HIPification and Profiling Tools

Julio Maia, Noel Chalmers, Paul T. Bauman, Nicholas Curtis, Nicholas Malaya, Damon McDougall, Rene van Oostrum, Noah Wolfe

May 2021

AMD

Agenda

Porting CUDA codes to HIP

Profiling tools

Objectives

This training:

- Demonstrates how to convert CUDA codes into HIP
- Explains the meaning of the term ‘hipify’
- Provides an idea of the common ‘gotchas’ of porting apps

Getting started with HIP

CUDA VECTOR ADD

```
__global__ void add(int n,
                    double *x,
                    double *y){
  int index = blockIdx.x * blockDim.x
              + threadIdx.x;
  int stride = blockDim.x * gridDim.x;

  for (int i = index; i < n; i +=
stride){
    y[i] = x[i] + y[i];
  }
}
```

HIP VECTOR ADD

```
__global__ void add(int n,
                    double *x,
                    double *y){
  int index = blockIdx.x * blockDim.x
              + threadIdx.x;
  int stride = blockDim.x * gridDim.x;

  for (int i = index; i < n; i += stride){
    y[i] = x[i] + y[i];
  }
}
```

KERNELS ARE SYNTACTICALLY THE SAME

CUDA APIs vs HIP API

CUDA	HIP
cudaMalloc(&d_x, N*sizeof(double));	hipMalloc(&d_x, N*sizeof(double));
cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);	hipMemcpy(d_x, x, N*sizeof(double), hipMemcpyHostToDevice);
cudaDeviceSynchronize();	hipDeviceSynchronize();

Launching a kernel

CUDA KERNEL LAUNCH SYNTAX

```
some_kernel<<<gridsize, blocksize,
              shared_mem_size, stream>>>
              (arg0, arg1, ...);
```

```
some_kernel<T_ARGS><<<gridsize, blocksize,    shared_
mem_size, stream>>>(arg0, arg1, ...);
```

HIP KERNEL LAUNCH SYNTAX

```
hipLaunchKernelGGL(some_kernel,
                   gridsize, blocksize,
                   shared_mem_size, stream,
                   arg0, arg1, ...);
```

```
hipLaunchKernelGGL(
  HIP_KERNEL_NAME(some_kernel<T_ARGS>),
  gridsize, blocksize,    shared_mem_size, stream,
      arg0, arg1, ...);
```


HIPification Tools for faster code porting

- ROCm provides 'HIPification' tools to do the heavy-lifting on porting CUDA codes to ROCm
  - Hipify-perl
  - Hipify-clang

- Good resource to help with porting: https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-porting-guide.html

- In practice, large portions of many HPC codes have been automatically Hipified:
  - ~90% of CUDA code in CORAL-2 HACC
  - ~80% of CUDA code in CORAL-2 PENNANT
  - ~80% of CUDA code in CORAL-2 QMCPack
  - ~95% of CUDA code in CORAL-2 Laghos

The remaining code requires programmer intervention

HIPify Tools

- Hipify-perl:
  - Easy to use – point at a directory and it will attempt to hipify CUDA code
  - Very simple string replacement technique: may make incorrect translations
    - `sed -e ‘s/cuda/hip/g’` (e.g., cudaMemcpy becomes hipMemcpy)
  - Recommended for quick scans of projects

- Hipify-clang:
  - Requires CLANG compiler
  - More robust translation of the code
  - Uses clang to parse files and perform semantic translation
  - Can generate warnings and assistance for code for additional user analysis
  - High quality translation, particularly for cases where the user is familiar with the make system

Hipify-perl

- Sits in $HIP/bin/ (**export PATH=$PATH:[MYHIP]/bin**)
- Command line tool: **hipify-perl foo.cu > new_foo.cpp**
- Compile: **hipcc new_foo.cpp**
- How does this this work in practice?
  - Hipify source code
  - Check it in to your favorite version control
  - Try to build
  - Manually work on the rest

Hipify-clang

- Available at https://github.com/ROCm-Developer-Tools/HIPIFY
- Build from source
- ‘Hipification’ requires same headers that would be needed to compile it with clang:
- `./hipify-clang foo.cu -I /usr/local/cuda-8.0/samples/common/inc`
- Understands how to translate many CUDA libraries (cuBLAS, cuFFT, cuSPARSE, etc.)
- Will get useful warning messages about unknown conversions

```
[10:59:29][pabauman@fry:~/work/qmcpack/build/hipify]$ /home/pabauman/work/hip-testing/hipify-clang-install/bin/hi
pify-clang /home/pabauman/work/qmcpack/src/src/QMCWaveFunctions/EinsplineSetCuda.cpp -o-dir=. -examine -I/usr/inc
lude/libxml2 -I/usr/include/hdf5/serial -I/home/pabauman/work/qmcpack/src/src -I/home/pabauman/work/qmcpack/build
/src -I/home/pabauman/work/qmcpack/build/include -I/home/pabauman/work/qmcpack/src/external_codes/mpi_wrapper -I/
home/pabauman/work/qmcpack/src/external_codes/boost_multi -I/home/pabauman/work/qmcpack/src/external_codes/catch
-I/usr/lib/x86_64-linux-gnu/openmpi/include
/tmp/EinsplineSetCuda.cpp-9b0c60.hip:135:5: warning: CUDA identifier is unsupported in HIP.
    cudaMemPrefetchAsync(pos, 3 * N * sizeof(float), curr_gpu, spline_streams[devicenr]);
    ^
/tmp/EinsplineSetCuda.cpp-9b0c60.hip:183:5: warning: CUDA identifier is unsupported in HIP.
    cudaMemPrefetchAsync(pos, 3 * N * sizeof(float), curr_gpu, spline_streams[devicenr]);
    ^
/tmp/EinsplineSetCuda.cpp-9b0c60.hip:226:5: warning: CUDA identifier is unsupported in HIP.
    cudaMemPrefetchAsync(pos, 3 * N * sizeof(float), curr_gpu, spline_streams[devicenr]);
    ^
```


Gotchas

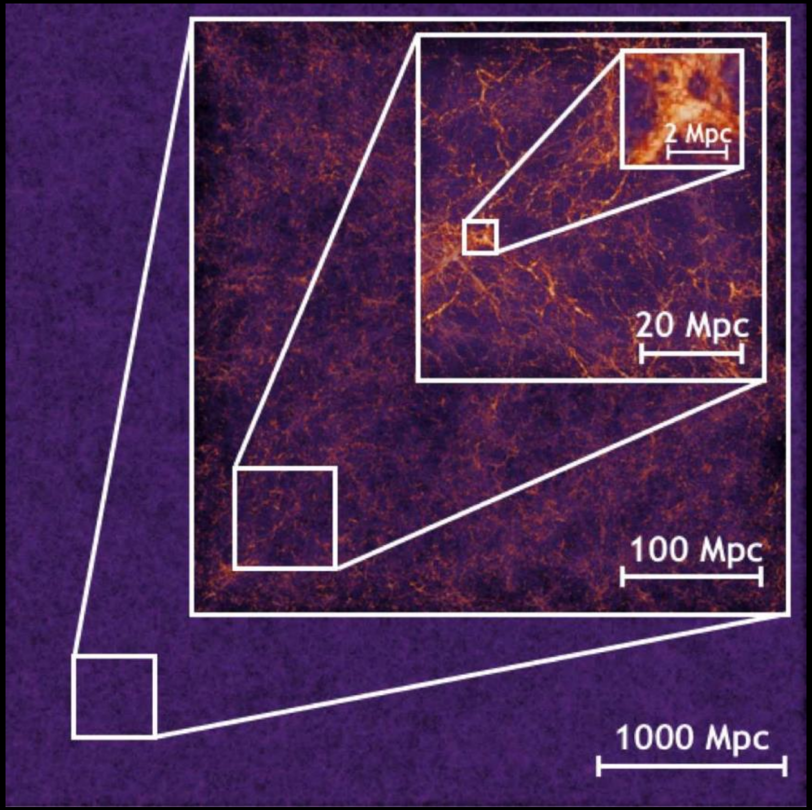
- Hipify tools are not running your application, or checking correctness
- Code relying on specific hardware aspects (e.g., warp size == 32) may need attention after conversion
- Hipifying can't handle inline PTX assembly
  - Can either use inline GCN ISA, or convert it to HIP
- Hipify-perl can't handle library calls, hipify-clang can handle library calls

What to look for when porting:

- Inline PTX assembly
- CUDA Intrinsics
- Hardcoded dependencies on warp size, or shared memory size
  - Grep for "32" *just in case*
  - Do not hardcode the warpsize! Use something like #define WARPSIZE *size*
- Code geared toward limiting size of register file on NVIDIA hardware
- Functions implicitly inlined
- Unified memory

Example: HACC

- Hardware Accelerated Cosmology Code
- Simulates time-evolution of universe
  - Mpc = Megaparsec = 3.09×10^{22} meters
- **Our HIP success story:**
  - **Ported in an afternoon**
- Profiling:
  - 10% of time is spent in the tree walk
  - >80% in the short force kernels
    - **(GPU kernel)**
  - 5% in the 3d Transposes / FFTs



$$f_{SR} = (s + \epsilon)^{-3/2} - f_{grid}(s)$$

where,

$$s = \mathbf{r} \cdot \mathbf{r}$$

and,

$$f_{grid}(s) = POLY_5(s)$$

HACC: What made it a success

- **What was easy?**
  - Simple GPU kernel
  - Few library dependencies (FFTW, not in kernel)
  - No advanced CUDA features
- **What was difficult?**
  - Inline PTX: required translation to AMD GCN
  - Hand-written wave-32 code (for a reduction)

Porting HACC

CUDA	HIP
`cudaMemcpyAsync(d_npos,h_npos,Nposbytes, cudaMemcpyHostToDevice,stream);`	`hipMemcpyAsync(d_npos,h_npos,Nposbytes, hipMemcpyHostToDevice,stream);`
`cudaMemcpyAsync(d_mask,h_mask,NmaskBytes, cudaMemcpyHostToDevice,stream);`	`hipMemcpyAsync(d_mask,h_mask,NmaskBytes, hipMemcpyHostToDevice,stream);`
`calcHHCullenDehnen<<<blocksPerGrid, threadsPerBlock, 0, stream>>> (cnt, SIZE, d_npos, d_mask, rsm);`	`hipLaunchKernelGGL(calcHHCullenDehnen, blocksPerGrid, threadsPerBlock, 0,stream, cnt, SIZE, d_npos, d_mask, rsm);`
`cudaMemcpyAsync(h_pos, d_npos+(SIZE-cnt),cntBytes, cudaMemcpyDeviceToHost,stream);`	`hipMemcpyAsync(h_pos, d_npos+(SIZE-cnt),cntBytes, hipMemcpyDeviceToHost,stream);`
`cudaMemcpyAsync(h_mask,d_mask,NmaskBytes, cudaMemcpyDeviceToHost,stream);`	`hipMemcpyAsync(h_mask,d_mask,NmaskBytes, hipMemcpyDeviceToHost,stream);`

Fortran + CUDA C/C++ -> Fortran + HIP C/C++

- The only difference here is that the CUDA C/C++ code is linked with some Fortran routines
- Assumption is these Fortran routines do not contain CUDA Fortran
- This behaves like you would expect:
  - hipify the CUDA
  - Compile your HIP C/C++ with hipcc
  - Compile your Fortran code
  - Link with hipcc
- Example scenario: your HIP C/C++ code makes calls to Fortran functions (e.g., LAPACK functions) on the host

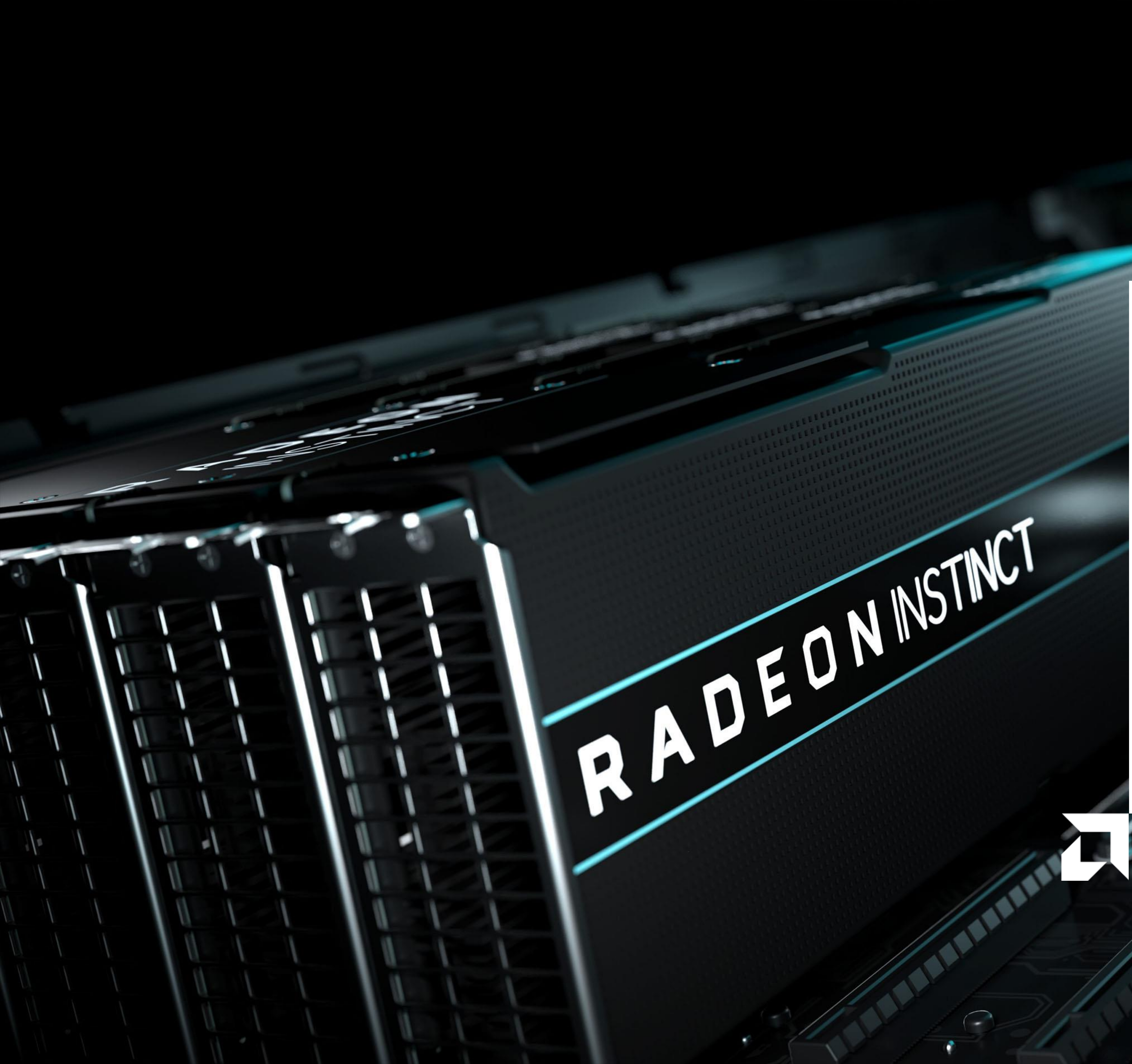
CUDA Fortran -> Fortran + HIP C/C++

- There is no HIP equivalent to CUDA Fortran
- But HIP functions are callable from C, using `extern C`, so they can be called directly from Fortran
- The strategy here is:
  - **Manually port** CUDA Fortran code to HIP kernels in C++
  - Wrap the kernel launch in a C function
  - Call the C function from Fortran through Fortran's ISO_C_binding
- This strategy should be usable by Fortran users since it is standard conforming Fortran
- ROCm has an interface layer, hipFort, which provides the wrapped bindings for use in Fortran
  - https://github.com/ROCmSoftwarePlatform/hipfort

Portability layers using HIP

Several portability layers are already supporting, or implementing, HIP

- RAJA
  - HIP kernel execution policies syntactically identical to CUDA
  - Official PRs under review
- Kokkos
  - HIP kernel execution policies syntactically identical to CUDA
  - Support is in beta and under development by Kokkos and AMD developers
- OCCA
  - OKL kernels can compile for HIP devices
  - Available in OCCA's master branch
- OpenMP 5.0
  - gcc and Cray's C++ compiler support target offload regions



Profiling

AMD GPU Profiling

- ROC-profiler (or simply 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 the collection of application tracing and counter collection
  - Under constant development
- Provided in the ROCm releases
- The output of rocprof can be visualized using the chrome browser with chrome tracing

rocprof: Getting started + useful flags

- To get help:
  - `$ /opt/rocm/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
- Advanced usage
  - -m <metric file>: Allows the user to define and collect custom metrics. See rocprofiler/test/tool/*.xml on GitHub for examples.

AMD

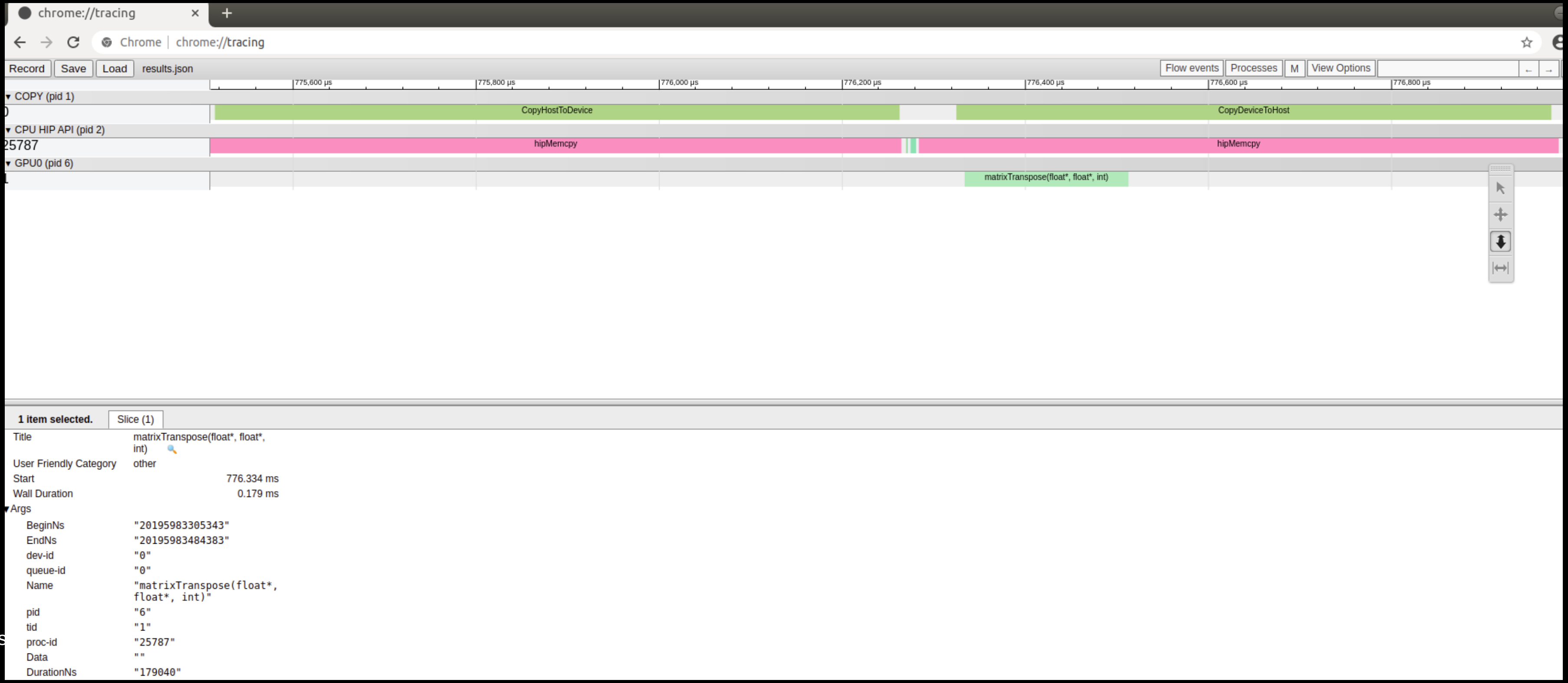
rocprof: Collecting application traces (1)

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

rocprof: Collecting application traces (2)

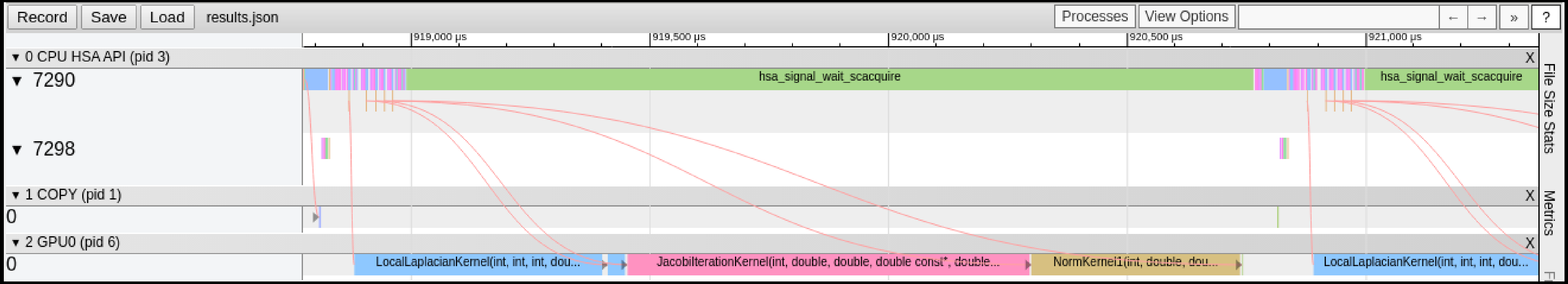
- rocprofiler can collect traces
  - `$ /opt/rocm/bin/rocprof --hip-trace <app with arguments>`
  - This will output a .json file that can be visualized using the chrome browser
  - Go to chrome://tracing and then load in the .json file.
    - The trace will display HIP calls, mem copies, kernels.



AMD

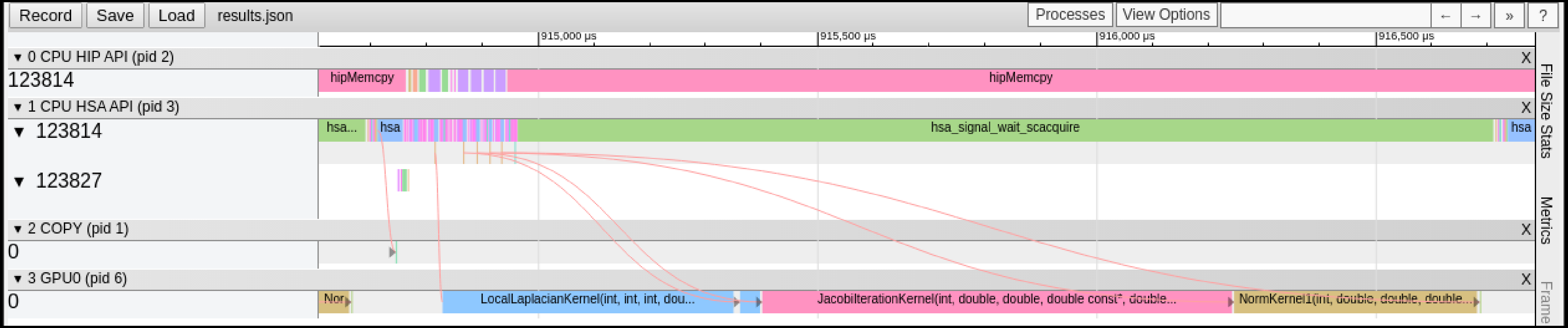
rocprof: Collecting application traces (3)

- rocprofiler can collect traces
  - `$ /opt/rocm/bin/rocprof --hsa-trace <app with arguments>`
  - This will output a .json file that can be visualized using the chrome browser
  - Go to chrome://tracing and then load in the .json file
    - The trace will display copies, hsa signals, and kernel calls
    - Slowest trace mode – Use with caution



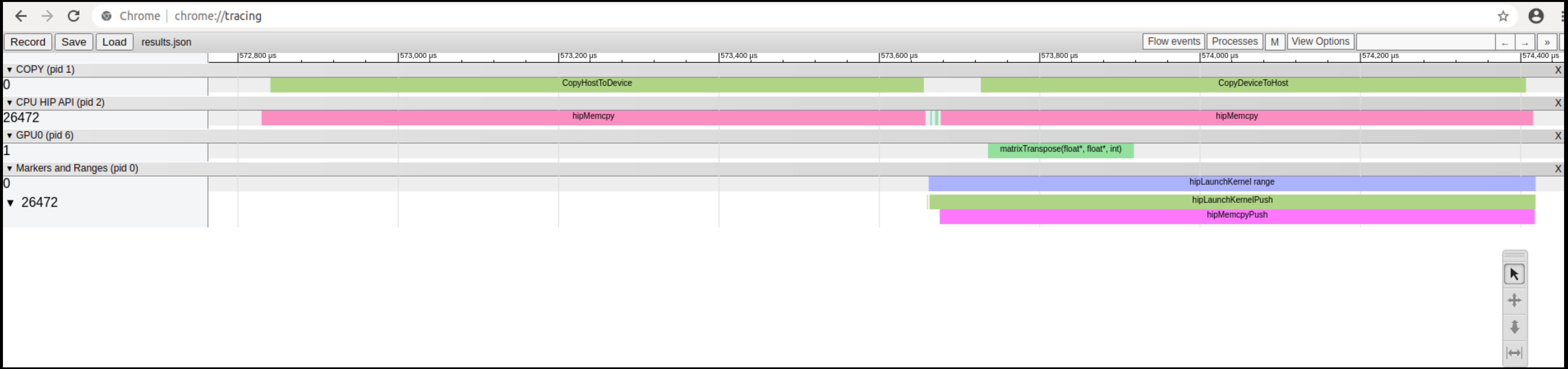
rocprof: Collecting application traces (4)

- rocprofiler can collect multiple trace modes simultaneously
  - `$ /opt/rocm/bin/rocprof --hsa-trace --hip-trace <app with arguments>`
  - This command will additionally add HIP API calls to the trace



rocprof: Collecting application traces (5)

- Rocprof can collect user code-markers using rocTX
  - See MatrixTranspose.cpp example on roctracer GitHub page for sample in-code usage
  - `$ /opt/rocm/bin/rocprof --hip-trace --roctx-trace <app with arguments>`



rocprof: Collecting hardware counters

- rocprofiler 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 by this command containing all of the requested counters

rocprof: Commonly Used 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 running that use 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 --timestamps 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 file will automatically include a column that calculates kernel duration
    - Note: the duration is aggregated over repeated calls to the same kernel

rocprof: Multiple MPI Ranks

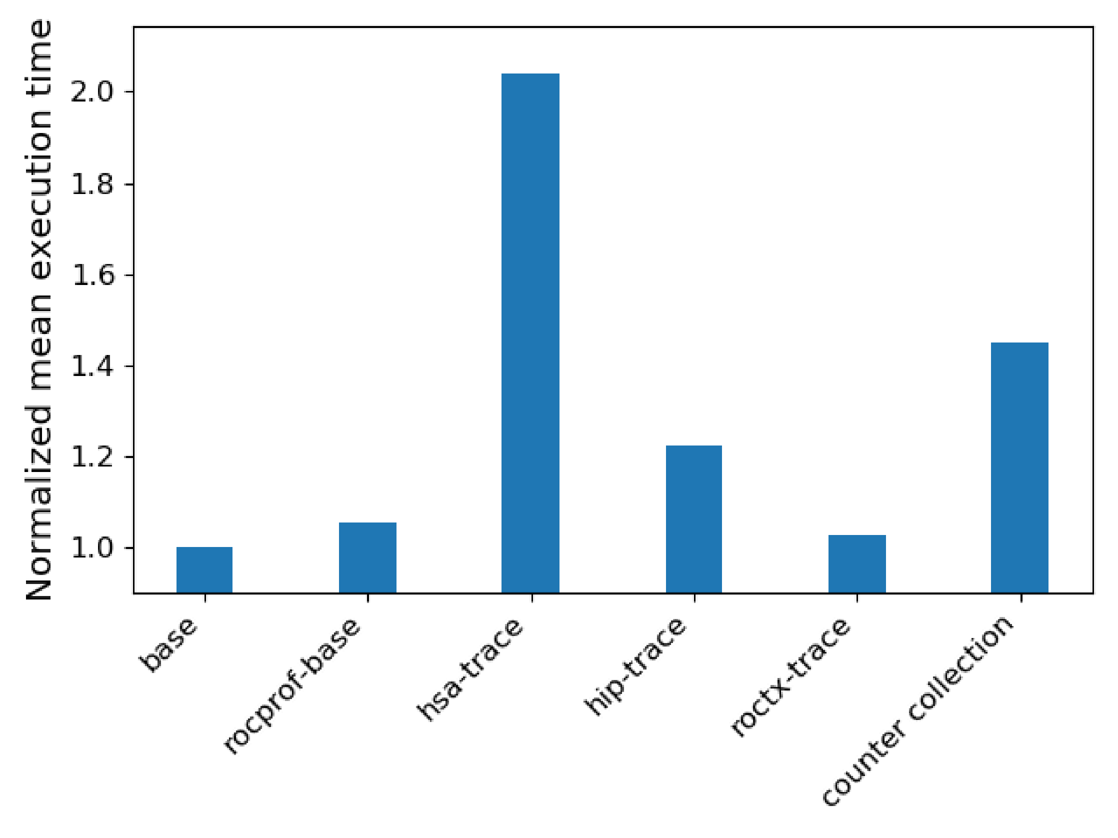
- rocprof can collect counters and traces for multiple MPI ranks.
- Say you want to profile an application usually called like this:
  - `mpiexec –np <n> ./Jacobi_hip –g <x> <y>`
  - Then invoke the profiler by executing:

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

- This will produce a single unified CSV file for all ranks
- Multi-node profiling currently isn’t supported

rocprof: Profiling Overhead

Simple estimation of profiling overhead, obtained via wall-clock timing of entire application run via Linux ‘time’ utility:



Disclaimer