Introduction to ROC-Profiler (`rocprof`)

Gina Sitaraman, Suyash Tandon, Justin Chang, Julio Maia, Noel Chalmers, Paul T. Bauman, Nicholas Curtis, Nicholas Malaya, Alessandro Fanfarillo, Jose Noudohouenou, Chip Freitag, Damon McDougall, Noah Wolfe, Jakub Kurzak, Samuel Antao, George Markomanolis, Bob Robey, Essam Morsi

Comprehensive General LUMI Course
April 23-26th, 2024

together we advance_

slides on LUMI in /project/project_465001098/Slides/AMD/

hands-on exercises: https://hackmd.io/@gmarkoma/lumi_finland

hands-on source code: /project/project_465001098/Exercises/AMD/HPCTrainingExamples/

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/)

- There are ROCProfiler V1 and V2 (roctracer and rocprofiler into single library, same API)

- A new rocprofiler-sdk is going to be released soon, the repository is public: https://github.com/ROCm/rocprofiler-sdk development is **still** going on, no version is released yet

Background – AMD Profilers

ROC-profiler (rocprof)

Hardware Counters

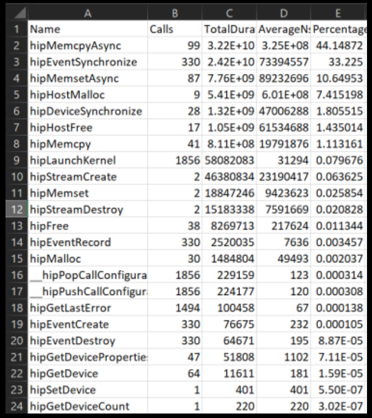
- Raw collection of GPU counters and traces
- Counter collection with user input files
- Counter results printed to a CSV

Traces and timelines

- Trace collection support for
- CPU copy
- HIP API
- HSA API
- GPU Kernels

Visualisation

- Traces visualized with Perfetto



	A	B	C	D	E
1	Name	Calls	TotalDura	AverageN	Percentage
2	hipMemcpyAsync	99	3.22E+10	3.25E+08	44.14872
3	hipEventSynchronize	330	2.42E+10	73394557	33.225
4	hipMemsetAsync	87	7.76E+09	89232696	10.64953
5	hipHostMalloc	9	5.41E+09	6.01E+08	7.415198
6	hipDeviceSynchronize	28	1.32E+09	47006288	1.805515
7	hipHostFree	17	1.05E+09	61534688	1.435014
8	hipMemcpy	41	8.11E+08	19791876	1.113161
9	hipLaunchKernel	1856	58082083	31294	0.079676
10	hipStreamCreate	2	46380834	23190417	0.063625
11	hipMemset	2	18847246	9423623	0.025854
12	hipStreamDestroy	2	15183338	7591669	0.020828
13	hipFree	38	8269713	217624	0.011344
14	hipEventRecord	330	2520035	7636	0.003457
15	hipMalloc	30	1484804	49493	0.002037
16	__hipPopCallConfigura	1856	229159	123	0.000314
17	__hipPushCallConfigur	1856	224177	120	0.000308
18	hipGetLastError	1494	100458	67	0.000138
19	hipEventCreate	330	76675	232	0.000105
20	hipEventDestroy	330	64671	195	8.87E-05
21	hipGetDevicePropertie	47	51808	1102	7.11E-05
22	hipGetDevice	64	11611	181	1.59E-05
23	hipSetDevice	1	401	401	5.50E-07
24	hipGetDeviceCount	1	220	220	3.02E-07

Omnitrace

Trace collection

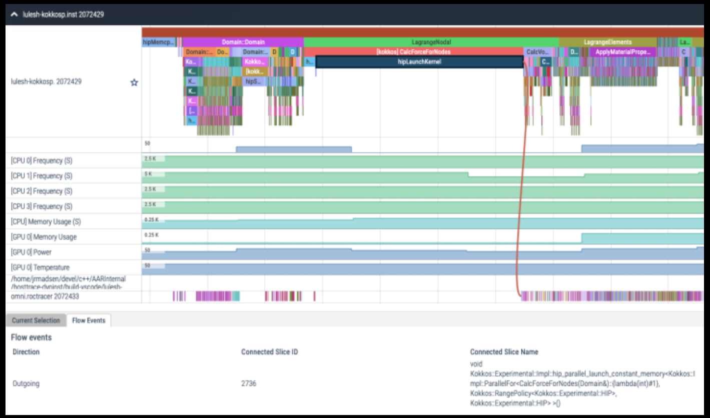
- Comprehensive trace collection
- CPU
- GPU

Supports

- CPU copy
- HIP API
- HSA API
- GPU Kernels
- OpenMP®
- MPI
- Kokkos
- p-threads
- multi-GPU

Visualisation

- Traces visualized with Perfetto



Flow events

Direction	Connected Slice ID	Connected Slice Name
Outgoing	2736	void Kokkos::Experimental::Impl::hip_parallel_launch_constant_memory<Kokkos::Impl::ParallelFor<CalcForceForNodes(Domain&)::{lambda(int)#1}, Kokkos::RangePolicy<Kokkos::Experimental::HIP>, Kokkos::Experimental::HIP> >()

Omniperf

Performance Analysis

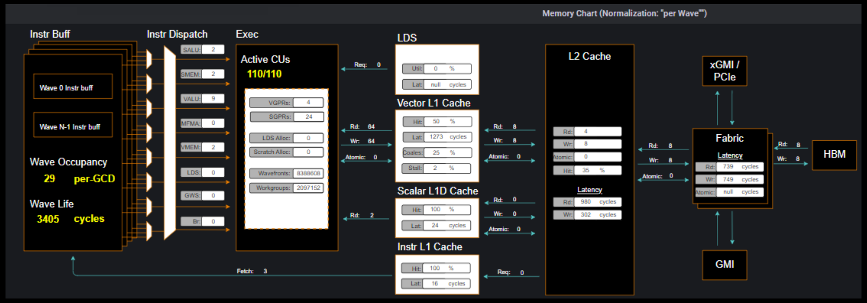
- Automated collection of hardware counters
- Analysis
- Visualization

Supports

- Speed of Light
- Memory chart
- Rooflines
- Kernel comparison

Visualisation

- With Grafana or standalone GUI



Wave Occupancy 29 per-GCD

Wave Life 3405 cycles

SALU: 2
SMEM: 2
VALU: 9
MFMA: 0
VMEM: 2
LDS: 0
GWS: 0
Br: 0

Active CUs 110/110

VGPRs: 4
SGPRs: 24
LDS Alloc: 0
Scratch Alloc: 0
Wavefronts: 8388608
Workgroups: 2097152

Req: 0
Rd: 64
Wr: 64
Atomic: 0
Rd: 2
Fetch: 3

LDS
Util: 0 %
Lat: null cycles

Vector L1 Cache
Hit: 50 %
Lat: 1273 cycles
Coales: 25 %
Stall: 2 %

Scalar L1D Cache
Hit: 100 %
Lat: 24 cycles

Instr L1 Cache
Hit: 100 %
Lat: 16 cycles

Rd: 8
Wr: 8
Atomic: 0
Rd: 0
Wr: 0
Atomic: 0
Req: 0

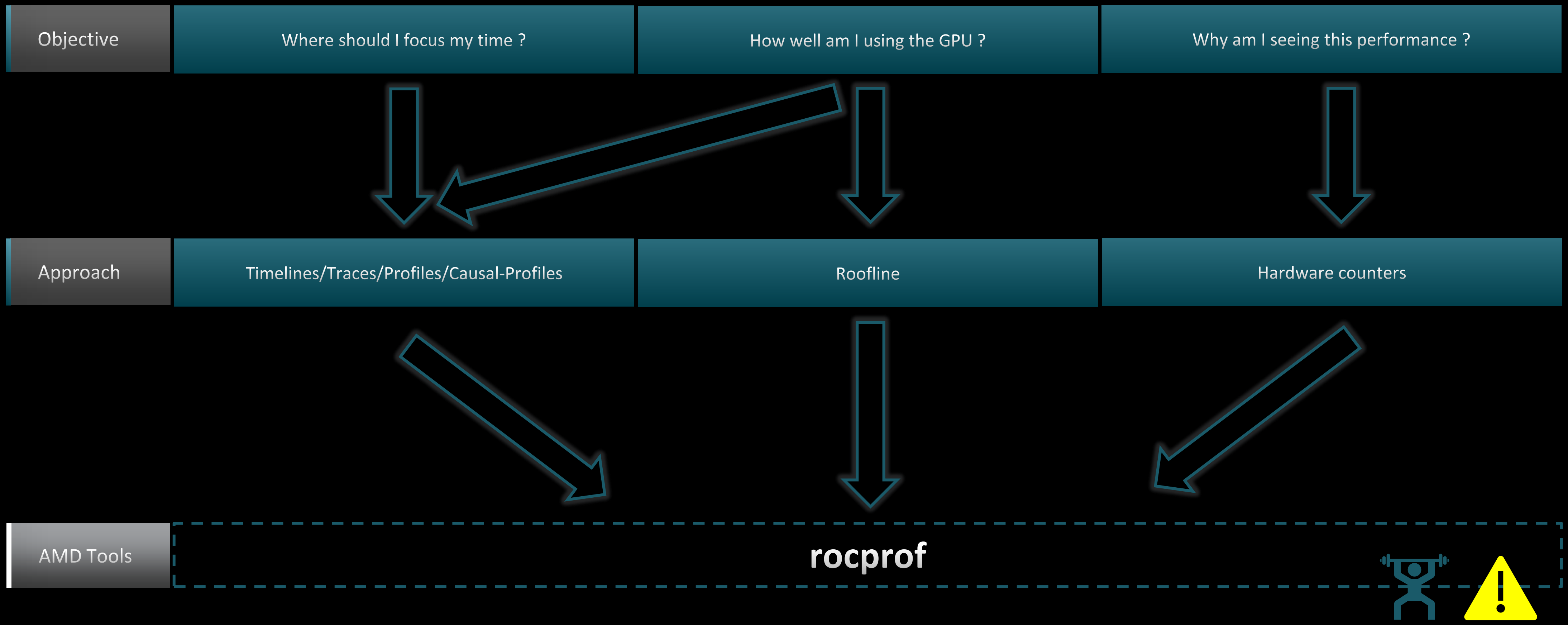
L2 Cache
Rd: 4
Wr: 8
Atomic: 0
Hit: 35 %
Latency
Rd: 980 cycles
Wr: 302 cycles

Rd: 8
Wr: 8
Atomic: 0

Fabric
Latency
Rd: 739 cycles
Wr: 749 cycles
Atomic: null cycles

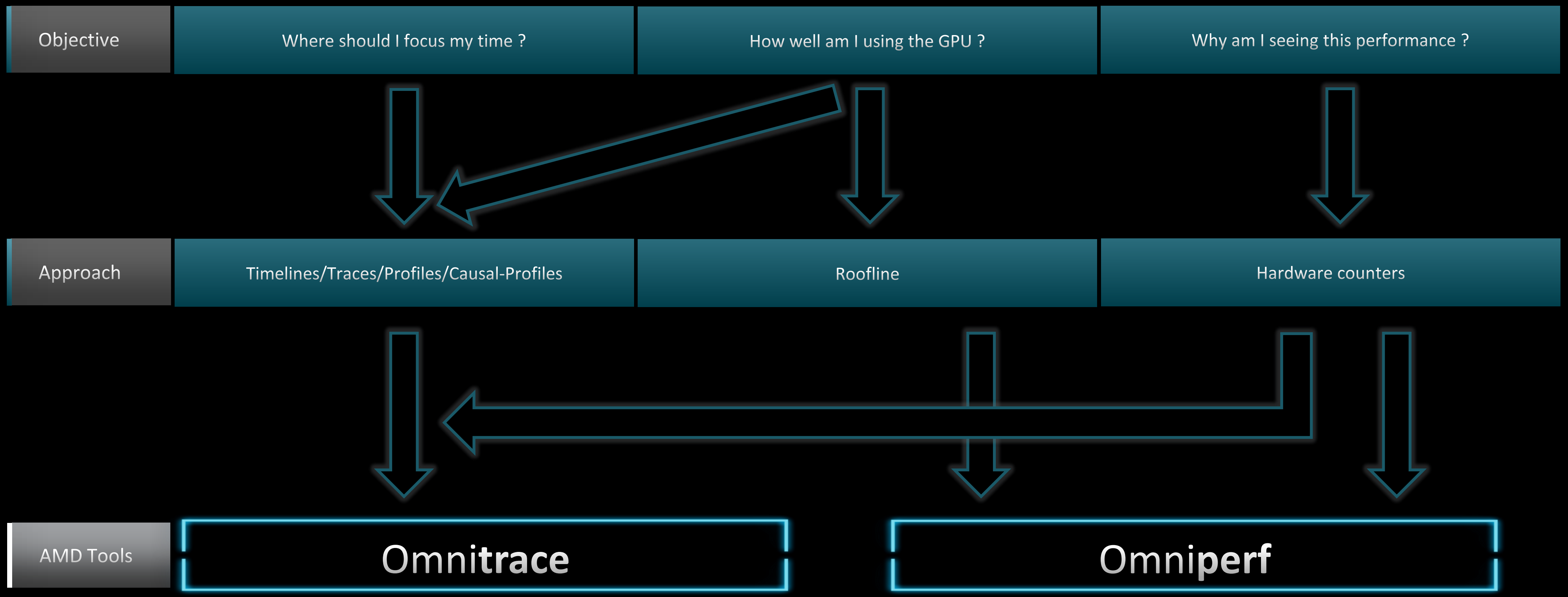
Rd: 8
Wr: 8

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



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

Omnitrace

Omniperf

rocprof: Getting Started + Useful Flags

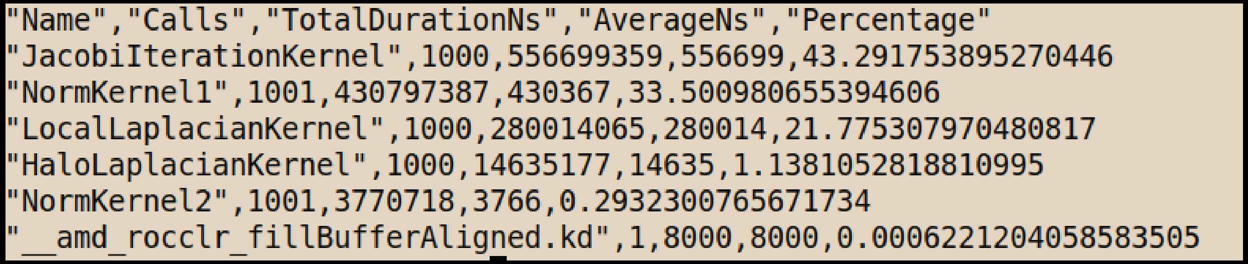
- To get help:
  `${ROCM_PATH}/bin/rocprof -h`
- Useful housekeeping flags:
  - `--timestamp <on|off>` - turn on/off gpu kernel timestamps
  - `--basenames <on|off>` - turn on/off truncating gpu kernel names (i.e., removing template parameters and argument types)
  - `-o <output csv file>` - Direct counter 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

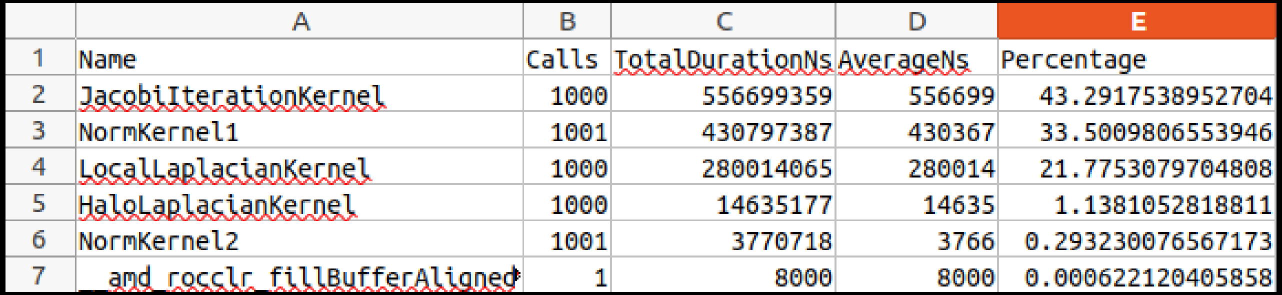
```
$ /opt/rocm/bin/rocprof --stats --basenames on <app with arguments>
```

- 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 to see the list of GPU kernels with their durations and percentage of total GPU time:



```
"Name","Calls","TotalDurationNs","AverageNs","Percentage"
"JacobiIterationKernel",1000,556699359,556699,43.291753895270446
"NormKernel1",1001,430797387,430367,33.500980655394606
"LocalLaplacianKernel",1000,280014065,280014,21.775307970480817
"HaloLaplacianKernel",1000,14635177,14635,1.1381052818810995
"NormKernel2",1001,3770718,3766,0.2932300765671734
"__amd_rocclr_fillBufferAligned.kd",1,8000,8000,0.0006221204058583505
```

- In a spreadsheet viewer, it is easier to read:



	A	B	C	D	E
1	Name	Calls	TotalDurationNs	AverageNs	Percentage
2	JacobiIterationKernel	1000	556699359	556699	43.2917538952704
3	NormKernel1	1001	430797387	430367	33.5009806553946
4	LocalLaplacianKernel	1000	280014065	280014	21.7753079704808
5	HaloLaplacianKernel	1000	14635177	14635	1.1381052818811
6	NormKernel2	1001	3770718	3766	0.293230076567173
7	__amd_rocclr_fillBufferAligned	1	8000	8000	0.000622120405858

rocprof: Collecting Application Traces

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

Trace Event	rocprof Trace Mode
HIP API call	`--hip-trace`
GPU Kernels	`--hip-trace`
Host <-> Device Memory copies	`--hip-trace`
CPU HSA Calls	`--hsa-trace`
User code markers	`--roctx-trace`

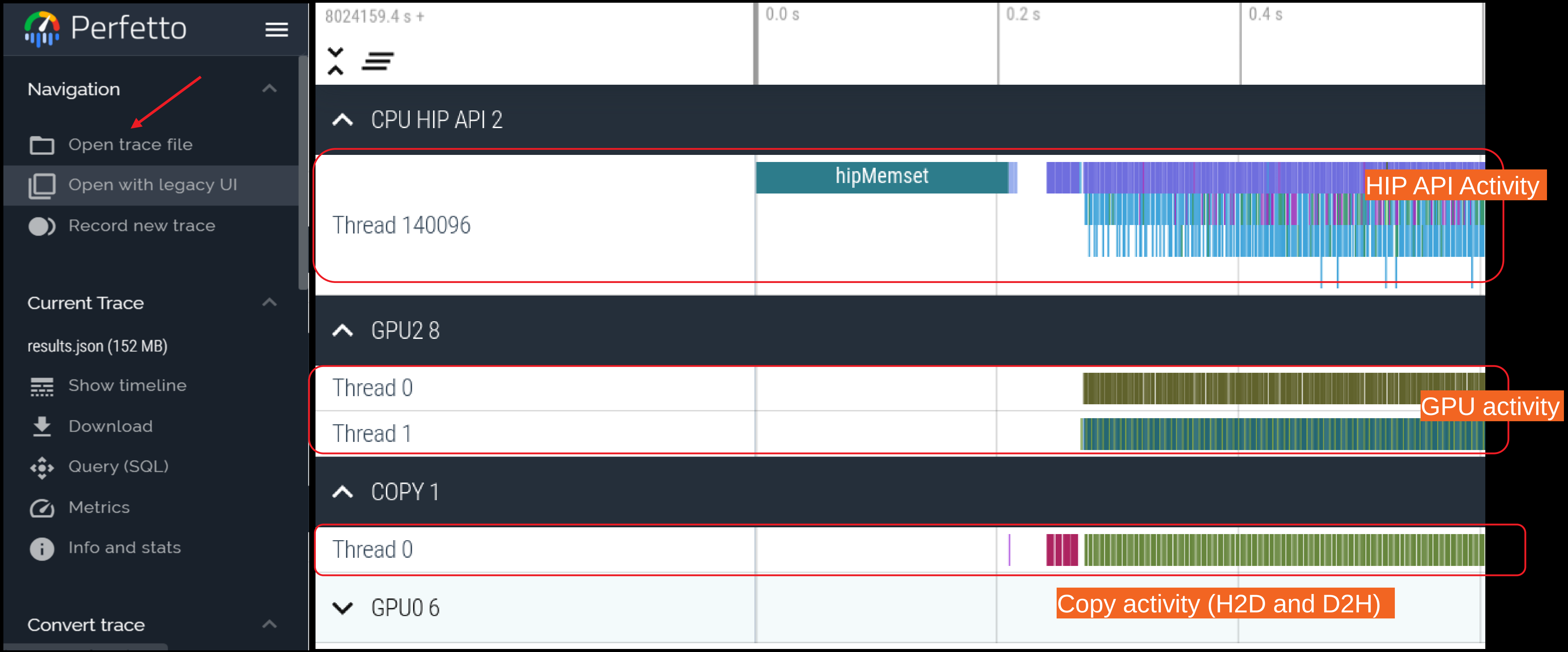
- You can combine modes like `--hip-trace --hsa-trace`
- If profiling OpenMP® offload code, `--hsa-trace` is required to show HSA activity

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/)



HIP API Activity

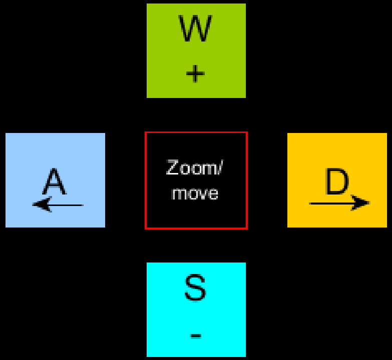
GPU activity

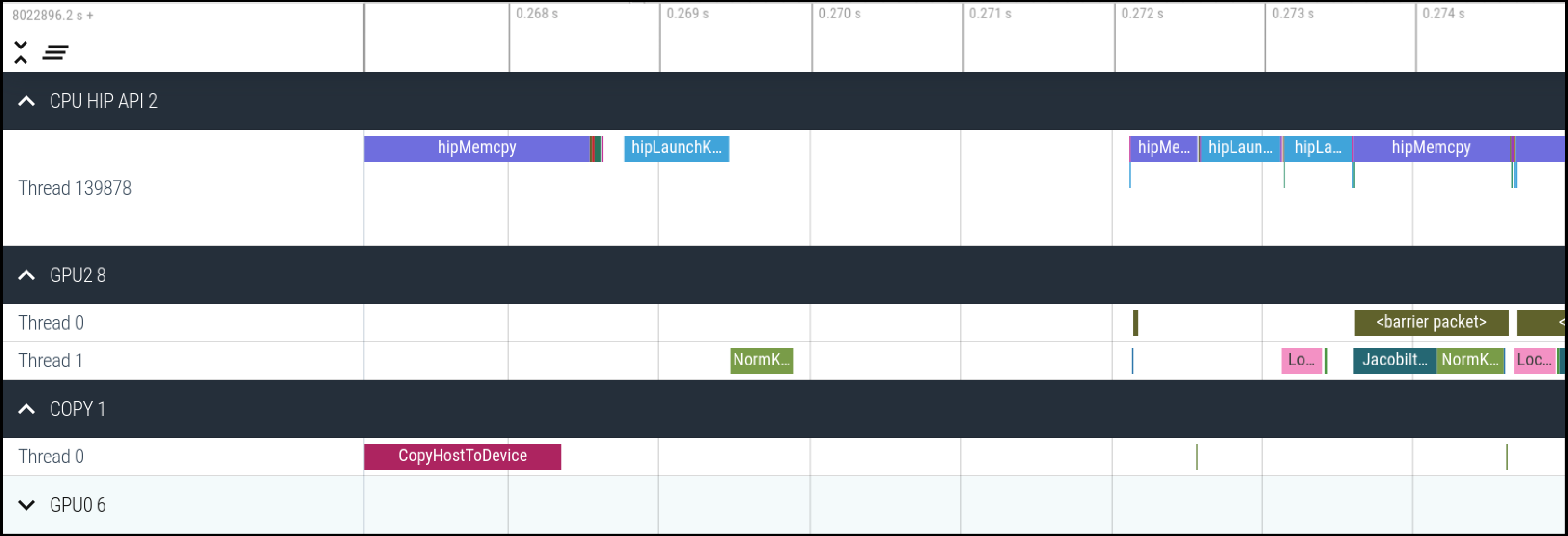
Copy activity (H2D and D2H)

AMD
together we advance_

Perfetto: Visualizing Application Traces

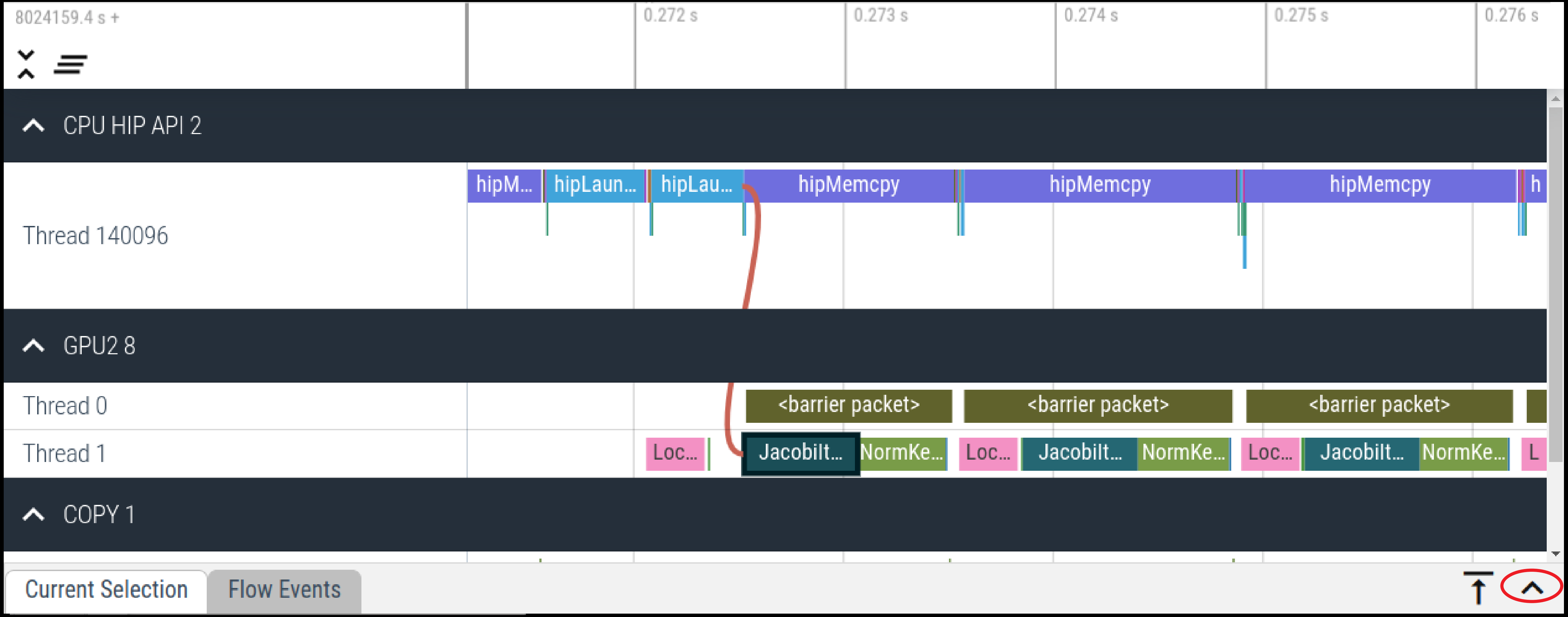
- Zoom in to see individual events
- Navigate trace using WASD keys





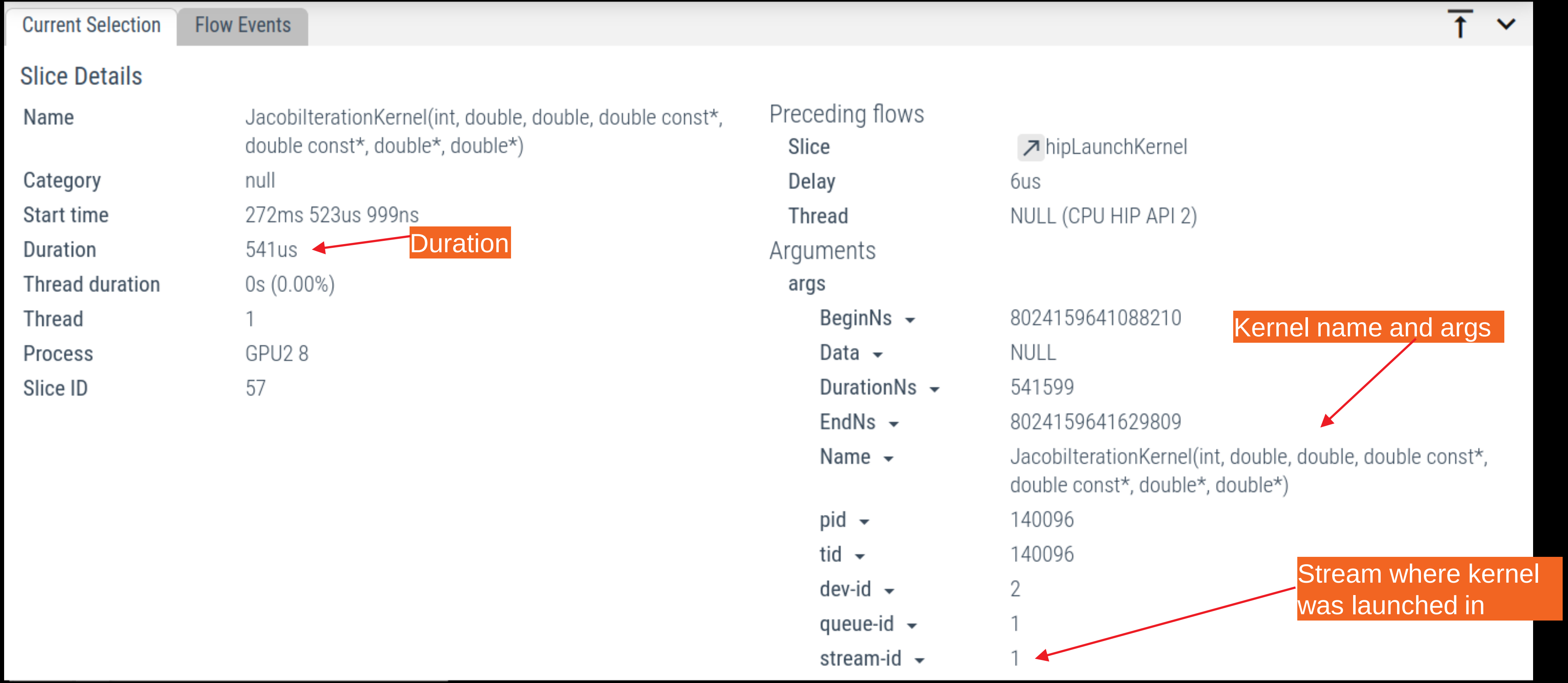
Perfetto: Kernel Information and Flow Events

- 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)



AMD
together we advance_

Perfetto: Kernel Information and Flow Events



Current Selection | Flow Events

Slice Details

Name	JacobiIterationKernel(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

Duration

Preceding flows

Slice	↗ hipLaunchKernel
Delay	6us
Thread	NULL (CPU HIP API 2)

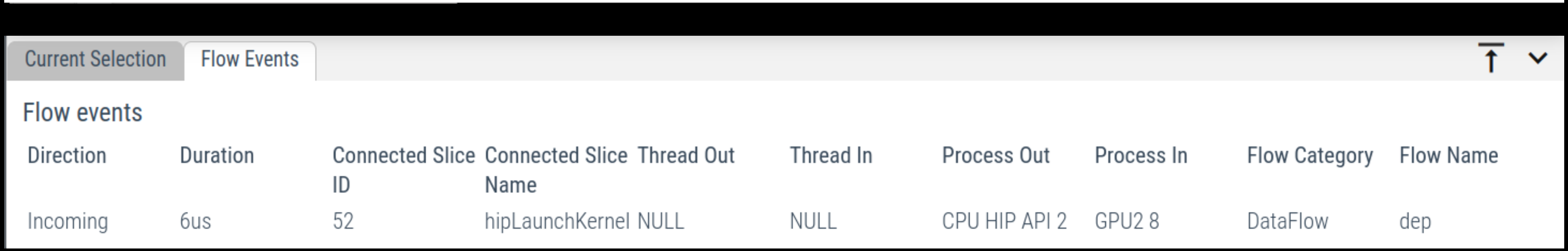
Arguments

args

BeginNs	8024159641088210
Data	NULL
DurationNs	541599
EndNs	8024159641629809
Name	JacobiIterationKernel(int, double, double, double const*, double const*, double*, double*)
pid	140096
tid	140096
dev-id	2
queue-id	1
stream-id	1

Kernel name and args

Stream where kernel was launched in



Current Selection | Flow Events

Flow events

Direction	Duration	Connected Slice ID	Connected Slice Name	Thread Out	Thread In	Process Out	Process In	Flow Category	Flow Name
Incoming	6us	52	hipLaunchKernel	NULL	NULL	CPU HIP API 2	GPU2 8	DataFlow	dep

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:

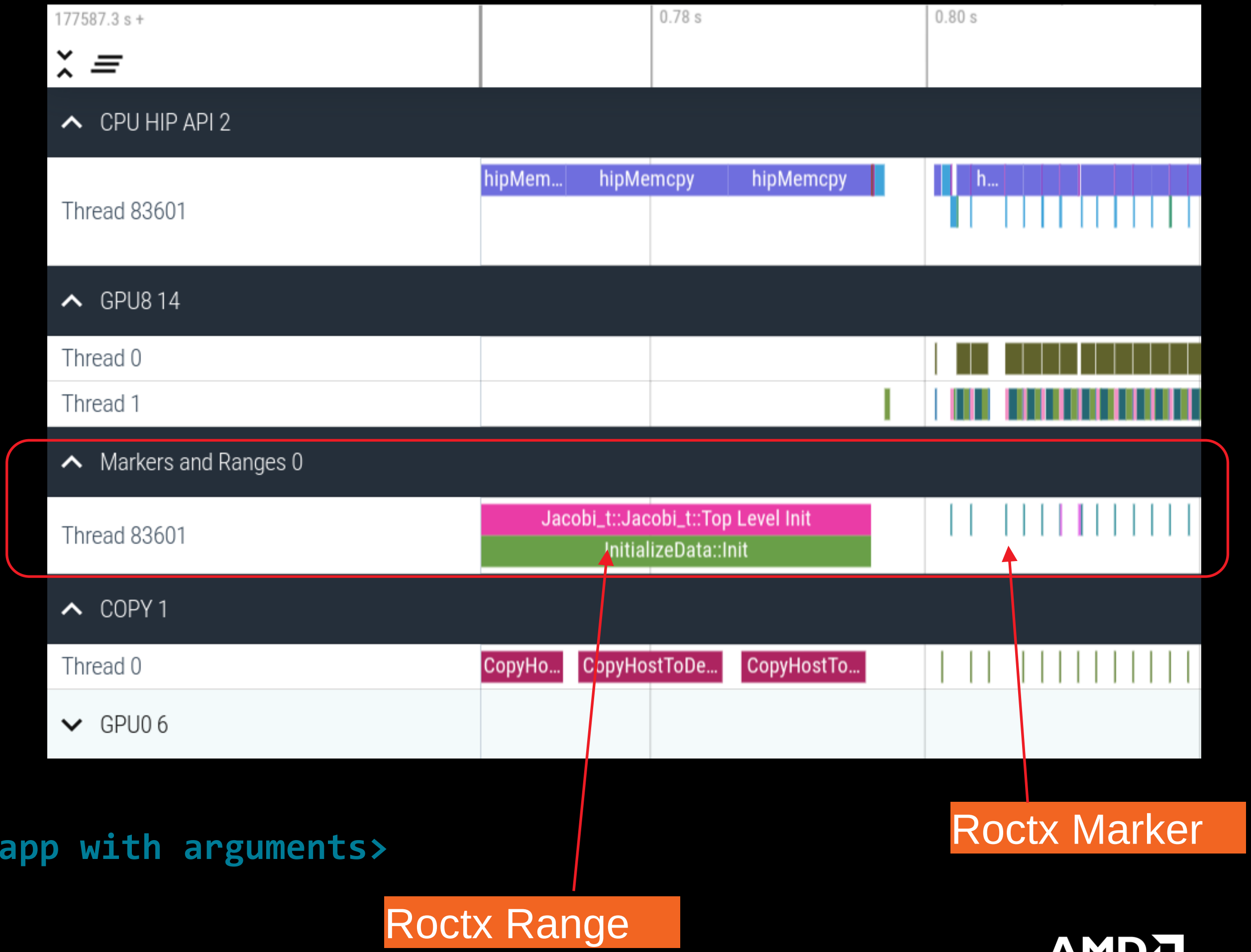
```
...
    roctxMark("start of some code");
    // some_code
    roctxMark("end of some code");
...
```

- 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>
```



Roctx Marker

Roctx Range

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

Larger Traces with Perfetto

- There is a memory limit in the Chrome browser. There is a way to read in the trace for the browser before starting it up.

Linux®

- `curl -LO https://get.perfetto.dev/trace_processor`
- `chmod +x ./trace_processor`
- `./trace_processor –httpd <path to trace file>`
- Open up Chrome browser and go to https://ui.perfetto.dev
- When prompted, click on "Yes, use loaded trace"

Windows®

- Open up https://get.perfetto.dev/trace_processor in a browser to download the python™ script
- `py trace_processor --httpd <trace file>`
  - You may need to download and install python on your windows system
- Open up Chrome browser and go to https://ui.perfetto.dev
- When prompted, click on "Yes, use loaded trace"

rocprof: Commonly Used GPU Counters

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

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

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>
```

- Then invoke the profiler by executing:

```
mpiexec -np <n> rocprof --hip-trace ./Jacobi_hip -g <x> <y>
or
srun --ntasks=n rocprof --hip-trace ./Jacobi_hip -g <x> <y>
```

- This will produce a single CSV file per MPI process
- Multi-node profiling currently isn’t supported

Profiling Per MPI Rank: From Another Node(1)

- Let’s consider a 3-step run:
  - `sbatch_profiling.sh` with sbatch command line to launch the app
  - `rocprof_batch.slurm` This file contains sbatch parameters and the call to srun command line
  - `rocprof_wrapper.sh` calls rocprof command line with input parameters to run the application to be profiled

-
  ```
  $ cat sbatch_profiling.sh
   sbatch -p <partition> -w <node> rocprof_batch.slurm
  ```

-
  ```
   $cat rocprof_batch.slurm
    #!/bin/bash
    #SBATCH --job-name=run
    #SBATCH --ntasks=2
    #SBATCH --ntasks-per-node=2
    #SBATCH --gpus-per-task=1
    #SBATCH --cpus-per-task=1
    #SBATCH --distribution=block:block
    #SBATCH --time=00:20:00
    #SBATCH --output=out.txt
    #SBATCH --error=err.txt
    #SBATCH -A XXXXX
    cd ${SLURM_SUBMIT_DIR}

     #load necessary modules
     #export necessary environment variables

    make clean all
    srun ./rocprof_wrapper.sh ${repository} triad_off_mpi triad_off_mpi
  ```

Profiling Per MPI Rank: From Another Node(2)

```
$cat rocprof_wrapper.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 ${OMPI_COMM_WORLD_RANK+z} ]]; then
        # mpich
        export MPI_RANK=${OMPI_COMM_WORLD_RANK}
    elif [[ -n ${MV2_COMM_WORLD_RANK+z} ]]; then
        # ompi
        export MPI_RANK=${MV2_COMM_WORLD_RANK}
    elif [[ -n ${SLURM_PROCID+z} ]]; then
        export MPI_RANK=${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}"
```

Output directory per rank

Filenames annotated with rank as well

Application and its arguments

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 rocprof_wrapper.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 ${OMPI_COMM_WORLD_RANK+z} ]]; then
        # mpich
        export MPI_RANK=${OMPI_COMM_WORLD_RANK}
    elif [[ -n ${MV2_COMM_WORLD_RANK+z} ]]; then
        # ompi
        export MPI_RANK=${MV2_COMM_WORLD_RANK}
    elif [[ -n ${SLURM_PROCID+z} ]]; then
        export MPI_RANK=${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}"
```

Output directory per rank

Filenames annotated with rank as well

Application and its arguments

AMD
together we advance_

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
- A new improved version is coming

Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

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.

Third-party content is licensed to you directly by the third party that owns the content and is not licensed to you by AMD. ALL LINKED THIRD-PARTY CONTENT IS PROVIDED “AS IS” WITHOUT A WARRANTY OF ANY KIND. USE OF SUCH THIRD-PARTY CONTENT IS DONE AT YOUR SOLE DISCRETION AND UNDER NO CIRCUMSTANCES WILL AMD BE LIABLE TO YOU FOR ANY THIRD-PARTY CONTENT. YOU ASSUME ALL RISK AND ARE SOLELY RESPONSIBLE FOR ANY DAMAGES THAT MAY ARISE FROM YOUR USE OF THIRD-PARTY CONTENT.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.
The OpenMP name and the OpenMP logo are registered trademarks of the OpenMP Architecture Review Board
Python
Windows is a registered trademark of Microsoft Corporation in the US and/or other countries.

AMD