AMD Developer Tools: Omnitrace & Omniperf

Suyash Tandon

Argonne Training Programme on Extreme-Scale Computing (ATPESC)
Aug 9th, 2023
Background – AMD Profilers

ROC-profiler (rocprof)

- **Hardware Counters**: Raw collection of GPU counters and traces. Counter collection with user input files. Counter results printed to a CSV.

- **Traces and timelines**: Trace collection support for CPU copy, HIP API, HSA API, GPU Kernels.

- **Visualisation**: Traces visualized with Perfetto.

References

- **ATPESC 2022** - ROC-profiler and debugger: An Overview of AMD ROCm™ Tools
- **ENCCS 2022** - Introduction to Rocprof Profiling Tools
- **AMD Lab notes** - Introduction to profiling tools for AMD hardware
Agenda – Introduction to Omni* tools

- **ROC-profiler (rocprof)**
  - Hardware Counters
  - Raw collection of GPU counters and traces
  - Counter collection with user input files
  - Counter results printed to a CSV

- **OmniTrace**
  - Trace collection
  - Comprehensive trace collection
  - Supports CPU copy, HIP API, HSA API, GPU Kernels, OpenMP®, MPI, Kokkos, p-threads, multi-GPU

- **Omniperf**
  - Performance Analysis
  - Automated collection of hardware counters
  - Analysis
  - Visualisation
  - Supports Speed of Light, Memory chart, Rooflines, Kernel comparison

- **Visualisation**
  - Traces visualized with Perfetto

---

Aug 9th, 2023

ATPESC 2023
Which profiler should we use?

Objective
- Where should I focus my time?
- How well am I using the GPU?
- Why am I seeing this performance?

Approach
- Timelines/Traces/Profiles/Causal-Profiles
- Roofline
- Hardware counters

AMD Tools
- rocprof
Which profiler should we use?

<table>
<thead>
<tr>
<th>Objective</th>
<th>Where should I focus my time?</th>
<th>How well am I using the GPU?</th>
<th>Why am I seeing this performance?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approach</td>
<td>Timelines/Traces/Profiles/Causal-Profiles</td>
<td>Roofline</td>
<td>Hardware counters</td>
</tr>
<tr>
<td>AMD Tools</td>
<td>Omnitrace</td>
<td></td>
<td>Omniperf</td>
</tr>
</tbody>
</table>

AMD Lab Notes: https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-profilers-readme/
Introduction to Omnitrace

**Omnitrace**

- **Trace collection**
  - CPU
  - GPU

- **Supports**
  - CPU copy
  - HIP API
  - HSA API
  - GPU Kernels
  - OpenMP®, MPI, Koikkos, p-threads, multi-GPU

- **Visualisation**
  - Traces visualized with Perfetto

- **ROC-profiler (rocprof)**
  - Raw collection of GPU counters and traces
  - Counter collection with user input files
  - Counter results printed to a CSV

- **Hardware Counters**

- **Traces and timelines**
  - Trace collection support for
    - CPU copy
    - HIP API
    - HSA API
    - GPU Kernels

- **Visualisation**
  - Traces visualized with Perfetto

---

*Aug 9th, 2023*  
ATPESC 2023
Omnitrace: Application Profiling, Tracing, and Analysis

Repository: https://github.com/AMDResearch/omnitrace
Not part of ROCm stack

C/C++, Fortran, Python, OpenCL™

Dynamic instrumentation, Statistical/process sampling, Causal Profiling

High-level summary, Comprehensive trace, Critical trace analysis

MPI, OpenMP®, Pthreads, HIP, HSA, Kokkos

HW counters, HSA API, HIP API, HIP trace, HSA trace, Memory & thermal

HW counters, Timing metrics, Memory access, Network, I/O, more...

Refer to current documentation for recent updates
Aug 9th, 2023
ATPESC 2023
Installation (if required)

To use pre-built binaries, select the version that matches your operating system, ROCm version, etc.

There are .rpm and .deb packages along with .sh scripts for installation.

Full documentation: https://amdresearch.github.io/omnitrace/

export OMNITRACE_VERSION=latest
export ROCM_VERSION=5.6.0
export OMNITRACE_INSTALL_DIR=/path/to/your/omnitrace/install>
wget https://github.com/AMDResearch/omnitrace/releases/${OMNITRACE_VERSION}/download/omnitrace-install.py
python3 omnitrace-install.py -p ${OMNITRACE_INSTALL_DIR} --rocm ${ROCM_VERSION}

Set up environment:
source ${OMNITRACE_INSTALL_DIR}/share/omnitrace/setup-env.sh

Note: If installing from source, remember to clone the omnitrace repo recursively
Omnitrace instrumentation Modes

**Runtime Instrumentation**
- Dynamic binary instrumentation
  - Characterize performance
  - Sample every invocation
  - Large overheads

**Sampling Instrumentation**
- Periodic sampling of entire application
  - Statistical sampling
  - Process sampling

**Basic command-line syntax:**

```
$ omnitrace [omnitrace-options] -- <CMD> <ARGS>
```

For more information or help use -h/--help/? flags:
```
$ omnitrace -h
```

Can also execute on systems using a job scheduler. For example, with SLURM, an interactive session can be used as:
```
$ srun [options] omnitrace [omnitrace-options] -- <CMD> <ARGS>
```

For problems, create an issue here: [https://github.com/AMDResearch/omnitrace/issues](https://github.com/AMDResearch/omnitrace/issues)

Documentation: [https://amdresearch.github.io/omnitrace/](https://amdresearch.github.io/omnitrace/)
# Omnitrace Configuration

```
$ omnitrace_avail --categories [options]
```

Get more information about run-time settings, data collection capabilities, and available hardware counters. For more information or help use `-h/--help flags:

```
$ omnitrace_avail -h
```

Collect information for omnitrace-related settings using shorthand `-c` for `--categories`:

```
$ omnitrace_avail -c perfetto
```

```
$ omnitrace_avail -c perfetto
```

<table>
<thead>
<tr>
<th>ENVIRONMENT VARIABLE</th>
<th>VALUE</th>
<th>CATEGORIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMNITRACE_PERFETTO_BACKEND</td>
<td>inprocess</td>
<td>custom, libomnitrace, omnitrace, perfetto</td>
</tr>
<tr>
<td>OMNITRACE_PERFETTO_BUFFER_SIZE_KB</td>
<td>1024000</td>
<td>custom, data, libomnitrace, omnitrace, perfetto</td>
</tr>
<tr>
<td>OMNITRACE_PERFETTO_FILL_POLICY</td>
<td>discard</td>
<td>custom, data, libomnitrace, omnitrace, perfetto</td>
</tr>
<tr>
<td>OMNITRACE_TRACE_DELAY</td>
<td>0</td>
<td>custom, libomnitrace, omnitrace, perfetto, profile, timemory, trace</td>
</tr>
<tr>
<td>OMNITRACE_TRACE_DURATION</td>
<td>0</td>
<td>custom, libomnitrace, omnitrace, perfetto, profile, timemory, trace</td>
</tr>
<tr>
<td>OMNITRACE_TRACE_PERIODS</td>
<td></td>
<td>custom, libomnitrace, omnitrace, perfetto, profile, timemory, trace</td>
</tr>
<tr>
<td>OMNITRACE_TRACE_PERIOD_CLOCK_ID</td>
<td>CLOCK_REALTIME</td>
<td>custom, libomnitrace, omnitrace, perfetto, profile, timemory, trace</td>
</tr>
<tr>
<td>OMNITRACE_USE_PERFETTO</td>
<td>true</td>
<td>backend, custom, libomnitrace, omnitrace, perfetto</td>
</tr>
</tbody>
</table>

Shows all runtime settings that may be tuned for perfetto
Omnitrace Configuration

Get more information about run-time settings, data collection capabilities, and available hardware counters. For more information or help use -h|--help/? flags:

```
$ omnitrace-avail --categories [options]
```

Collect information for omnitrace-related settings using shorthand -c for --categories:

```
$ omnitrace-avail -c omnitrace
```

For brief description, use the options:

```
$ omnitrace-avail -bd
```

Create a config file

Create a config file in $HOME:

```
$ omnitrace-avail -G $HOME/.omnitrace.cfg
```

To add description of all variables and settings, use:

```
$ omnitrace-avail -G $HOME/.omnitrace.cfg --all
```

Modify the config file $HOME/.omnitrace.cfg as desired to enable and change settings:

```
<snip>
OMNITRACE_USE_PERFETTO = true
OMNITRACE_USE_TIMEMORY = false
OMNITRACE_USE_SAMPLING  = true
OMNITRACE_USE_ROCTRACER = true
OMNITRACE_USE_ROCM_SMI  = true
OMNITRACE_USE_KOKKOSP  = false
OMNITRACE_USE_CAUSAL  = false
OMNITRACE_USE_MPIP    = true
OMNITRACE_USE_PID     = true
OMNITRACE_USE_ROCPROFILE = true
OMNITRACE_USE_ROCTX  = true
<snip>
```

Declare which config file to use by setting the environment:

```
$ export OMNITRACE_CONFIG_FILE=/path-to/.omnitrace.cfg
```

Contents of the config file
Dynamic Instrumentation

Runtime Instrumentation
Dynamic Instrumentation – Jacobi Example

Clone jacobi example:

```bash
$ git clone https://github.com/amd/HPCTrainingExamples.git
$ cd HPCTrainingExamples/HIP/jacobi
```

Requires ROCm and MPI install, compile:

```bash
$ make
```

Run the non-instrumented code on a single GPU as:

```bash
$ time mpirun -np 1 ./Jacobi_hip -g 1 1
real 0m3.77s
```

```
Dynamic instrumentation

```bash
$ time mpirun -np 1 omnitrace-instrument -- ./Jacobi_hip -g 1 1
real 1m45.74s
```

Extra time is the overhead of dyninst reading every binary that is loaded, not overhead of omnitrace during app execution

Parsing libraries

Functions instrumented

Outputs that will be created

Aug 9th, 2023
Dynamic Instrumentation – Jacobi Example

Clone jacobi example:

```bash
$ git clone https://github.com/amd/HPCTrainingExamples.git
$ cd HPCTrainingExamples/HIP/jacobi
```

Requires ROCm and MPI install, compile:

```bash
$ make
```

Run the non-instrumented code on a single GPU as:

```bash
$ time mpirun -np 1 ./Jacobi_hip -g 1 1
real 0m2.115s
```

Dynamic instrumentation

```bash
$ time mpirun -np 1 omnitrace-instrument -- ./Jacobi_hip -g 1 1
real 1m45.742s
```

Available functions to instrument:

```bash
$ mpirun -np 1 omnitrace-instrument -v 1 --simulate --print-available-functions -- ./Jacobi_hip -g 1 1
```

Here, -v gives a verbose output from omnitrace

Functions found in each module detected by omnitrace

- HaloExchange.cpp
  - HaloExchange.cold.21[14]
  - HaloExchange[1207]
  - _GLOBAL_sub_I_HaloExchange.cpp[8]

- Input.cpp
  - [ExtractNumber][19]
  - [FindAndClearArgument][38]
  - [ParseCommandLineArguments][206]
  - [PrintUsage][12]

- JacobiIteration.cpp
  - [JacobiIteration][71]

- JacobiMain.cpp
  - [main.cold.0][5]
  - [main][35]

- JacobiRun.cpp
  - [Jacobi::Run][155]

- JacobiSetup.cpp
  - [FormatNumber][53]
  - [Jacobi::ApplyTopology][234]
  - [Jacobi::CreateMesh][459]
  - [Jacobi::InitializeData][552]
  - [Jacobi::Jacobi::t.cold.30][15]
  - [Jacobi::Jacobi::t][1043]
  - [Jacobi::PrintResults][187]
  - [Jacobi::Jacobi::t][167]
  - [PrintPerCounter][34]
  - _GLOBAL_sub_I_JacobiSetup.cpp[8]

- std::cxx11::basic_stringbuf<char, std::char_traits<char>, std::allocator<char>> :>:-basic_stringbuf[16]

- std::cxx11::basic_stringbuf<char, std::char_traits<char>, std::allocator<char>> :>:-basic_stringbuf[18]

The simulate flag does not run the executable, but only demonstrates the available functions

Aug 9th, 2023

ATPESC 2023
Dynamic Instrumentation – Jacobi Example

Clone jacobi example:

```bash
$ git clone https://github.com/amd/HPCTrainingExamples.git
$ cd HPCTrainingExamples/HIP/jacobi
```

Requires ROCm and MPI install, compile:

```bash
$ make
```

Run the non-instrumented code on a single GPU as:

```bash
$ time mpirun -np 1 ./Jacobi_hip -g 1 1
real   0m2.115s
```

Dynamic instrumentation

```bash
$ time mpirun -np 1 omnitrace-instrument -- ./Jacobi_hip -g 1 1
real   1m45.742s
```

Available functions to instrument:

```bash
$ mpirun -np 1 omnitrace-instrument -v 1 --simulate --print-available-functions -- ./Jacobi_hip -g 1 1
```

Custom include/exclude functions* with -I or -E, resp. For e.g:

```bash
$ mpirun -np 1 omnitrace-instrument -v 1 -I
'Jacobi_t::Run' 'JacobiIteration' -- ./Jacobi_hip -g 1 1
```

Only these two functions are shown to be instrumented:

```bash
$ time mpirun -np 1 omnitrace-instrument --v 1 -- ./Jacobi_hip -g 1 1
real   1m45.742s
```

Aug 9th, 2023

ATPESC 2023
Dynamic Instrumentation

Binary Rewrite
Binary Rewrite – Jacobi Example

Generating a new executable/library with instrumentation built-in:

```bash
$ omnitrace-instrument [omnitrace-options] -o <new-name-of-exec> -- <CMD> <ARGS>
```

This new binary will have instrumented functions

Subroutine Instrumentation

Default instrumentation is main function and functions of 1024 instructions and more (for CPU)

To instrument routines with 50 or more cycles, add option "-i 50" (more overhead)

---

Aug 9th, 2023

ATPESC 2023
Binary Rewrite – Jacobi Example

Binary Rewrite

$ omnitrace-instrument [omnitrace-options] -o <new-name-of-exec> -- <CMD> <ARGS>

Generating a new /library with instrumentation built-in:

$ omnitrace-instrument -o Jacobi_hip.inst -- ./Jacobi_hip

Run the instrumented binary:

$ mpirun -np 1 omnitrace-run -- ./Jacobi_hip.inst -g 1 1

subroutine instrumentation
Default instrumentation is main function and functions of 1024 instructions and more (for CPU)

To instrument routines with 50 or more cycles, add option "-i 50" (more overhead)

Binary rewrite is recommended for runs with multiple ranks as omnitrace produces separate output files for each rank

Aug 9th, 2023
### List of Instrumented GPU Functions

```
$ cat omnitrace-Jacobi_hip.inst-output/2023-03-15_13.57/roctracer-0.txt
```

<table>
<thead>
<tr>
<th>LABEL</th>
<th>COUNT</th>
<th>DEPTH</th>
<th>METRIC</th>
<th>UNITS</th>
<th>SUM</th>
<th>MEAN</th>
<th>% SELF</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread create</td>
<td>1</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000353</td>
<td>0.000353</td>
<td>0.0</td>
</tr>
<tr>
<td>l_start thread</td>
<td>1</td>
<td>1</td>
<td>roctracer</td>
<td>sec</td>
<td>2.344864</td>
<td>2.344864</td>
<td>100.0</td>
</tr>
<tr>
<td>HipInit</td>
<td>1</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>hipGetDeviceCount</td>
<td>1</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>hipSetDevice</td>
<td>1</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>hipHostMalloc</td>
<td>3</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>hipMalloc</td>
<td>7</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>hipMemset</td>
<td>1</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>hipStreamCreate</td>
<td>2</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>hipMempy</td>
<td>1065</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>LocalLaplacianKernel(int, int, int, double, double, double, const*, double*)</td>
<td>999</td>
<td>1</td>
<td>roctracer</td>
<td>sec</td>
<td>0.279360</td>
<td>0.00280</td>
<td>100.0</td>
</tr>
<tr>
<td>HaloLaplacianKernel(int, int, int, double, double, double, const*, double*, double*)</td>
<td>990</td>
<td>1</td>
<td>roctracer</td>
<td>sec</td>
<td>0.014761</td>
<td>0.000015</td>
<td>100.0</td>
</tr>
<tr>
<td>JaccobiIterationKernel(int, double, double, const*, double*, double*, double*)</td>
<td>995</td>
<td>1</td>
<td>roctracer</td>
<td>sec</td>
<td>0.531156</td>
<td>0.008554</td>
<td>100.0</td>
</tr>
<tr>
<td>NormKernel(int, double, double, double, const*, double*, double*)</td>
<td>997</td>
<td>1</td>
<td>roctracer</td>
<td>sec</td>
<td>0.430196</td>
<td>0.004431</td>
<td>100.0</td>
</tr>
<tr>
<td>NormKernel2(int, double const*, double*)</td>
<td>999</td>
<td>1</td>
<td>roctracer</td>
<td>sec</td>
<td>0.004342</td>
<td>0.000004</td>
<td>100.0</td>
</tr>
<tr>
<td>hipEventCreate</td>
<td>2</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>hipLaunchKernel</td>
<td>5062</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>JaccobiIterationKernel(int, double, double, double, const*, double*, double*, double*)</td>
<td>1</td>
<td>1</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000552</td>
<td>0.000552</td>
<td>100.0</td>
</tr>
<tr>
<td>NormKernel(int, double, double, double, const*, double*)</td>
<td>1</td>
<td>1</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000425</td>
<td>0.000425</td>
<td>100.0</td>
</tr>
<tr>
<td>hipDeviceSynchronize</td>
<td>1001</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>NormKernel(int, double, double, double, const*, double*)</td>
<td>2</td>
<td>1</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000805</td>
<td>0.000425</td>
<td>100.0</td>
</tr>
<tr>
<td>NormKernel2(int, double const*, double*)</td>
<td>1</td>
<td>1</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000804</td>
<td>0.000804</td>
<td>100.0</td>
</tr>
<tr>
<td>HaloLaplacianKernel(int, int, int, double, double, double, const*, double*, double*, double*)</td>
<td>9</td>
<td>1</td>
<td>roctracer</td>
<td>sec</td>
<td>0.001133</td>
<td>0.000815</td>
<td>100.0</td>
</tr>
<tr>
<td>JaccobiIterationKernel(int, double, double, const*, double*, double*, double*, double*)</td>
<td>40</td>
<td>1</td>
<td>roctracer</td>
<td>sec</td>
<td>0.022204</td>
<td>0.005555</td>
<td>100.0</td>
</tr>
<tr>
<td>LocalLaplacianKernel(int, int, int, double, double, double, const*, double*, double*)</td>
<td>1</td>
<td>1</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000281</td>
<td>0.000281</td>
<td>100.0</td>
</tr>
<tr>
<td>hipEventRecord</td>
<td>2000</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>hipStreamSynchronize</td>
<td>2000</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>hipEventElapsed</td>
<td>1000</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>HaloLaplacianKernel(int, int, int, double, double, double, const*, double*, double*)</td>
<td>1</td>
<td>1</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000615</td>
<td>0.000615</td>
<td>100.0</td>
</tr>
<tr>
<td>hipFree</td>
<td>4</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
<tr>
<td>hipHostFree</td>
<td>2</td>
<td>0</td>
<td>roctracer</td>
<td>sec</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Roctracer-0.txt shows duration of HIP API calls and GPU kernels
Visualizing Trace

Use Perfetto
Copy perfetto-trace-0.proto to your laptop, go to https://ui.perfetto.dev/, Click "Open trace file", select perfetto-trace-0.proto

Traces of CPU functions

CPU metrics

Aug 9th, 2023

ATPESC 2023
# Visualizing Trace

**Use Perfetto**  
Zoom in to investigate regions of interest

![Image of Perfetto visualization]

<table>
<thead>
<tr>
<th>CPU Context Switches (S)</th>
<th>25 K</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Frequency (0) (S)</td>
<td>5 K</td>
</tr>
<tr>
<td>CPU Frequency (1) (S)</td>
<td>2.5K</td>
</tr>
<tr>
<td>CPU Frequency (2) (S)</td>
<td>2.5K</td>
</tr>
<tr>
<td>CPU Frequency (3) (S)</td>
<td>2.5K</td>
</tr>
<tr>
<td>CPU Frequency (4) (S)</td>
<td>2.5K</td>
</tr>
<tr>
<td>CPU Frequency (5) (S)</td>
<td>2.5K</td>
</tr>
<tr>
<td>CPU Frequency (6) (S)</td>
<td>2.5K</td>
</tr>
<tr>
<td>CPU Frequency (7) (S)</td>
<td>2.5K</td>
</tr>
<tr>
<td>CPU Frequency (8) (S)</td>
<td>2.5K</td>
</tr>
<tr>
<td>CPU Frequency (9) (S)</td>
<td>2.5K</td>
</tr>
<tr>
<td>CPU Frequency (10) (S)</td>
<td>2.5K</td>
</tr>
</tbody>
</table>
Visualizing Trace

Use Perfetto
Zoom in to investigate regions of interest

Flow Events
Select metrics of interest to view close together

GPU characteristics

Aug 9th, 2023
ATPESC 2023
Hardware Counters
# Hardware Counters – List All

```bash
$ mpirun -np 1 omnitrace-avail --all
```

## Components, Categories

<table>
<thead>
<tr>
<th>COMPONENT</th>
<th>AVAILABLE</th>
<th>VALUE</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>alltime</td>
<td>false</td>
<td>void</td>
<td>void</td>
<td>alltime, alltime, caliper marker, caliper loop marker, cpu_clock, cpu_util, perf::cycles, perf::cycles+</td>
</tr>
<tr>
<td>caliper</td>
<td>false</td>
<td>void</td>
<td>void</td>
<td>caliper, caliper, caliper marker, caliper loop marker, cpu_clock, cpu_util, perf::cycles, perf::cycles+</td>
</tr>
<tr>
<td>caliper</td>
<td>false</td>
<td>void</td>
<td>void</td>
<td>caliper, caliper, caliper marker, caliper loop marker, cpu_clock, cpu_util, perf::cycles, perf::cycles+</td>
</tr>
<tr>
<td>caliper</td>
<td>true</td>
<td>std::vector&lt;long, long&gt;</td>
<td>void</td>
<td>perf::cycles, cpu_utilization</td>
</tr>
<tr>
<td>cpu_clock</td>
<td>true</td>
<td>void</td>
<td>void</td>
<td>cpu_clock</td>
</tr>
<tr>
<td>cpu_util</td>
<td>true</td>
<td>void</td>
<td>void</td>
<td>cpu_util</td>
</tr>
<tr>
<td>creepset</td>
<td>false</td>
<td>void</td>
<td>void</td>
<td>creepset counters</td>
</tr>
<tr>
<td>std::vector&lt;long, long&gt;</td>
<td>void</td>
<td>void</td>
<td>void</td>
<td>creepset counters</td>
</tr>
</tbody>
</table>

## Environment Variables

<table>
<thead>
<tr>
<th>ENVIRONMENT VARIABLE</th>
<th>VALUE</th>
<th>DATA TYPE</th>
<th>DESCRIPTION</th>
<th>CATEGORIES</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMNITRACE_CAUSAL_BINARY_EXCLUDE</td>
<td>OMNITRACE CAUSAL BINARY SCOPE</td>
<td>string</td>
<td>Excludes binaries matching the list of patterns</td>
<td>analysis, causal, custom, libomitrace, o...</td>
</tr>
<tr>
<td>OMNITRACE_CAUSAL_BINARY_SCOPE</td>
<td>OMNITRACE CAUSAL BINARIES</td>
<td>string</td>
<td>Limits causal experiments to the binaries on the list</td>
<td>analysis, causal, custom, libomitrace, o...</td>
</tr>
<tr>
<td>OMNITRACE_CAUSAL_DELAY</td>
<td>OMNITRACE_CAUSAL_DURATION</td>
<td>double</td>
<td>Length of time to wait (in seconds) before performing causal experiments</td>
<td>analysis, causal, custom, libomitrace, o...</td>
</tr>
<tr>
<td>OMNITRACE_CAUSAL_FUNCTION</td>
<td>OMNITRACE_CAUSAL_FUNCTION_SCOPE</td>
<td>string</td>
<td>Limits of function scope entries for causal analysis</td>
<td>analysis, causal, custom, libomitrace, o...</td>
</tr>
<tr>
<td>OMNITRACE_CAUSAL_RANDOM_SEED</td>
<td>OMNITRACE_CAUSAL_SOURCE_EXCLUDE</td>
<td>unsigned long</td>
<td>Seed for random number generator which seeded all causal seeds</td>
<td>analysis, causal, custom, libomitrace, o...</td>
</tr>
<tr>
<td>OMNITRACE_CAUSAL_SOURCE_SCOPE</td>
<td>OMNITRACE_CAUSAL_SOURCE_SCOPE</td>
<td>string</td>
<td>Limits causal experiments to the source scope</td>
<td>analysis, causal, custom, libomitrace, o...</td>
</tr>
</tbody>
</table>

## CPU Hardware Counters

<table>
<thead>
<tr>
<th>HARDWARE COUNTER</th>
<th>AVAILABLE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>true</td>
<td>Level 1 data cache misses</td>
</tr>
<tr>
<td>PAPI_L1_DCM</td>
<td>false</td>
<td>Level 1 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L2_DCM</td>
<td>true</td>
<td>Level 2 data cache misses</td>
</tr>
<tr>
<td>PAPI_L2_DCM</td>
<td>false</td>
<td>Level 2 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L3_DCM</td>
<td>true</td>
<td>Level 3 data cache misses</td>
</tr>
<tr>
<td>PAPI_L3_DCM</td>
<td>false</td>
<td>Level 3 instruction cache misses</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>true</td>
<td>Level 1 cache misses</td>
</tr>
</tbody>
</table>

## GPU Hardware Counters

<table>
<thead>
<tr>
<th>HARDWARE COUNTER</th>
<th>AVAILABLE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCC_NORMAL_READBACK_sum:device0</td>
<td>true</td>
<td>Total number of readbacks from the video memory to the CPU.</td>
</tr>
<tr>
<td>TCC_ALL_TP_IB_WRITE_sum:device0</td>
<td>true</td>
<td>Total number of write operations to the IB.</td>
</tr>
<tr>
<td>TCC_NORMAL_WRITE_EVICTION_sum:device0</td>
<td>true</td>
<td>Total number of evictions per device.</td>
</tr>
<tr>
<td>TCC_ALL_TP_IB_EVICTION_sum:device0</td>
<td>true</td>
<td>Total number of evictions per device.</td>
</tr>
<tr>
<td>TCC_EA_READ_EVICTION_sum:device0</td>
<td>true</td>
<td>Total number of evictions due to read operations.</td>
</tr>
<tr>
<td>TCC_EA_WRITE_EVICTION_sum:device0</td>
<td>true</td>
<td>Total number of evictions due to write operations.</td>
</tr>
<tr>
<td>CPU_WIDTH</td>
<td>true</td>
<td>Width of the CPU.</td>
</tr>
<tr>
<td>WRITE_SIZE</td>
<td>device0</td>
<td>Total number of bytes written to the video memory.</td>
</tr>
<tr>
<td>WRITE_SIZE</td>
<td>device0</td>
<td>Total number of bytes written to the video memory.</td>
</tr>
<tr>
<td>GDSize</td>
<td>device0</td>
<td>Total number of bytes written to the video memory.</td>
</tr>
<tr>
<td>MemInitBusy</td>
<td>device0</td>
<td>Total number of bytes written to the video memory.</td>
</tr>
<tr>
<td>ALUStalledByDS</td>
<td>device0</td>
<td>Total number of bytes written to the video memory.</td>
</tr>
</tbody>
</table>

A very small subset of the counters shown here

---

**Aug 9th, 2023**

**ATPESC 2023**
## Commonly Used GPU Counters

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUUtilization</td>
<td>The percentage of ALUs active in a wave. Low VALUUtilization is likely due to high divergence or a poorly sized grid</td>
</tr>
<tr>
<td>VALUBusy</td>
<td>The percentage of GPUTime vector ALU instructions are processed. Can be thought of as something like compute utilization</td>
</tr>
<tr>
<td>FetchSize</td>
<td>The total kilobytes fetched from global memory</td>
</tr>
<tr>
<td>WriteSize</td>
<td>The total kilobytes written to global memory</td>
</tr>
<tr>
<td>L2CacheHit</td>
<td>The percentage of fetch, write, atomic, and other instructions that hit the data in L2 cache</td>
</tr>
<tr>
<td>MemUnitBusy</td>
<td>The percentage of GPUTime the memory unit is active. The result includes the stall time</td>
</tr>
<tr>
<td>MemUnitStalled</td>
<td>The percentage of GPUTime the memory unit is stalled</td>
</tr>
<tr>
<td>WriteUnitStalled</td>
<td>The percentage of GPUTime the write unit is stalled</td>
</tr>
</tbody>
</table>


### Modify config file

Create a config file in $HOME:

```bash
$ omnitrace-avail -G $HOME/.omnitrace.cfg
```

Modify the config file $HOME/.omnitrace.cfg to add desired metrics and for concerned GPU#ID:

```
... OMNITRACE_ROCM_EVENTS = GPUBusy:device=0, Wavefronts:device=0, MemUnitBusy:device=0 ...
```

To profile desired metrics for all participating GPUs:

```
... OMNITRACE_ROCM_EVENTS = GPUBusy, Wavefronts, MemUnitBusy ...
```
Execution with Hardware Counters

(after modifying cfg file to set up OMNITRACE_ROCM_EVENTS with GPU metrics)

```bash
$ mpirun -np 1 omnitrace-run -- ./Jacobi_hip.inst -g 1 1
```
Visualization with Hardware Counters

CPU activity

GPU hardware counters

GPU activity

ROCTX Regions

Aug 9th, 2023

ATPESC 2023
Tracing Multiple Ranks
Profiling Multiple MPI Ranks – Jacobi Example

Binary Rewrite
Generating a new /library with instrumentation built-in:

$ omnitrace-instrument -o Jacobi_hip.inst -- ./Jacobi_hip

Run the instrumented binary with 2 ranks:

$ mpirun -np 2 omnitrace-run --./Jacobi_hip.inst -g 2 1

All output files are generated for each rank
Visualizing Traces from Multiple Ranks - Separately
Visualizing Traces from Multiple Ranks - Combined

**Merge Perfetto**

Use the following command to merge and concatenate multiple traces:

```
$ cat perfetto-trace-0.proto perfetto-trace-1.proto > allprocesses.proto
```

Two processes seen in combined trace file

Zooming in helps understand load balance issues

Aug 9th, 2023
Statistical Sampling
Sampling Call-Stack (I)

OMNITRACE_USE_SAMPLING = false

OMNITRACE_USE_SAMPLING = true; OMNITRACE_SAMPLING_FREQ = 100 (100 samples per second)

Each sample shows the call stack at that time

Scroll down all the way in Perfetto to see the sampling output!
**Sampling Call-Stack (II)**

**Zoom in call-stack sampling**

<table>
<thead>
<tr>
<th>Function</th>
<th>Samples (omnitrace)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi_...</td>
<td>Jacobi_t::Run()</td>
</tr>
<tr>
<td>Norm(gr...</td>
<td>LocalLaplacian(gr...</td>
</tr>
<tr>
<td>hipMync...</td>
<td>hipLaunchKernel</td>
</tr>
<tr>
<td>hipApiN...</td>
<td>std::basic_string&lt;...</td>
</tr>
<tr>
<td>hiprtl...</td>
<td>OnUnload</td>
</tr>
<tr>
<td>hiprtl...</td>
<td>std::ostream&amp; std::...</td>
</tr>
<tr>
<td>hiprtl...</td>
<td>std::ostreambuf_it...</td>
</tr>
<tr>
<td>roctrac...</td>
<td>roctracer_disabl...</td>
</tr>
<tr>
<td>hsa_amd...</td>
<td>hsa_amd_image_ge....</td>
</tr>
</tbody>
</table>

Sampling data is annotated with (S)

Thread 0 (S) 3625610
Other Features
Kernel Durations

If you do not see a `wall_clock.txt` dumped by omnitrace, try modify the config file `$HOME/.omnitrace.cfg` and enable `OMNITRACE_USE_TIMEMORY`:

```
OMNITRACE_USE_PERFETTO = true
OMNITRACE_USE_TIMEMORY = true
OMNITRACE_USE_SAMPLING = false
```

Text file is for quick reference. JSON output is easy to script for and can be read by Hatchet, a Python package ([https://hatchet.readthedocs.io/en/latest/](https://hatchet.readthedocs.io/en/latest/))
Kernel Durations (flat profile)

Edit in your omnitrace.cfg:

<table>
<thead>
<tr>
<th>OMNITRACE_USE_TIMEMORY</th>
<th>= true</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMNITRACE_FLAT_PROFILE</td>
<td>= true</td>
</tr>
</tbody>
</table>

Use flat profile to see aggregate duration of kernels and functions
# User API

Omnitrace provides an API to control the instrumentation

<table>
<thead>
<tr>
<th>API Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int omnitrace_user_start_trace(void)</td>
<td>Enable tracing on this thread and all subsequently created threads</td>
</tr>
<tr>
<td>int omnitrace_user_stop_trace(void)</td>
<td>Disable tracing on this thread and all subsequently created threads</td>
</tr>
<tr>
<td>int omnitrace_user_start_thread_trace(void)</td>
<td>Enable tracing on this specific thread. Does not apply to subsequently created threads</td>
</tr>
<tr>
<td>int omnitrace_user_stop_thread_trace(void)</td>
<td>Disable tracing on this specific thread. Does not apply to subsequently created threads</td>
</tr>
<tr>
<td>int omnitrace_user_push_region(void)</td>
<td>Start user defined region</td>
</tr>
<tr>
<td>int omnitrace_user_pop_region(void)</td>
<td>End user defined region, FILO (first in last out) is expected</td>
</tr>
</tbody>
</table>

All the API calls: [https://amdresearch.github.io/omnitrace/user_api.html](https://amdresearch.github.io/omnitrace/user_api.html)

Aug 9th, 2023
We use the example omnitrace/examples/openmp/

Build the code with CMake:
```
$ cmake -B build
```

Use the openmp-lu binary, which can be executed with:
```
$ export OMP_NUM_THREADS=4
$ srun -n 1 -c 4 ./openmp-lu
```

Create a new instrumented binary:
```
$ cmake -DINSTRUMENT=1 ..
$ make
```

Execute the new binary:
```
$ srun -n 1 -c 4 ./openmp-lu
```

![Real-Clock Timer](image.png)

Aug 9th, 2023
OpenMP® Visualization
Python™

The omnitrace Python package is installed in 
/path/omnitrace_install/lib/pythonX.Y/site-packages/omnitrace

Setup the environment:

```
$ export PYTHONPATH=/path/omnitrace/lib/python/site-packages/:
```

We use the Fibonacci example in 
omnitrace/examples/python/source.py

Execute the python program with:

```
$ omnitrace-python ./external.py
```

Profiled data is dumped in output directory:

```
$ cat omnitrace-source-output/timestamp/wall_clock.txt
```

Python documentation: https://amdresearch.github.io/omnitrace/python.html
Visualizing Python™ Perfetto Tracing
Omnitrace can instrument Kokkos applications too.

Edit the $HOME/.omnitrace.cfg file and enable omnitrace:

```bash
OMNITRACE_USE_KOKKOSP = true
```

Profiling with omnitrace produces "kokkos*.txt" files:

```bash
$ cat kokkos_memory0.txt
```

| B>> | [kokes][dev0] kokkos::deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes][deep_copy] Host=DataBlock_A2_mirror HIP=DataBlock_A2 | 1 | 2 | kokkos_memory MB | 142 | 142 | 100 |
| B>> | [kokes][dev0] kokkos::deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes] kokkos::deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes][dev0] kokkos::deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes] kokkos::deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes][dev0] kokkos::deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes][deep_copy] Host=DataBlock_A2_mirror HIP=DataBlock_A2 | 1 | 2 | kokkos_memory MB | 140 | 140 | 100 |
| B>> | [kokes][dev0] kokkos::deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes] kokkos::deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes][dev0] kokkos::deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes] kokkos::deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes][dev0] Host=SyncToDevice() | 1 | 1 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes][deep_copy] HIP=Hydro_Vc Host=Hydro_Vc_mirror | 1 | 2 | kokkos_memory MB | 1124 | 1124 | 100 |
| B>> | [kokes][dev0] kokkos::deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes] kokkos::deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes][dev0] kokkos::deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes] kokkos::deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes][dev0] kokkos::deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes][deep_copy] HIP=Hydro_InvD Host=Hydro_InvD_mirror | 1 | 2 | kokkos_memory MB | 140 | 140 | 100 |
| B>> | [kokes][dev0] kokkos::deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes] kokkos::deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes][dev0] kokkos::deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes] kokkos::deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes][dev0] Host=Hydro_Vs Host=Hydro_Vs_mirror | 1 | 2 | kokkos_memory MB | 426 | 426 | 100 |
| B>> | [kokes][dev0] kokkos::deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes] kokkos::deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes][dev0] kokkos::deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
| B>> | [kokes] kokkos::deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory MB | 0 | 0 | 0 |
Visualizing Kokkos with Perfetto Trace

- Visualize perfetto-trace-0.proto (with sampling enabled)
Other Executables

- **omnitrace-sample**
  - For sampling with low overhead, use omnitrace-sample
  - Use omnitrace-sample --help to get relevant options
  - Settings in the OmniTrace config file will be used by omnitrace-sample
  - Example invocation to get a flat tracing profile on Host and Device (-PTHD), excluding all components (-E all) and including only rocm-smi, roctracer, rocprofiler and roctx components (-I ...)
    ```
    mpirun -np 1 omnitrace-sample -PTHD -E all -I rocm-smi -I roctracer -I rocprofiler -I roctx -- ./Jacobi_hip -g 1 1
    ```
- **omnitrace-causal**
  - Invokes causal profiling
- **omnitrace-critical-trace**
  - Post-processing tool for critical-trace data output by omnitrace
Tips & Tricks

• My Perfetto timeline seems weird how can I check the clock skew?
  • Set OMNITRACE_VERBOSE=1 or higher for verbose mode and it will print the timestamp skew
• It takes too long to map rocm-smi samples to kernels.
  • Temporarily set OMNITRACE_USE_ROCM_SMI=OFF
• What is the best way to profile multi-process runs?
  • Use OmniTrace's binary rewrite (-o) option to instrument the binary first, run the instrumented binary with mpirun/srun
• If you are doing binary rewrite and you do not get information about kernels, set:
  • HSA_TOOLS_LIB=libomnitrace.so in the env. and set OMNITRACE_USE_ROCTRACER=ON in the cfg file
• My HIP application hangs in different points, what do I do?
  • Try to set HSA_ENABLE_INTERRUPT=0 in the environment, this changes how HIP runtime is notified when GPU kernels complete
• My Perfetto trace is too big, can I decrease it?
  • Yes, with v1.7.3 and later declare OMNITRACE_PERFETTO_ANNOTATIONS to false
• I want to remove the many rows of CPU frequency lines from the Perfetto trace
  • Declare the OMNITRACE_USE_PROCESS_SAMPLING = false
Introduction to Omniperf

**ROC-profiler (rocprof)**
- Hardware Counters: Raw collection of GPU counters and traces
- Traces and timelines: Trace collection support for CPU copy, HIP API, HSA API, GPU Kernels
- Visualisation: Traces visualized with Perfetto

**Omniace**
- Trace collection: Comprehensive trace collection
- Supports: CPU copy, HIP API, HSA API, GPU Kernels, OpenMP®, MPI, Kokkos, p-threads, multi-GPU
- Visualisation: Traces visualized with Perfetto

**Omniperf**
- Performance Analysis: Automated collection of hardware counters
  - Analysis: Speed of Light, Memory chart, Rooflines, Kernel comparison
  - Visualisation: With Grafana or standalone GUI

---

Aug 9th, 2023

ATPESC 2023
Omniperf: Automated Collection of Hardware Counters and Analysis

Repository: https://github.com/AMDResearch/omnip erf

Not part of ROCm stack
Built on top of ROC-profiler

Integrated Performance Analyzer for AMD GPUs

- Speed-of-Light
- Roofline
- Memory chart
- Baseline comparison
- Sub-system performance analysis
- LDS
- v1.0
- L2 Cache
- HBM
- Shader
- Compute
- Wavefront
- Instruction mix
- Latencies

INSTINCT™ Support

- MI200
- MI100

User Interfaces

- Grafana™ GUI
- Standalone GUI
- Command Line (CLI)

Refer to current documentation for recent updates

Aug 9th, 2023
ATPESC 2023
# Omniperf features

<table>
<thead>
<tr>
<th>Omniperf Features</th>
<th>MI200 support</th>
<th>MI100 support</th>
<th>Standalone GUI Analyzer</th>
<th>Grafana/MongoDB GUI Analyzer</th>
<th>Dispatch Filtering</th>
<th>Kernel Filtering</th>
<th>GPU ID Filtering</th>
<th>Baseline Comparison</th>
<th>Multi-Normalizations</th>
<th>System Info Panel</th>
<th>System Speed-of-Light Panel</th>
<th>Kernel Statistic Panel</th>
<th>Memory Chart Analysis Panel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Roofline Analysis Panel (Supported on MI200 only, SLES 15 SP3 or RHEL8)</td>
<td>Command Processor (CP) Panel</td>
<td>Shader Processing Input (SPI) Panel</td>
<td>Wavefront Launch Panel</td>
<td>Compute Unit - Instruction Mix Panel</td>
<td>Compute Unit - Pipeline Panel</td>
<td>Local Data Share (LDS) Panel</td>
<td>Instruction Cache Panel</td>
<td>Scalar L1D Cache Panel</td>
<td>Texture Addresser and Data Panel</td>
<td>Vector L1D Cache Panel</td>
<td>L2 Cache Panel</td>
<td>L2 Cache (per-Channel) Panel</td>
</tr>
</tbody>
</table>
Client-side installation (if required)

Download the latest version from here: https://github.com/AMDResearch/omniperf/releases

Full documentation: https://amdresearch.github.io/omniperf/

```
wget https://github.com/AMDResearch/omniperf/releases/download/v1.0.8/omniperf-v1.0.8.tar.gz

tar zxvf omniperf-v1.0.8.tar.gz

cd omniperf-v1.0.8/
python3 -m pip install -t ${INSTALL_DIR}/python-libs -r requirements.txt
mkdir build

cd build
export PYTHONPATH=${INSTALL_DIR}/python

cmake -DCMAKE_INSTALL_PREFIX=${INSTALL_DIR}/1.0.8 \
-DPYTHON_DEPS=${INSTALL_DIR}/python-libs \
-DMODULE_INSTALL_PATH=${INSTALL_DIR}/modulefiles

make install
export PATH=${INSTALL_DIR}/1.0.8/bin:$PATH
```
**Omniperf modes**

<table>
<thead>
<tr>
<th>Profile</th>
<th>Target application is launched using AMD ROC profiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernels</td>
<td>Dispatches</td>
</tr>
<tr>
<td>IP Blocks</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Analyze</th>
<th>Profiled data is loaded to omniperf CLI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate access to metrics</td>
<td>Lightweight standalone GUI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Database</th>
<th>Profiled data is imported to Grafana™ database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grafana™ GUI is based on MongoDB</td>
<td>Interact with saved workload database</td>
</tr>
</tbody>
</table>

**Basic command-line syntax:**

**Profile:**

$ omniperf profile -n workload_name [profile options] [roofline options] -- <CMD> <ARGS>

**Analyze:**

$ omniperf analyze -p <path/to/workloads/workload_name/mi200/>

To use a lightweight standalone GUI with CLI analyzer:

$ omniperf analyze -p <path/to/workloads/workload_name/mi200/> --gui

**Database:**

$ omniperf database <interaction type> [connection options]

For more information or help use -h/--help/? flags:

$ omniperf profile --help

---

For problems, create an issue here: [https://github.com/AMDResearch/omniperf/issues](https://github.com/AMDResearch/omniperf/issues)

Documentation: [https://amdresearch.github.io/omniperf](https://amdresearch.github.io/omniperf)
We use the example sample/vcopy.cpp from the Omniperf installation folder:

```bash
```

Compile with hipcc:

```bash
$ hipcc -o vcopy vcopy.cpp
```

Profile with Omniperf:

```bash
$ omniperf profile -n vcopy_all -- ./vcopy 1048576 256
```

Profile only

```bash
omniperf ver: 1.0.4
Path: /pfs/lustrep4/scratch/project_462000075/markoman/omniperf-1.0.4/build/workloads
Target: mi200
Command: ./vcopy 1048576 256
Kernel Selection: None
Dispatch Selection: None
IP Blocks: All
```

A new directory will be created called workloads/vcopy_all

**Note:** Omniperf executes the code as many times as required to collect all HW metrics. Use kernel/dispatch filters especially when trying to collect roofline analysis.
Omniperf workflows

Do you have access to a Grafana server?

- Omniperf Import
  - Load Grafana page, visualize and analyze

I want only CLI

- Omniperf Analyze
  - Do CLI analysis on the terminal

I want a GUI but I have no access to a Grafana server*

- Download data
  - Omniperf Analyze with standalone GUI
  - Load the mentioned web page from your web browser, visualize and analyze

* Option to use ssh forward and not download data
Omniperf analyze

We use the example sample/vcopy.cpp from the Omniperf installation folder:

```
```

Compile with hipcc:

```
$ hipcc -o vcopy vcopy.cpp
```

Profile with Omniperf:

```
$ omniperf profile -n vcopy_all --./vcopy 1048576 256
```

A new directory will be created called workloads/vcopy_all

Analyze the profiled workload:

```
$ omniperf analyze -p workloads/vcopy_all/mi200/ &> vcopy_analyze.txt
```

---

### 7.1 Wavefront Launch Stats

<table>
<thead>
<tr>
<th>Index</th>
<th>Metric</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1.0</td>
<td>Grid Size</td>
<td>1048576.00</td>
<td>1048576.00</td>
<td>1048576.00</td>
<td>Work items</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Workgroup Size</td>
<td>256.00</td>
<td>256.00</td>
<td>256.00</td>
<td>Work items</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Total Wavefronts</td>
<td>16384.00</td>
<td>16384.00</td>
<td>16384.00</td>
<td>Wavefronts</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Saved Wavefronts</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>Wavefronts</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Restored Wavefronts</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>Wavefronts</td>
</tr>
<tr>
<td>7.1.5</td>
<td>VGP Rs</td>
<td>44.00</td>
<td>44.00</td>
<td>44.00</td>
<td>Registers</td>
</tr>
<tr>
<td>7.1.6</td>
<td>SEPRs</td>
<td>48.00</td>
<td>48.00</td>
<td>48.00</td>
<td>Registers</td>
</tr>
<tr>
<td>7.1.7</td>
<td>LDS Allocation</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>Bytes</td>
</tr>
<tr>
<td>7.1.8</td>
<td>PCIe Allocation</td>
<td>16496.00</td>
<td>16496.00</td>
<td>16496.00</td>
<td>Bytes</td>
</tr>
</tbody>
</table>

---

2. System Speed-of-Light

<table>
<thead>
<tr>
<th>Index</th>
<th>Metric</th>
<th>Value</th>
<th>Unit</th>
<th>Peak</th>
<th>Pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.0</td>
<td>VALU FLOPS</td>
<td>8.00</td>
<td>GFlop</td>
<td>23930.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2.1.1</td>
<td>VALU IOPS</td>
<td>89.14</td>
<td>Glop</td>
<td>23930.0</td>
<td>0.37</td>
</tr>
<tr>
<td>2.1.2</td>
<td>MPRA FLOPS</td>
<td>0.00</td>
<td>GFlop</td>
<td>95748.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2.1.3</td>
<td>MPRA FLOPS</td>
<td>0.00</td>
<td>GFlop</td>
<td>191488.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2.1.4</td>
<td>MPRA FLOPS</td>
<td>0.00</td>
<td>GFlop</td>
<td>67972.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2.1.5</td>
<td>MPRA FLOPS</td>
<td>0.00</td>
<td>GFlop</td>
<td>67972.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2.1.6</td>
<td>MPRA IOPS</td>
<td>0.00</td>
<td>Glop</td>
<td>191488.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2.1.7</td>
<td>Active CGU</td>
<td>58.00</td>
<td>Cpu</td>
<td>210</td>
<td>3.86</td>
</tr>
<tr>
<td>2.1.8</td>
<td>SALU UTIL</td>
<td>5.90</td>
<td>Pct</td>
<td>160</td>
<td>5.89</td>
</tr>
<tr>
<td>2.1.9</td>
<td>SALU UTIL</td>
<td>5.90</td>
<td>Pct</td>
<td>160</td>
<td>5.89</td>
</tr>
<tr>
<td>2.1.10</td>
<td>SALU UTIL</td>
<td>0.00</td>
<td>Pct</td>
<td>160</td>
<td>0.00</td>
</tr>
<tr>
<td>2.1.11</td>
<td>VALU Active Threads/Wave</td>
<td>32.71</td>
<td>Threads</td>
<td>64</td>
<td>51.18</td>
</tr>
<tr>
<td>2.1.12</td>
<td>VALU Active Channels/Wave</td>
<td>0.00</td>
<td>Channels</td>
<td>1</td>
<td>10.71</td>
</tr>
</tbody>
</table>
Omniperf Analyze

- Execute omniperf analyze -h to see various options
- Use specific IP block (-b) Example: -b 0 shows the Top Stat block shown below

Top kernels:
```bash
$ srun -n 1 --gpus 1 omniperf analyze -p workloads/vcopy_all/mi200/ -b 0
```

IP Block of wavefronts
```bash
$ srun -n 1 --gpus 1 omniperf analyze -p workloads/vcopy_all/mi200/ -b 7.1.2
```

<table>
<thead>
<tr>
<th>KernelName</th>
<th>Count</th>
<th>Sum(ns)</th>
<th>Mean(ns)</th>
<th>Median(ns)</th>
<th>Pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>vecCopy[double*, double*, double*, int, int] [clone.kd]</td>
<td>1</td>
<td>20960.00</td>
<td>20960.00</td>
<td>20960.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>

---

7. Wavefront
7.1 Wavefront Launch Stats

<table>
<thead>
<tr>
<th>Index</th>
<th>Metric</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1.2</td>
<td>Total Wavefronts</td>
<td>16384.00</td>
<td>16384.00</td>
<td>16384.00</td>
<td>Wavefronts</td>
</tr>
</tbody>
</table>

Aug 9th, 2023
Omniperf analyze

To see available options and usage instructions:

```
$ omniperf analyze -h
...
```

Help:
- `-h, --help` show this help message and exit

General Options:
- `-v, --version` show program's version number and exit
- `-V, --verbose` Increase output verbosity

Analyze Options:
- `-p [ ...], --path [ ...]` Specify the raw data root dirs or desired results directory.
- `-o, --output` Specify the output file.
- `-l, --list-kernels` List kernels. Top 10 kernels sorted by duration (descending order).
- `-m, --list-metrics` List metrics can be customized to analyze on specific arch:
  - `gfx906`
  - `gfx908`
  - `gfx920`

- `-b [ ...], --metric [ ...]` Specify IP block/metric id(s) from --list-metrics for filtering.
- `-k [ ...], --kernel [ ...]` Specify kernel id(s) from --list-kernels for filtering.
- `-d, --dispatch [ ...]` Specify dispatch id(s) for filtering.
- `-g, --gpu-id [ ...]` Specify GPU id(s) for filtering.
- `-n, --normal-unit` Specify the normalization unit: (DEFAULT: per_wave)
  - `per_wave`
  - `per_cycle`
  - `per_second`
  - `per_kernel`
- `--config-dir` Specify the directory of customized configs.
- `-t, --time-unit` Specify display time unit in kernel top stats: (DEFAULT: ns)
  - `s`
  - `ms`
  - `us`
  - `ns`
- `--decimal` Specify the decimal to display. (DEFAULT: 2)
- `-c, --cols [ ...]` Specify column indices to display.
- `-g` Debug single metric.
- `-p` List the installation dependency.
- `--gui [GUI]` Activate a GUI to interate with Omniperf metrics.

Optionally, specify port to launch application (DEFAULT: 8050)
Easy things you can check

- Are all the CUs being used?
  - If not, more parallelism is required (for most of the cases)

- Are all the VGPRs being spilled?
  - Try smaller workgroup sizes

- Is the code Integer limited?
  - Try reducing the integer ops, usually in the index calculation
Omniperf analyze with standalone GUI

We use the example sample/vcopy.cpp from the Omniperf installation folder:

```bash
```

Compile with hipcc:

```bash
$ hipcc -o vcopy vcopy.cpp
```

Profile with Omniperf:

```bash
$ omniperf profile -n vcopy_all --vcopy 1048576 256
```

A new directory will be created called `workloads/vcopy_all`

Analyze the profiled workload:

```bash
$ omniperf analyze -p workloads/vcopy_all/mi200/ --gui
```

Open web page http://IP:8050/
We use the example sample/vcopy.cpp from the Omniperf installation folder:

```
```

Compile with hipcc:

```
$ hipcc -o vcopy vcopy.cpp
```

Profile with Omniperf:

```
$ omniperf profile -n vcopy_all -- ~/vcopy 1048576 256
```

A new directory will be created called `workloads/vcopy_all`.

Import the database to analyze in Grafana™ GUI:

```
$ omniperf database --import [connection options] -w workloads/vcopy_demo/mi200/
```

ROC Profiler: `/usr/bin/rocprof`

Import Profiling Results

```
Pulling data from /root/test/workloads/vcopy_demo/mi200
The directory exists
Found sysinfo file
Kernel name verbose level: 2
Password:
Password received
-- Conversion & Upload in Progress --
... ...
9 collections added.
Workload name uploaded
-- Complete! --
```
Key Insights from Omniperf Analyzer
When assessing performance difference between two workloads (current and baseline), it’s good to know the differences between underlying systems.
Initial assessment with kernel statistics
Roofline: the first-step characterization of workload performance

<table>
<thead>
<tr>
<th>Name</th>
<th>Calls</th>
<th>Performance</th>
<th>HBM BW</th>
<th>Total Duration</th>
<th>Avg Duration</th>
<th>AI (Vector)</th>
<th>AI (L1 Cache)</th>
<th>AI (L2 Cache)</th>
<th>AI (MEM)</th>
<th>Total FLOPs</th>
<th>VALU FLOPs</th>
<th>MFMA FLOPs (F16)</th>
<th>MFMA FLOPs (BF16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>void dot_kernel-double...</td>
<td>100</td>
<td>86.5 GFLOPS</td>
<td>689 GB/s</td>
<td>244 ms</td>
<td>2.44 ms</td>
<td>0.063</td>
<td>0.136</td>
<td>0.126</td>
<td>0.126</td>
<td>210,563,552</td>
<td>210,563,552</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>void triad_kernel-doubl...</td>
<td>100</td>
<td>111 GFLOPS</td>
<td>1.33 TB/s</td>
<td>189 ms</td>
<td>1.89 ms</td>
<td>0.042</td>
<td>0.083</td>
<td>0.083</td>
<td>0.083</td>
<td>209,715,200</td>
<td>209,715,200</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>void add_kernel-doubl...</td>
<td>100</td>
<td>55.7 GFLOPS</td>
<td>1.34 TB/s</td>
<td>188 ms</td>
<td>1.88 ms</td>
<td>0.021</td>
<td>0.042</td>
<td>0.043</td>
<td>0.043</td>
<td>104,857,600</td>
<td>104,857,600</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>void copy_kernel-doubl...</td>
<td>100</td>
<td>9 GFLOPS</td>
<td>1.37 TB/s</td>
<td>122 ms</td>
<td>1.22 ms</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>void matmul_kernel-doubl...</td>
<td>100</td>
<td>86.1 GFLOPS</td>
<td>1.38 TB/s</td>
<td>122 ms</td>
<td>1.22 ms</td>
<td>0.031</td>
<td>0.063</td>
<td>0.063</td>
<td>0.063</td>
<td>104,857,600</td>
<td>104,857,600</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Background - What is a roofline?
Background – What is Roofline

- **Attainable FLOPs/s**
  - FLOPs/s rate as measured empirically on a given device
  - FLOP = floating point operation
  - FLOP counts for common operations
    - Add: 1 FLOP
    - Mul: 1 FLOP
    - FMA: 2 FLOP
  - FLOPs/s = Number of floating-point operations performed per second
Background – What is Roofline

- Arithmetic Intensity (AI)
  - characteristic of the workload indicating how much compute (FLOPs) is performed per unit of data movement (Byte)
  - Ex: $x[i] = y[i] + c$
    - FLOPs = 1
    - Bytes = $1 \times RD + 1 \times WR = 4 + 4 = 8$
    - AI = $1 / 8$

![Diagram showing attainable FLOPs/s vs. arithmetic intensity (FLOPs/Byte)]
Background – What is Roofline

- Log-Log plot
  - makes it easy to doodle, extrapolate performance along Moore's Law, etc...

![Roofline Diagram](image-url)

- Attainable FLOPs/s vs. Arithmetic Intensity (FLOPs/Byte)
Background – What is Roofline

- Roofline Limiters
  - Compute
    - Peak FLOPs/s
  - Memory BW
    - AI * Peak GB/s

- Note:
  - These are empirically measured values
  - Different SKUs will have unique plots
  - Individual devices within a SKU will have slightly different plots based on thermal solution, system power, etc.
  - Omniperf uses suite of simple kernels to empirically derive these values
  - These are NOT theoretical values indicating peak performance under “unicorn” conditions
Background – What is Roofline

- **Attainable FLOPs/s =**
  - \[ \text{min} \left( \frac{\text{Peak FLOPs/s}}{AI \times \text{Peak GB/s}} \right) \]

- **Machine Balance:**
  - Where \( AI = \frac{\text{Peak FLOPs/s}}{\text{Peak GB/s}} \)
  - Typical machine balance: 5-10 FLOPs/B
    - 40-80 FLOPs per double to exploit compute capability
  - MI250x machine balance: \( \sim 16 \) FLOPs/B
    - 128 FLOPs per double to exploit compute capability

![Diagram showing Roofline analysis](image)
Background – What is Roofline

- Attainable FLOPs/s =
  \[ \min \{ \text{Peak FLOPs/s} \} \]
  \[ \cdot \frac{\text{Peak FLOPs/s}}{\text{Peak GB/s}} \]

- Machine Balance:
  Where \( AI = \frac{\text{Peak FLOPs/s}}{\text{Peak GB/s}} \)

- Five Performance Regions:
  - Unattainable Compute

![Diagram showing the relationship between Attainable FLOPs/s, Arithmetic Intensity, HBM GB/s, and Peak FLOPs/s.](image)
Background – What is Roofline

• Attainable FLOPs/s =
  $$\min \left\{ \frac{\text{Peak FLOPs/s}}{\text{AI} \times \text{Peak GB/s}} \right\}$$

• Machine Balance:
  Where $$\text{AI} = \frac{\text{Peak FLOPs/s}}{\text{Peak GB/s}}$$

• Five Performance Regions:
  • Unattainable Compute
  • Unattainable Bandwidth

Note:
FLOP: Floating Point Operation
FLOPs: plural
FLOPS: Floating Point Operations per Second (alternately FLOPs/s)
Background – What is Roofline

- Attainable FLOPs/s =
  \[
  \min \left\{ \frac{\text{Peak FLOPs/s}}{\text{AI} \times \text{Peak GB/s}} \right\}
  \]

- Machine Balance:
  - Where \( \text{AI} = \frac{\text{Peak FLOPs/s}}{\text{Peak GB/s}} \)

- Five Performance Regions:
  - Unattainable Compute
  - Unattainable Bandwidth
  - Compute Bound
Background – What is Roofline

- Attainable FLOPs/s =
  - \( \min \left\{ \text{Peak FLOPs/s}, \frac{\text{Peak FLOPs/s}}{\text{AI} \times \text{Peak GB/s}} \right\} \)

- Machine Balance:
  - Where \( \text{AI} = \frac{\text{Peak FLOPs/s}}{\text{Peak GB/s}} \)

- Five Performance Regions:
  - Unattainable Compute
  - Unattainable Bandwidth
  - Compute Bound
  - Bandwidth Bound

![Diagram showing Arithemetic Intensity vs. Attainable FLOPs/s with regions for Unattainable Compute, Unattainable Bandwidth, Compute Bound, and Bandwidth Bound]
Background – What is Roofline

- Attainable FLOPs/s =
  \[ \min \left( \frac{\text{Peak FLOPs/s}}{\text{AI} \times \text{Peak GB/s}} \right) \]

- Machine Balance:
  \[ \text{AI} = \frac{\text{Peak FLOPs/s}}{\text{Peak GB/s}} \]

- Five Performance Regions:
  - Unattainable Compute
  - Unattainable Bandwidth
  - Compute Bound
  - Bandwidth Bound
  - Poor Performance
Background – What is Roofline

• Attainable FLOPs/s =
  • \( \min \left\{ \frac{\text{Peak FLOPs/s}}{AI \times \text{Peak GB/s}} \right\} \)

• Final result is a single roofline plot presenting the peak attainable performance (in terms of FLOPs/s) on a given device based on the arithmetic intensity of any potential workload.

• We have an application independent way of measuring and comparing performance on any platform.
Background – What is “Good” Performance?

- Example:
  - We run a number of kernels and measure FLOPs/s
Background – What is “Good” Performance?

- Example:
  - We run a number of kernels and measure FLOPs/s
  - Sort kernels by arithmetic intensity
Background – What is “Good” Performance?

- Example:
  - We run a number of kernels and measure FLOPs/s
  - Sort kernels by arithmetic intensity
  - Compare performance relative to hardware capabilities

![Graph showing relationship between arithmetic intensity and attainable FLOPs/s](image)

- Peak FLOPs/s
- Attainable FLOPs/s

Background – What is “Good” Performance?

- Example:
  - We run a number of kernels and measure FLOPs/s
  - Sort kernels by arithmetic intensity
  - Compare performance relative to hardware capabilities

![Graph showing relationship between arithmetic intensity and attainable FLOPs/s](image)

- Peak FLOPs/s
- Attainable FLOPs/s

Aug 9th, 2023

ATPESC 2023
Background – What is “Good” Performance?

- Example:
  - We run a number of kernels and measure FLOPs/s
  - Sort kernels by arithmetic intensity
  - Compare performance relative to hardware capabilities
  - Kernels near the roofline are making good use of computational resources
    - Kernels can have low performance (FLOPS/s), but make good use of BW
Background – What is “Good” Performance?

- Example:
  - We run a number of kernels and measure FLOPs/s
  - Sort kernels by arithmetic intensity
  - Compare performance relative to hardware capabilities
  - Kernels near the roofline are making good use of computational resources
    - Kernels can have low performance (FLOPS/s), but make good use of BW
  - Increase arithmetic intensity when bandwidth limited
    - Reducing data movement increases AI
  - Kernels not near the roofline *should* have optimizations that can be made to get closer to the roofline
Roofline Calculations on AMD Instinct™ MI200 GPUs
Overview - AMD Instinct™ MI200 Architecture

Graphics Compute Die (GCD)

Compute Unit
Local Data Share (LDS)
- Scalar Unit
- SIMD0
- SIMD1
- SIMD2
- SIMD3
- SGPR
- VGPR
- Vector L1 Data Cache (vL1D)

Compute Unit
Local Data Share (LDS)
- SIMD0
- SIMD1
- SIMD2
- SIMD3
- SGPR
- VGPR
- Vector L1 Data Cache (vL1D)

Compute Unit
Local Data Share (LDS)
- SIMD0
- SIMD1
- SIMD2
- SIMD3
- SGPR
- VGPR
- Vector L1 Data Cache (vL1D)

L2 Cache (L2)

Data Fabric

Memory Controller

HBM Memory

Remote Socket
(CPU, GPU)

(Peer GCD)

Aug 9th, 2023

ATPESC 2023
Empirical Hierarchical Roofline on MI200 - Overview

- Peak MFMA GFLOP/sec
- Peak VALU GFLOP/sec
- Peak HBM BW
- Peak LDS BW
- Peak vL1D BW
- Peak L2 BW
- Workload Perf: (GFLOP/sec, AI)
Empirical Hierarchical Roofline on MI200 – Roofline Benchmarking

- Empirical Roofline Benchmarking
  - Measure achievable Peak FLOPS
    - VALU: F32, F64
    - MFMA: F16, BF16, F32, F64
  - Measure achievable Peak BW
    - LDS
    - Vector L1D Cache
    - L2 Cache
    - HBM

- Internally developed micro benchmark algorithms
  - Peak VALU FLOP: axpy
  - Peak MFMA FLOP: Matrix multiplication based on MFMA intrinsic
  - Peak LDS/L1D/L2 BW: Pointer chasing
  - Peak HBM BW: Streaming copy
Empirical Hierarchical Roofline on MI200 – Perfmon counters

- **Weight**
  - ADD: 1
  - MUL: 1
  - FMA: 2
  - Transcendental: 1

- **FLOP Count**
  - VALU: derived from VALU math instructions (assuming 64 active threads)
  - MFMA: count FLOP directly, in unit of 512

- **Transcendental Instructions (7 in total)**
  - $e^x$, $\log(x)$: F16, F32
  - $\frac{1}{x}$, $\frac{1}{\sqrt{x}}$: F16, F32, F64
  - $\sin x$, $\cos x$: F16, F32

- **Profiling Overhead**
  - Require 3 application replays

---

<table>
<thead>
<tr>
<th>ID</th>
<th>HW Counter</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SQ_INSTS_VALU_ADD_F16</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>2</td>
<td>SQ_INSTS_VALU_MUL_F16</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>3</td>
<td>SQ_INSTS_VALU_FMA_F16</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>4</td>
<td>SQ_INSTS_VALU_TRANS_F16</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>5</td>
<td>SQ_INSTS_VALU_ADD_F32</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>6</td>
<td>SQ_INSTS_VALU_MUL_F32</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>7</td>
<td>SQ_INSTS_VALU_FMA_F32</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>8</td>
<td>SQ_INSTS_VALU_TRANS_F32</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>9</td>
<td>SQ_INSTS_VALU_ADD_F64</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>10</td>
<td>SQ_INSTS_VALU_MUL_F64</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>11</td>
<td>SQ_INSTS_VALU_FMA_F64</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>12</td>
<td>SQ_INSTS_VALU_TRANS_F64</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>13</td>
<td>SQ_INSTS_VALU_INT32</td>
<td>IOP counter</td>
</tr>
<tr>
<td>14</td>
<td>SQ_INSTS_VALU_INT64</td>
<td>IOP counter</td>
</tr>
<tr>
<td>15</td>
<td>SQ_INSTS_VALU_MFMA_MOPS_F16</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>16</td>
<td>SQ_INSTS_VALU_MFMA_MOPS_F16</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>17</td>
<td>SQ_INSTS_VALU_MFMA_MOPS_BF16</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>18</td>
<td>SQ_INSTS_VALU_MFMA_MOPS_F32</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>19</td>
<td>SQ_INSTS_VALU_MFMA_MOPS_F64</td>
<td>FLOP counter</td>
</tr>
<tr>
<td>20</td>
<td>SQ_LDS_IDX_ACTIVE</td>
<td>LDS Bandwidth</td>
</tr>
<tr>
<td>21</td>
<td>SQ_LDS_BANK_CONFLICT</td>
<td>LDS Bandwidth</td>
</tr>
<tr>
<td>22</td>
<td>TCP_TOTAL_CACHE_ACCESSSES_sum</td>
<td>L1D Bandwidth</td>
</tr>
<tr>
<td>23</td>
<td>TCP_TCC_WRITE_REQ_sum</td>
<td>L2 Bandwidth</td>
</tr>
<tr>
<td>24</td>
<td>TCP_TCC_ATOMIC_WITH_RET_REQ_sum</td>
<td>L2 Bandwidth</td>
</tr>
<tr>
<td>25</td>
<td>TCP_TCC_ATOMIC_WITHOUT_RET_REQ_sum</td>
<td>L2 Bandwidth</td>
</tr>
<tr>
<td>26</td>
<td>TCP_TCC_READ_REQ_sum</td>
<td>L2 Bandwidth</td>
</tr>
<tr>
<td>27</td>
<td>TCC_EA_RDREQ_sum</td>
<td>HBM Bandwidth</td>
</tr>
<tr>
<td>28</td>
<td>TCC_EA_RDREQ_32B_sum</td>
<td>HBM Bandwidth</td>
</tr>
<tr>
<td>29</td>
<td>TCC_EA_WRREQ_sum</td>
<td>HBM Bandwidth</td>
</tr>
<tr>
<td>30</td>
<td>TCC_EA_WRREQ_64B_sum</td>
<td>HBM Bandwidth</td>
</tr>
</tbody>
</table>
Total_FLOP = 64 * (SQ_INSTS_VALU_ADD_F16 + SQ_INSTS_VALU_MUL_F16 + SQ_INSTS_VALU_TRANS_F16 + 2 * SQ_INSTS_VALU_FMA_F16) 
+ 64 * (SQ_INSTS_VALU_ADD_F32 + SQ_INSTS_VALU_MUL_F32 + SQ_INSTS_VALU_TRANS_F32 + 2 * SQ_INSTS_VALU_FMA_F32) 
+ 64 * (SQ_INSTS_VALU_ADD_F64 + SQ_INSTS_VALU_MUL_F64 + SQ_INSTS_VALU_TRANS_F64 + 2 * SQ_INSTS_VALU_FMA_F64) 
+ 512 * SQ_INSTS_VALU_MFMA_MOPS_F16 
+ 512 * SQ_INSTS_VALU_MFMA_MOPS_BF16 
+ 512 * SQ_INSTS_VALU_MFMA_MOPS_F32 
+ 512 * SQ_INSTS_VALU_MFMA_MOPS_F64

Total_IOP = 64 * (SQ_INSTS_VALU_INT32 + SQ_INSTS_VALU_INT64)

LDS_{BW} = 32 * 4 * (SQ_LDS_IDX_ACTIVE – SQ_LDS_BANK_CONFLICT)

vl1D_{BW} = 64 * TCP_TOTAL_CACHE_ACCESSSES_sum

L2_{BW} = 64 * TCP_TCC_READ_REQ_sum 
+ 64 * TCP_TCC_WRITE_REQ_sum 
+ 64 * (TCP_TCC_ATOMIC_WITH_RET_REQ_sum + TCP_TCC_ATOMIC_WITHOUT_RET_REQ_sum)

HBM_{BW} = 32 * TCC_EA_RDREQ_32B_sum + 64 * (TCC_EA_RDREQ_sum - TCC_EA_RDREQ_32B_sum) 
+ 32 * (TCC_EA_WRREQ_sum – TCC_EA_WRREQ_64B_sum) + 64 * TCC_EA_WRREQ_64B_sum

* All calculations are subject to change

Aug 9th, 2023

ATPESC 2023
Empirical Hierarchical Roofline on MI200 - Manual Rocprof

- For those who like getting their hands dirty
- Generate input file
  - See example roof-counters.txt →
- Run rocprof
  
```bash
foo@bar:~$ rocprof -i roof-counters.txt --timestamp on ./myCoolApp
```
- Analyze results
  - Load `results.csv` output file in csv viewer of choice
  - Derive final metric values using equations on previous slide
- Profiling Overhead
  - Requires one application replay for each pmc line

# roof-counters.txt

- FP32 FLOPs
  - pmc: SQ_INSTS_VALU_ADD_F32 SQ_INSTS_VALU_MUL_F32 SQ_INSTS_VALU_FMA_F32 SQ_INSTS_VALU_TRANS_F32

- HBM Bandwidth
  - pmc: TCC_EA_RDREQ_sum TCC_EA_RDREQ_32B_sum TCC_EA_WRREQ_sum TCC_EA_WRREQ_64B_sum

- LDS Bandwidth
  - pmc: SQ_LDS_IDX_ACTIVE SQ_LDS_BANK_CONFLICT

- L2 Bandwidth
  - pmc: TCP_TCC_READ_REQ_sum TCP_TCC_WRITE_REQ_sum TCP_TCC_ATOMIC_WITH_RET_REQ_sum
    TCP_TCC_ATOMIC_WITHOUT_RET_REQ_sum

- vL1D Bandwidth
  - pmc: TCP_TOTAL_CACHE_ACCESSSES_sum
Omniperf Performance Analyzer (cont..)
Subsystem performance analysis

Memory subsystems
- L2 Cache
- HBM access
- LDS
- vl1D

Omniperf tooling support
- L2 Cache SOL
- L2 fabric metrics
- Per-channel statistics

Speed-of-Light: L2 Cache

L2 Util
- Cache Hit
- L2-RA Hit SW
- L2-CA Hit SW

Cache Hit Rate % (Channel 16 - 31)

L2 - Fabric Transactions

<table>
<thead>
<tr>
<th>Metric</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read BW</td>
<td>693,148,700,953</td>
<td>664,965,096,054</td>
<td>695,197,543,698</td>
<td>Bytes per Sec</td>
</tr>
<tr>
<td>Write BW</td>
<td>692,659,058,092</td>
<td>664,696,624,666</td>
<td>694,762,646,665</td>
<td>Bytes per Sec</td>
</tr>
<tr>
<td>Read (128)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Req per Sec</td>
</tr>
<tr>
<td>Read (Uncached 32...)</td>
<td>2,804,240</td>
<td>1,634,649</td>
<td>2,579,819</td>
<td>Req per Sec</td>
</tr>
<tr>
<td>Read (64B)</td>
<td>10,830,448,452</td>
<td>10,283,828,376</td>
<td>10,835,794,324</td>
<td>Req per Sec</td>
</tr>
<tr>
<td>HBM Read</td>
<td>10,830,262,679</td>
<td>10,283,794,324</td>
<td>10,836,291,922</td>
<td>Req per Sec</td>
</tr>
<tr>
<td>Write (128)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Req per Sec</td>
</tr>
<tr>
<td>Write (Uncached 32...)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Req per Sec</td>
</tr>
<tr>
<td>Write (64B)</td>
<td>10,822,805,916</td>
<td>10,377,509,917</td>
<td>10,838,780,416</td>
<td>Req per Sec</td>
</tr>
<tr>
<td>HBM Write</td>
<td>10,822,801,389</td>
<td>10,377,488,192</td>
<td>10,854,762,613</td>
<td>Req per Sec</td>
</tr>
<tr>
<td>Read Latency</td>
<td>799</td>
<td>792</td>
<td>801</td>
<td>Cycles</td>
</tr>
<tr>
<td>Write Latency</td>
<td>749</td>
<td>737</td>
<td>784</td>
<td>Cycles</td>
</tr>
<tr>
<td>Atomic Latency</td>
<td></td>
<td></td>
<td></td>
<td>Cycles</td>
</tr>
<tr>
<td>Read Stall</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>pct</td>
</tr>
<tr>
<td>Write Stall</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>pct</td>
</tr>
</tbody>
</table>

L2 - Fabric Interface Stats (Cycles "per Wave")

- HBM Stall
- Peer (00) Stall
- Remote Socket Stall
- Credit Stalls
- HBM Stall
- Peer (00) Stall
- Remote Socket Stall

Aug 9th, 2023

ATPESC 2023
# Shader compute components

## Wavefront Runtime Stats

<table>
<thead>
<tr>
<th>Metric</th>
<th>Avg</th>
<th>Min</th>
<th>Max</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Time (Nanosec)</td>
<td>6,178,998</td>
<td>6,178,719</td>
<td>6,463,519</td>
<td>ns</td>
</tr>
<tr>
<td>Kernel Time (Cycles)</td>
<td>9,007,899</td>
<td>8,905,122</td>
<td>9,137,368</td>
<td>Cycle</td>
</tr>
<tr>
<td>Inst/wavefront</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>Inst/wavefront</td>
</tr>
<tr>
<td>Wave Cycles</td>
<td>3,465</td>
<td>3,335</td>
<td>3,455</td>
<td>Cycles/wave</td>
</tr>
<tr>
<td>Dependency Wait Cycles</td>
<td>3,209</td>
<td>3,186</td>
<td>3,240</td>
<td>Cycles/wave</td>
</tr>
<tr>
<td>Issue Wait Cycles</td>
<td>165</td>
<td>112</td>
<td>193</td>
<td>Cycles/wave</td>
</tr>
<tr>
<td>Active Cycles</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>Cycles/wave</td>
</tr>
<tr>
<td>Wavefront Occupancy</td>
<td>3,190</td>
<td>3,166</td>
<td>3,210</td>
<td>Wavefronts</td>
</tr>
</tbody>
</table>

## Speed-of-Light: Compute Pipeline

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALU (FLOPS)</td>
<td>0.0%</td>
<td></td>
</tr>
<tr>
<td>MFMA-BF16 (FLOPS)</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>MFMA-F16 (FLOPS)</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>MFMA-F32 (FLOPS)</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>MFMA-F64 (FLOPS)</td>
<td>36.5%</td>
<td></td>
</tr>
<tr>
<td>MFMA-iB (IDPs)</td>
<td>0%</td>
<td></td>
</tr>
</tbody>
</table>
**Omniperf profile – Roofline only**

Profile with roofline:

```
$ omniperf profile -n roofline_case_app --roof-only -- <CMD> <ARGS>
```

Analyze the profiled workload:

```
$ omniperf analyze -p path/to/workloads/roofline_case_app/mi200 --gui
```

Open web page http://IP:8050/

---

When profile with --roof-only, a PDF with the roofline will be created. In order to see the name of the kernels, add the --kernel-names and a second PDF will be created with names for the kernel markers:

```
$ omniperf profile -n roofline_case_app --roof-only --kernel-names -- <CMD> <ARGS>
```
What if Grafana and web GUI crashes when loading performance data? (real case)
When profiling produces too large data...

- We had an application that the realistic case was dispatching 6.7 million calls to kernels.
- Executing Omniperf without any options, it would take up to 36 hours to finish while single non instrumented execution takes less than 1 hour.
- HW counters add overhead.
- We had totally around 9 GB of profiling data from 1 MPI process.
- Uploading the data to a Grafana server was crashing Grafana server and we had to reboot the service.
- Using standalone GUI was never finishing loading the data.

- Omniperf profile has an option called –k where you define which specific kernel to profile. You can define the id 0-9 of the top 10 kernels.
- This creates profiling data only for the selected kernel.
- This way you can split the profiling data to 10 executions, one per kernel:
  - You can use different resources to do the experiments in parallel (remember there can be performance variation between different GPUs).
  - You can visualize each kernel.

Profile with roofline for a specific kernel:

```
$ srun -N 1 -n 1 --tasks-per-node=1 --gpus=1 --hint=nomultithread omniperf profile -n kernel_roof -k kernel_name --roof-only -- ./binary args
```

Aug 9th, 2023
Example – DAXPY with a loop in the kernel
DAXPY – with a loop in the kernel

```c
#include <hip/hip_runtime.h>

__constant__ double a = 1.0f;

__global__
void daxpy (int n, double const* x, int incx, double* y, int incy)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n)
        for(int ll=0; ll<20; ll++)
            y[i] = a*x[i] + y[i];
}

int main()
{
    int n = 1<<24;
    std::size_t size = sizeof(double)*n;

double* d_x;
double* d_y;
hipMalloc(&d_x, size);
hipMalloc(&d_y, size);

int num_groups = (n+255)/256;
int group_size = 256;
daxpy<<<num_groups, group_size>>>(n, d_x, 1, d_y, 1);
hipDeviceSynchronize();
```
Roofline

Empirical Roofline Analysis (FP32/FP64)

- Performance: almost 330 GFLOPs
Kernel execution time and L1D Cache Accesses

<table>
<thead>
<tr>
<th>Kernel Name</th>
<th>Count</th>
<th>Sum(ns)</th>
<th>Mean(ns)</th>
<th>Median(ns)</th>
<th>Pct</th>
</tr>
</thead>
<tbody>
<tr>
<td>daxpy(int, double const*, int, double*, int) [clone .kd]</td>
<td>1.00</td>
<td>2024491.00</td>
<td>2024491.00</td>
<td>2024491.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>

16. Vector L1 Data Cache

16.1 Speed-of-Light

16.2 L1D Cache Stalls

16.3 L1D Cache Accesses
DAXPY – with a loop in the kernel - Optimized

```c
#include <hip/hip_runtime.h>

__constant__ double a = 1.0f;

__global__
void daxpy (int n, double const* __restrict__ x, int incx, double* __restrict__ y, int incy)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n)
        for(int ll=0; ll<256; ll++)
            y[i] = a*x[i] + y[i];
}

int main()
{
    int n = 1<<24;
    std::size_t size = sizeof(double)*n;

double* d_x;
double *d_y;
hipMalloc(&d_x, size);
hipMalloc(&d_y, size);

int num_groups = (n+255)/256;
int group_size = 256;
daxpy<<<num_groups, group_size>>>(n, d_x, 1, d_y, 1);
hipDeviceSynchronize();
```
Roofline - Optimized

Empirical Roofline Analysis (FP32/FP64)

- Performance: almost 2 TFLOPs
# Kernel execution time and L1D Cache Accesses - Optimized

## 6.2 times faster!
Summary – Omnitrace

- OmniTrace is a powerful tool to understand CPU + GPU activity
  - Ideal for an initial look at how an application runs

- Leverages several other tools and combines their data into a comprehensive output file
  - Some tools used are AMD uProf, rocprof, rocm-smi, roctracer, perf, etc.

- Easy to visualize traces in Perfetto

- Includes several features:
  - Dynamic Instrumentation either at Runtime or using Binary Rewrite
  - Statistical Sampling for call-stack info
  - Process sampling, monitoring of system metrics during application run
  - Causal Profiling
  - Critical Path Tracing
Summary – Omniperf

- Omniperf is an integrated performance analyzer for AMD GPUs built on ROCprofiler
- Omniperf executes the code many times to collect various hardware counters (over 100 counters default behavior)
- Using specific filtering options (kernel, dispatch ID, metric group), the overhead of profiling can be reduced
- Roofline analysis is supported on MI200 GPUs
- Omniperf shows many panels of metrics based on hardware counters,
- Typical Omniperf workflows:
  - Profile + Analyze with CLI or visualize with standalone GUI
  - Profile + Import to database and visualize with Grafana
- Omniperf targets MI100 and MI200 and future generation AMD GPUs
- Omniperf requires to use just 1 MPI process
Questions?
DISCLAIMERS AND ATTRIBUTIONS

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD’s products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18

THIS INFORMATION IS PROVIDED ‘AS IS.’ AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

© 2023 Advanced Micro Devices, Inc. All rights reserved.

AMD, the AMD Arrow logo, Radeon™, Instinct™, EPYC, Infinity Fabric, ROCm™, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

The OpenMP name and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board.