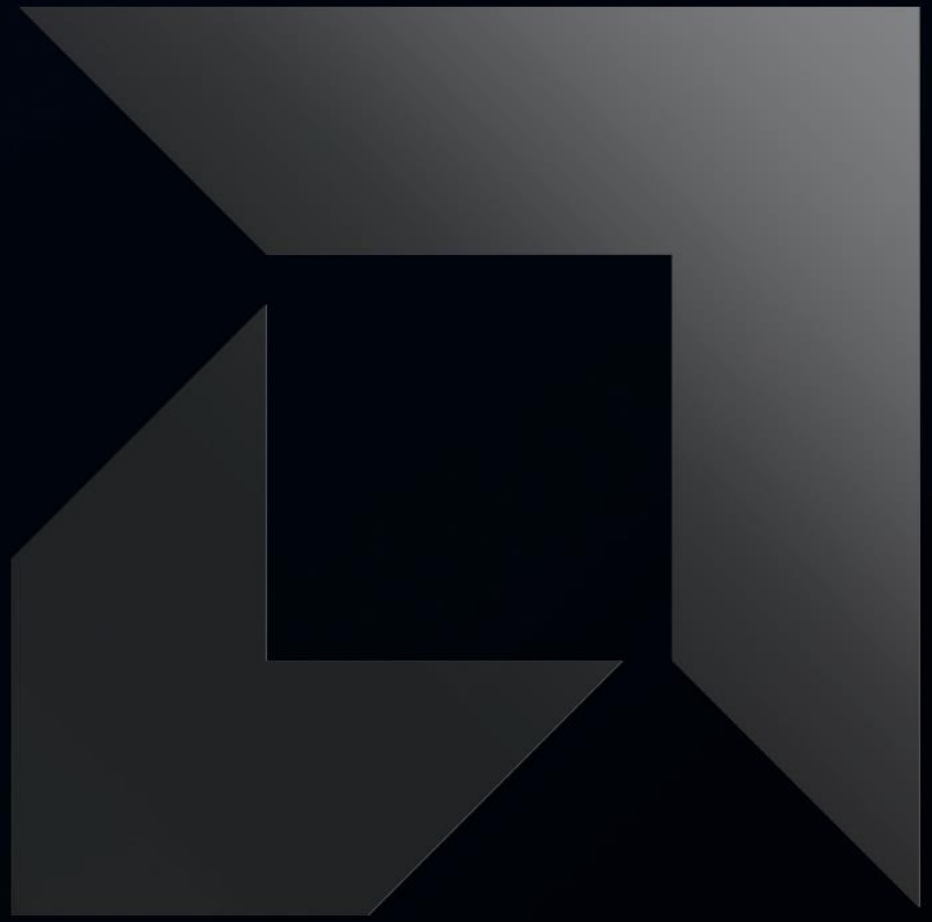


ROCm Performance Analysis SDK and Tooling Ecosystem

Jonathan R. Madsen, PhD
ROCm Profiling Group
jonathan.madsen@amd.com

Scalable Tools Workshop
July 07, 2025

AMD
together we advance_

Performance Analysis Overview

APIs

- Rocprofiler SDK
- Rocprofiler SDK ROCTx
  - new ROCTx
- AMD-SMI
  - formerly: ROCm-SMI

Tools

- ROCm Systems Profiler
  - formerly: Omnitrace
- ROCm Compute Profiler
  - formerly: Omniperf
- rocprofv3

Analysis

- ROCpd (beta)

Rocprofiler-SDK

ROCm Profiling APIs

Requested Features & Improvements From External Tool Developers

- General performance improvement
- Dramatically improve reliability and stability between ROCm releases
- Robust initialization for rocprofiler and tools
- Multi-tool Support
- Make external correlation IDs more useful
- Ability for tool to force initialization (instead of relying on rocprofiler to call tool)
- Notifications for when rocprofiler creates internal threads
- Thread pool for buffer callbacks
- Improved thread-safety, error handling, memory access correctness
- Eliminate memory leaks
- Improve counter collection performance
- Topology/agent information without having to use (and initialize) HSA-runtime and/or HIP

Initialization of Rocprofiler and Tools (1/2)

- Tool libraries provide special symbol “rocprofiler_configure” which can returns a simple struct containing callbacks for initialization and/or finalization
  - This is a similar scheme to how OpenMP Tools works
- Introduced **rocprofiler-register** library
  - github.com/ROCm/rocprofiler-register-internal
  - Link dependency for ROCr (HSA-runtime), HIP, ROCTx
  - Scans for special symbol: “rocprofiler_configure”
  - Canonicalizes environment variable ROCP_TOOL_LIBRARIES
    - E.g., HSA_TOOLS_LIB is HSA-specific
  - When special symbol is found or env variable is set finds loaded or loads rocprofiler-sdk library, passes dispatch tables to rocprofiler-sdk
- “rocprofiler_force_configure” can be used to force initialization before a runtime triggers initialization

Initialization of Rocprofiler and Tools (2/2)

```
typedef int (*rocprofiler_tool_initialize_t)(rocprofiler_client_finalize_t finalize_func,
                                             void*                        tool_data);
```

```
typedef void (*rocprofiler_tool_finalize_t)(void* tool_data);
```

```
typedef struct
{
    size_t                        size;  ///< size of this struct (in case of future extensions)
    rocprofiler_tool_initialize_t initialize;  ///< context creation
    rocprofiler_tool_finalize_t   finalize;    ///< cleanup
    void*                         tool_data;   ///< data to provide to init and fini callbacks
} rocprofiler_tool_configure_result_t;
```

```
rocprofiler_tool_configure_result_t*
rocprofiler_configure(uint32_t                 version,
                      const char*              runtime_version,
                      uint32_t                 priority,
                      rocprofiler_client_id_t* client_id) ROCPROFILER_PUBLIC_API;
```

Multi-Tool Support (1/3)

- Each instance of special symbol is assigned unique client ID
- Initialize function (provided to rocprofiler by tool in “rocprofiler_configure”) provides tool opportunity for tool to create as many “contexts” as desired
  - All contexts that are created here are associated with the unique client ID
  - ROCm Runtimes (e.g. HSA, HIP, etc.) must not be called during initialization
- Contexts ***cannot*** be created or modified outside of initialization function
  - In other words, during initialization, tools tells us exactly which feature they wish to be made available; even if they don’t intend to use this feature right away

Multi-Tool Support (2/3)

- Contexts
  - Bundle one or more services into a single handle
  - Improve and simplify activation/deactivation/reactivation of one or more services
    - “rocprofiler_start_context” will activate all the services associated with that context
    - “rocprofiler_stop_context” will disactivate all the services associated with that context
      - If this is called while a context is in use, the completion of the services associated with that context will not be aborted
  - Support multiple active contexts for certain services
    - Supported: API tracing, kernel tracing, code object tracing, etc.
    - Not supported: dispatch counter collection, PC sampling

Multi-Tool Support (3/3)

```
auto hsa_context = rocprofiler_context_id_t{};

rocprofiler_create_context(&hsa_context);

for(auto itr : {ROCPROFILER_CALLBACK_TRACING_HSA_CORE_API,
                ROCPROFILER_CALLBACK_TRACING_HSA_AMD_EXT_API,
                ROCPROFILER_CALLBACK_TRACING_HSA_IMAGE_EXT_API,
                ROCPROFILER_CALLBACK_TRACING_HSA_FINALIZE_EXT_API})
{
    rocprofiler_configure_callback_tracing_service(
        hsa_context, itr, nullptr, 0, tool_tracing_callback, nullptr);
}
```

External Correlation IDs

- External correlation IDs are now unions of 64-bit unsigned int or void pointer
- New “external correlation ID request” service
- External correlation IDs are thread-specific and context-specific
  - Supports setting the external correlation on a different thread than current thread
- Callback and buffer record correlation IDs are pairs of the internal and external correlation ID

```
typedef union rocprofiler_user_data_t
{
    uint64_t value;  ///< e.g., set to thread id
    void*    ptr;    ///< e.g., set to data allocation
} rocprofiler_user_data_t;

typedef struct rocprofiler_correlation_id_t
{
    uint64_t                internal;
    rocprofiler_user_data_t external;
} rocprofiler_correlation_id_t;
```

API and Activity Tracing

- API & Activity Tracing API
  - HIP API tracing split into two groups
  - ROCTx API tracing split into three groups
  - HSA API tracing split into four groups
  - Kernel dispatch tracing
  - Memory copy tracing
  - Memory allocation tracing
  - Scratch allocations
  - KFD page migration, page faults, + more
  - Tool and runtime thread creation
  - HIP stream tracing

- Initialization scheme enables new optimizations
- **Implementation scheme enables ABI stability across ROCm releases**
- Implementation scheme is much more maintainable
- Consistent string ⇔ enum mapping
- Iterate over arguments in callback and buffer tracing

Buffer Improvements

- Improved Buffered Implementation
  - Improved data storage efficiency (i.e., variable sized records)
  - Concurrent copies into buffer
- Ability to create dedicated threads for each buffer, direct multiple buffers to one thread, or somewhere in between → thread-pool for buffer records
- Callbacks with buffer records are easily parallelizable despite variable distance between adjacent records

Counter Collection

- Support for Instances and Dimensions
  - Exposes performance counter data on an individual XCC/WGPs/etc. level
- Customizable Instance Aggregation and Reduction Operations
  - User selectable methods for how to aggregate and reduce counter data across instances (max/min/avg across multiple dimensions)
- Dispatch kernel serialization to improve counter result accuracy on a per kernel level
- Performance and stability improvements over v1/v2

PC Sampling

- Host-trap PC Sampling available on MI200 and MI300
  - Software-based
- Stocastic PC Sampling available on MI300
  - Supported in the hardware
- HPCToolkit and TAU utilize PC Sampling

Rocprofiler-Register

ROCm Profiling APIs (Internal)

rocprofiler-register (Background)

- HIP, HSA-runtime, ROCTx use dispatch tables (AKA intercept tables) for their public APIs
  - A dispatch table is a struct of function pointers containing all/some of the public API
  - These function pointers are all initially set to point to the implementation of the public API function
    - `hipSetDevice_fn` in table initially points to `hip::hipSetDevice` , which is the implementation of the function
  - The public API functions, e.g. `hipSetDevice`, simply call the function pointer
    - `hipSetDevice` → `tbl.hipSetDevice_fn` → `hip::hipSetDevice`
  - Using dispatch tables allows profiling tools to replace the function pointer with a wrapper that calls into the tool first
    - `hipSetDevice` → `tbl.hipSetDevice_fn` → `rocprofiler::hipSetDevice` → `hip::hipSetDevice`

rocprofiler-register (Background)

- Immediately after the dispatch table is initially constructed, the dispatch table needs to be made available to a profiling library (i.e. rocprofiler-sdk) to add the wrappers if profiling is requested
  - In the past, HSA-runtime used an env variable “HSA_TOOLS_LIB” to deduce profiling was requested
    - Relying on environment variables has many problems
      - Race to update environment variable before HSA reads it
      - Each additional runtime would use their own specific environment variable, e.g. HIP_TOOLS_LIB
      - … Many more, I could talk about this for an hour

rocprofiler-register (Purpose)

- rocprofiler-register automates, unifies, and simplifies the communication between a runtime using a dispatch table and the rocprofiler-sdk library
  - Simplifies implementation in runtimes: typically, only requires about 1-2 dozen lines of code
  - Ensures all runtimes use same automated procedure for deducing whether a profiling request has been made, one environment variable for all runtimes instead of one per runtime (environment variables are available as fallback)
  - Automates by looking for “special symbol” (i.e. publicly visible function) implemented by tool libraries
    - If symbol doesn’t exist (and env variable is unset/empty) → no profiling request is being made

Rocprofiler-SDK ROCTx

ROCm Profiling APIs

Rocprofiler-SDK ROCTx

- New **rocprofiler-sdk-roctx** package
  - Replaces ROCTx distributed as part of roctracer
  - Library: **librocprofiler-sdk-roctx.so**
  - Header: **rocprofiler-sdk-roctx/roctx.h**
- Contains all original ROCTx function (mark, range push, range start, etc.)
- Adds new functions for controlling profiler
  - roctxProfilerPause(…)
  - roctxProfilerResume(…)
  - Contexts make it *extremely* easy for tools to support these functions
- Adds new functions for “hinting” to tool how you would like resources to be labeled in their output
  - roctxNameOsThread(…), roctxNameHipDevice(…), roctxNameHipStream(…), etc.
  - Rocprofiler SDK does nothing to enforce using these names – each tool decides if/how it is supported

Rocprofiler-SDK ROCTx – Additional Notes

- Python bindings in ROCm 7.0
- Planned features:
  - Annotation support (i.e. pass key-value pairs of data in addition to string message)
  - Feature parity with NVTX3

AMD-SMI

ROCm Profiling APIs

AMD-SMI

- Previous iteration: ROCm-SMI
- System Monitoring:
  - GPU temperature, power level, utilization, memory usage, etc.
- Tools use this API to periodically sample device-level metrics

ROCm Profiling Tools

ROCm Systems Profiler

ROCm Profiling Tools

Tool: ROCm Systems Profiler (rocprofiler-systems)

- A comprehensive profiling and tracing tool used for applications running on the CPU or GPU
- Gathers performance data through binary instrumentation, call-stack sampling, hardware counter sampling, user APIs, and Python Interpreter hooks
- Detailed traces are visualized with Perfetto
- High-level summaries (profiles) are also produced via JSON or TXT

rocprofiler-systems - Configuration

- We use `rocprof-sys-avail` for more information about settings and capabilities.

ROCPROFSYS_PROFILE	Enable timemory backend
ROCPROFSYS_ROCM_EVENTS	ROCm hardware counters. Use ':device=N' syntax to specify collection on device number N, e.g. ':device=0'. If no device specification is provided, the event is collected on every available device
ROCPROFSYS_SAMPLING_CPUS	CPUs to collect frequency information for. Values should be separated by commas and can be explicit or ranges, e.g. 0,1,5-8. An empty value implies 'all' and 'none' suppresses all CPU frequency sampling
ROCPROFSYS_SAMPLING_DELAY	Time (in seconds) to wait before the first sampling signal is delivered, increasing this value can fix deadlocks during init
ROCPROFSYS_SAMPLING_DURATION	If > 0.0, time (in seconds) to sample before stopping
ROCPROFSYS_SAMPLING_FREQ	Number of software interrupts per second when OMNITTRACE_USE_SAMPLING=ON
ROCPROFSYS_SAMPLING_GPUS	Devices to query when ROCPROFSYS_USE_ROCM_SMI=ON. Values should be separated by commas and can be explicit or ranges, e.g. 0,1,5-8. An empty value implies 'all' and 'none' suppresses all GPU sampling

- These values can be set as environment variables, but we can also create a configuration file.

```
$ rocprof-sys-avail -G -O $HOME/.rocprofsys.cfg
```

- Modify the newly created file, as desired, to tailor your profiling.
- Then, declare which configuration file to use by setting just one environment variable

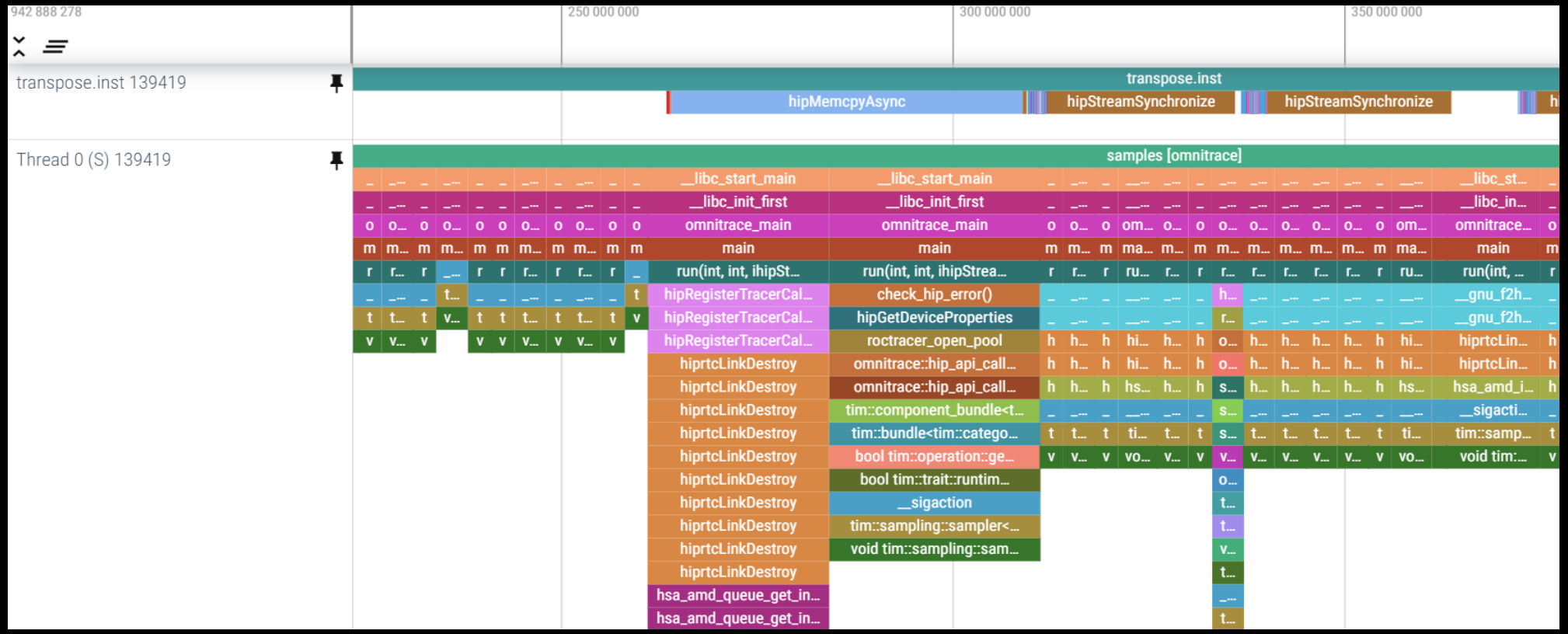
```
$ export ROCPROFSYS_CONFIG_FILE=rocprofsys.cfg
```

rocprofiler-systems – Call-stack Sampling

- The `rocprof-sys-sample` application is used to execute call-stack sampling on an application.

```
usage: $ rocprof-sys-sample <rocprofsys-options> -- <exe> <exe-options>
```

- Visualized in Perfetto



rocprofiler-systems – Binary Instrumentation

- We can use `rocprof-sys-instrument` to perform a binary rewrite of the test application

```
$ rocprof-sys-instrument -o <output-filename> <rocprofsys-options> -- <exe>
```

- Once the binary has been rewritten, we use `rocprof-sys-run` to run the instrumented binary

```
$ rocprof-sys-run <rocprofsys-options> -- <instrumented-exe> <exe-arguments>
```

- For example, let us instrument and run our *transpose* application

```
$ rocprof-sys-instrument -o transpose.inst -- ./transpose
$ rocprof-sys-run -- ./transpose.inst
```

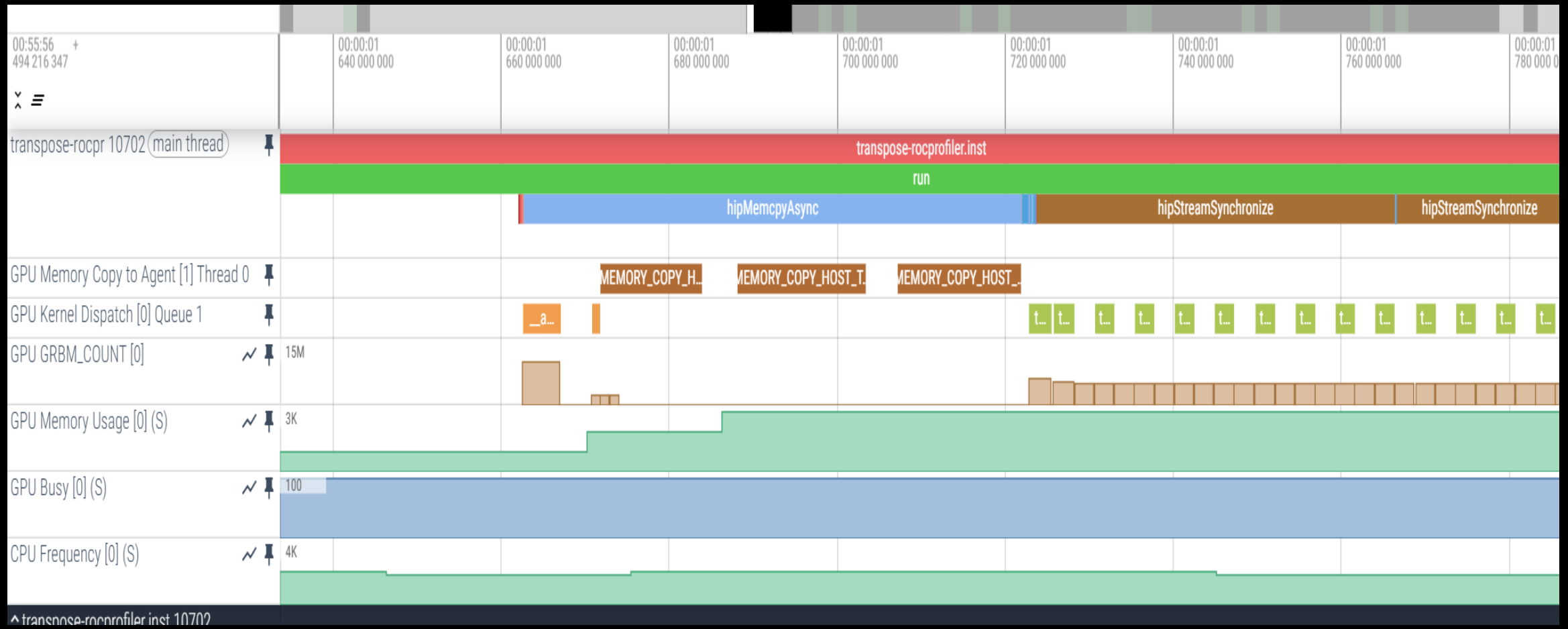
```
ROCPROFSYS: LD_PRELOAD=/home/gliff/code/rocprofiler-systems/build/ubuntu/22.04/lib/librocprof-sys-dl.so.1.0.0
ROCPROFSYS: OMP_TOOL_LIBRARIES=/home/gliff/code/rocprofiler-systems/build/ubuntu/22.04/lib/librocprof-sys-dl.so.1.0.0
[rocprof-sys][167973][rocprofsys_init_tooling] Instrumentation mode: Trace

   rocprof-sys v1.0.0 (rev: e8ff3d0Bebd5014d39537551387edc54b0cf0aab, x86_64-linux-gnu, compiler: GNU v11.4.0, rocm: v6.3.x)
[397.105]      perfetto.cc:47606 Configured tracing session 1, #sources:1, duration:0 ms, #buffers:1, total buffer size:1024000 KB, total sessions:1, uid:0
 session name: ""
[rocprof-sys][167973][0][rocprofiler_configure] rocprofsys is using rocprofiler-sdk v0.5.0 (0.5.0)
[rocprof-sys][dl][167973] rocprofsys_main
[transpose] Number of threads: 2
[transpose] Number of iterations: 500
[transpose] Syncing every 10 iterations
[transpose] Number of devices found: 1
[transpose] Rank 0 assigned to device 0
[0][1] M: 9920 N: 9920
[0][0] M: 9920 N: 9920
[0][1] Runtime of transpose is 2.3378 sec
The average performance of transpose is 156.811 GBytes/sec
[0][0] Runtime of transpose is 2.33963 sec
The average performance of transpose is 156.688 GBytes/sec

[rocprof-sys][167973][0][rocprofsys_finalize] finalizing...
[rocprof-sys][167973][0][rocprofsys_finalize]
[rocprof-sys][167973][0][rocprofsys_finalize] rocprofsys/process/167973 : 4.344695 sec wall_clock, 1258.180 MB peak_rss,  196.432 MB page_rss, 9.760000 sec
cpu_clock,  224.6 % cpu_util [laps: 1]
[rocprof-sys][167973][0][rocprofsys_finalize] rocprofsys/process/167973/thread/0 : 4.338187 sec wall_clock, 4.209750 sec thread_cpu_clock,   97.0 % thread_c
pu_util, 1258.180 MB peak_rss [laps: 1]
[rocprof-sys][167973][0][rocprofsys_finalize] rocprofsys/process/167973/thread/2 : 0.000051 sec wall_clock, 0.000052 sec thread_cpu_clock,  100.1 % thread_c
pu_util,    0.000 MB peak_rss [laps: 1]
```

rocprofiler-systems – Binary Instrumentation

Visualization of Application Trace



ROCm Compute Profiler

ROCm Profiling Tools

Tool: ROCm Compute Profiler (rocprofiler-compute)

- rocprofiler-compute is an GPU kernel level architecture performance analyzer for AMD MI series GPUs(>= MI50)
- rocprofiler-compute executes the code many times to collect various hardware counters through ROCprofiler
- rocprofiler-compute shows 17+ panels of derived metrics based on hardware raw counters with rich customization capability

Rocprofiler-compute - Feature

0. Top Stat	12. Local Data Share(LDS)
1. System Info	13. Instruction Cache
2. System Speed-of-Light	14. Scalar L1 Cache
3. Memory Chart	15. Texture Address and Texture Data
4. Roofline Analysis(MI200 only)	16. L1 Cache
5. Command Processor	17. L2 Cache
6. Shader Processing Input	18. L2 Cache (per Channel)
7. Wavefront Launch Stat	
10. Compute Unit - Instruction Mix	
11. Compute Unit - Compute Pipeline	

Rocprofiler-compute - System Speed-of-Light

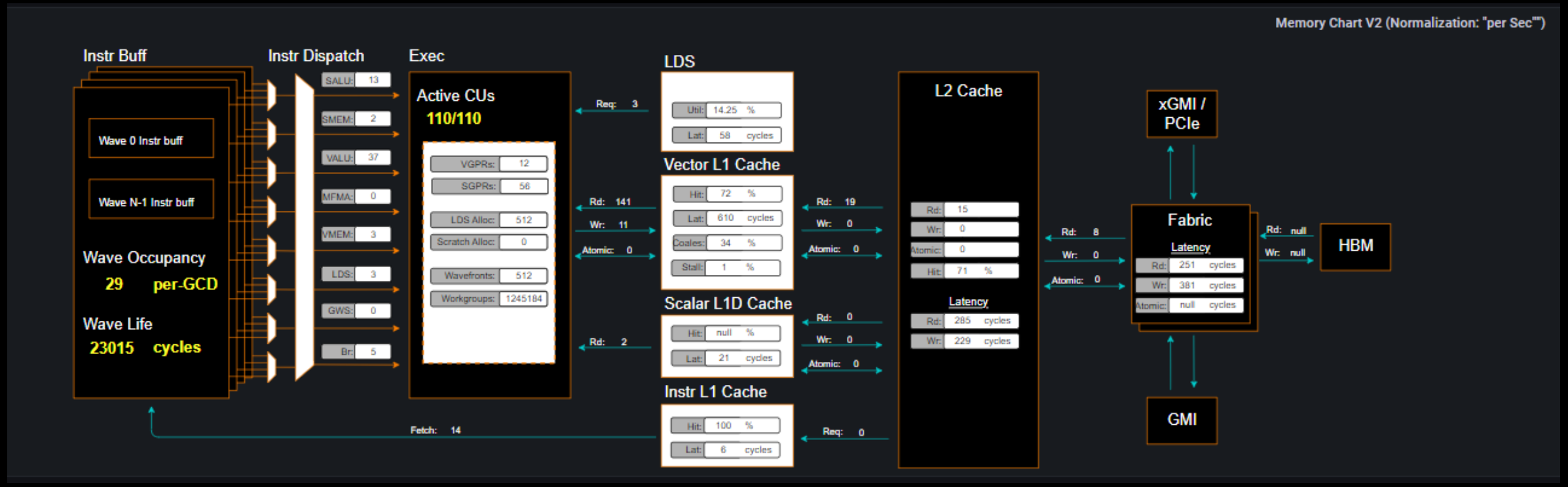
Speed of Light

Metric	Avg	Unit	Theoretical Max	Pct-of-Peak
VALU FLOPs	249	GFLOP	23,936	1%
VALU IOPs	1,370	GIOP	23,936	6%
MFMA FLOPs (BF16)	0	GFLOP	95,744	0%
MFMA FLOPs (F16)	0	GFLOP	191,488	0%
MFMA FLOPs (F32)	0	GFLOP	47,872	0%
MFMA FLOPs (F64)	0	GFLOP	47,872	0%
MFMA IOPs (Int8)	0	GIOP	191,488	0%
Active CUs	110	CUs	110	100%
SALU Util - %	17	pct	100	17%
VALU Util - %	42	pct	100	42%
MFMA Util - %	0	pct	100	0%
VALU Active Threads/Wave	60	Threads	64	94%
IPC - Issue	1	Instr/cycle	5	18%
LDS BW	1,625	GB/sec	23,936	7%
LDS Bank Conflict	0	Conflicts/access	32	0%
Instr Cache Hit Rate	100	pct	100	100%
Instr Cache BW	876	GB/s	6,093	14%
Const Cache Hit Rate	100	pct	100	100%
Const Cache BW	123	GB/s	6,093	2%
L1 Cache Hit Rate	72	pct	100	72%
L1 Cache BW	4,335	GB/s	11,968	36%
TC2TD Stall	57	pct	100	57%
TD Busy	100	pct	100	100%
L2 Cache Hit Rate	71	pct	100	71%
L2-EA Read BW	534	GB/s	1,638	33%
L2-EA Write BW	2	GB/s	1,638	0%
L2-EA Read Latency	251	Cycles		
L2-EA Write Latency	381	Cycles		
Wave Occupancy	3,171	Wavefronts	3,520	90%
Instr Fetch BW	536	GB/s	3,046	18%
Instr Fetch Latency	16	Cycles		

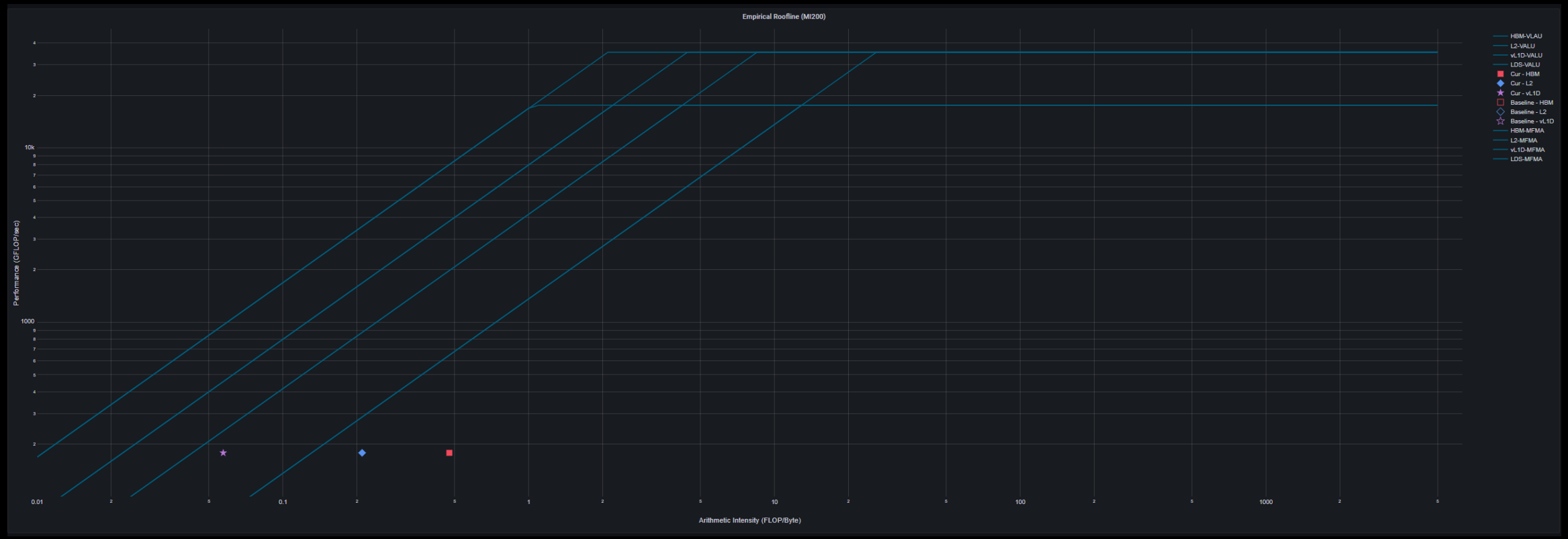
Hierarchy Analysis:

Check key metrics in System Speed-of-Light,
then go to detailed blocks if you have any suspicions there.

Rocprofiler-compute - Memory Chart



Rocprofiler-compute - Empirical Hierarchical Roofline



Rocprofiler-compute - Modes

- Profiling

```
rocprof-compute profile -n workload_name [profile options] [roofline options] -- <profile_cmd>
```

- CLI Analysis

```
rocprof-compute analyze -p workloads/workload_name/MIxxx/
```

- Standalone GUI Analysis

```
rocprof-compute analyze -p workloads/workload_name/MIxxx/ --gui
```

Then follow the instructions to open the web page for the GUI

- Grafana GUI import

```
rocprof-compute database --import [connection options]
```

- Grafana GUI Analysis

Connect to Grafana server, choose the case to analyze

Rocprofiler-compute - CLI Analysis

- Use vcopy example:

```
rocprof-compute analyze -p workloads/vcopy_kernel/mi200/ &> vcopy_analyze.txt
```

0. Top Stat

	KernelName	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct
0	vecCopy(double*, double*, double*, int, int) [clone .kd]	1	341123.00	341123.00	341123.00	100.00

2. System Speed-of-Light

Index	Metric	Value	Unit	Peak	PoP
2.1.0	VALU FLOPs	0.00	Gflop	23936.0	0.0
2.1.1	VALU IOPs	89.14	Giop	23936.0	0.37242200388114116
2.1.2	MFMA FLOPs (BF16)	0.00	Gflop	95744.0	0.0
2.1.3	MFMA FLOPs (F16)	0.00	Gflop	191488.0	0.0
2.1.4	MFMA FLOPs (F32)	0.00	Gflop	47872.0	0.0
2.1.5	MFMA FLOPs (F64)	0.00	Gflop	47872.0	0.0
2.1.6	MFMA IOPs (Int8)	0.00	Giop	191488.0	0.0
2.1.7	Active CUs	58.00	Cus	110	52.72727272727273
2.1.8	SALU Util	3.69	Pct	100	3.6862586934167525
2.1.9	VALU Util	5.90	Pct	100	5.895531580380328
2.1.10	MFMA Util	0.00	Pct	100	0.0
2.1.11	VALU Active Threads/Wave	32.71	Threads	64	51.10526315789473
2.1.12	IPC - Issue	0.98	Instr/cycle	5	19.576640831930312

7.1 Wavefront Launch Stats

Index	Metric	Avg	Min	Max	Unit
7.1.0	Grid Size	1048576.00	1048576.00	1048576.00	Work items
7.1.1	Workgroup Size	256.00	256.00	256.00	Work items
7.1.2	Total Wavefronts	16384.00	16384.00	16384.00	Wavefronts
7.1.3	Saved Wavefronts	0.00	0.00	0.00	Wavefronts
7.1.4	Restored Wavefronts	0.00	0.00	0.00	Wavefronts
7.1.5	VGPRs	44.00	44.00	44.00	Registers
7.1.6	SGPRs	48.00	48.00	48.00	Registers
7.1.7	LDS Allocation	0.00	0.00	0.00	Bytes
7.1.8	Scratch Allocation	16496.00	16496.00	16496.00	Bytes

rocprofv3

ROCm Profiling Tools

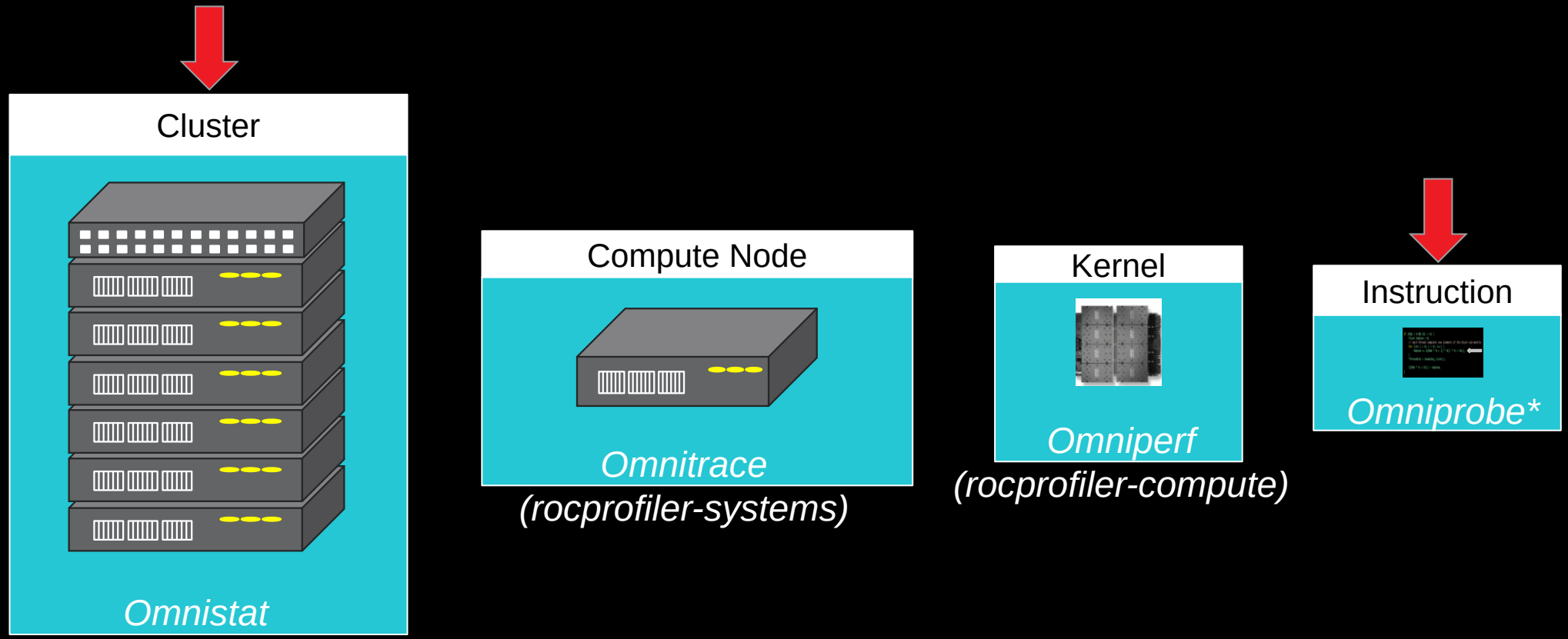
Tool: rocprofv3

- Built on top of rocprofiler-sdk
- Tracing capabilities: HSA, HIP, RCCL, kernel, memory copy and allocation, scratch memory
- GPU HW counter collection (kernel dispatch)
- Output formats
  - ROCm <= 7.0: CSV, JSON, Perfetto, OTF2
  - ROCm > 7.0: SQLite3 database
    - Note overlap for ROCm 7.0… more on this later

AMD Research Tools

AMD Research Tools

- Contact: Keith Lowery (keith.lowery@amd.com)



Research: Omniprobe Areas of Interest

- Intra-kernel region-based timing through timestamp injection
- Memory access pattern traces
- Memory coalescing and shared memory bank conflicts
- Cache misses

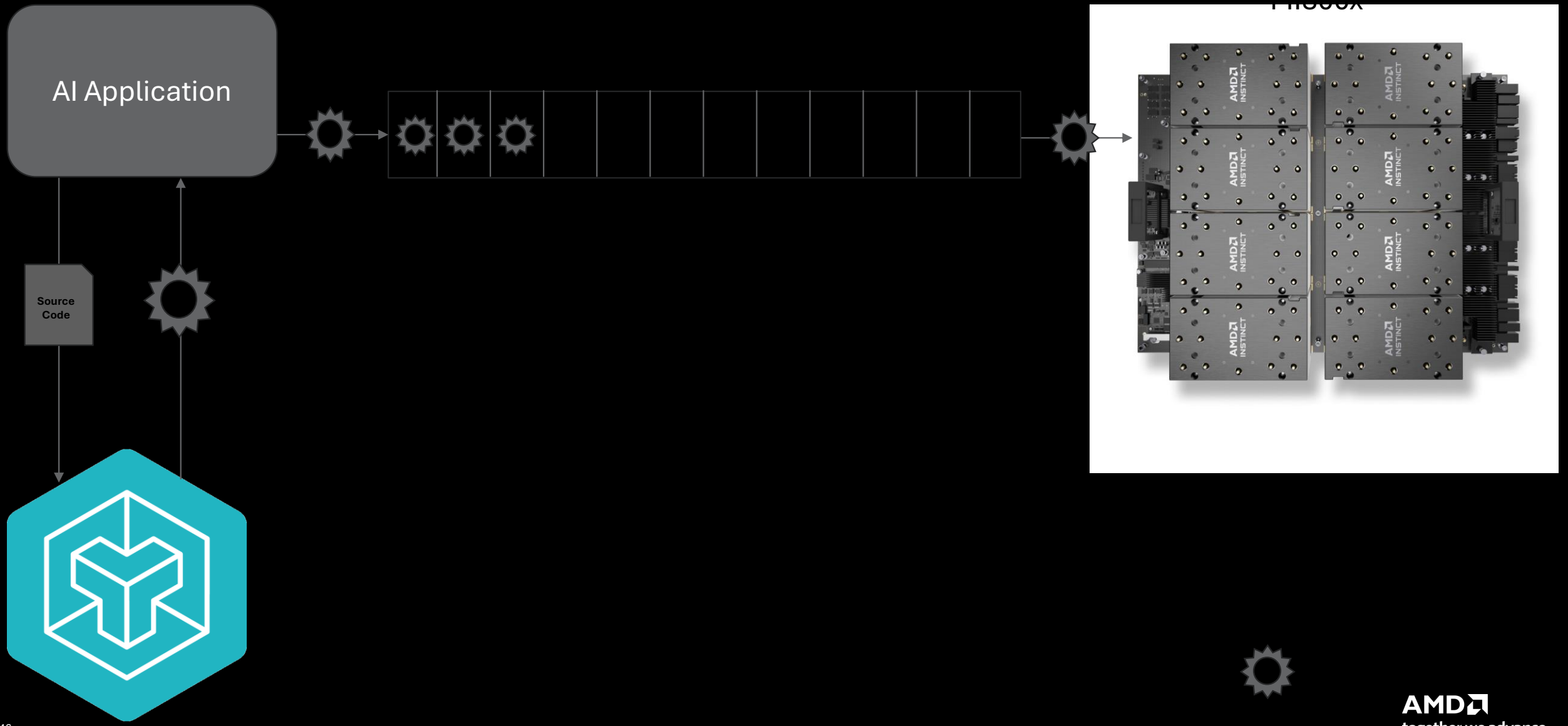
Research: What motivated to create Omniprobe

- Limited observability by hardware and existing tools.
  - Current tools provide only aggregate understanding of kernel performance.
    - You know bank conflicts occurred but you don’t know where.
    - You know what the average memory b/w was, but you don’t know why.
    - You might know average FLOPS, but you don’t know anything about performance variability in different parts of a kernel.
- Key customers are asking questions our current tools can’t answer
  - What line of code caused a bank conflict?
  - Where in my code do have have memory accesses that don’t coalesce?
  - Where are the opportunities for overlapping compute and data movement?
  - Do I have memory address alignment issues that are affecting my kernel performance?
  - Where is most of my time being consumed in a kernel and why?
- We wanted to overcome some of the limits of our current perfmon features.

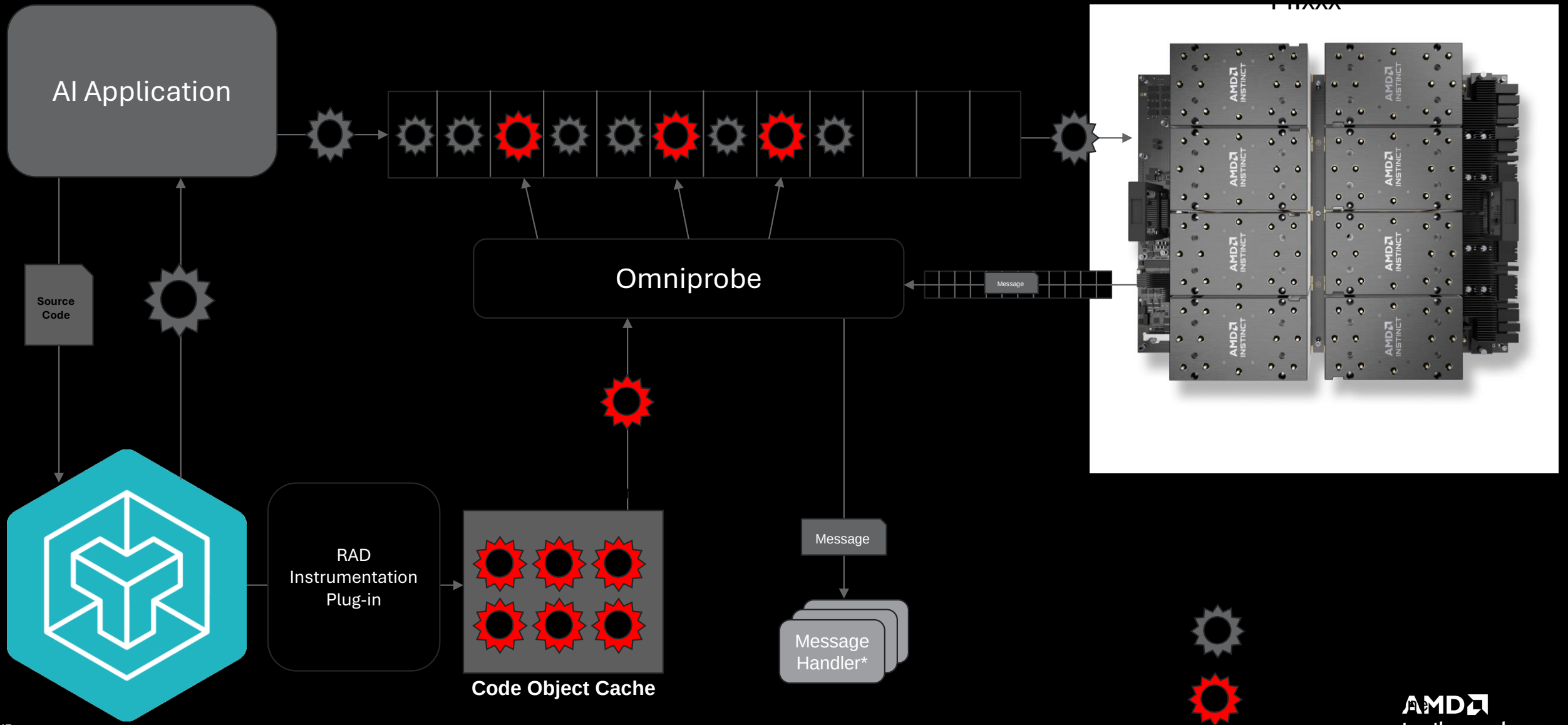
Research: Omniprobe Components

- Instrumentation compiler plug-in
  - An LLVM mid-end plugin to clone and instrument GPU kernels.
- Data/Message Streaming GPU->Host
  - A host and device library for sending messages for host analysis from instrumented kernels.
- Omniprobe HSA runtime plug-in (TBD: transition to rocprofiler-sdk)
  - An HSA plug-in for selectively injecting instrumented kernels into HSA queues.

Omniprobe Runtime - Triton

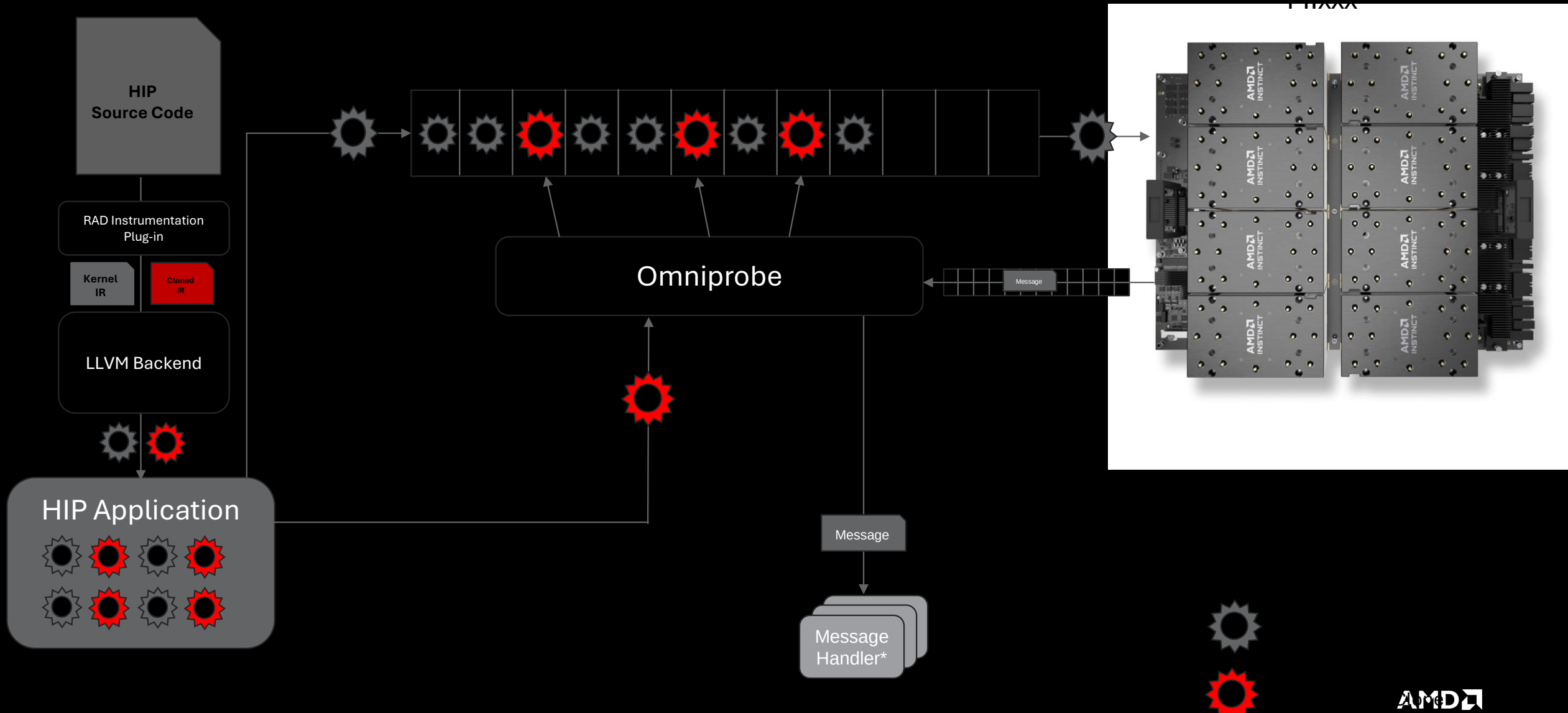


Omniprobe Runtime - Triton



*Developers can create their own message handlers for doing custom

Omniprobe Runtime - HIP



*Developers can create their own message handlers for doing custom

ROCm Profiling Analysis

ROCpd

ROCm Profiling Analysis

Legacy Approach to Profiling Data: Gaps and Deficiencies

- Large traces are very difficult to digest
  - Where am I spending the most compute time? What kernels are slowest? What is the total exclusive time spent in that function? How much data am I copying? And what is the average size of the copies?
  - Our current tools do very little, if any, to help answer these questions
- If you reach the Perfetto buffer limit and data was discarded… too bad, change the settings and run app again.
- If you did not request profiles/summary/stats at the start of the run… too bad, run again.
- If the collection period you added to reduce the data size missed something important… too bad, run again.
- If you removed trace annotations to reduce the data size but now need to see it… too bad, run again.
- If you removed ROCm API backtraces to reduce data size but now need them… too bad, run again.
- If you want the current profiling data converted to another supported format… too bad, run again.

Legacy Approach to Profiling Data: Gaps and Deficiencies

- There is a core problem at the heart of this approach…

THE ORIGINAL DATA COLLECTED BY THE TOOL IS NOT PRESERVED

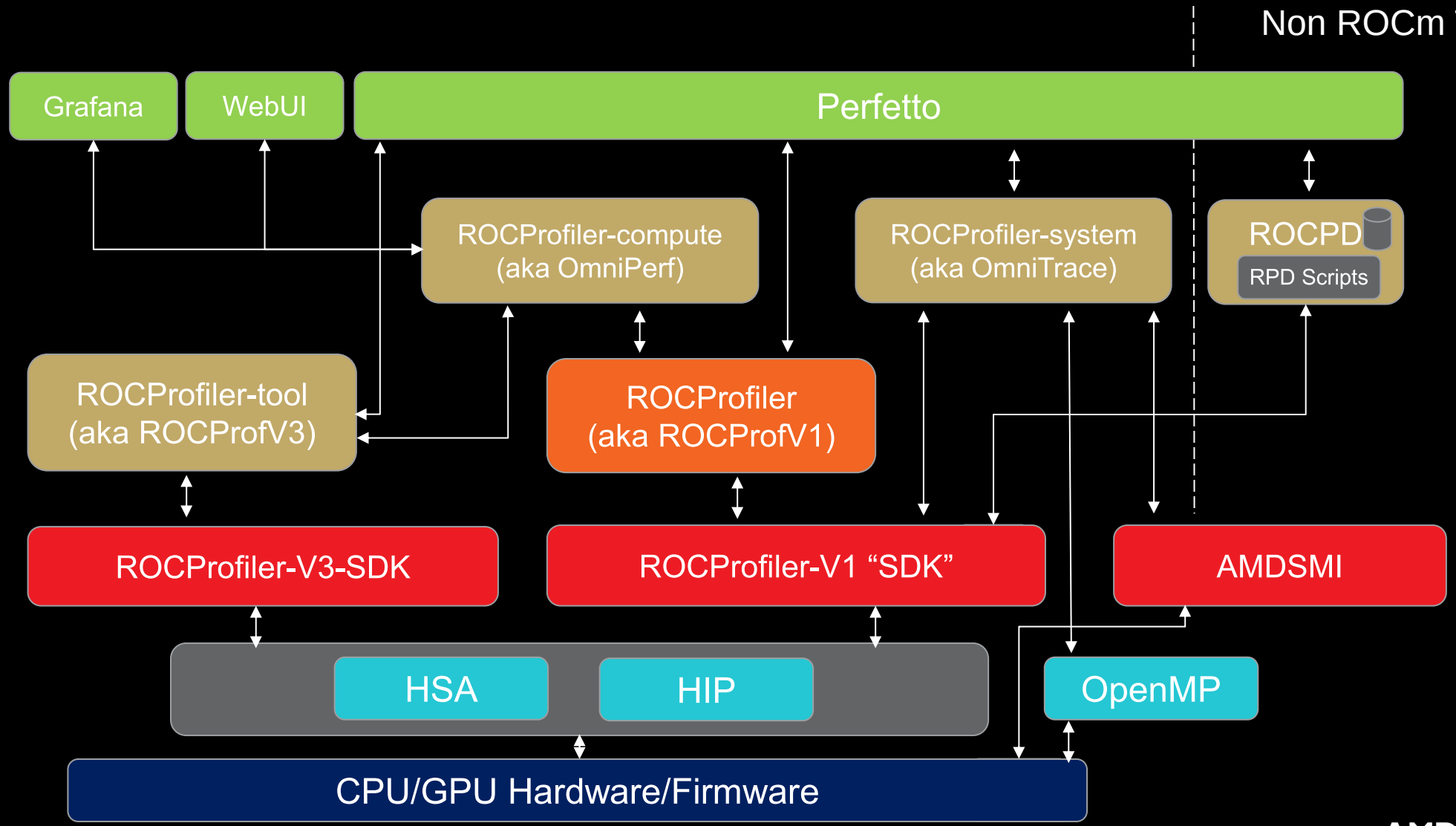
- This is a natural by-product of the expectation that the command-line tool that collects the data should also be responsible for the visualization output format and/or data analysis.

New Approach to Profiling Data: Overview

- rocprofv3 and rocprofiler-systems will eventually only support outputting the raw data collected by the tool
  - i.e., these command-line tools will only handle the data collection phase.
- ROCm Profiling Data (ROCpd) will provide the data analysis phase.
  - Conversion to CSV, Perfetto, OTF2 + more in future
  - Generation of summaries, statistics
  - Comparison of two different profiling runs
  - Guided Analysis
  - Integration with Jupyter notebooks
  - Exporting analysis to HTML, PDF, Markdown, Excel, etc.
  - Exporting plots to PNG, JPEG, etc.
  - Multi-process and multi-node aggregation
  - Data filtering

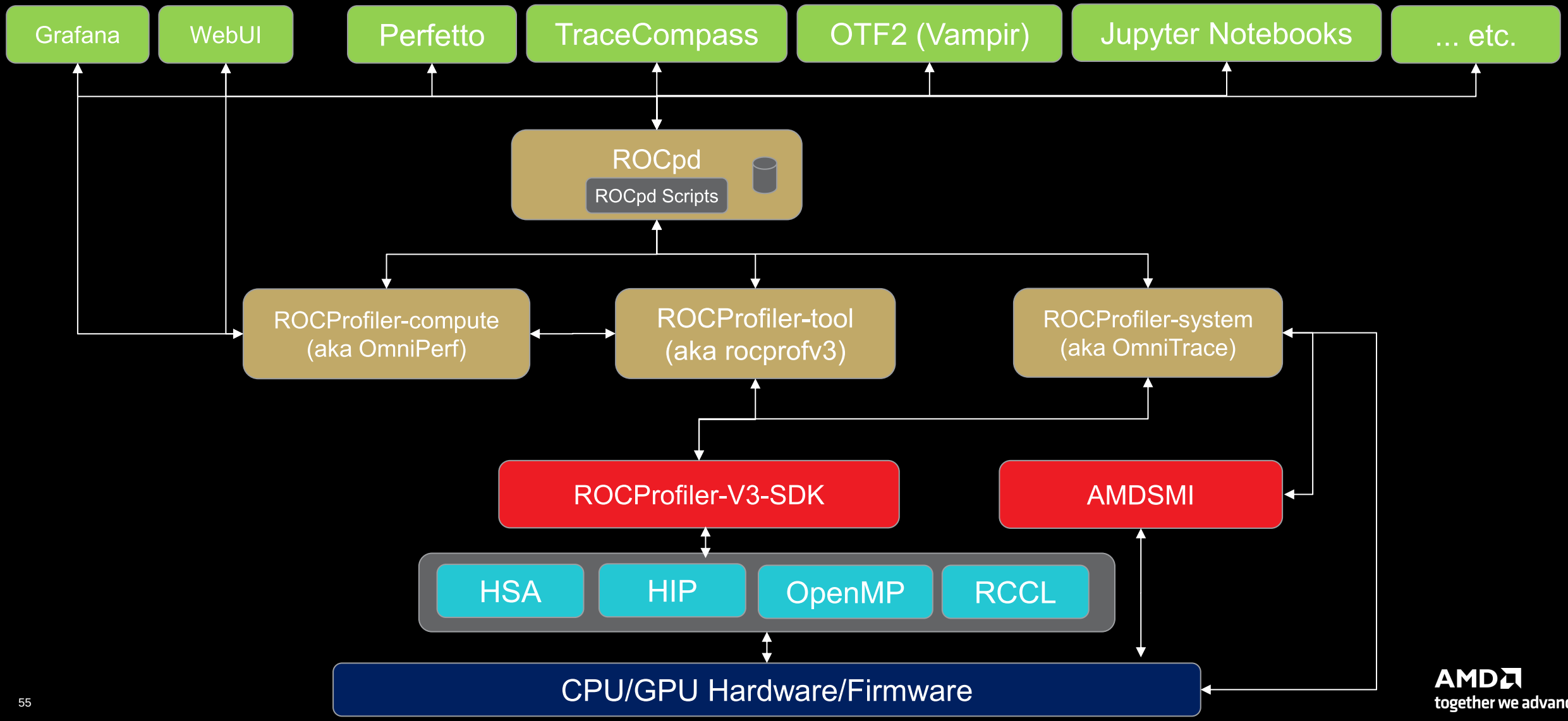
ROCm Tools Overview (Present)

Green – Visualization Tool
Gold – User Facing Tools
Red/Orange – Underlying Tools
Light Blue – SW layer
Dark Blue – Hardware/Firmware



ROCm Tools Overview (Goal)

Green – Visualization Tool
Gold – User Facing Tools
Red/Orange – Underlying Tools
Light Blue – SW layer
Dark Blue – Hardware/Firmware



ROCpd: Overview

- ROCpd is a Python package distributed by rocprofiler-sdk
  - Standalone DEB and RPM packages: rocprofiler-sdk-rocpd
  - PyPi distribution is in the roadmap (i.e. `pip install rocpd`)
- ROCpd can be utilized as a command-line tool and as a Python API
  - Users are encouraged to build and contribute analysis workflows
- ROCpd takes one or more SQL databases as input
  - Data collection tool writes raw data into a SQL database adhering to ROCpd SQL schema.
  - Data collection tool can generate as many SQL databases as necessary or just one SQL database.
- ROCpd takes multiple input SQL databases and stitches them together into a single database connection without ever modifying the original databases of raw data on disk.

Why SQL database?

- SQL (Structured Query Language):
  - A declarative language for creating, retrieving, updating, and deleting relational data.
- SQLite3:
  - A self-contained, serverless, zero-configuration, file-based RDBMS implementing SQL standards.
    - RDBMS == Relational Database Management System
- There are many SQL engines, SQLite3 is just the easiest to use.

ROCpd Database Schema - Tables

- `rocpd_metadata` general info
- `rocpd_string` string storage optimization
- `rocpd_info_node` "info" tables index
- `rocpd_info_process` details of the collected
- `rocpd_info_thread` data, e.g.,
- `rocpd_info_agent` kernel properties,
- `rocpd_info_queue` counter info, etc.
- `rocpd_info_stream` for efficient data
- `rocpd_info_pmc` storage.
- `rocpd_info_code_object`
- `rocpd_info_kernel_symbol`

- `rocpd_track` location of record (PID, TID, etc.)
- `rocpd_timestamp` timestamp storage
- `rocpd_event` general perf data
- `rocpd_arg` arguments for event
- `rocpd_pmc_event` measurement for event
- `rocpd_region` event with start + end
- `rocpd_sample` event with one timestamp
- `rocpd_kernel_dispatch` GPU kernel event
- `rocpd_memory_copy` copy event
- `rocpd_memory_allocate` alloc/free event

ROCpd Database Schema - Views

- Views below are “human readable” compositions of their `rocpd_` prefixed analogues.

- `regions`
- `samples`
- `kernels`
- `memory_copies`
- `memory_allocations`

- Views below return analysis on the current data, e.g. `top` returns regions, kernels, etc. sorted by their duration.

- `top`
- `top_kernels`
- `busy`
- `kernel_summary`
- `kernel_summary_region`
- `memory_copy_summary`
- `memory_allocation_summary`

ROCpd Database Schema

- The list of ROCpd SQL tables in one of the prior slides was a lie…
- This is an example of a real ROCpd table name:

```
rocpd_region_00000135_fbe0_7be0_9db7_25c2efbbca00
```

- What I presented to you is this view:

```
CREATE VIEW
    rocpd_region AS
SELECT
    *
FROM
    rocpd_region_00000135_fbe0_7be0_9db7_25c2efbbca00;
```

- Why? This allows ROCpd to trivially manipulate multiple databases as if they were one database

ROCpd Database Schema

```
ATTACH DATABASE rank0.db AS db0;
ATTACH DATABASE rank1.db AS db1;

CREATE TEMPORARY VIEW
    rocpd_region AS
SELECT
    *
FROM
    db0.rocpd_region_00000135_fbe0_7be0_9db7_25c2efbbca00
UNION ALL
SELECT
    *
FROM
    db1.rocpd_region_00000136_01cf_71cf_8140_d7934d48440c;
```

New Approach to Profiling Data: Summary

- Common package for post-processing ROCm profiling data
- Aggregates multi-process and multi-node data
- Reads database(s) → Generates output for visualization
- Built-in utilities for filtering, generating summaries, time-windowing, etc.
- Provides framework for building new analysis tools and workflows


AMD