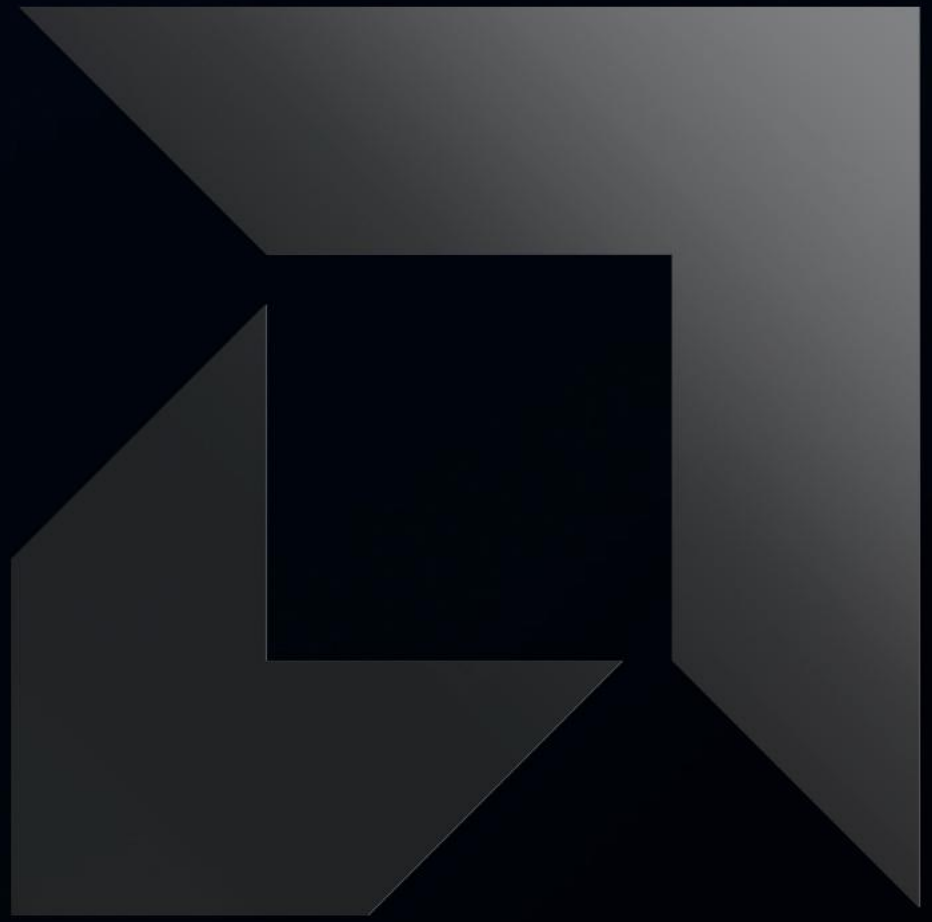


PyTorch Profiling by Example

Presenter: Luka Stanisic
Oct 13-16, 2025
AMD @ CASTIEL AI Workshop

AMD
together we advance_

PyTorch Profiling

1. Overview
2. Example Problem
3. PyTorch Profiler
4. ROCprofiler-SDK CLI tool – `rocprofv3`
5. ROCm Systems Profiler – `rocprof-sys`
6. ROCm Compute Profiler – `rocprof-compute`

Overview: AI application profiling

- AI application's complex multi-layer structure complicates profiling and understanding performance

- Python™ top-level
- Underlying libraries in C/C++/Fortran
- Heavy reliance on GEMMs (matrix multiplication libraries)
- Both cross-node and intra-node communication
- Memory requirements
- I/O for processing large data sets

- Different parts of an AI application require different tools for performance analysis and optimization

Profiling Tools: From Simplest to the most Advanced

- PyTorch profiler – built-in to PyTorch, operator-level performance analysis
  - Works on AMD MI300 accelerators
- `rocprofv3` (AMD) – collects data from hardware counters on AMD Instinct™ GPUs
  - Can display kernel runtimes and a timeline view of the operations on the GPU
- ROCm Systems Profiler (AMD) – a timeline trace collection tool showing GPU + CPU activity
  - `rocprof-sys` collects data from a wide array of profiling sources and displays them all together in one view
- ROCm Compute Profiler (AMD) – focusing on detailed GPU kernel performance analysis
  - Can profile and analyze performance metrics of individual kernels running on the GPU

Example Problem: Train CIFAR100

- Train a classifier on CIFAR-100 data

```
for epoch in range(args.max_epochs):
   for i, (source, targets) in enumerate(train_data):
      opt.zero_grad()
      with precision_context:
            output = model(source)
            logits = output["logits"]
            loss = criterion(logits, targets)
      accuracy = (torch.argmax(logits, axis=-1) == targets).to(torch.float32).mean()
      if args.precision == "automixed":
            scaler.scale(loss).backward()
            scaler.step(opt)
            scaler.update()
      else:
            loss.backward()
            opt.step()
```

- Example in HPCTrainingExamples/MLExamples/PyTorch_Profiling

Repository Organization

- `README.md` contains complete description of how to run the exercises
- `train_cifar_100.py` main script that trains CIFAR100
- `download-data.sh` utility script for downloading the required input data, run it once in the beginning
- `setup.sh` utility script for loading the right modules (`rocm` and `pytorch`)

- 5 exercises in separate directories with READMEs and helper scripts calling `train_cifar_100.py`:
  1. no-profiling
  2. torch-profiler
  3. rocprofv3
  4. rocm-systems-profiler
  5. rocm-compute-profiler

train_cifar_100.py Script

```
-h, --help            show this help message and exit
--data-path DATA_PATH, -dp DATA_PATH # default="data/"
                      Top level data storage
--batch-size BATCH_SIZE, -bs BATCH_SIZE # default="256"
                      Batch size per rank
--download-only # disabled by default
--precision {float32,automixed,bfloat16} # default="automixed"
--max-epochs MAX_EPOCHS # default="1"
                      Number of epochs (maximum) to run. Ignored if max_steps is set and is reached first.
--max-steps MAX_STEPS, -ms MAX_STEPS # default="20"
                      Maximum number of steps to run for profiling
--torch-profile # disabled by default
                      Activate the pytorch profiler
--model {resnet,swinv2,vit} # default="resnet"
                      Vision classification model to use
```

Setup

- Set environment variables

```
export PROFILER_TOP_DIR=$PWD
export MASTER_ADDR=`hostname`
export MASTER_PORT=1234
```

- OpenMPI will set the following
  - `OMPI_COMM_WORLD_RANK`; for serial do `export OMPI_COMM_WORLD_RANK=0`
  - `OMPI_COMM_WORLD_SIZE`; for serial do `export OMPI_COMM_WORLD_SIZE=1`
- Slurm will set the following
  - `SLURM_PROCID`
  - `SLURM_NPROCS`
- Environment setup via modules
  - `module load rocm pytorch # pytorch module automatically loads openmpi`

- Get data (only once, but on a compute node)
  - `./download-data.sh # Calls python3 train_cifar_100.py --download-only --data-path data`

Running Example

- Run one of the bash scripts (e.g., `./single_process.sh`) or simply do:
  - `python3 train_cifar_100.py`

- Expected output (performance greatly depends on the underlying hardware and software stack

```
Namespace(data_path='data', batch_size=512, download_only=False, precision='automixed', max_epochs=1, max_steps=20,
torch_profile=False, model='resnet’)
0 / 0: loss 5.00, acc 0.39%, images / second / gpu: 564.89.
0 / 1: loss 5.06, acc 0.00%, images / second / gpu: 11532.09.
0 / 2: loss 5.02, acc 0.00%, images / second / gpu: 13469.08.
0 / 3: loss 5.07, acc 1.95%, images / second / gpu: 13489.73.
0 / 4: loss 4.92, acc 1.95%, images / second / gpu: 4789.11.
0 / 5: loss 5.70, acc 1.17%, images / second / gpu: 10993.46.
0 / 6: loss 5.66, acc 1.17%, images / second / gpu: 12273.44.
0 / 7: loss 5.45, acc 1.95%, images / second / gpu: 12632.55.
0 / 8: loss 5.27, acc 1.56%, images / second / gpu: 12162.64.
0 / 9: loss 5.26, acc 1.95%, images / second / gpu: 12356.06.
0 / 10: loss 5.41, acc 1.17%, images / second / gpu: 12972.44.
0 / 11: loss 5.22, acc 1.56%, images / second / gpu: 12873.21.
0 / 12: loss 5.27, acc 0.39%, images / second / gpu: 13010.79.
0 / 13: loss 5.12, acc 0.39%, images / second / gpu: 12716.79.
0 / 14: loss 5.08, acc 2.34%, images / second / gpu: 12882.02.
0 / 15: loss 5.16, acc 0.39%, images / second / gpu: 12878.00.
0 / 16: loss 4.92, acc 1.17%, images / second / gpu: 12432.17.
0 / 17: loss 5.44, acc 1.56%, images / second / gpu: 12922.48.
0 / 18: loss 4.83, acc 0.78%, images / second / gpu: 12857.34.
0 / 19: loss 4.95, acc 1.95%, images / second / gpu: 13012.37.
0 / 20: loss 4.85, acc 1.56%, images / second / gpu: 12677.30.
```

No Profiling (yet): Establishing Baseline

- Enter the right folder (`PROFILER_TOP_DIR` should still point to the main, parent directory)
  - `cd no-profiling`

- Running the example
  - In serial using a single process: `./single_process.sh`
  - In parallel using multiple MPI ranks (make sure sufficient number of GPUs is allocated): `./mpi.sh`
  - With Slurm: `./slurm.sh`

- Check if code is really running on a GPU at all using `rocm-smi` or `AMD_LOG_LEVEL`

- Inspect scaling efficiency by comparing `images/second/gpu` from serial and parallel runs
- Modify and inspect impact of `train_cifar_100.py` default argument (e.g., `--batch-size`, `--precision`)
- In the following examples, additionally compare profiling outputs of serial vs parallel runs

PyTorch Profiler: Operator-level Evaluation

- The PyTorch profiler is built into PyTorch and supports AMD GPUs
- In Python™ programs, the profiler can be used as a context with:

Initialization

```
if args.torch_profile == True:
   from torch.profiler import profile, record_function, ProfilerActivity, schedule
   this_schedule = schedule(skip_first=3, wait=5, warmup=1, active=3,repeat=1)
   profiling_context = profile(activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA], record_shapes=True, schedule=this_schedule)
…
```

Inside training loop

```
if args.torch_profile and rank == 0:
   profiling_context.step()
…
```

Finalization

```
if args.torch_profile and rank == 0:
   profiling_context.export_chrome_trace(f"trace_{epoch}_{i}.json")
   print(profiling_context.key_averages(group_by_stack_n=5).table(sort_by="cuda_time_total", row_limit=10))
```

- Check PyTorch Profiler docs for more info: https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html

- In this training, activate PyTorch profiler with `--torch-profile` option (specific to `train_cifar_100.py`)
  - Scripts in torch-profiler folder already have it enabled

PyTorch Profiler: Single Process Example

- Start by running single process example:
  - `cd torch-profiler`
  - `./single-process.sh`

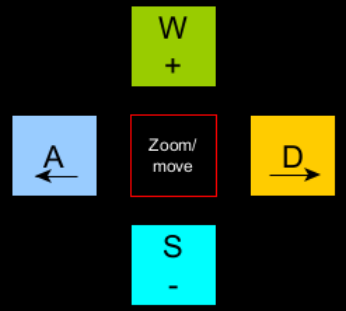
- At the end of the execution, check profiling summary for the training run:

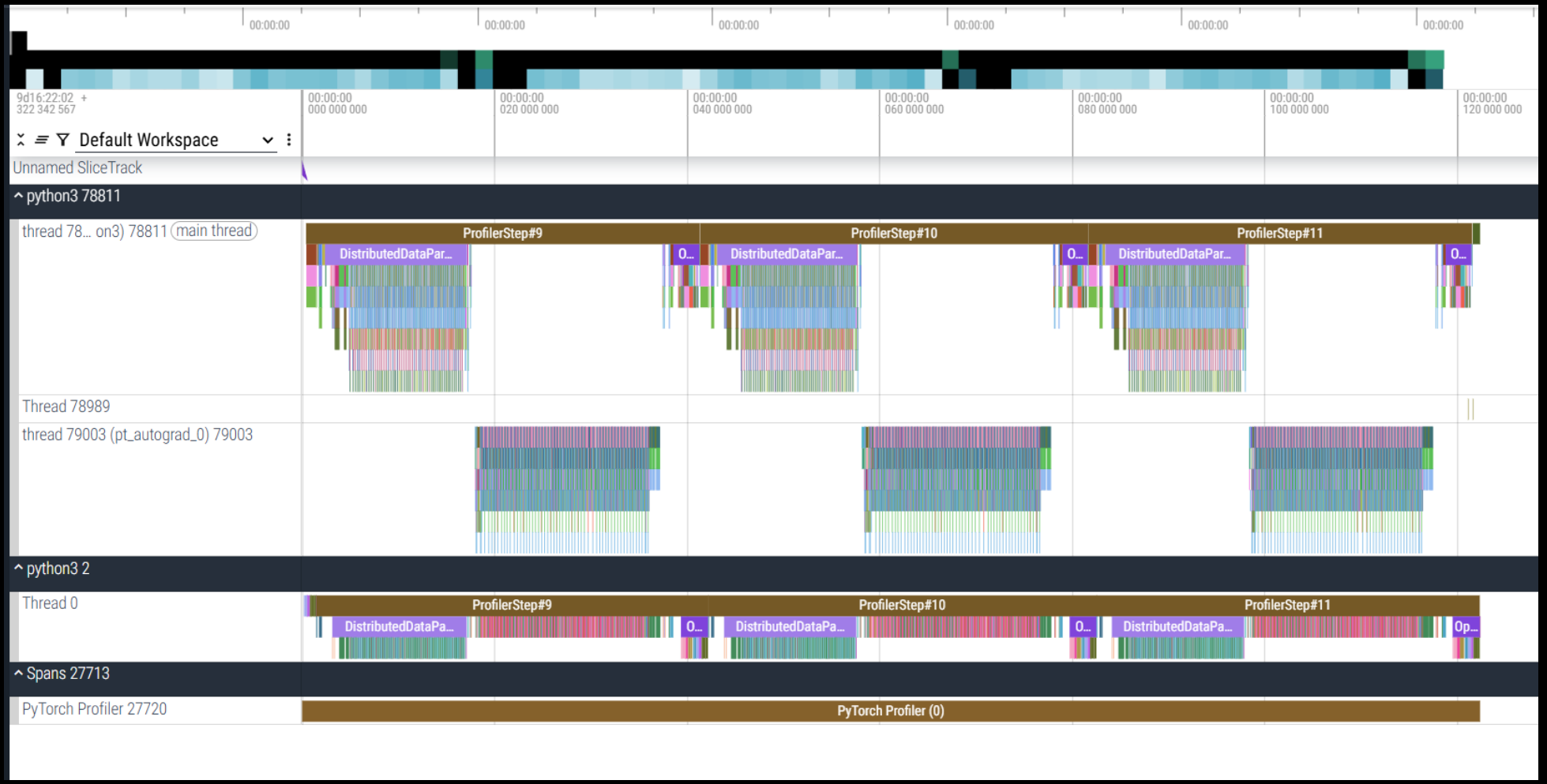
Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA total	CUDA time avg	# of Calls
ProfilerStep*	0.00%	0.000us	0.00%	0.000us	0.000us	119.800ms	185.55%	119.800ms	39.933ms	3
DistributedDataParallel.forward	0.00%	0.000us	0.00%	0.000us	0.000us	41.607ms	64.44%	41.607ms	13.869ms	3
ProfilerStep*	35.57%	59.977ms	70.90%	119.564ms	39.855ms	0.000us	0.00%	25.539ms	8.513ms	3
autograd::engine::evaluate_function: ConvolutionBack...	0.95%	1.597ms	11.26%	18.992ms	119.449us	0.000us	0.00%	16.641ms	104.662us	159
ConvolutionBackward0	0.46%	780.579us	10.05%	16.941ms	106.548us	0.000us	0.00%	16.073ms	101.087us	159
aten::convolution_backward	7.11%	11.987ms	9.58%	16.160ms	101.638us	16.073ms	24.89%	16.073ms	101.087us	159
DistributedDataParallel.forward	7.91%	13.338ms	26.36%	44.458ms	14.819ms	0.000us	0.00%	15.269ms	5.090ms	3
aten::conv2d	0.49%	827.956us	14.04%	23.679ms	74.462us	0.000us	0.00%	12.709ms	39.967us	318
autograd::engine::evaluate_function: torch::autograd...	1.82%	3.067ms	6.47%	10.907ms	22.581us	0.000us	0.00%	11.217ms	23.224us	483
record_param_comms	0.50%	840.427us	0.79%	1.338ms	49.566us	9.403ms	14.56%	9.403ms	348.255us	27

- Analyze additionally generated timeline trace output `trace_0_##.json`
  - `scp aac6:<directory>/trace_0_###.json .`
  - Open Chrome (or another browser) and enter ui.perfetto.dev in the search field
  - Open downloaded `trace_0_###.json` trace file

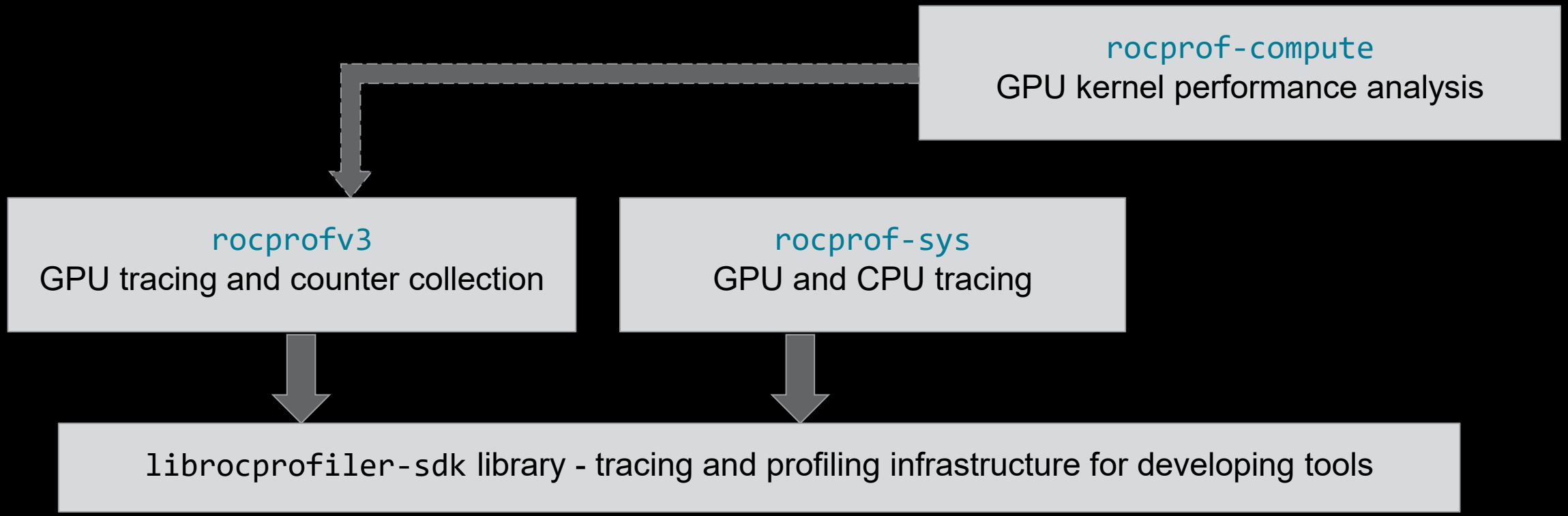
PyTorch Profiler: Analyzing Timeline with Perfetto

Zoom in or scroll with the keys shown





AMD Tools for Profiling PyTorch Apps



rocprofv3: Profiling Basic AMD GPU Activity

- rocprofv3 – performance analysis tool for ROCm based applications (single or multi process)
  - Introduced in ROCm 6.2 (successor of rocprof and rocprofv2)

- Main capabilities:
  - GPU Hotspot analysis – identify performance bottlenecks
  - Device activity tracing – visualize HIP, HSA , GPU kernels, and data transfers a GUI
  - Performance counter collection – analyze kernel performance further

- Supports Python and OpenMP® offload profiling

- Documented at: https://rocm.docs.amd.com/projects/rocprofiler-sdk/en/latest/how-to/using-rocprofv3.html

rocprofv3: Collecting GPU Hotspots

- `rocprofv3 --stats --kernel-trace -- python3 <app with arguments>`

  Combine `--stats` with `--kernel-trace`

  Note: Starting from ROCm 7.0 need `--output-format csv`

- Run `rocprofv3 --help` to see all available options

- For profiling examples, either write your own `rocprofv3` command or use existing helper scripts:
  - `cd rocprofv3`
  - `./kernels.sh # Calls rocprofv3 --stats --kernel-trace --output-directory single_process --output-file kernels --output-format csv -- python3 ${PROFILER_TOP_DIR}/train_cifar_100.py --data-path ${PROFILER_TOP_DIR}/data`

- Inspect all output files in the directory:
  - `ls –lrt single_process/`
  - `head single_process/kernels_kernel_stats.csv`

rocprofv3: GPU Kernels Summary

- For a better look at the data, you can download and view csv files in a spreadsheet viewer

Name	Calls	TotalDurationNs	AverageNs	Percentage	MinNs	MaxNs	StdDev
__amd_rocclr_copyBuffer	7084	30631280	4324.009034	8.62	2080	19040	1934.775497
MIOpenBatchNormBwdSpatial	1113	30550321	27448.62624	8.6	11200	75201	16883.7312
MIOpenBatchNormFwdTrainSpatial	1113	24243730	21782.32704	6.82	5920	71201	17522.50777
igemm_wrw_gtcx2_nhwc_fp16_bx0_ex0_bt256x256x32_wt32x32x8_ws2x2_wr2x2_ta1x4x1x8_1x8x1x	378	17533328	46384.46561	4.94	36480	57920	5469.769233
void at::native::vectorized_elementwise_kernel<4, at::native::AUnaryFunctor<float, float, float, at::n	3402	14283234	4198.481481	4.02	2080	17600	2158.050196
igemm_wrw_gtcx2_nhwc_fp16_bx0_ex1_bt256x256x32_wt32x32x8_ws2x2_wr2x2_ta1x4x1x8_1x8x1x	252	12408987	49242.01191	3.49	40320	67840	5987.652596
batched_transpose_64x64_pack_4x4_ediv_4x4_half	1176	12372186	10520.56633	3.48	4000	23681	5811.585469
batched_transpose_16x64_pack_1x4_ediv_1x2_half	1554	11818266	7605.061776	3.33	3200	16800	3330.13436
igemm_bwd_gtcx2_nhwc_fp16_bx0_ex1_bt128x128x32_wt32x32x8_ws1x1_wr2x2_ta1x8x2x1_1x4x1x	336	11483075	34175.81845	3.23	22880	63520	10757.01903
void at::native::(anonymous namespace)::multi_tensor_apply_kernel<at::native::(anonymous name	68	11101944	163263.8824	3.12	35521	285281	94983.75626
igemm_bwd_gtcx2_nhwc_fp16_bx0_ex0_bt128x128x32_wt32x32x8_ws1x1_wr2x2_ta1x8x2x1_1x4x1x	462	10481465	22687.15368	2.95	17920	33600	3562.70101
SubTensorOpWithScalar1d	2919	10130103	3470.40185	2.85	1920	8480	1047.285154
igemm_fwd_gtcx2_nhwc_fp16_bx0_ex1_bt128x128x32_wt32x32x8_ws1x1_wr2x2_ta1x8x2x1_1x4x1x8	336	8665942	25791.49405	2.44	21440	45920	5144.190298
void at::native::vectorized_elementwise_kernel<4, at::native::BinaryFunctor<c10::Half, c10::Half, c	1029	8436021	8198.271137	2.37	3040	23681	5170.276324
igemm_fwd_gtcx2_nhwc_fp16_bx0_ex0_bt128x128x32_wt32x32x8_ws1x1_wr2x2_ta1x8x2x1_1x4x1x8	399	8200017	20551.42105	2.31	15680	38081	5711.322989
void at::native::elementwise_kernel<512, 1, at::native::gpu_kernel_impl<at::native::direct_copy_ker	1218	7820974	6421.16092	2.2	2560	22080	4078.022098
void at::native::(anonymous namespace)::multi_tensor_apply_kernel<at::native::(anonymous name	51	7767059	152295.2745	2.19	29280	305281	111865.7566
void at::native::(anonymous namespace)::multi_tensor_apply_kernel<at::native::(anonymous name	68	7210418	106035.5588	2.03	25920	199200	65999.72723
void at::native::vectorized_elementwise_kernel<4, at::native::CUDAFunctor_add<c10::Half>, std::a	672	7009622	10430.98512	1.97	4000	24641	6035.759189
void at::native::vectorized_elementwise_kernel<4, at::native::float16_copy_kernel_cuda(at::Tensorl	1197	6781615	5665.509607	1.91	2080	17121	3272.189212
batched_transpose_4x256_half	588	6183049	10515.38946	1.74	8160	14080	1529.750706
void at::native::vectorized_elementwise_kernel<4, at::native::(anonymous namespace)::launch_cla	1029	5847855	5683.046647	1.65	2720	19520	3531.67456
void at::native::(anonymous namespace)::multi_tensor_apply_kernel<at::native::(anonymous name	42	5763852	137234.5714	1.62	75041	203681	60556.35715
void at::native::(anonymous namespace)::multi_tensor_apply_kernel<at::native::(anonymous name	51	5586095	109531.2745	1.57	26401	248001	96033.17791
void at::native::vectorized_elementwise_kernel<4, at::native::CUDAFunctorOnSelf_add<long>, std::	1113	5349130	4806.046721	1.51	3680	6241	595.201643
batched_transpose_4x128_half	1008	5031055	4991.125992	1.42	3360	9601	662.491398
void at::native::(anonymous namespace)::multi_tensor_apply_kernel<at::native::(anonymous name	51	4781450	93753.92157	1.35	31680	177280	54880.39791
void at::native::(anonymous namespace)::multi_tensor_apply_kernel<at::native::(anonymous name	34	4774733	140433.3235	1.34	64320	224961	74552.62726

- Many different kernels (with many calls) executed on a GPU
- Performance not dominated by a single GPU kernel (all <10%)

rocprofv3: Profiling Parallel Runs

- Use `./mpi_kernels.sh` or `./slurm_kernels.sh`

	A	B	C	D	E	F	G	H
1	Name	Calls	TotalDurationNs	AverageNs	Percentage	MinNs	MaxNs	StdDev
2	ncclDevKernel_Generic_4(ncclDevComm*, channelMasks, ncclWork*)	142	338296342	2382368.606	48.89	19840	7913589	2110456.496
3	__amd_rocclr_copyBuffer	6768	35006027	5172.285313	5.06	2080	154560	2538.907308
4	MIOpenBatchNormBwdSpatial	1060	29530041	27858.52925	4.27	8320	128640	17340.88685
5	MIOpenBatchNormFwdTrainSpatial	1060	22611807	21331.8934	3.27	5440	71360	17232.56029
6	igemm_wrw_gtcx2_nhwc_fp16_bx0_ex0_bt256x256x32_wt32x32x8_ws2x2_wr2x2_ta1x4x1x8_1x8x1x	360	16198221	44995.05833	2.34	22720	59520	7561.591281
7	void at::native::vectorized_elementwise_kernel<4, at::native::AUnaryFunctor<float, float, float, at::n	3220	13847025	4300.318323	2	1920	18240	2199.061579
8	igemm_wrw_gtcx2_nhwc_fp16_bx0_ex1_bt256x256x32_wt32x32x8_ws2x2_wr2x2_ta1x4x1x8_1x8x1x	240	13724619	57185.9125	1.98	38240	84640	11646.49477
9	batched_transpose_64x64_pack_4x4_ediv_4x4_half	1114	11978541	10752.7298	1.73	4000	31680	6002.444733
10	batched_transpose_16x64_pack_1x4_ediv_1x2_half	1477	11458708	7758.096141	1.66	3520	27840	3477.469607
11	void at::native::vectorized_elementwise_kernel<4, at::native::BinaryFunctor<c10::Half, c10::Half, c	980	10980944	11205.0449	1.59	2880	180799	21225.63142
12	igemm_bwd_gtcx2_nhwc_fp16_bx0_ex1_bt128x128x32_wt32x32x8_ws1x1_wr2x2_ta1x8x2x1_1x4x1x	308	10708150	34766.72078	1.55	22080	66560	12044.05304

- RCCL/NCCL communication takes majority of the time in the parallel run (unlike single process execution)

rocprofv3: Generating Timeline Trace

- `rocprofv3 --runtime-trace --output-format pftrace -- python3 <app with arguments>`

  For smaller runs use `--sys-trace`

  Generate timeline trace for Perfetto

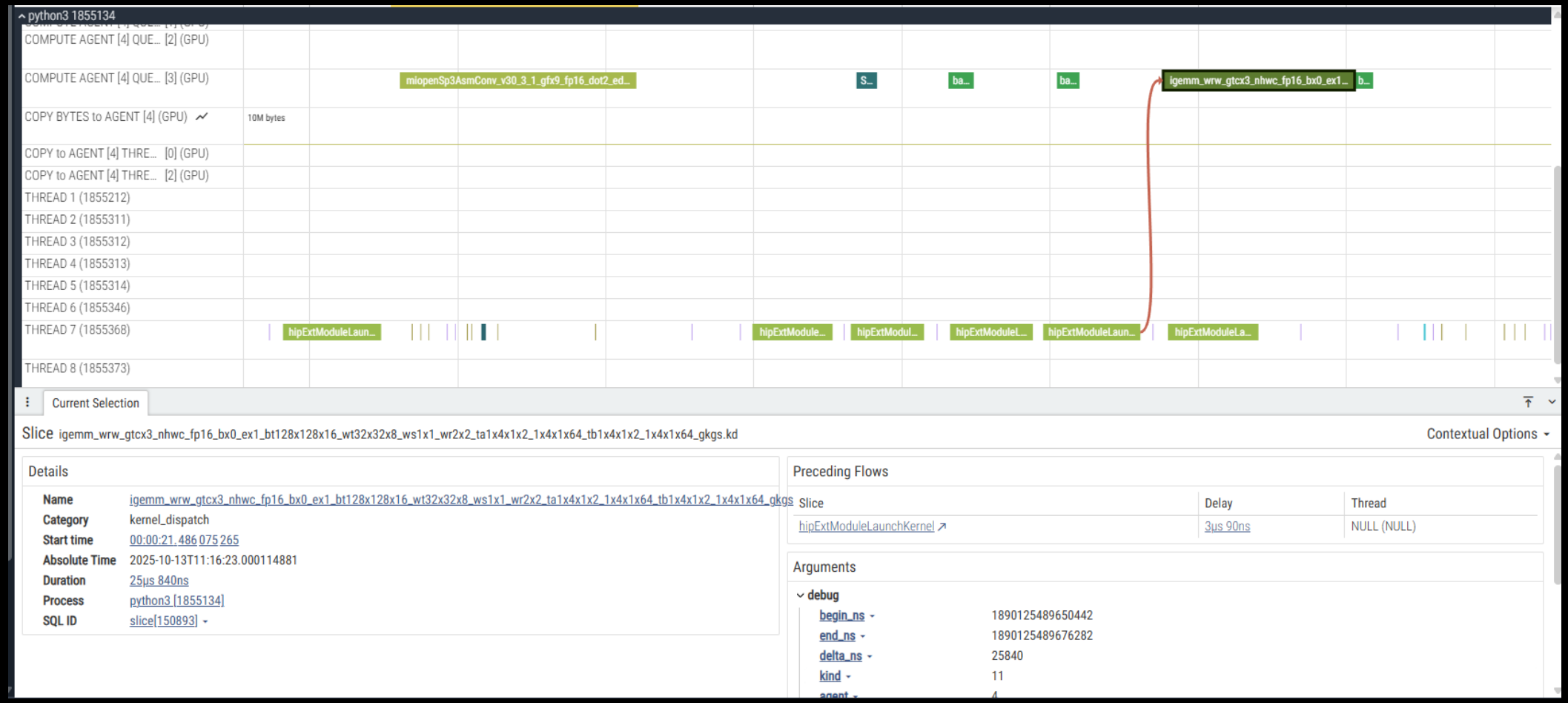
- For profiling examples, either write your own `rocprofv3` command or use existing helper scripts:
  - `./traces.sh` or `./mpi_traces.sh` or `./slurm_traces.sh`

- Similar to PyTorch Profiler, download traces (with `.pftrace` extension) and open with Perfetto

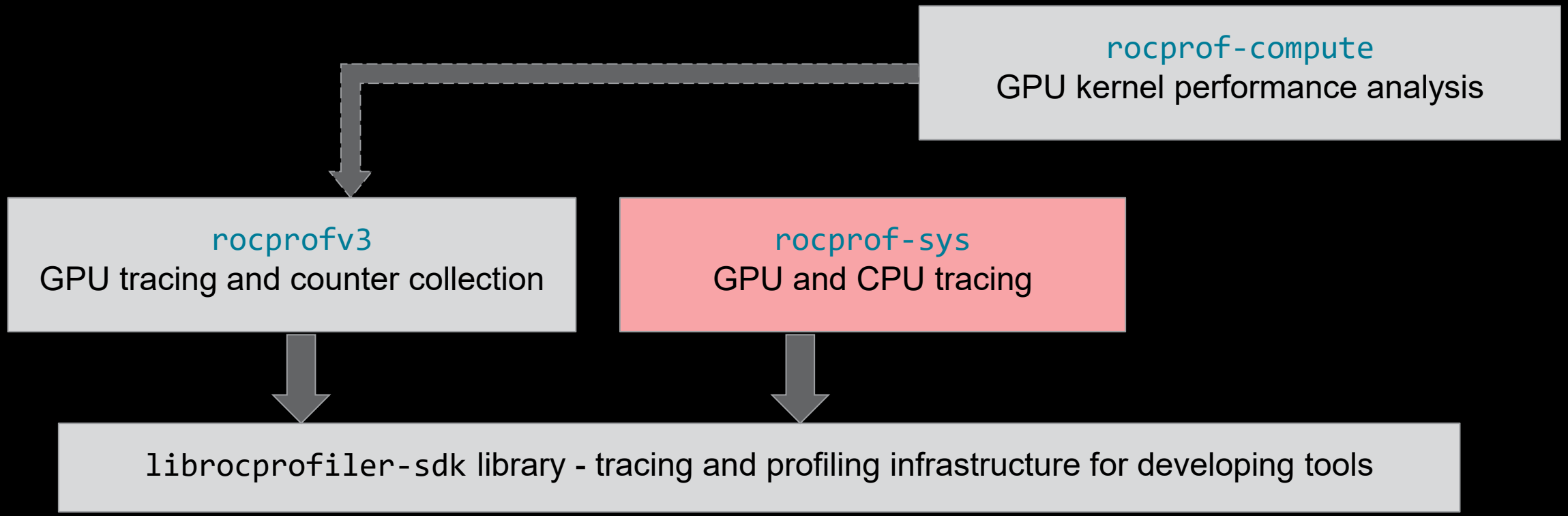
- Advanced: if the trace is too large to open in Perfetto use the `trace_processor` tool
  - `curl -LO https://get.perfetto.dev/trace_processor`
  - `chmod +x ./trace_processor`
  - `trace_processor --httpd /path/to/trace.pftrace`
  - `# Reload the browser user interface. It will prompt to use the HTTP+RPC interface`

rocprofv3: Visualizing Application Timeline

- Check for compute kernels running on GPUs, their durations and CPU threads that started them



AMD Tools for Profiling PyTorch Apps



ROCm Systems Profiler (`rocprof-sys`)

- Profiling and comprehensive tracing of applications on CPU and GPU
- Several data collection modes: sampling, dynamic instrumentation, binary rewrite, causal profiling, etc.
- Collect CPU and GPU metrics
- Visualization format: protobuf files (.proto) viewed in Perfetto

- ROCm Systems Profiler for Python support not fully mature but constantly improving
  - `module load rocm pytorch`
  - `module load rocprofiler-systems`

- `rocprof-sys-run --profile --trace -- python3 <app with arguments>`
- `rocprof-sys-sample -c rocprofsys.cfg -- python3 <app with arguments>`

rocprof-sys: Typical Workflow

Create run-time config (optional, one-time only)

```
rocprof-sys-avail -G
```

Instrument binary (optional)

```
rocprof-sys-instrument -o app.inst -- <app>
```

Collect trace

```
rocprof-sys-run -- ./app.inst
```

(or)

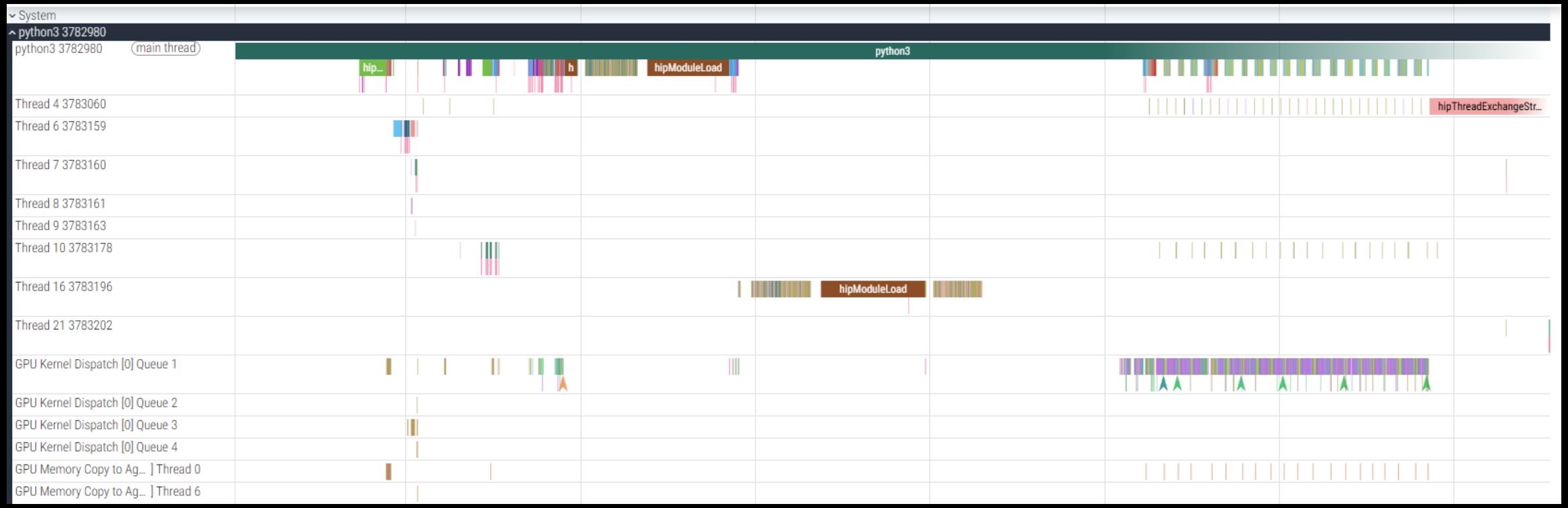
```
mpirun -np 4 rocprof-sys-run -- ./app.inst
```

- `rocprof-sys` documentation: https://rocm.docs.amd.com/projects/rocprofiler-systems/en/latest/index.html
  - Python: https://rocm.docs.amd.com/projects/rocprofiler-systems/en/latest/how-to/profiling-python-scripts.html

- Configuration can be performed in different ways: config file, env variables, command line arguments
  - Run-time configuration parameters: https://rocm.docs.amd.com/projects/rocprofiler-systems/en/latest/how-to/configuring-runtime-options.html

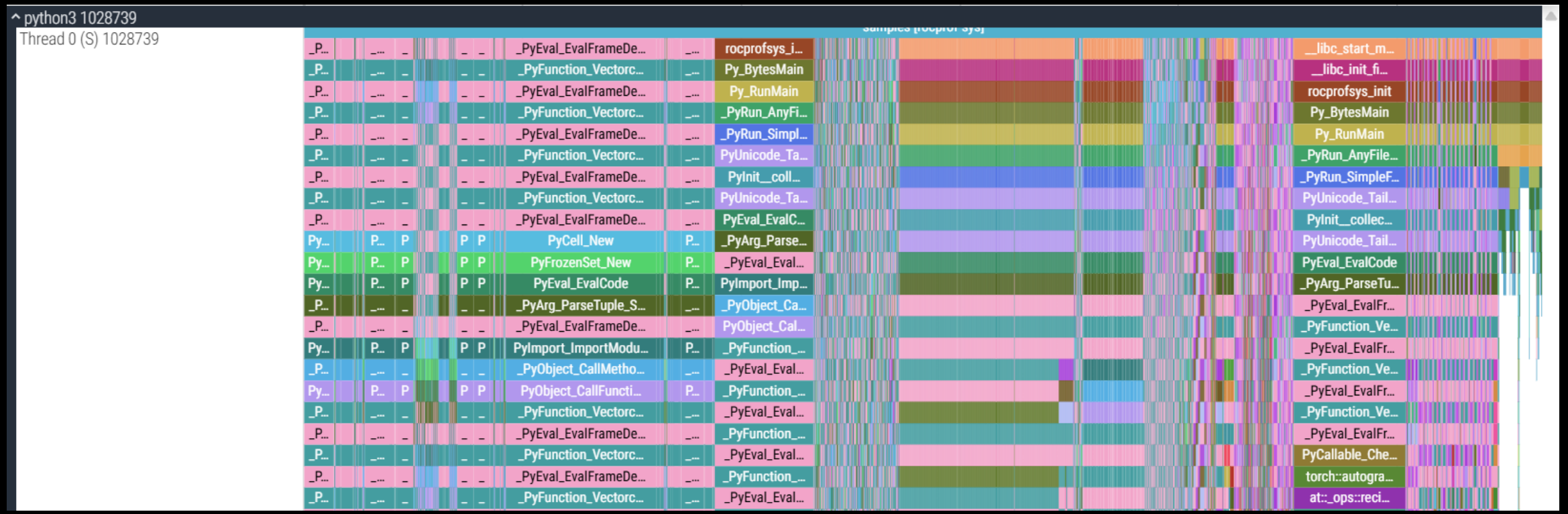
rocprof-sys: Serial Perfetto Trace

- `rocprof-sys-run --profile --trace -- python3 <app with arguments>`
- Similar output to `rocprofv3`, just with potentially more information

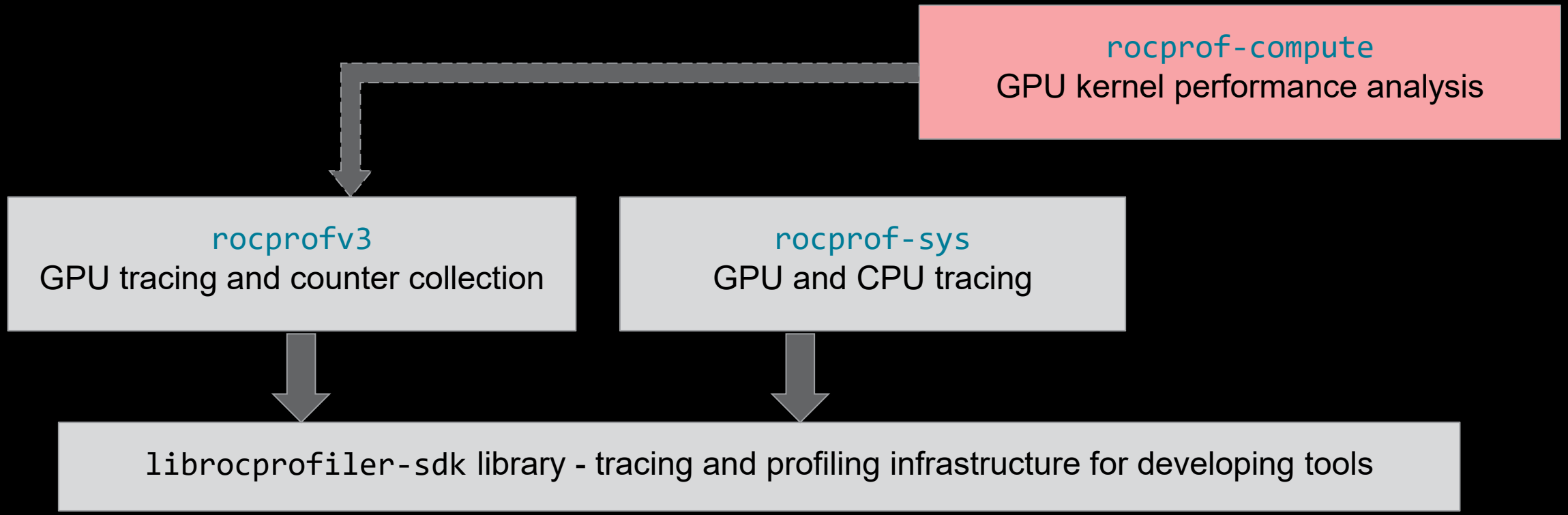


rocprof-sys: Sampling

- `rocprof-sys-sample -c rocprofsys.cfg -- python3 <app with arguments>`
- By using sample profiling, one can explore call stack



AMD Tools for Profiling PyTorch Apps



ROCm Compute Profiler (`rocprof-compute`)

- `rocprof-compute`: a GPU kernel performance analysis tool added to ROCm with version 6.3
  - Originally (before ROCm 6.3) an AMD Research tool called Omniperf that needed separate install

- Most notable features:
  - Roofline analysis to quantify performance of GPU kernels based on hardware limits
  - Kernel comparison to quantify improvements and visualize their impact on hardware memory
  - Executes code many times for automatic hardware counter collection providing many derived metrics
  - Support for speed of light and memory chart (memory chart available only in GUI)

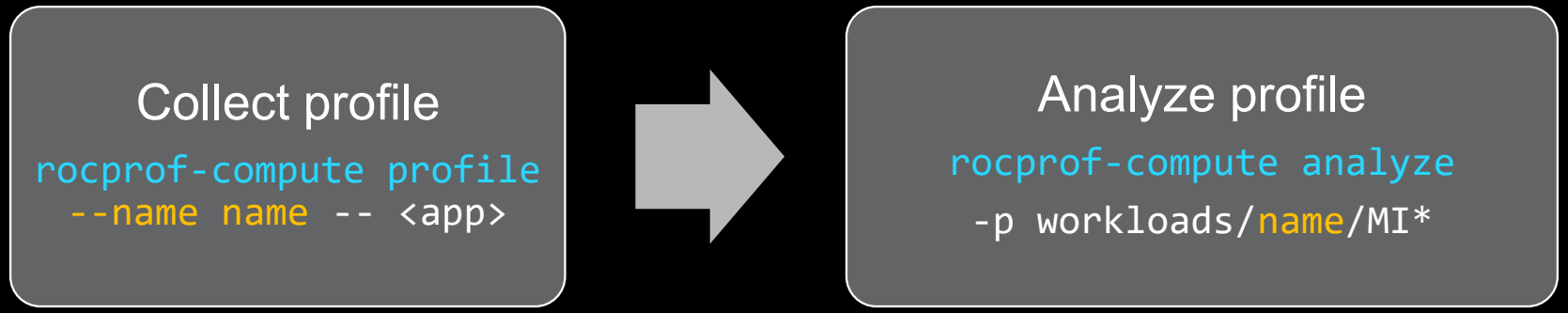
- ROCm Systems Profiler for Python support not fully mature but constantly improving
  - `module load rocm pytorch`
  - `module load rocprofiler-compute`

- `rocprof-compute` may have challenges with some non-deterministic AI programs
  - Example shown in the slides and exercises should work
  - More robust solution under development

rocprof-compute: Visualize Roofline Models

- `rocprof-compute profile --name roofline --roof-only --kernel-names --device 0 -- <app>`

How to run rocprof-compute?



- Profile single process application (runs ~10 times to collect all the counters):
  - `rocprof-compute profile --no-roof --name cifar_100_single_proc -- <app>`
- Analyzed profiled data, focusing on speed-of-light:
  - `rocprof-compute analyze -p workloads/cifar_100_single_proc/MI* -b 2.1.2 2.1.3 2.1.4 2.1.5`
- Compare two rocprof-compute profiles to assess optimization effects
  - `rocprof-compute analyze -p workloads/exp1/MI300A_A1 -p workloads/exp2/MI300A_A1 -b 7.1.0`

rocprof-compute: Top Kernels Stats

- Top ten kernels stats always shown
- Initialization and copying buffers on top
  - Not surprising for this small problem
- Three igemm (integer matrix multiplies) operations, two transposes
- Output consistent with rocprofv3

```
--------------------------------------------------------------------------------
0. Top Stats
0.1 Top Kernels
```

	Kernel_Name	Count	Sum(ns)	Mean(ns)	Median(ns)	Pct
0	MIOpenBatchNormBwdSpatial.kd	1113.00	29481198.00	26488.05	16960.00	8.43
1	__amd_rocclr_copyBuffer.kd	7084.00	25225023.00	3560.84	2720.00	7.21
2	MIOpenBatchNormFwdTrainSpatial.kd	1113.00	23670135.00	21266.97	16320.00	6.77
3	igemm_wrw_gtcx2_nhwc_fp16_bx0_ex0_bt256x256x32_wt32x32x8_ws2x2_wr2x2_ta1x4x1x8_1x8x1x32_tb1x4x1x8_1x8x1x32_gkgs.kd	378.00	17705325.00	46839.48	46880.50	5.06
4	void at::native::vectorized_elementwise_kernel<4, at::native::AUnaryFunctor<float, float, float, at::native::binary_i...	3402.00	14961319.00	4397.80	3840.00	4.28
5	batched_transpose_64x64_pack_4x4_ediv_4x4_half.kd	1176.00	12659557.00	10764.93	7520.00	3.62
6	igemm_wrw_gtcx2_nhwc_fp16_bx0_ex1_bt256x256x32_wt32x32x8_ws2x2_wr2x2_ta1x4x1x8_1x8x1x32_tb1x4x1x8_1x8x1x32_gkgs.kd	252.00	12326116.00	48913.16	49920.00	3.53
7	batched_transpose_16x64_pack_1x4_ediv_1x2_half.kd	1554.00	11979227.00	7708.64	5600.00	3.43
8	igemm_bwd_gtcx2_nhwc_fp16_bx0_ex1_bt128x128x32_wt32x32x8_ws1x1_wr2x2_ta1x8x2x1_1x4x1x64_tb1x8x1x2_1x4x1x64_mh_gkgs.kd	336.00	11269146.00	33539.12	30080.00	3.22
9	SubTensorOpWithScalar1d.kd	2919.00	10965945.00	3756.75	3200.00	3.14

`rocprof-compute`: Speed-of-Light Table

- `rocprof-compute` shows many tables that can help with the performance analysis of an application.

```
--------------------------------------------------------------------------------
2. System Speed-of-Light
2.1 Speed-of-Light
```

Metric_ID	Metric	Avg	Unit	Peak	Pct of Peak
2.1.2	MFMA FLOPs (BF16)	2219.04	Gflop	181043.20	1.23
2.1.3	MFMA FLOPs (F16)	1750.61	Gflop	181043.20	0.97
2.1.4	MFMA FLOPs (F32)	0.00	Gflop	45260.80	0.00
2.1.5	MFMA FLOPs (F64)	0.00	Gflop	45260.80	0.00

- Majority of the matrix operations work in BF16/F16, no single/double floating point operations
- FLOPs amount very small indicating that a much bigger problem can be used to occupy GPU

rocprof-compute: Analyzing Particular Kernel Dispatch

- During analysis phase, get list of hotspots and dispatches:
  - `rocprof-compute analyze -p workloads/cifar_100_single_proc/MI* --list-stats >& stats.txt`

- Find dispatch of kernel you want to analyze, and then analyze only that dispatch:
  - `rocprof-compute analyze -p workloads/cifar_100_single_proc/MI* --dispatch <N> >& dispatchN.txt`

- When specific dispatch is not provided, values shown are averaged over all dispatches of that kernel

rocprof-compute: Tips

- Filtering by kernel name and metrics during `rocprof-compute profile` will cut down on profiling time
  - `rocprof-compute profile -k "<kernel1>" "<kernel2>"` filters two kernel names
  - Surrounding kernel name in quotes allows spaces to appear in your kernel search string
  - `rocprof-compute` applies wildcard automatically, so only unique kernel names substring required

- Use a subset of metrics for `rocprof-compute profile` to reduce the number of profiler runs
  - `rocprof-compute profile --block SQ SQC -n <workload name> -- <app>`
  - `rocprof-compute profile --help` displays all block strings you can filter by
  - Performance model doc goes over some of the meaning behind lower-level hardware units and metrics

- MPI/srun support still brittle, safest way is to use node interactively and run only with 1 MPI rank

- Don't know where to start? → There are few easy things to 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

Conclusion

- Each of the presented tools has a role in understanding AI application performance and how to optimize it

- PyTorch Profiler is a operator-level profiler built into PyTorch
- `rocprofv3` is a CLI based tool for device and runtime API tracing, and raw hardware counter collection
- `rocprof-sys` is the comprehensive CPU + GPU tracing tool
- `rocprof-compute` is used to study kernel performance via automated counter collection and analysis

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.


AMD