New Developments in the ROCm Tools Ecosystem

July 27, 2026
Scalable Tools Workshop

AMD
together we advance_

ROCm Systems Profiler

Application-level profiler for system-wide tracing and performance analysis

OVERVIEW

A system-wide tracing tool for heterogeneous application. It captures the entire application stack in a single unified timeline.

PAIN POINTS

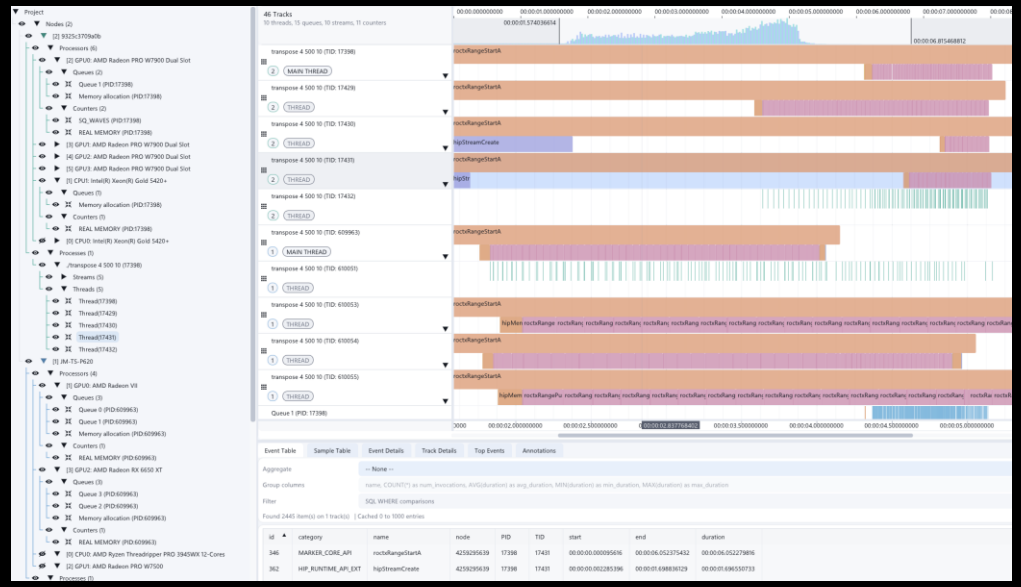
- Where is my application spending time?
- Which kernel is taking the most GPU time?

KEY FEATURES

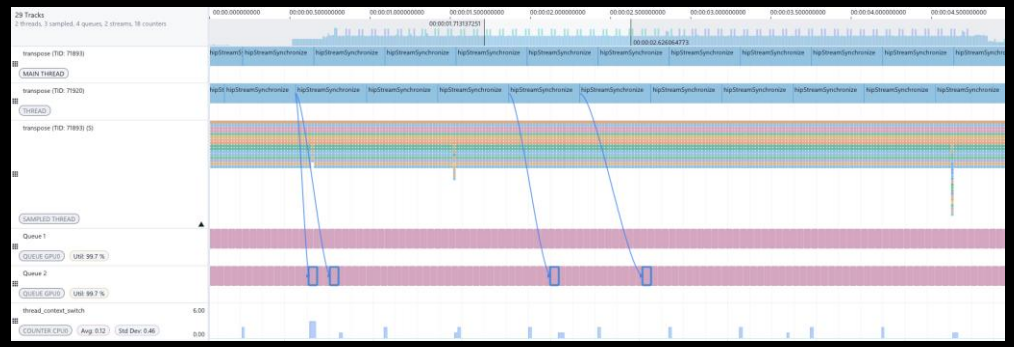
- Holistic view of CPU, GPU, Memory, Communication, I/O metrics
- CPU/GPU synchronization analysis
- Causal Profiling: predict optimization impact before code changes
- Dynamic instrumentation, statistical & process-level sampling
- GUI support via ROCm Optiq
- Support for multi-node environments

API SUPPORT

- Parallelism: HIP, HSA, Pthreads, Kokkos, OpenMP
- Multimedia: rocDecode, rocJPEG
- Communication: MPI, UCX, RCCL, OpenSHMEM



View a hierarchical representation of hardware and software components



Visualize CPU and GPU activities, events and performance metrics in a timeline

ROCm Systems Profiler Documentation

AMD
together we advance_

ROCm Compute Profiler

Kernel-level profiling for ML and HPC workloads on AMD GPUs

OVERVIEW

A deep kernel-level profiling tool that helps developers identify which kernels are slow, explains what causes the performance bottleneck at hardware level, and guide optimization decisions.

PAIN POINTS

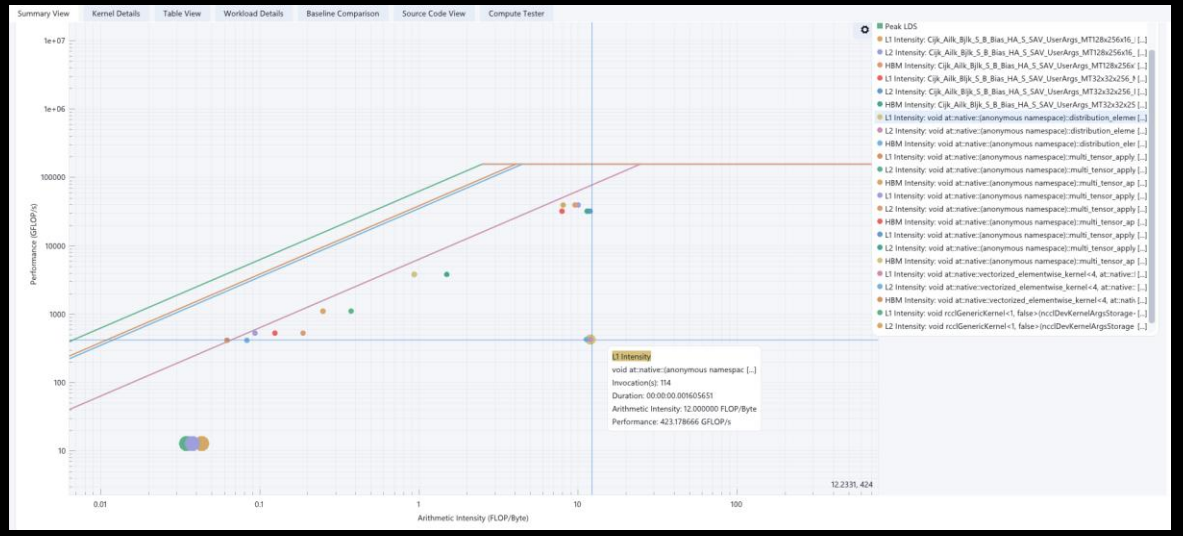
- Is this kernel compute- or memory-bound?
- Which hardware block is the bottleneck?
- How did my optimization improve GPU resource utilization?
- Did my optimization improve performance?

KEY FEATURES

- Roofline analysis
- Kernel level performance metrics
- PC Sampling
- System & Hardware block-level Speed-of-Light (SOL)
- In-depth memory analysis
- Baseline comparisons
- Support for multi-node / multi-rank
- GUI support via ROCm Optiq

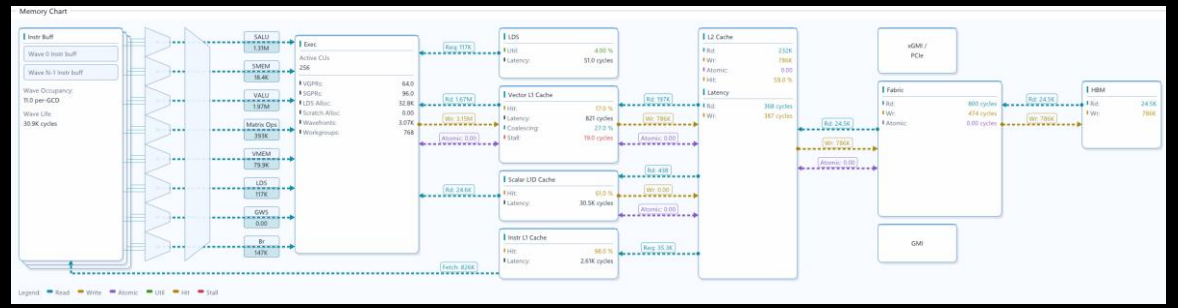
[ROCm Compute Profiler Documentation]

ROOFLINE ANALYSIS



The roofline chart plots kernel performance against empirical hardware ceilings.

MEMORY CHART



Visualization of memory transactions and throughput at each level of the memory sub-system

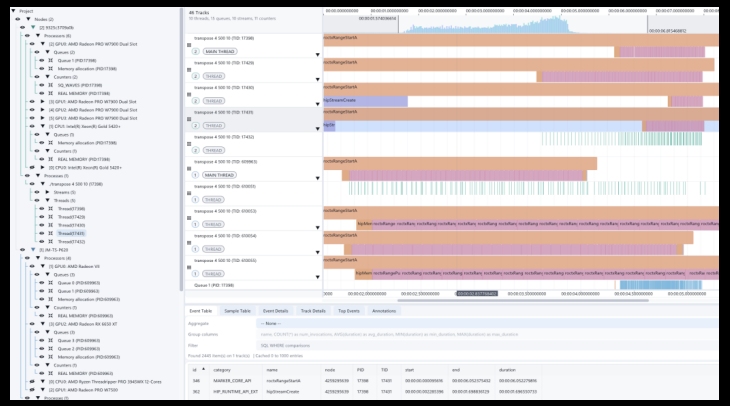
AMD
together we advance_

BETA

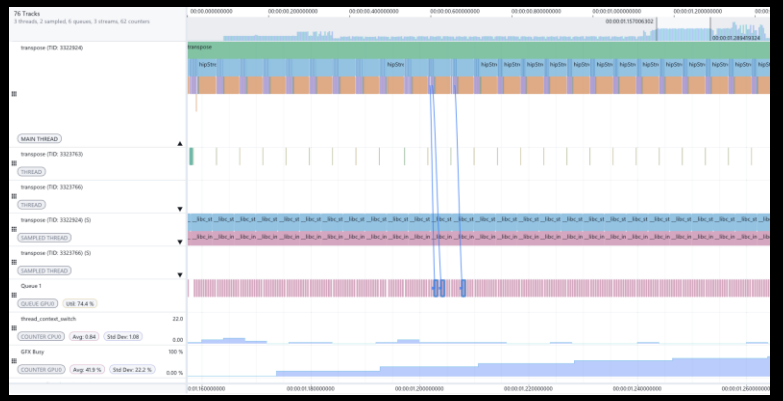
ROCm Optiq

Interactive visualization and analysis – a companion for ROCm Systems and Compute Profiler

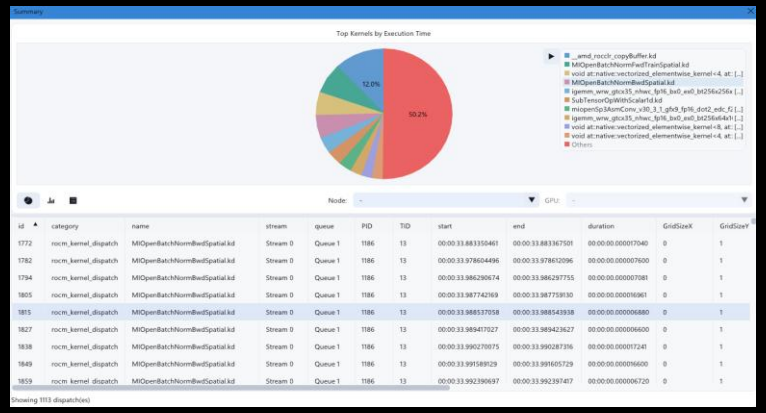
ROCm Systems Profiler



Hierarchical view of nodes, processes, GPU queues, memory operations and threads.

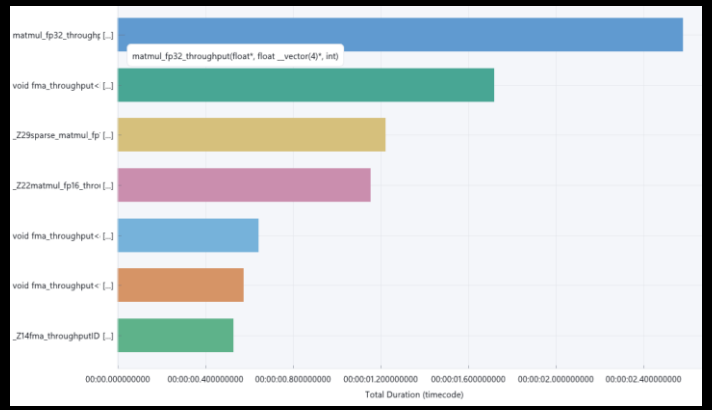


CPU/GPU events and performance counters over time.

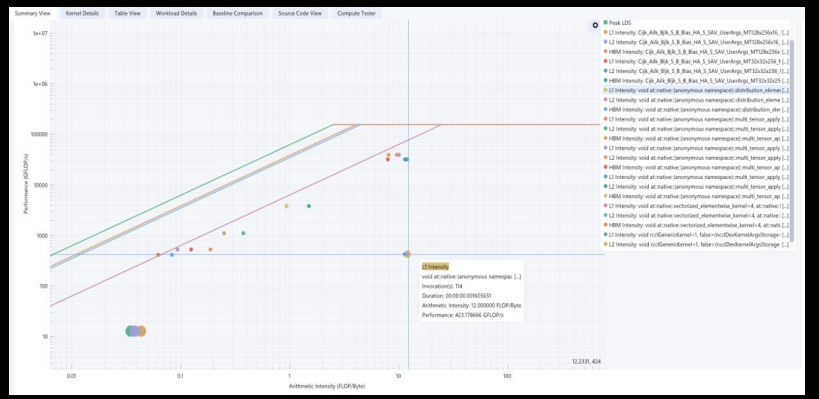


Top 10 kernels by execution time

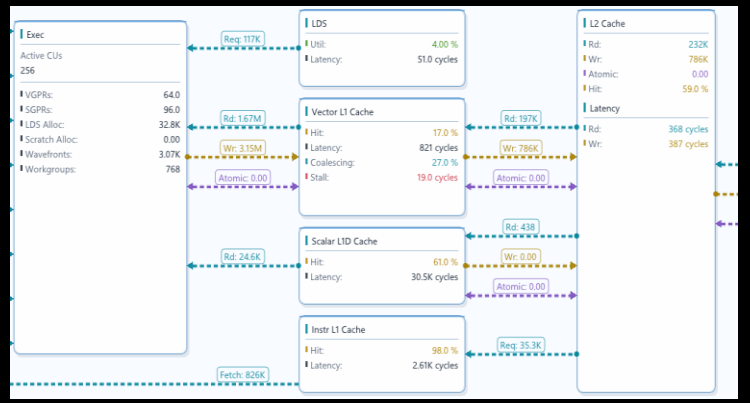
ROCm Compute Profiler



Summary view highlighting longest-running kernels.



Analyze roofline chart to identify if kernel is compute- or memory-bound.



Display memory transactions and throughput at each cache hierarchy level.

- Customizable interface for fast navigation
- Designed to handle large trace files
- Windows and Linux support (macOS coming soon)

ROCm Optiq Documentation

AMD
together we advance_

ROCprofiler-SDK

Foundational profiling infrastructure for GPU compute applications on the AMD GPUs

OVERVIEW

A C++ library that provides a tracing infrastructure that helps tool developers, framework integrators, and advanced users to build custom tools to trace GPU applications and collect hardware performance counters for AMD GPUs.

PAIN POINTS

- How can I add AMD GPU support to an existing profiling tool?

KEY FEATURES

- Powers higher-level ROCm and third-party profilers
  - ML Frameworks: PyTorch, JAX, TensorFlow, Triton, OpenXL
  - Community Tools: HPC Toolkit, TAU, PAPI, SCORE-P, Linaro, HPE Internal Profiling Tool
- Runtime-independent APIs for tracing HIP, HAS, RCCL, memory operations, and asynchronous GPU activities.
- Enables both high-level execution tracing and low-level hardware performance counter collection
- PC Sampling (software and hardware based) detects performance hotspots and bottlenecks in GPU kernels
- Advanced Thread Trace (ATT) captures fine-grained GPU thread execution details, essential for deep performance analysis and debugging.
- Dynamic attach/detach API enables profiling of running processes
- Profiles GPU workloads launched via OpenMP offloading in HPC and multi-threaded applications

ROCprofiler-SDK Documentation

What’s New in ROCprof SDK

- Use ROCprof Trace Decoder to convert SQTT data into actionable GPU performance insights – now available as the open-source rocprof-trace-decoder library.
- Attach to and detach from running processes using rocattach.so library - now available as public API.
- Start profiling late with rocprofiler_force_configure() to profile already-initialized HSA and HIP runtimes, without requiring application restart. Useful for applications that dynamically load tools at runtime, use plug-in architectures where ROCprofiler-SDK is loaded after GPU initialization, or need to attach to already running workload.
- PC Sampling (support in MI4xx), Frameworks support (Pytorch, JAX, Triton), MI4xx counters
- Streaming Performance Monitor (SPM) samples counters at (high frequency) regular intervals.

Rocprof-V3

Lightweight command-line profiling tool built on ROCProfiler-SDK

OVERVIEW

A command-line profiling tool for accessing rocprofiler-sdk capabilities. It provides a reference implementation and a template for custom tools that will be built on ROCProfiler-SDK. It is also useful for developers who need a lightweight profiler with minimal overhead.

PAIN POINTS

- Which instruction is causing the stall? What are the reasons for the stall?
- I just need a quick trace or counter dump.
- What hardware counters are available?

KEY FEATURES

- PC Sampling and Advanced Thread Trace (SQTT) deliver deep GPU kernel performance insights
- Trace generation per MPI process for distributed applications
- ROCTx annotations support enables custom region marking for enhanced trace readability
- Profiles single-process, multi-process, containerized, and distributed workloads
- Dynamically attaches to running applications via process ID
- rocpd (database) output enables scriptable trace analysis and merging

ROCprof-V3 Documentation

ROCprof Compute Viewer (RCV)

Visualization and analysis tool designed for GPU thread trace data

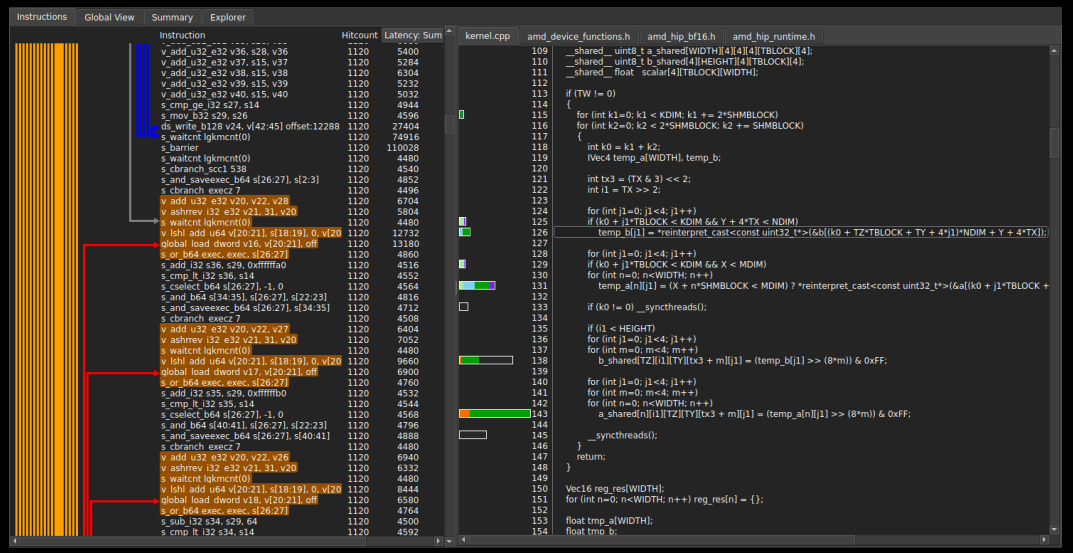
Pain Points:

- What are the performance hotspots in my GPU kernels at the source and assembly level?
- Which specific instructions and wavefronts are causing performance bottlenecks?
- What is the hardware utilization over time?

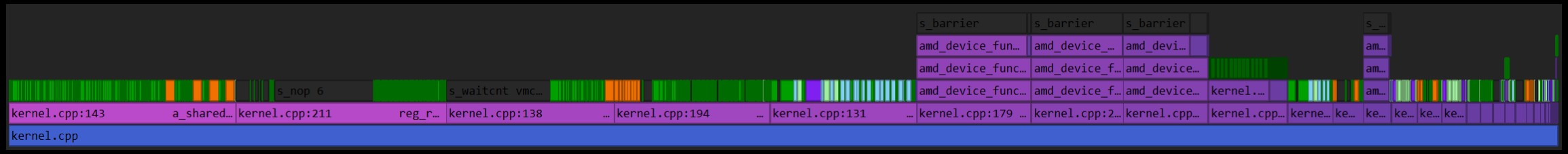
Key Features:

- Instructions Visualization: Displays ISA and source code (python/c++/fortran) latency hotspots, as well as the source<->ISA correspondence.
- Compute Unit and Utilization: Displays individual execution timing and latency per wave and hardware pipe, and the correspondence to the ISA.
- Occupancy and Kernel plots: Visualizes occupancy per SE and Kernel ID.

ROCprof Compute Viewer Documentation



Instruction view is best suited for quickly finding hotspot in assembly form.



Flamegraph suitable for general hotspots finding in source form. Select widths using total or non-hidden latency.

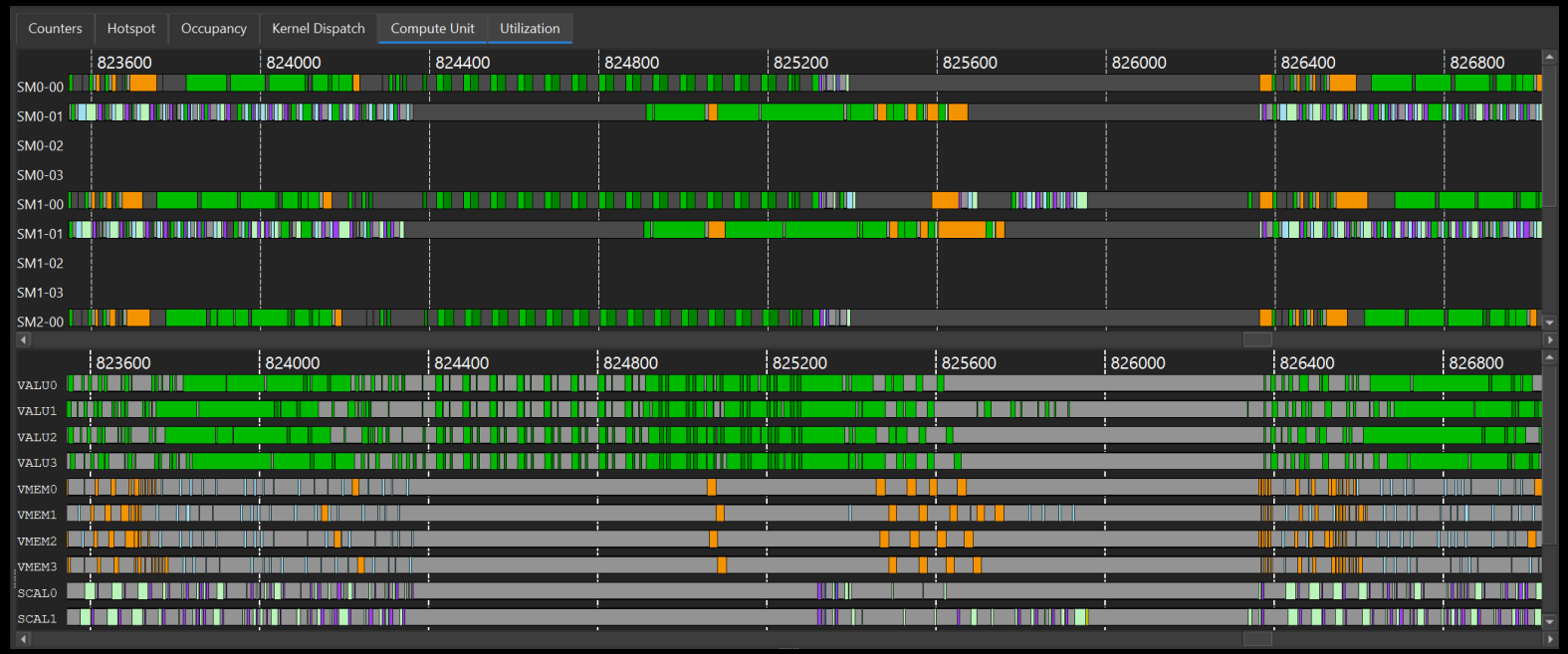
AMD
together we advance_

ROCprof Compute Viewer – The low level information

Cycle-by-cycle view of instruction latency per wave and per hardware pipe

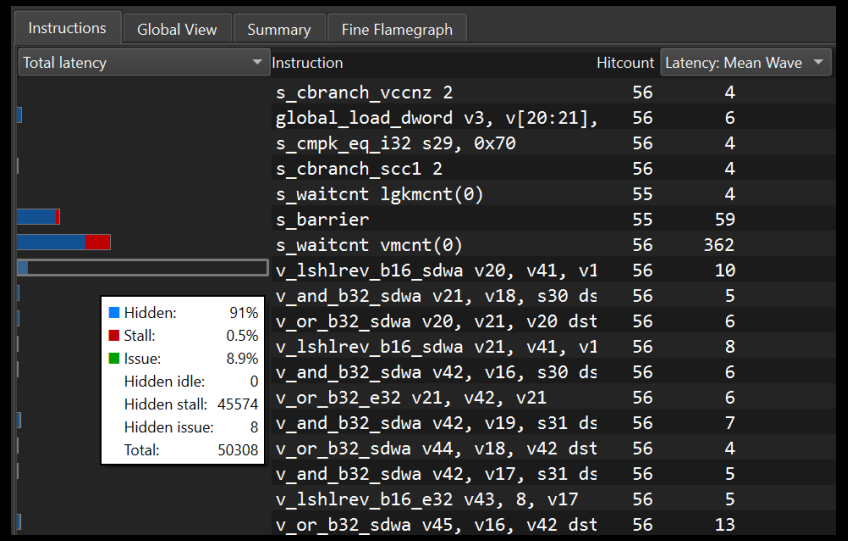
Highlight: Capable of answering questions beyond “where I’m stalling”:

- When am I stalling?
- When are my hardware pipes idle?
- What are other waves doing the moment I’m stalling?
- What is hiding the latency of my instructions, if anything?



Cycle-by-cycle view of waves and hardware pipes: Suited for high efficiency kernels

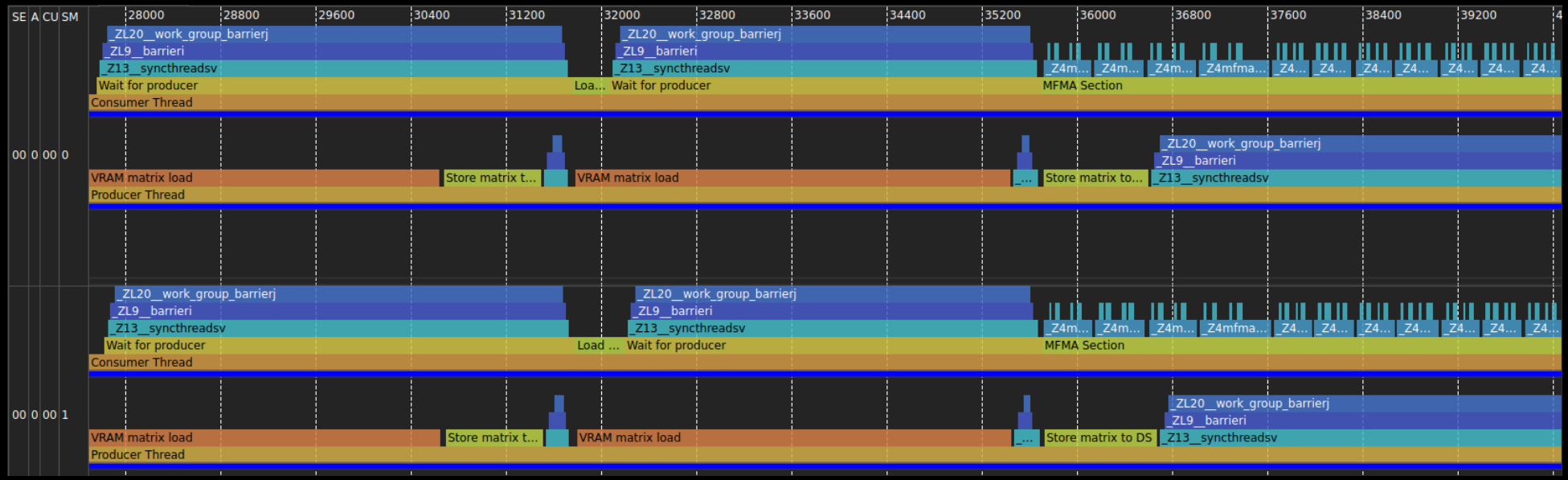
Hidden Latency Documentation



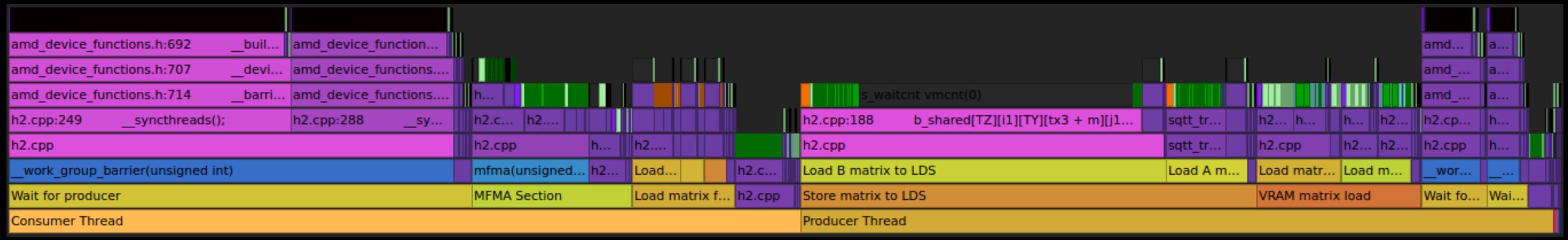
Experimental: Approximate hidden latency derived from utilization

ROCprof Compute Viewer – Trace with sqtt markers

- **Almost** cycle-accurate view of (potentially) all waves on the GPU.
- Environment variables to decide filters for:
  - CU / WGP
  - SIMD
  - WAVE_ID (Slot)
  - WorkgroupID



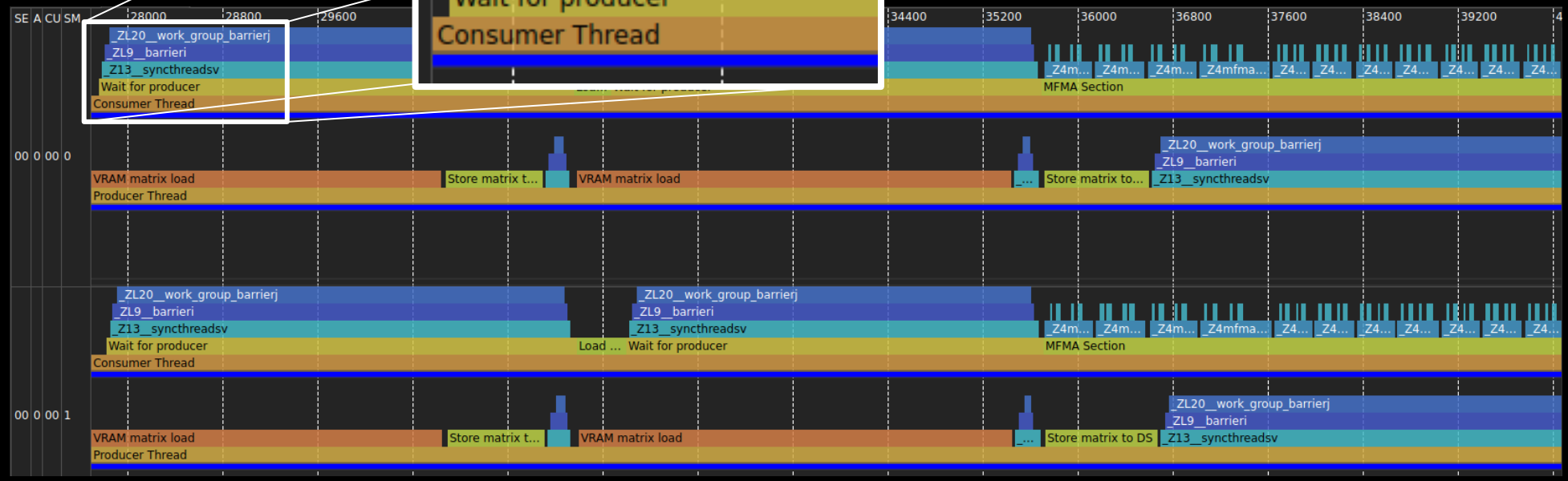
Global view: Per-wave trace seen with sqtt markers



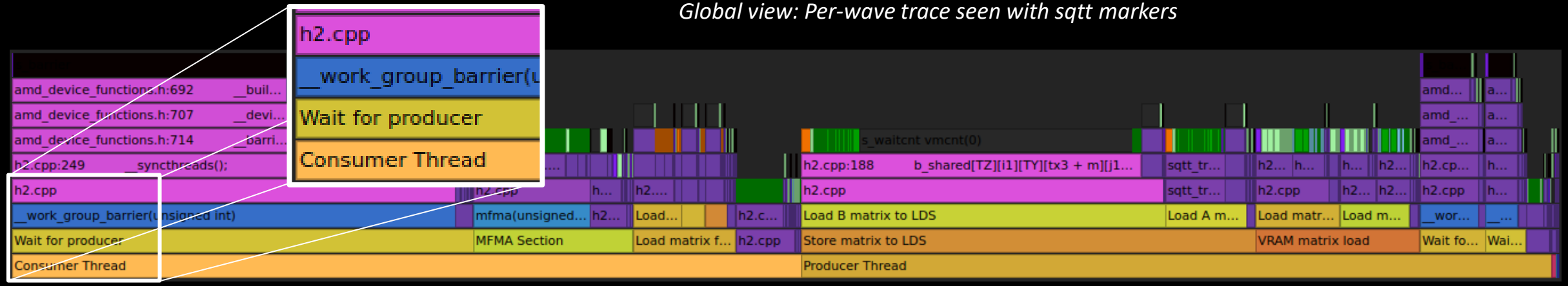
Fine flamegraph: seen with sqtt markers. Also: Coarse flamegraph for all waves.

ROCprof Compute Viewer … t markers

- **Almost** cycle-accurate view of (potentially) all waves on the GPU.
- Environment variables to decide filters for:
  - CU / WGP
  - SIMD
  - WAVE_ID (Slot)
  - WorkgroupID



Global view: Per-wave trace seen with sqtt markers



Fine flamegraph: seen with sqtt markers. Also: Coarse flamegraph for all waves.

ROCprof Compute Viewer (RCV) – User markers in device code

- Automatically generated at function entry/exit by the compiler.
- Hand-inserted user markers (similar to roctx for GPUs).
- Automatic annotation of global, buffer and flat operations.
- Or all the above combined.
- Compiled out in normal builds for zero overhead in production/release.
- Low overhead (only when used sparsely).

```
sqtt_marker_enter("Consumer Thread");

// ...

for (int k1=0; k1 < KDIM; k1 += 2*SHMBLOCK)
{
    // ...
    for (int k2=0; k2 < 2*SHMBLOCK; k2 += SHMBLOCK)
    {
        sqtt_marker_enter("Wait for producer");
        __syncthreads();
        sqtt_marker_exit("Wait for producer");

        sqtt_marker_enter("Load matrix from DS");
        // ... load from shared memory into registers ...
        sqtt_marker_exit("Load matrix from DS");

        sqtt_marker_enter("MFMA Section");
        for (int n=0; n<HEIGHT; n++) for (int r1=0; r1<WIDTH; r1++)
        {
            Vec4 res = mfma(a0_load[r1], a1_load[r1], b0_load[n], b1_load[n]);
            for (int m=0; m<4; m++) reg_res[r1][m*HEIGHT + n] += scal[r1][m] * res[m];
        }
        sqtt_marker_exit("MFMA Section");
    }
}

sqtt_marker_exit("Consumer Thread");
```

SQTT Instrumentation Documentation

Performance analysis: the known unknowns

What tools do I use for performance analysis?

Which tool do I use first?

What parameters do I pass the tool?

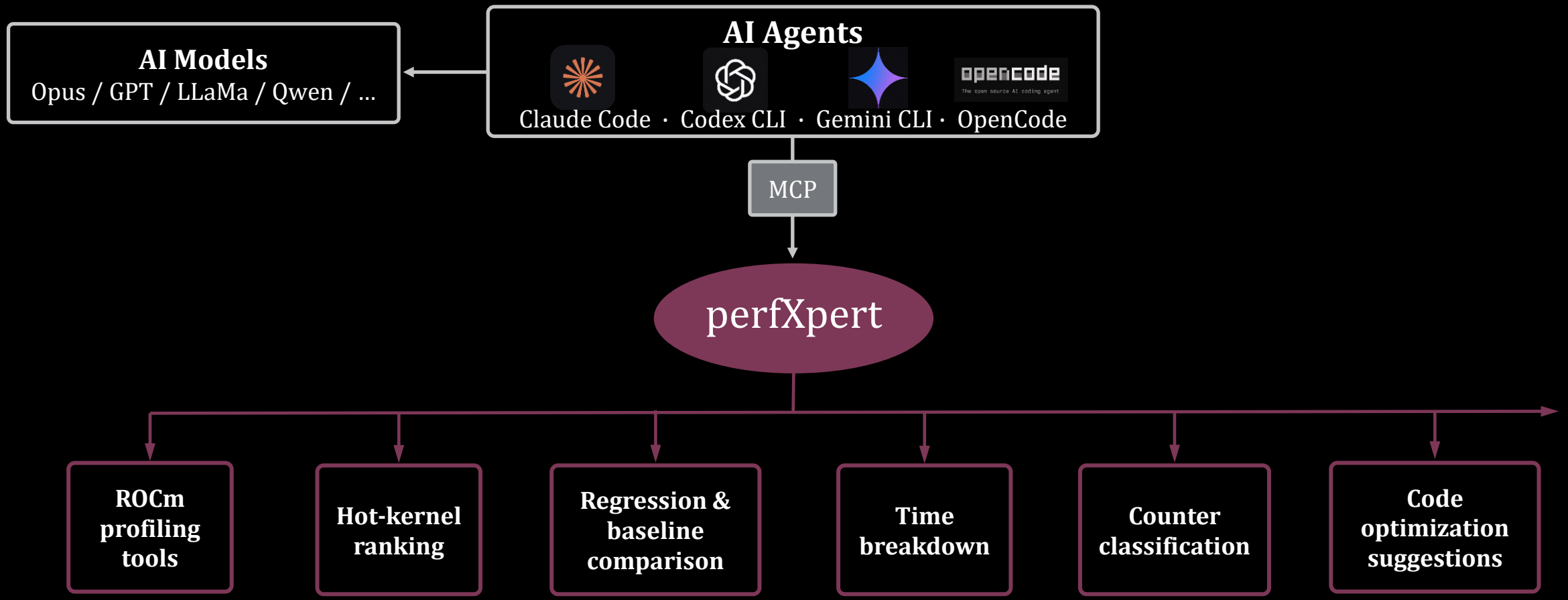
How do I interpret the output?

What do all these counters mean?

Which part of my code do I optimize?

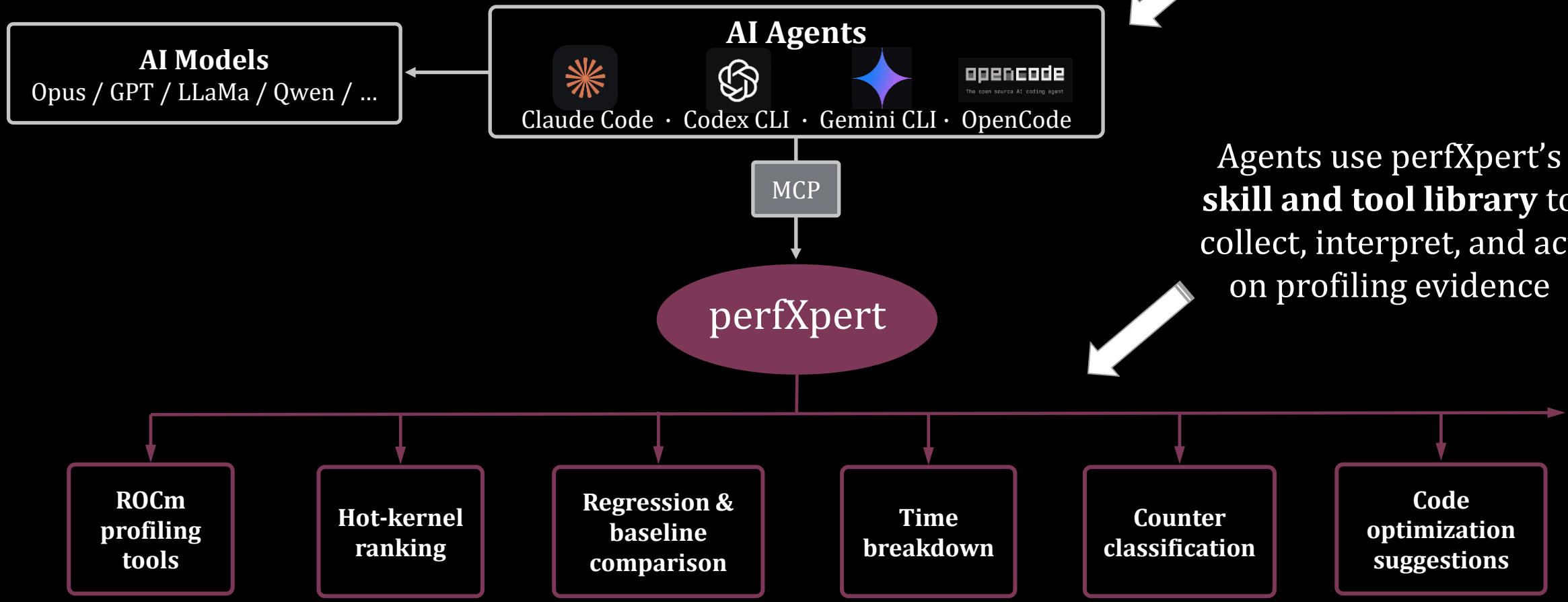
What compiler flags would benefit my code?

Seemless integration: perfXpert-mcp



Seemless integration: perfXpert-mcp

You use the AI agents and models you **know and trust.**



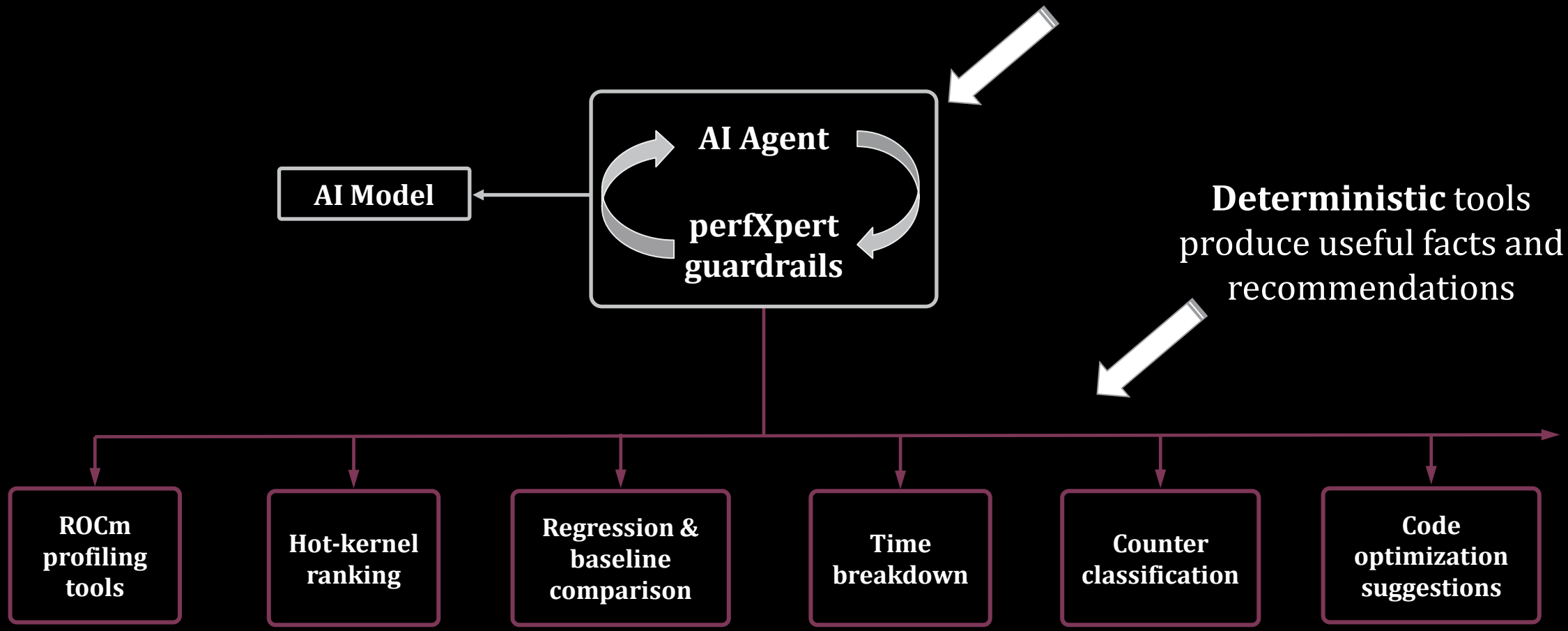
Agents use perfXpert's **skill and tool library** to collect, interpret, and act on profiling evidence

Correctness & Determinism

- How can I trust the results?
- Did my agent run the tools correctly?
- Is the result analysis accurate?
- Is my agent hallucinating the whole thing?

Determinism: perfXpert-code

perfXpert-code **controls agent** behavior and **verifies output** at every step to **avoid bogus results**



Deterministic tools produce useful facts and recommendations

Extending perfXpert

- Open-source project in public rocm-systems Github repository
  - https://github.com/ROCm/rocm-systems/tree/develop/experimental/python/perfxpert
- Skills and tools are text-based and human readable
- External tool developers can contribute and integrate with their tool
- Application/kernel developers can add knowledge about their code
- Users can add tests, optimization strategies, architecture details

Useful Links

ROCm Tools Documentation

ROCm Developer Tools GitHub Super Repo

ROCm Optiq GitHub Repo

ROCgdb GitHub Repo

Omnistat GitHub Repo

Check these blog posts:

- Introduction to Profiling Tools for AMD Hardware
- Performance Profiling on AMD GPUs – Part 1: Foundations
- Performance Profiling on AMD GPUs – Part 2: Basic Usage
- Performance Profiling on AMD GPUs – Part 3: Advanced Usage