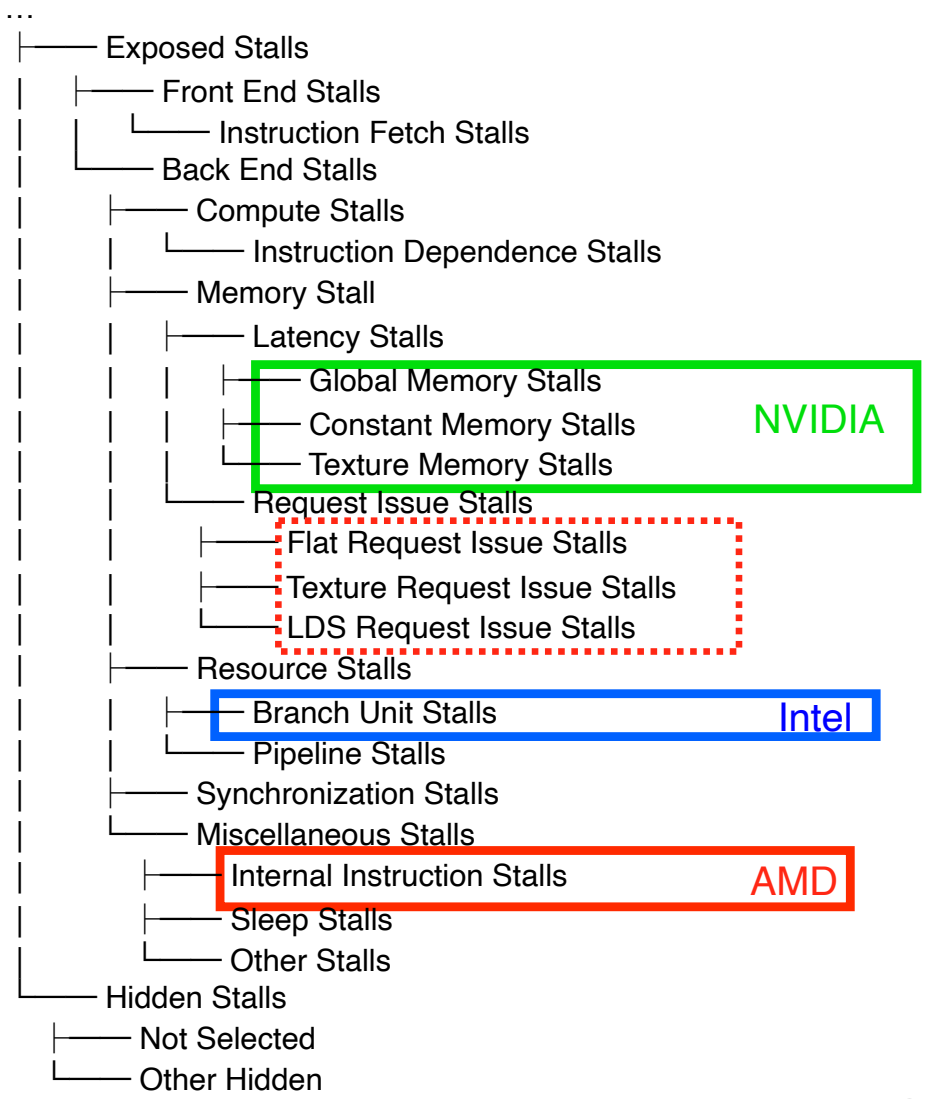


A. Yasin. A Top-Down method for performance analysis and counters architecture. ISPASS 2014, Monterey, CA, USA, 2014, pp. 35-44, doi: 10.1109/ISPASS.2014.6844459.

```
CPU (Total slots)
├── Active
│   ├── Retiring
│   │   ├── Light operations
│   │   │   ├── FP
│   │   │   │   ├── Scalar
│   │   │   │   ├── Vector
│   │   │   │   └── X87
│   │   │   ├── Int
│   │   │   ├── Memory
│   │   │   ├── Fused
│   │   │   ├── Non-fused
│   │   │   └── Others
│   │   └── Heavy operations
│   └── Bad speculation
│       ├── Misprediction
│       └── Machine clear
└── Stall
    ├── Front-end bound
    │   ├── Fetch latency
    │   └── Fetch bandwidth
    └── Back-end bound
        ├── Core bound
        │   ├── Divider
        │   ├── Serialization
        │   ├── AMX
        │   └── Port utilization
        └── Memory-bound
            ├── Store bound
            ├── L1
            ├── L2
            ├── L3
            └── DRAM
                ├── Bandwidth
                └── Latency
```

HPCToolkit's GPU Top Down Model





Request Issue Stalls on MI300A

- There are no such stall reasons
- HPCToolkit synthesizes memory request issue stalls on MI300A
- NOT SELECTED
  - originally interpreted as hidden stall (abundant instruction parallelism)
  - made no sense where we saw it!
- NOT SELECTED = sampled instruction didn't issue
  - if sampled instruction is a flat memory operation and
    - another flat memory operation
      - attempted to issue
      - stalled
  - no one is issuing flat memory operations!
- PIPE BUSY: sampled instruction attempted to issue but failed (e.g. flat memory pipe full)

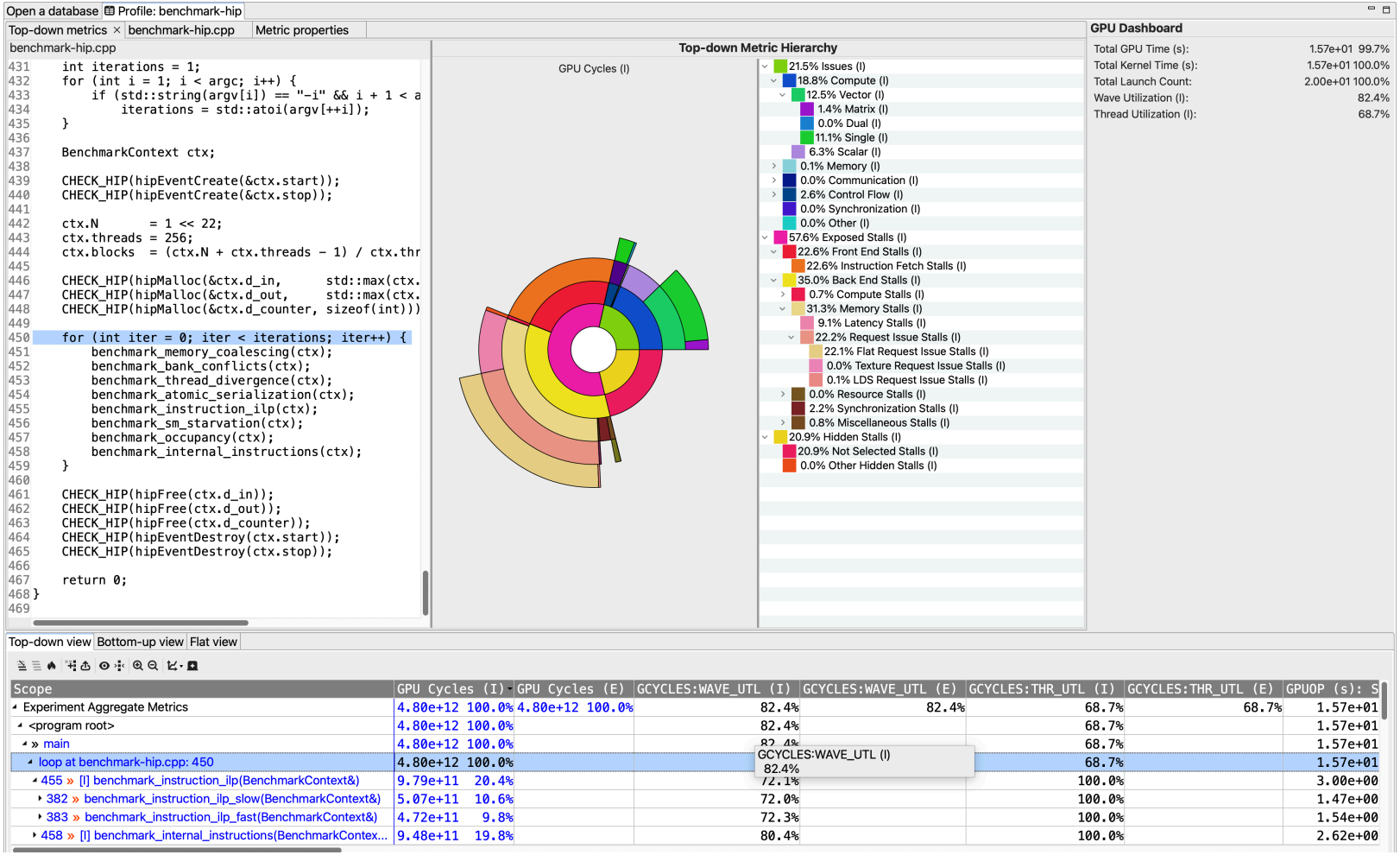
Request Issue Stalls on MI300A

```
if (iclass == GPU_INST_TYPE_FLAT) {
  if (stall == GPU_INST_STALL_PIPE_BUSY ||
      (stall == GPU_INST_STALL_NSEL && ISSUE_STALL(ga->details.pc_sampling.pipeline_info, flat))) {
    stall = GPU_INST_STALL_MEM_REQ_FLAT;
  }
} else if (iclass == GPU_INST_TYPE_TEXTURE) {
  if (stall == GPU_INST_STALL_PIPE_BUSY ||
      (stall == GPU_INST_STALL_NSEL && ISSUE_STALL(ga->details.pc_sampling.pipeline_info, texture))) {
    stall = GPU_INST_STALL_MEM_REQ_TEX;
  }
} else if (iclass == GPU_INST_TYPE_LDS) {
  if (stall == GPU_INST_STALL_PIPE_BUSY ||
      (stall == GPU_INST_STALL_NSEL && ISSUE_STALL(ga->details.pc_sampling.pipeline_info, lds))) {
    stall = GPU_INST_STALL_MEM_REQ_LDS;
  }
}
```

Our Implementation in HPCToolkit is not the Solution!

- HPCToolkit synthesizes several kinds of request issue stalls
- Ideally, other tools would explain stalls in the same style
- Implement the same code in every tool? Madness!
- Rocprofiler-sdk could post-process
  - *instruction type* (to identify pipeline)
  - *pipe busy*
  - *not selected*
  - *arbiter pipeline stall information*

HPCToolkit's GPU Top Down Model: AMD MI300A




```
431     int iterations = 1;
432     for (int i = 1; i < argc; i++) {
433         if (std::string(argv[i]) == "-i" && i + 1 < a
434             iterations = std::atoi(argv[++i]);
435     }
436
437     BenchmarkContext ctx;
438
439     CHECK_HIP(hipEventCreate(&ctx.start));
440     CHECK_HIP(hipEventCreate(&ctx.stop));
441
442     ctx.N       = 1 << 22;
443     ctx.threads = 256;
444     ctx.blocks  = (ctx.N + ctx.threads - 1) / ctx.thr
445
446     CHECK_HIP(hipMalloc(&ctx.d_in,      std::max(ctx.
447     CHECK_HIP(hipMalloc(&ctx.d_out,     std::max(ctx.
448     CHECK_HIP(hipMalloc(&ctx.d_counter, sizeof(int)))
449
450     for (int iter = 0; iter < iterations; iter++) {
451         benchmark_memory_coalescing(ctx);
452         benchmark_bank_conflicts(ctx);
453         benchmark_thread_divergence(ctx);
454         benchmark_atomic_serialization(ctx);
455         benchmark_instruction_ilp(ctx);
456         benchmark_sm_starvation(ctx);
457         benchmark_occupancy(ctx);
458         benchmark_internal_instructions(ctx);
459     }
460
461     CHECK_HIP(hipFree(ctx.d_in));
462     CHECK_HIP(hipFree(ctx.d_out));
463     CHECK_HIP(hipFree(ctx.d_counter));
464     CHECK_HIP(hipEventDestroy(ctx.start));
465     CHECK_HIP(hipEventDestroy(ctx.stop));
466
467     return 0;
468 }
469
```

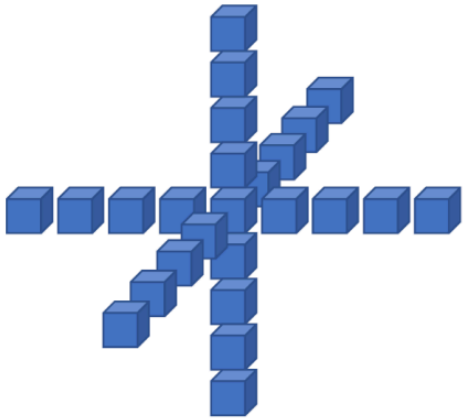
Scope	GPU Cycles (I)	GPU Cycles (E)	GCYCLES:WAVE_UTL (I)	GCYCLES:WAVE_UTL (E)	GCYCLES:THR_UTL (I)	GCYCLES:THR_UTL (E)	GPUOP (s): S
Experiment Aggregate Metrics	4.80e+12 100.0%	4.80e+12 100.0%	82.4%	82.4%	68.7%	68.7%	1.57e+01
<program root>	4.80e+12 100.0%		82.4%		68.7%		1.57e+01
» main	4.80e+12 100.0%		82.4%		68.7%		1.57e+01
loop at benchmark-hip.cpp: 450	4.80e+12 100.0%				68.7%		1.57e+01
455 » [I] benchmark_instruction_ilp(BenchmarkContext&)	9.79e+11 20.4%		72.1%		100.0%		3.00e+00
382 » benchmark_instruction_ilp_slow(BenchmarkContext&)	5.07e+11 10.6%		72.0%		100.0%		1.47e+00
383 » benchmark_instruction_ilp_fast(BenchmarkContext&)	4.72e+11 9.8%		72.3%		100.0%		1.54e+00
458 » [I] benchmark_internal_instructions(BenchmarkContex...	9.48e+11 19.8%		80.4%		100.0%		2.62e+00

Example: Minimod - A Code for Seismic Data Analysis

- Minimod is an OpenMP code that analyzes seismic data by applying a star-shaped 25-point stencil to a 3D array (as shown to the right)

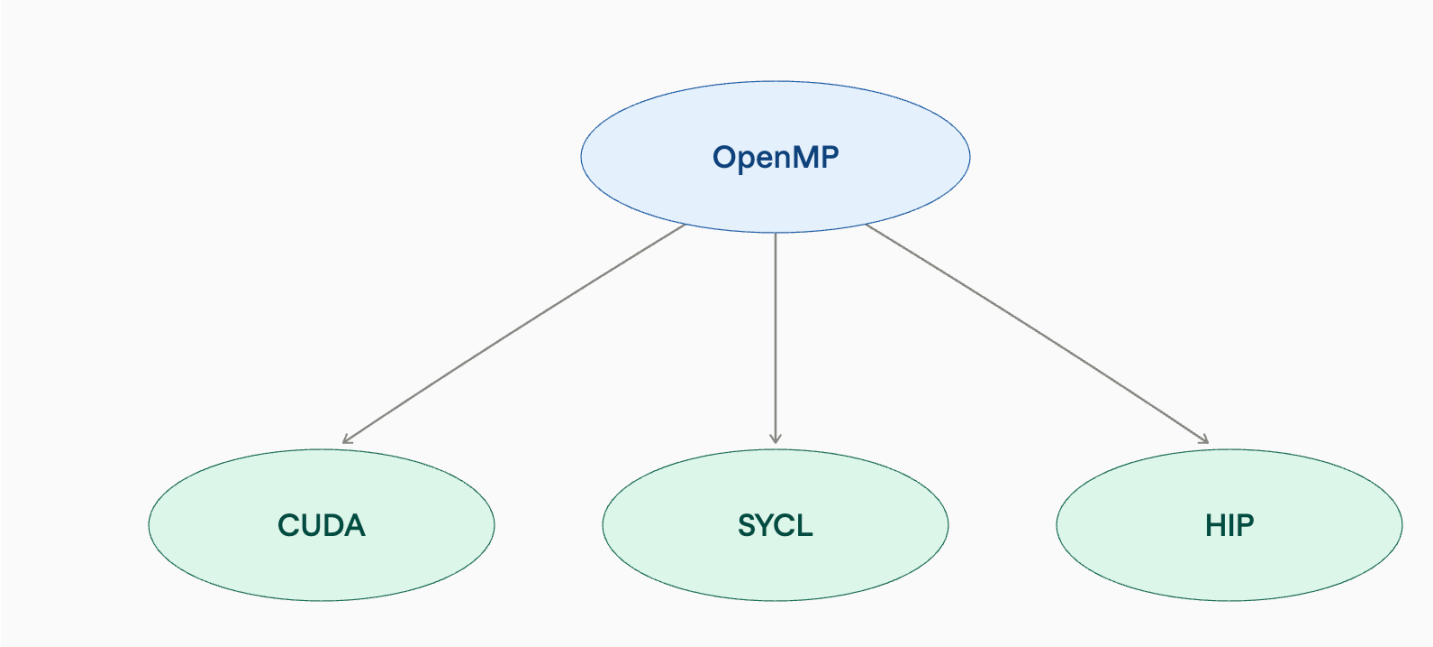
- The GPU code accesses the 3D data using the following indexing MACRO

```
#define IDX3_l(i,j,k) ((nz+2*lz)*(ny+2*ly)*((i)+lx) + \
                       (nz+2*lz)*((j)+ly) + ((k)+lz))
```



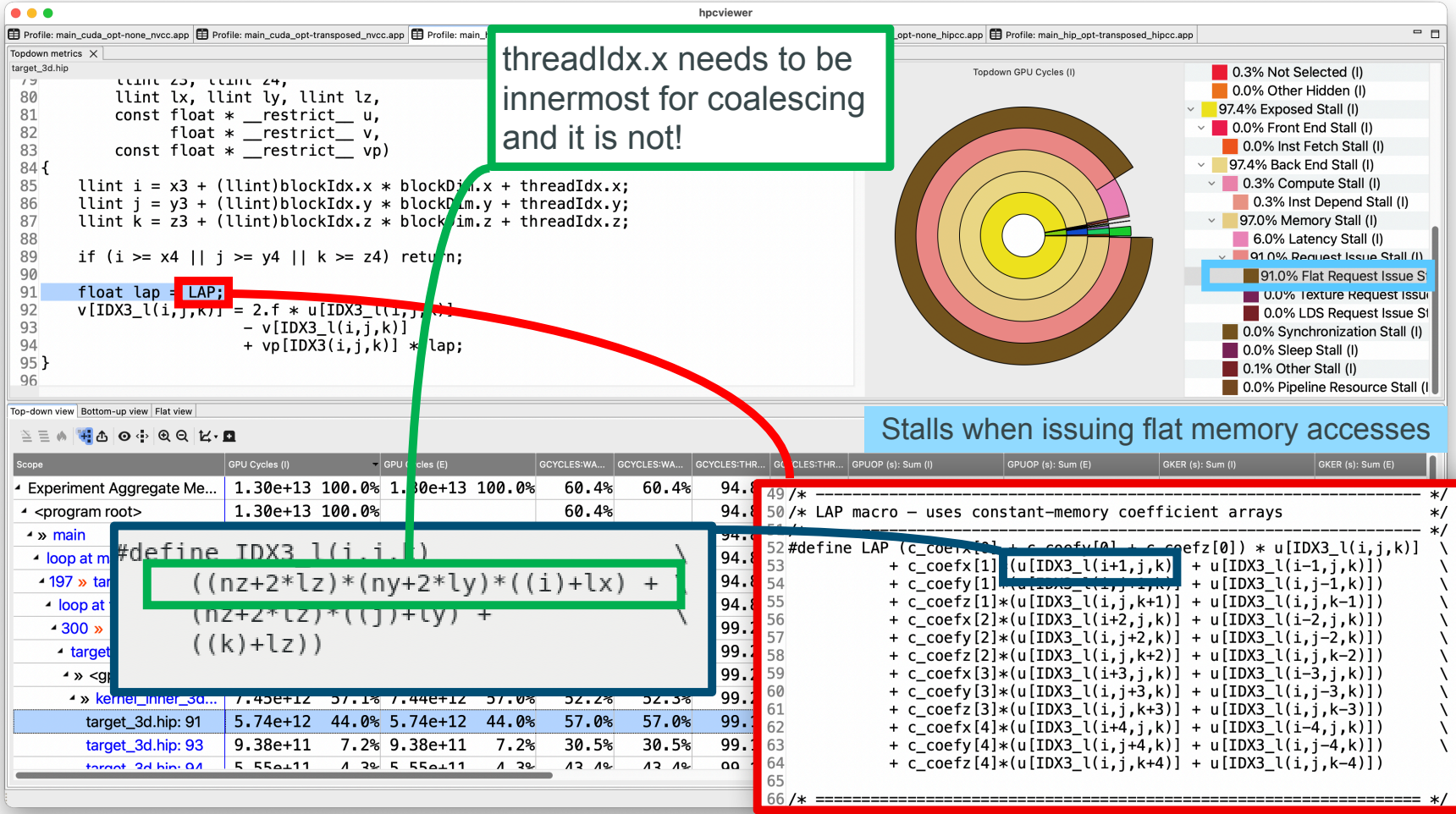
Automatically Generating and Tuning GPU Codes

- Claude Sonnet 4.6 mapped OpenMP implementation of Minimod to into CUDA, HIP, SYCL



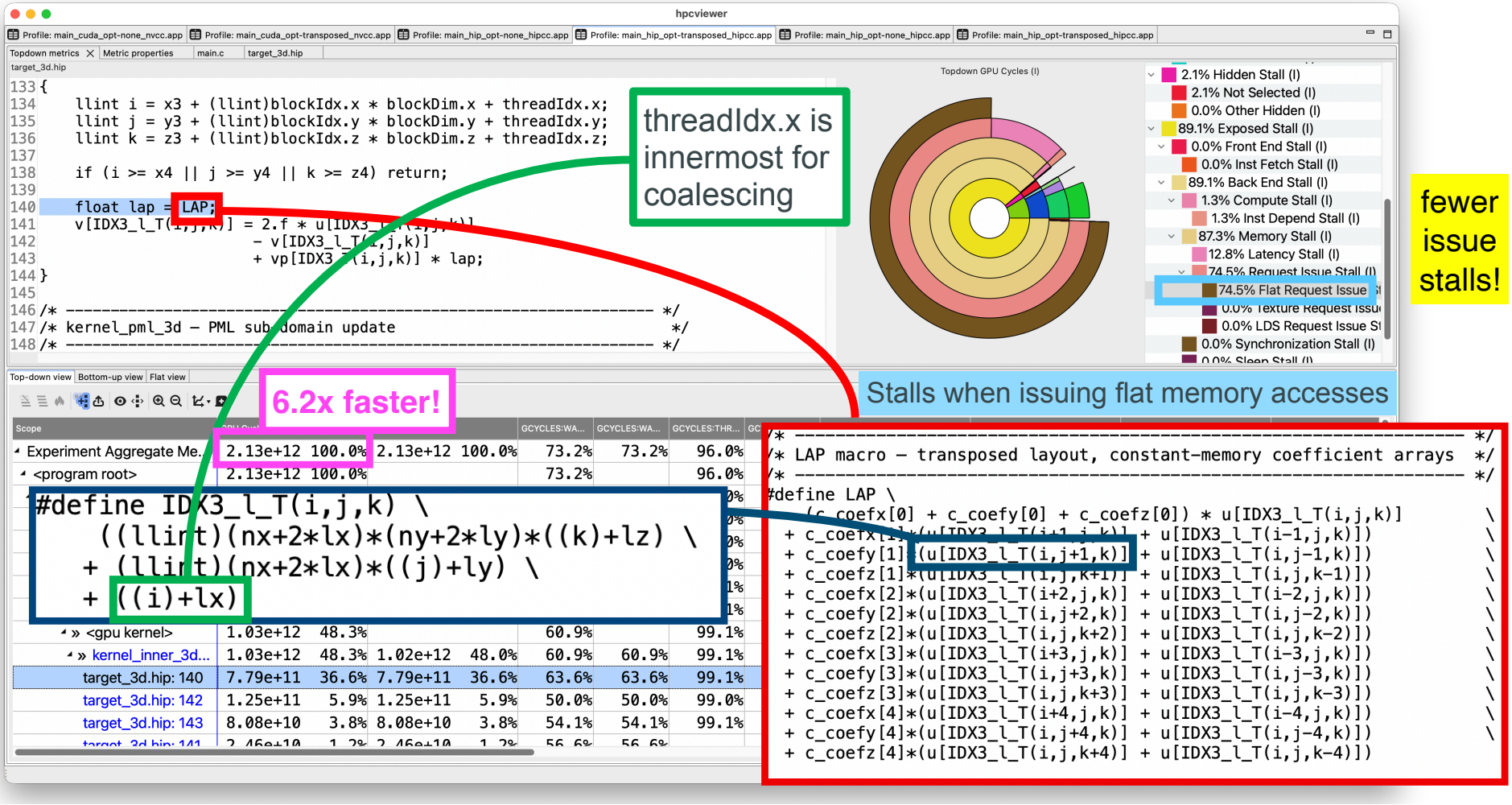
HPCToolkit Measurement of Minimod on AMD MI300A

Measure HIP version of minimod written by Claude Sonnet 4.6



HPCToolkit Measurement of Minimod Transposed on AMD MI300A

Measure HIP version of minimod written by Claude Sonnet 4.6



Attributing Stalls to Memory Accesses

- AMD waitcnt
  - we know that we are waiting for some memory access but not which
- Intel and NVIDIA
  - we know specifically what we are waiting for and can trace it backward through the control flow

Musings

- Want to know how much HW is running a kernel
  - Tried counting compute units generating samples on AMD MI300A
    - proof of concept code: identify chipsets and CUs running a kernel
      - why? report magnitude of HW running the kernel (a profile)
      - currently, the implementation reports a max of 23 CUs active (bug?)
  - NVIDIA and Intel don’t have measurement capabilities for this
- Can identify where threads are used poorly in waves with a waste calculation
- Want to measure idleness
  - On MI300A, only sample active CUs; can’t measure idle CUs
- Should we approximate idleness by
  - expected samples = ((kernel time in s)/(gpu clock period)) * (number of CUs)
  - idleness = expected samples - measured samples

Missing capabilities

- AMD provides correlation ids for PC samples from concurrent kernels
  - NVIDIA and Intel do not!
- Rather than guessing the root cause of issue request stalls, could GPUs provide a stall reason like “uncoalesced memory access”
- Want a GPU capability like Intel’s Load Latency Facility
  - tell a tool not just “waiting for an access to complete”
  - want information like
    - access completed from shared memory/LDS
    - access completed from L2
    - access completed from main memory