GPU Profiling:
Performance Timelines with Rocprof and Omnitrace

Suyash Tandon
HIP Lecture Series – 04
October 2nd, 2023
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

**Omnitrace**
- **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
- **Supports**
  - Speed of Light
  - Memory chart
  - Rooflines
  - Kernel comparison
- **Visualisation**
  - With Grafana or standalone GUI
Agenda – AMD Profilers with timeline profiling support

- ROC-profile (rocpof)
  - 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

- Omnitrace
  - Trace collection
  - Comprehensive trace collection
    - CPU
    - GPU
  - 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
    - Visualisation
  - Supports
    - Speed of Light
    - Memory chart
    - Rooflines
    - Kernel comparison

*Will be covered in HIP Training Series 05*
Background – AMD Profilers

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
Background – AMD Profilers

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

*Will be covered in HIP Training Series 05
Why use a timeline profile

GPU centric
• Visualize the application performance
• Understand the interactions between a program’s operations
• Reveal hidden data transfer and other implicit operations
• Understand multiple stream dependencies and performance

CPU and GPU timelines
• Understand interaction between CPU and GPU
• Understand MPI communication process
• Track memory usage
• See hardware temperatures and frequencies
What is ROC-Profiler?

• ROC-profiler (also referred to as rocprof) is the command line front-end for AMD's GPU profiling libraries
  • Repo: https://github.com/ROCm-Developer-Tools/rocprofiler

• rocprof contains the central components allowing application traces and counter collection
  • Under constant development

• Distributed with ROCm

• The output of rocprof can be visualized in the Chrome browser with Perfetto (https://ui.perfetto.dev/)
rocprof: Getting Started + Useful Flags

• To get help:
  `${ROCM_PATH}/bin/rocprof -h`

• Useful housekeeping flags:
  • --timestamp <on|off> - turn on/off gpu kernel timestamps
  • --basenames <on|off> - turn on/off truncating gpu kernel names (i.e., removing template parameters and argument types)
  • -o <output csv file> - Direct counter information to a particular file name
  • -d <data directory> - Send profiling data to a particular directory
  • -t <temporary directory> - Change the directory where data files typically created in /tmp are placed. This allows you to save these temporary files.

• Flags directing rocprofiler activity:
  • -i input<.txt|.xml> - specify an input file (note the output files will now be named input.*)
  • --hsa-trace - to trace GPU Kernels, host HSA events (more later) and HIP memory copies.
  • --hip-trace - to trace HIP API calls
  • --roctx-trace - to trace roctx markers
  • --kfd-trace - to trace GPU driver calls

• Advanced usage
  • -m <metric file> - Allows the user to define and collect custom metrics. See rocprofiler/test/tool/*.xml on GitHub for examples.
rocprof: Kernel Information

- rocprof can collect kernel(s) execution stats
  
  `$ /opt/rocm/bin/rocprof --stats --basenames on <app with arguments>`

- This will output two csv files:
  - `results.csv`: information per each call of the kernel
  - `results.stats.csv`: statistics grouped by each kernel

- Content of `results.stats.csv` provides the list of GPU kernels with their durations and percentage of total GPU time:

<table>
<thead>
<tr>
<th>Name</th>
<th>Calls</th>
<th>TotalDurationNs</th>
<th>AverageNs</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>JacobiIterationKernel</td>
<td>1000</td>
<td>556699359</td>
<td>556699</td>
<td>43.291753895270446</td>
</tr>
<tr>
<td>NormKernel1</td>
<td>1001</td>
<td>430797387</td>
<td>430367</td>
<td>33.500980655394606</td>
</tr>
<tr>
<td>LocalLaplacianKernel</td>
<td>1000</td>
<td>280014665</td>
<td>280014</td>
<td>21.77538797048817</td>
</tr>
<tr>
<td>HaloLaplacianKernel</td>
<td>1000</td>
<td>14635177</td>
<td>14635</td>
<td>1.138105281810995</td>
</tr>
<tr>
<td>NormKernel2</td>
<td>1001</td>
<td>3770718</td>
<td>3766</td>
<td>0.293236076567173</td>
</tr>
<tr>
<td>amd_rocclr fillBufferAligned.kd</td>
<td>1</td>
<td>8000</td>
<td>8000</td>
<td>0.00062212040583505</td>
</tr>
</tbody>
</table>

- In a spreadsheet viewer, it is easier to read:
rocprof: Collecting Application Traces

- rocprof can collect a variety of trace event types, and generate timelines in JSON format for use with Perfetto, currently:

<table>
<thead>
<tr>
<th>Trace Event</th>
<th>rocprof Trace Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIP API call</td>
<td>--hip-trace</td>
</tr>
<tr>
<td>GPU Kernels</td>
<td>--hip-trace</td>
</tr>
<tr>
<td>Host &lt;-&gt; Device Memory copies</td>
<td>--hip-trace</td>
</tr>
<tr>
<td>CPU HSA Calls</td>
<td>--hsa-trace</td>
</tr>
<tr>
<td>User code markers</td>
<td>--roctx-trace</td>
</tr>
</tbody>
</table>

- You can combine modes like --hip-trace --hsa-trace
- If profiling OpenMP offload code, --hsa-trace is required to show HSA activity
rocprof + Perfetto: Collecting and Visualizing Application Traces

- rocprof can collect traces
  
  ```
  $ /opt/rocm/bin/rocprof --hip-trace <app with arguments>
  ```

  This will output a `.json` file that can be visualized using the chrome browser and Perfetto (https://ui.perfetto.dev/)

Perfetto – timeline visualization tool

- Both rocprof and Omnitrace currently use Perfetto for timeline presentation
- Perfetto is a built-in timeline visualization tool in the Chrome browser
- Original purpose was to profile android applications
- Open-source application from Google®
- Accessed through https://ui.perfetto.dev/ to invoke tool
  
  * Does not go over internet
  * Works even if off-line
  * No server interaction
- Opens file and reads local profile data in several formats
rocprof + Perfetto: Collecting and Visualizing Application Traces

- rocprof can collect traces
  
  `$ /opt/rocm/bin/rocprof --hip-trace <app with arguments>`

  This will output a `.json` file that can be visualized using the chrome browser and Perfetto (https://ui.perfetto.dev/)
Perfetto: Visualizing Application Traces

- Zoom in to see individual events
- Navigate trace using WASD keys

| 8022896.2 s + | 0.268 s | 0.269 s | 0.270 s | 0.271 s | 0.272 s | 0.273 s | 0.274 s |

- CPU HIP API 2

Thread 139878

- GPU2 8

Thread 0

Thread 1

- COPY 1

Thread 0

- GPU0 6
Perfetto: Kernel Information and Flow Events

- Zoom and select a kernel, you can see the link to the HIP call launching the kernel
- Try to open the information for the kernel (button at bottom right)
## Perfetto: Kernel Information and Flow Events

### Current Selection

#### Flow Events

<table>
<thead>
<tr>
<th>Current Selection</th>
<th>Flow Events</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Slice Details

- **Name**: JacobIterationKernel(int, double, double, double const*, double const*, double*, double*)
- **Category**: null
- **Start time**: 272ms 523us 999ns
- **Duration**: 541us
- **Thread duration**: 0s (0.00%)
- **Thread**: 1
- **Process**: GPU2 8
- **Slice ID**: 57

### Preceding flows

- **Slice**: hipLaunchKernel
- **Delay**: 6us
- **Thread**: NULL (CPU HIP API 2)
- **Arguments**:
  - `BeginNs`: 8024159641088210
  - `Data`: NULL
  - `DurationNs`: 541599
  - `EndNs`: 8024159641629809
  - **Name**: JacobIterationKernel(int, double, double, double const*, double const*, double*, double*)
  - `pid`: 140096
  - `tid`: 140096
  - `dev-id`: 2
  - `queue-id`: 1
  - `stream-id`: 1

### Stream where kernel was launched in

- **Duration**: 541us

### Flow events

<table>
<thead>
<tr>
<th>Direction</th>
<th>Duration</th>
<th>Connected Slice ID</th>
<th>Connected Slice Name</th>
<th>Thread Out Name</th>
<th>Thread In</th>
<th>Process Out</th>
<th>Process In</th>
<th>Flow Category</th>
<th>Flow Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incoming</td>
<td>6us</td>
<td>52</td>
<td>hipLaunchKernel NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>CPU HIP API 2</td>
<td>GPU2 8</td>
<td>DataFlow</td>
<td>dep</td>
</tr>
</tbody>
</table>
rocprof: Collecting Application Traces with rocTX Markers and Regions

- rocprof can collect user defined regions or markers using rocTX
- Annotate code with roctx regions:
  ```
  #include <roctx.h>
  ...
  roctxRangePush("reduce_for_c");
  reduce_function();
  roctxRangePop();
  ...
  ```

- Annotate code with roctx markers:
  ```
  roctxMark("start of some code");
  // some code
  roctxMark("end of some code");
  ...
  ```

- Add roctx and roctracer libraries to link line:
  ```
  -L${ROCM_PATH}/lib -lroctx64 -lroctracer64
  ```

- Profile with --roctx-range option:
  ```
  $ /opt/rocm/bin/rocprof --hip-trace --roctx-trace <app with arguments>
  ```
rocprof: Collecting Hardware Counters

- rocprof can collect a number of hardware counters and derived counters
  - $ /opt/rocm/bin/rocprof --list-basic
  - $ /opt/rocm/bin/rocprof --list-derived

- Specify counters in a counter file. For example:
  - $ /opt/rocm/bin/rocprof -i rocprof_counters.txt <app with args>
  - $ cat rocprof_counters.txt
    
    pmc : Wavefronts VALUInsts VFetchInsts VWriteInsts VALUUtilization VALUBusy WriteSize
    pmc : SALUInsts SFetchInsts LDSInsts FlatLDSInsts GDSInsts SALUBusy FetchSize
    pmc : L2CacheHit MemUnitBusy MemUnitStalled WriteUnitStalled ALUStalledByLDS LDSBankConflict

- A limited number of counters can be collected during a specific pass of code
  - Each line in the counter file will be collected in one pass
  - You will receive an error suggesting alternative counter ordering if you have too many / conflicting counters on one line

- A csv file will be created containing all the requested counters for each invocation of every kernel
# rocprof: Commonly Used GPU Counters

<table>
<thead>
<tr>
<th>Counter</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>

Performance Counters Tips and Tricks

• GPU Hardware counters are global
  • Kernel dispatches are serialized to ensure that only one dispatch is ever in flight
  • It is recommended that no other applications are using the GPU when collecting performance counters

• Use --basenames on which will report only kernel names, leaving off kernel arguments

• How do you time a kernel’s duration?
  • $ /opt/rocm/bin/rocprof --timestamp on -i rocprof_counters.txt <app with args>
    • This produces four times: DispatchNs, BeginNs, EndNs, and CompleteNs
    • Closest thing to a kernel duration: EndNs - BeginNs
    • If you run with “--stats” the resultant results.stats.csv file will include a kernel duration column
      • Note: the duration is aggregated over repeated calls to the same kernel
rocprof: Multiple MPI Ranks

• rocprof can collect counters and traces for multiple MPI ranks
• Say you want to profile an application usually called like this:
  ```
  mpiexec -np <n> ./Jacobi_hip -g <x> <y>
  ```

• Invoke the profiler by executing:
  ```
  mpiexec -np <n> rocprof <rocprof_options> ./Jacobi_hip -g <x> <y>
  or
  srun --ntasks=n rocprof <rocprof_options> ./Jacobi_hip -g <x> <y>
  ```

• By directing output files from each rank to different directories, we can collect traces for each rank separately
  • Use a helper script for this, and run your program as shown below:
    ```
    mpiexec -np <n> helper_rocprof.sh ./Jacobi_hip -g <x> <y>
    ```

• Multi-node profiling currently isn’t supported
Profiling Multiple MPI Ranks

$cat helper_rocprof.sh

#!/bin/bash
set -euo pipefail
# depends on ROCM_PATH being set outside; input arguments are the output directory & the name
outdir="$1"
name="$2"
if [[ -n ${OMPI_COMM_WORLD_RANK} ]]; then
    # mpich
    export MPI_RANK=${OMPI_COMM_WORLD_RANK}
elif [[ -n ${MV2_COMM_WORLD_RANK} ]]; then
    # ompi
    export MPI_RANK=${MV2_COMM_WORLD_RANK}
elif [[ -n ${SLURM_PROCID} ]]; then
    export MPI_RANK=${SLURM_PROCID}
else
    echo "Unknown MPI layer detected! Must use OpenMPI, MVAPICH, or SLURM"
    exit 1
fi
rocprof="${ROCM_PATH}/bin/rocprof"
pid="$$"
outdir="${outdir}/rank_${pid}_${MPI_RANK}"
outfile="${name}_${pid}_${MPI_RANK}.csv"
${rocprof} -d ${outdir} --hsa-trace -o ${outdir}/${outfile} "${@:3}"
rocprof: Profiling Overhead

- As with every profiling tool, there is an overhead
- The percentage of the overhead depends on the profiling options used
  - For example, tracing is faster than hardware counter collection
- When collecting many counters, the collection may require multiple passes
- With rocTX markers/regions, tracing can take longer and the output may be large
  - Sometimes too large to visualize
- The more data collected, the more the overhead of profiling
  - Depends on the application and options used
Summary

• rocprof is the open source, command line AMD GPU profiling tool distributed with ROCm
• Many other tools are built over rocprof
• rocprof provides tracing of GPU kernels, HIP API, HSA API and Copy activity
• rocprof can be used to collect GPU hardware counters with additional overhead
• JSON Traces can be viewed in Perfetto UI
• Other output files are in text/CSV format
Agenda – AMD Profilers with timeline profiling support

**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

**Omni.trace**
- Comprehensive trace collection
  - CPU
  - GPU
- 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
- Visualisation
  - Speed of Light
  - Memory chart
  - Rooflines
  - Kernel comparison
- Visualisation
  - With Grafana or standalone GUI
Omnitrace: Application Profiling, Tracing, and Analysis

Repository: https://github.com/AMDResearch/omnitrace

Not part of ROCm stack

Language Support:
- C/C++
- Fortran
- Python
- OpenCL™

Data Collection Modes:
- Dynamic instrumentation
- Statistical/process sampling
- Causal Profiling

Data Analysis:
- High-level summary
- Comprehensive trace
- Critical trace analysis

Parallelism Support:
- MPI
- OpenMP®
- Pthreads
- HIP
- HSA
- Kokkos

GPU Metrics:
- HW counters
- HSA API
- HIP API
- HIP trace
- HSA trace
- Memory & thermal

CPU Metrics:
- HW counters
- Timing metrics
- Memory access
- Network
- I/O
- more...

Refer to current documentation for recent updates
**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/](https://amdresearch.github.io/omnitrace/)

```bash
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} --roc ${ROCM_VERSION}
```

Set up environment:

```bash
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/AMDR&DResearch/omnitrace/issues

Documentation: 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

<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

---

<table>
<thead>
<tr>
<th>ENVIRONMENT VARIABLE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMNITRACE_CAUSAL_BINARY_EXCLUDE</td>
<td>Excludes binaries matching the list of provided regexes from causal experiments (separated by tab, sem...</td>
</tr>
<tr>
<td>OMNITRACE_CAUSAL_BINARY_SCOPE</td>
<td>Limits causal experiments to the binaries matching the provided list of regular expressions (separated...</td>
</tr>
<tr>
<td>OMNITRACE_CAUSAL_DELAY</td>
<td>Length of time to wait (in seconds) before starting the first causal experiment</td>
</tr>
<tr>
<td>OMNITRACE_CAUSAL_FUNCTION_EXCLUDE</td>
<td>Excludes functions matching the list of provided regexes from causal experiments (separated by tab, sem...</td>
</tr>
<tr>
<td>OMNITRACE_CAUSAL_FUNCTION_SCOPE</td>
<td>List of (&lt;function&gt;) regex entries for causal profiling (separated by tab, semi-colon, and/or quotes ({)...</td>
</tr>
<tr>
<td>OMNITRACE_CAUSAL_RANDOM_SEED</td>
<td>Seed for random number generator which selects speedups and experiments -- please note that random lines...</td>
</tr>
<tr>
<td>OMNITRACE_CAUSAL_SOURCE_EXCLUDE</td>
<td>Excludes source files or source file + linem pair (i.e. &lt;file&gt; or &lt;file&gt;&lt;lin&gt; ) matching the list of...</td>
</tr>
<tr>
<td>OMNITRACE_CAUSAL_SOURCE_SCOPE</td>
<td>Limits causal experiments to the source files or source file + linem pair (i.e. &lt;file&gt; or &lt;file&gt;&lt;lin&gt;...</td>
</tr>
<tr>
<td>OMNITRACE_CONFIG_FILE</td>
<td>Configuration file for omnitrace</td>
</tr>
<tr>
<td>OMNITRACE_CRITICAL_TRACE</td>
<td>Enable generation of the critical trace</td>
</tr>
<tr>
<td>OMNITRACE_OUTPUT_PATH</td>
<td>Activation state of timemory</td>
</tr>
<tr>
<td>OMNITRACE_OUTPUT_PREFIX</td>
<td>Explicitly specify the output folder for results</td>
</tr>
<tr>
<td>OMNITRACE_PAPI_EVENTS</td>
<td>PAPI events and events to collect (see also: papi avail)</td>
</tr>
<tr>
<td>OMNITRACE_PERFETTO_BACKEND</td>
<td>Specify the perfetto backend to activate. Options are: 'lpsprocess', 'system', or 'all'</td>
</tr>
<tr>
<td>OMNITRACE_PERFETTO_BUFFER_SIZE_KB</td>
<td>Size of perfetto buffer (in KB)</td>
</tr>
<tr>
<td>OMNITRACE_PERFETTO_FILL_POLICY</td>
<td>Behavior when perfetto buffer is full. ‘discard’ will ignore new entries, ‘ring buffer’ will overwrite...</td>
</tr>
<tr>
<td>OMNITRACE_PROCESS_SAMPLING DURATION</td>
<td>Number of measurements per second when OMNITRACE_USE_PROCESS_SAMPLING=ON. If set to zero, uses OMNITR...</td>
</tr>
<tr>
<td>OMNITRACE_RUCH EVENTS</td>
<td>NROC hardware counters. Use ‘:device=n’ syntax to specify collection on device number (n), e.g. ‘...</td>
</tr>
<tr>
<td>OMNITRACE_SAMPLING_CPU0</td>
<td>Number of objects per second when OMNITRACE_USE_SAMPLING=ON. Values should be separated by commas and can be expli...</td>
</tr>
<tr>
<td>OMNITRACE_SAMPLING_FREQ</td>
<td>Time (in seconds) to wait before the first sampling signal is delivered, increasing this value can fix...</td>
</tr>
<tr>
<td>OMNITRACE_SAMPLING_DURATION</td>
<td>Devices to query when OMNITRACE_USE_RUCHishops=ON. Values should be separated by commas and can be expli...</td>
</tr>
<tr>
<td>OMNITRACE_SAMPLING_FREQ</td>
<td>Time (in seconds) to sample before stopping</td>
</tr>
<tr>
<td>OMNITRACE_SAMPLING_FREQ</td>
<td>Number of objects per second when OMNITRACE_USE_SAMPLING=ON. Values should be separated by commas and can be expli...</td>
</tr>
<tr>
<td>OMNITRACE_SAMPLING_FREQ</td>
<td>Time (in seconds) to wait before the first sampling signal is delivered, increasing this value can fix...</td>
</tr>
</tbody>
</table>

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 = true
OMNITRACE_USE_SAMPLING = false
OMNITRACE_USE_ROCTRACER = true
OMNITRACE_USE_ROOM_SMI = true
OMNITRACE_USE_KOKKOSP = false
OMNITRACE_USE_CAUSAL = false
OMNITRACE_USE_MP1P = true
OMNITRACE_USE_PID = true
OMNITRACE_USE_ROCTRACER = 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:

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

Requires ROCm and MPI install, compile:

$ make

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

$ time ./Jacobi_hip -g 1 1
real 0m2.115s

Dynamic instrumentation

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

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
Dynamic Instrumentation – Jacobi Example

Clone jacobi example:

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

Requires ROCm and MPI install, compile:

$ make

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

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

Dynamic instrumentation

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

Available functions to instrument:

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

Here, -v gives a verbose output from omnitrace

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

Functions found in each module detected by omnitrace

- HaloExchange.cpp:
  - HaloExchange.cold.21
  - HaloExchange
  - _GLOBAL_sub_I_HaloExchange.cpp

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

- JacobiIteration.cpp:
  - [JacobiIteration][71]

- JacobiMain.cpp:
  - [main.cold.8][5]
  - [main][35]

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

- JacobiSetup.cpp:
  - [FormatNumber][53]
  - [Jacobi_t::ApplyTopology][234]
  - [Jacobi_t::CreateMesh][459]
  - [Jacobi_t::InitializeData][552]
  - [Jacobi_t::Jacobi_t.cold.38][15]
  - [Jacobi_t::Jacobi_t][1043]
  - [Jacobi_t::PrintResults][107]
  - [Jacobi_t::Jacobi_t][167]
  - [PrintPerfCounter][34]
  - [GLOBAL_sub_I_JacobiSetup.cpp][8]

- std::cxx11::basic_stringbuf<char, std::char_traits<char>, std::allocator<char>> ::::basic_stringbuf<char, std::char_traits<char>, std::allocator<char>>
Dynamic Instrumentation – Jacobi Example

Clone jacobi example:

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

Requires ROCm and MPI install, compile:

$ make

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

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

Dynamic instrumentation

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

Available functions to instrument:

$ 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:

$ 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
Dynamic Instrumentation

Binary Rewrite
Binary Rewrite – Jacobi Example

Binary Rewrite

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

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

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

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)

Path to new instrumented binary
Binary Rewrite – Jacobi Example

Binary Rewrite

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

Generating a new /library with instrumentation built-in:

$ omitractrace-instrument -o JacobiHip.inst -- ./Jacobi_hip

Run the instrumented binary:

$ mpirun -np 1 omitractrace-run -- ./JacobiHip.inst -g l 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 omitractrace produces separate output files for each rank

Generates traces for application run
### List of Instrumented GPU Functions

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

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
Visualizing Trace

Use Perfetto
Zoom in to investigate regions of interest
Visualizing Trace

Use Perfetto
Zoom in to investigate regions of interest
Hardware Counters
$ mpirun -np 1 omnitrace-avail --all

A very small subset of the counters shown here
Commonly Used GPU Counters

<table>
<thead>
<tr>
<th>Counter</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:

```
$ 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
```

Full list at: https://github.com/ROCm-Developer-Tools/rocprofiler/blob/amd-master/test/tool/metrics.xml
Execution with Hardware Counters

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

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

[Image] GPU hardware counters
Visualization with Hardware Counters

CPU activity

GPU hardware counters

GPU activity

ROCTX Regions
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
Sampling Call-Stack (I)

OMNITRACE_USE_SAMPLING = false

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

Scroll down all the way in Perfetto to see the sampling output!

Each sample shows the call stack at that time.
## Sampling Call-Stack (II)

### Zoom in call-stack sampling

![Sampling Call-Stack Diagram]

**Thread 0 (S) 3625610**

Sampling data is annotated with (S)
Kernel Durations

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

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
```

Call Stack

```
<table>
<thead>
<tr>
<th>Call Stack</th>
<th>Durations</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0) FP Allreduce</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
<tr>
<td>(0) __libc_DeriveSynchronize</td>
<td>1</td>
</tr>
</tbody>
</table>
```

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/).
Kernel Durations (flat profile)

Edit in your omnitrace.cfg:

OMNITRACE_USE_TIMEMORY = true
OMNITRACE_FLAT_PROFILE = true

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)
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:
```
$ srun -n 1 omnitrace-instrument -o openmp-lu.inst -- ./openmp-lu
```

Execute the new binary:
```
$ srun -n 1 -c 4 omnitrace-run -- ./openmp-lu.inst
```
OpenMP® Visualization
The omnitrace Python package is installed in `/path/omnitrace_install/lib/pythonX.Y/site-packages/omnitrace`

Setup the environment:

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

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

Execute the python program with:

```bash
$ omnitrace-python ./external.py
$ cat omnitrace-source-output/timestamp/wall_clock.txt
```

Python documentation: https://amdresearch.github.io/omnitrace/python.html
Kokkos

Omnitrace can instrument Kokkos applications too.

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

```
OMNITRACE_USE_KOKKOSP = true
```

Profiling with omnitrace produces *kokkos*.txt files:

```
$ cat kokkos_memory0.txt
```

```
| B> | [kokkos][devU] kokkos:deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory | MB | 0 | 0 | 0 |
| B> | [kokkos] kokkos:deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory | MB | 0 | 0 | 0 |
| B> | [kokkos][deep_copy] Host=DataBlock_dv_mirror HIP=DataBlock_dv | 1 | 2 | kokkos_memory | MB | 142 | 142 | 100 |
| B> | [kokkos][devU] kokkos:deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory | MB | 0 | 0 | 0 |
| B> | [kokkos] kokkos:deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory | MB | 0 | 0 | 0 |
| B> | [kokkos] kokkos:deep_copy: copy between contiguous views, post deep copy fence | 1 | 3 | kokkos_memory | MB | 0 | 0 | 0 |
| B> | [kokkos] Host=DataBlock_dv_mirror HIP=DataBlock_dv | 1 | 2 | kokkos_memory | MB | 140 | 140 | 100 |
| B> | [kokkos][devU] kokkos:deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory | MB | 0 | 0 | 0 |
| B> | [kokkos] kokkos:deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory | MB | 0 | 0 | 0 |
| B> | [kokkos][deep_copy] Host=DataBlock_dv_mirror HIP=DataBlock_dv | 1 | 2 | kokkos_memory | MB | 1124 | 1124 | 100 |
| B> | [kokkos][devU] kokkos:deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory | MB | 0 | 0 | 0 |
| B> | [kokkos] kokkos:deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory | MB | 0 | 0 | 0 |
| B> | [kokkos][deep_copy] HIP=Hydro_Vc Host=Hydro_Vc_mirror | 1 | 2 | kokkos_memory | MB | 140 | 140 | 100 |
| B> | [kokkos][devU] kokkos:deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory | MB | 0 | 0 | 0 |
| B> | [kokkos] kokkos:deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory | MB | 0 | 0 | 0 |
| B> | [kokkos][deep_copy] HIP=Hydro_Vs Host=Hydro_Vs_mirror | 1 | 2 | kokkos_memory | MB | 426 | 426 | 100 |
| B> | [kokkos][devU] kokkos:deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory | MB | 0 | 0 | 0 |
| B> | [kokkos] kokkos:deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory | MB | 0 | 0 | 0 |
| B> | [kokkos][deep_copy] HIP=Hydro_Vs Host=Hydro_Vs_mirror | 1 | 2 | kokkos_memory | MB | 426 | 426 | 100 |
| B> | [kokkos][devU] kokkos:deep_copy: copy between contiguous views, pre view equality check | 1 | 3 | kokkos_memory | MB | 0 | 0 | 0 |
| B> | [kokkos]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 ...)
    
    ```bash
    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

Current documentation: [https://amdresearch.github.io/omnitrace/development.html#executables](https://amdresearch.github.io/omnitrace/development.html#executables)
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`.
Summary

- 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
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.