



Introduction to AMD ROCm™ Ecosystem

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

LUMI, 2nd phase pilot projects training on LUMI-G
23/08/2022

AMD
together we advance_

Agenda

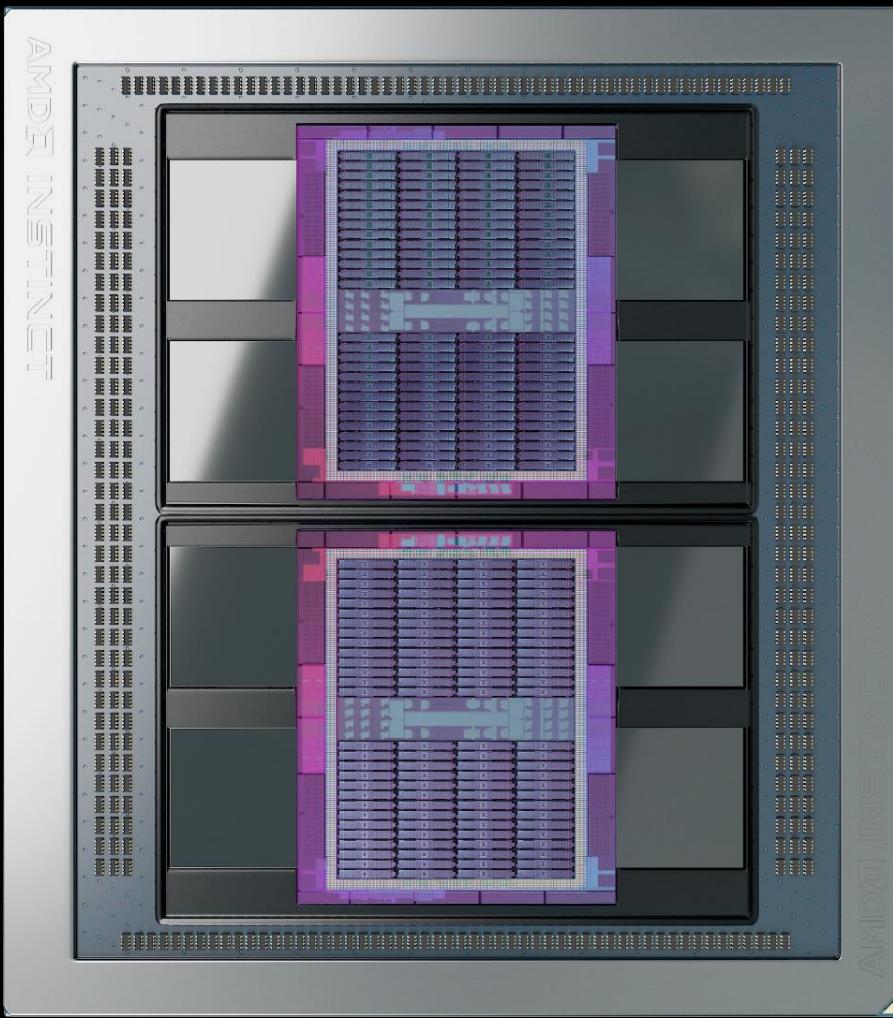
-
1. Introduction to the Architecture
 2. Introduction to ROCm and HIP
 3. Porting Applications to HIP
 4. ROCm libraries
 5. Profiling
 6. Debugging

Introduction/Expectations

- This talk is a high level of our ecosystem presentation
- We avoid deep dive topics as the audience is from various domains and levels
- We plan to give more extensive introduction and advanced training
- We hope that you can identify topics that you would like further training
- Contact the LUMI User Support Team for further training requests



Introduction to the Architecture



AMD INSTINCT™ MI250X

WORLD'S MOST ADVANCED DATA CENTER ACCELERATOR

58B

Transistors in 6nm

220

Compute Units

880

2nd Gen Matrix Cores

128

GB HBM2E @ 3.2 TB/s

<https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf>



2ND GENERATION CDNA ARCHITECTURE TAILORED-BUILT FOR HPC & AI

TSMC 6NM
TECHNOLOGY

UP TO 110 CU PER
GRAPHICS CORE DIE

4 MATRIX CORES PER
COMPUTE UNIT

MATRIX CORES
ENHANCED FOR HPC

8 INFINITY FABRIC
LINKS PER DIE

SPECIAL FP32 OPS FOR
DOUBLE THROUGHPUT

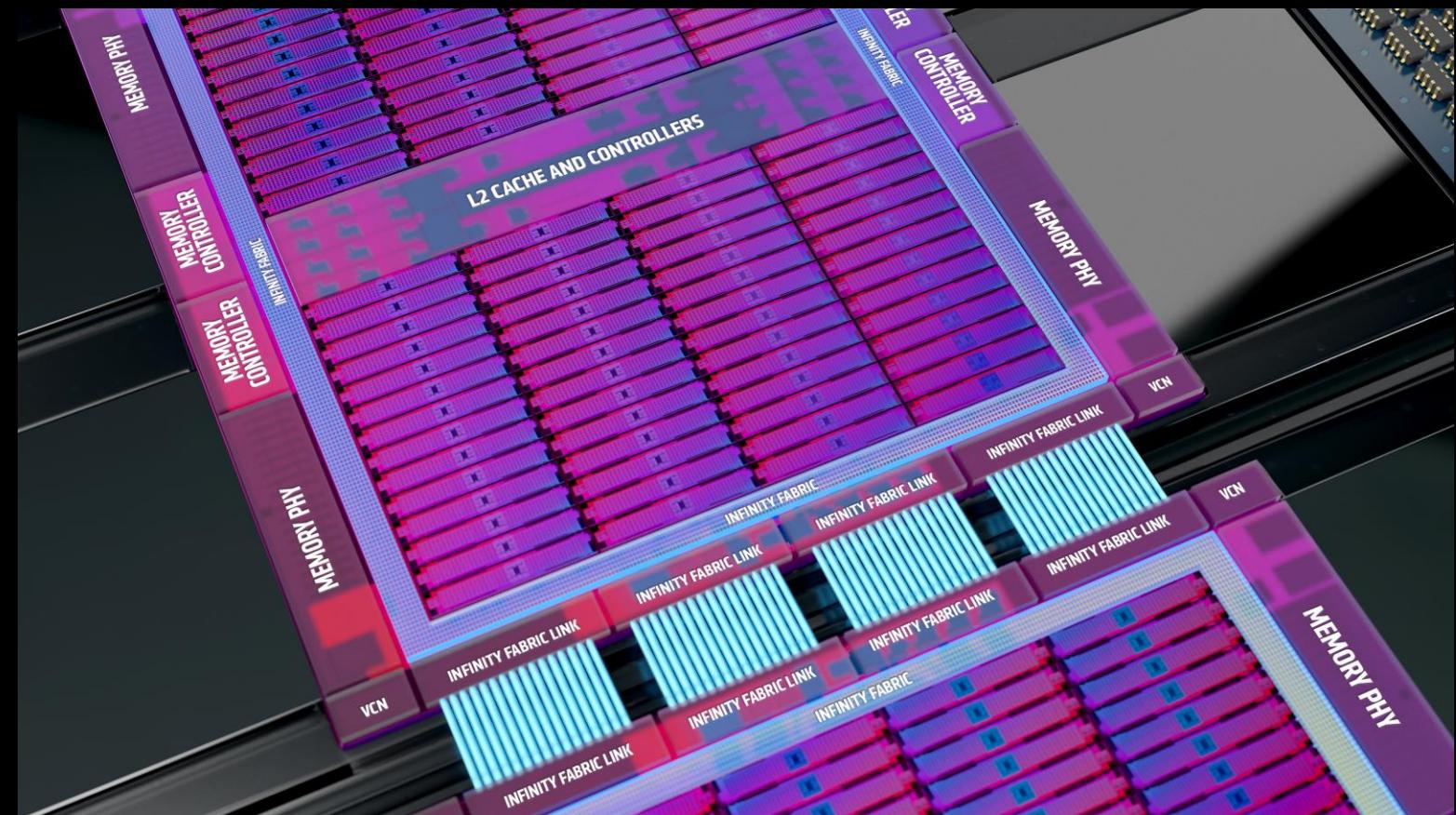
MULTI-CHIP DESIGN

TWO GPU DIES IN PACKAGE TO MAXIMIZE COMPUTE & DATA THROUGHPUT

INFINITY FABRIC FOR CROSS-DIE CONNECTIVITY

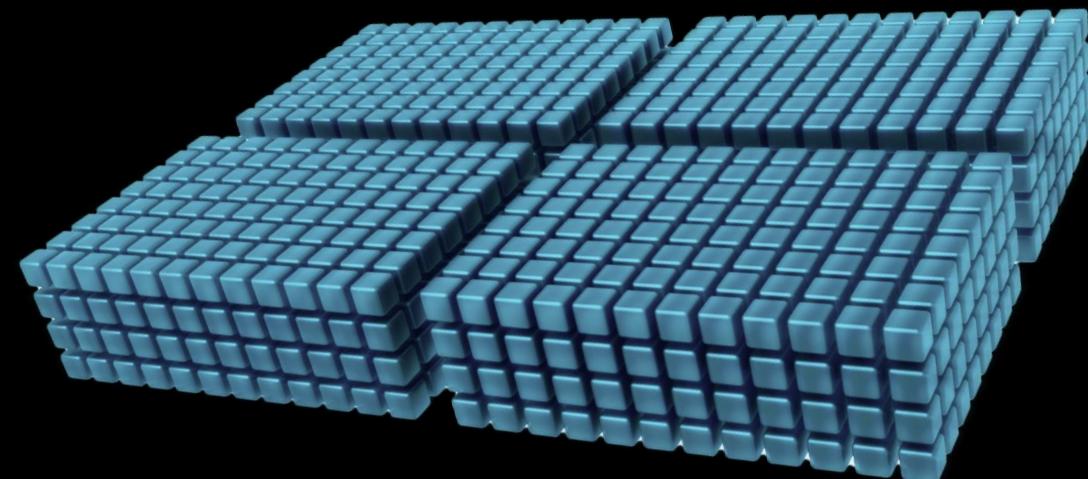
4 LINKS RUNNING AT 25GBPS

400GB/S OF BI-DIRECTIONAL BANDWIDTH



2nd GENERATION MATRIX CORES

OPTIMIZED COMPUTE UNITS FOR SCIENTIFIC COMPUTING



DOUBLE PRECISION (FP64)
MATRIX CORE THROUGHPUT
REPRESENTATION

MI100 MATRIX CORES

OPS/CLOCK/COMPUTE UNIT

No FP64 Matrix Core

256 FP32

1024 FP16

512 BF16

512 INT8

MI250X MATRIX CORES

OPS/CLOCK/COMPUTE UNIT

256 FP64

256 FP32

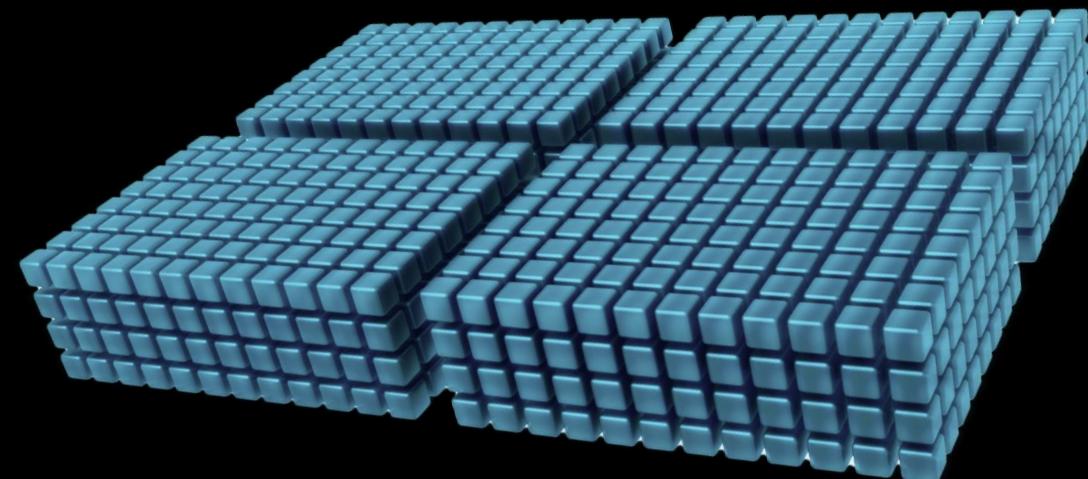
1024 FP16

1024 BF16

1024 INT8

2nd GENERATION MATRIX CORES

OPTIMIZED COMPUTE UNITS FOR SCIENTIFIC COMPUTING



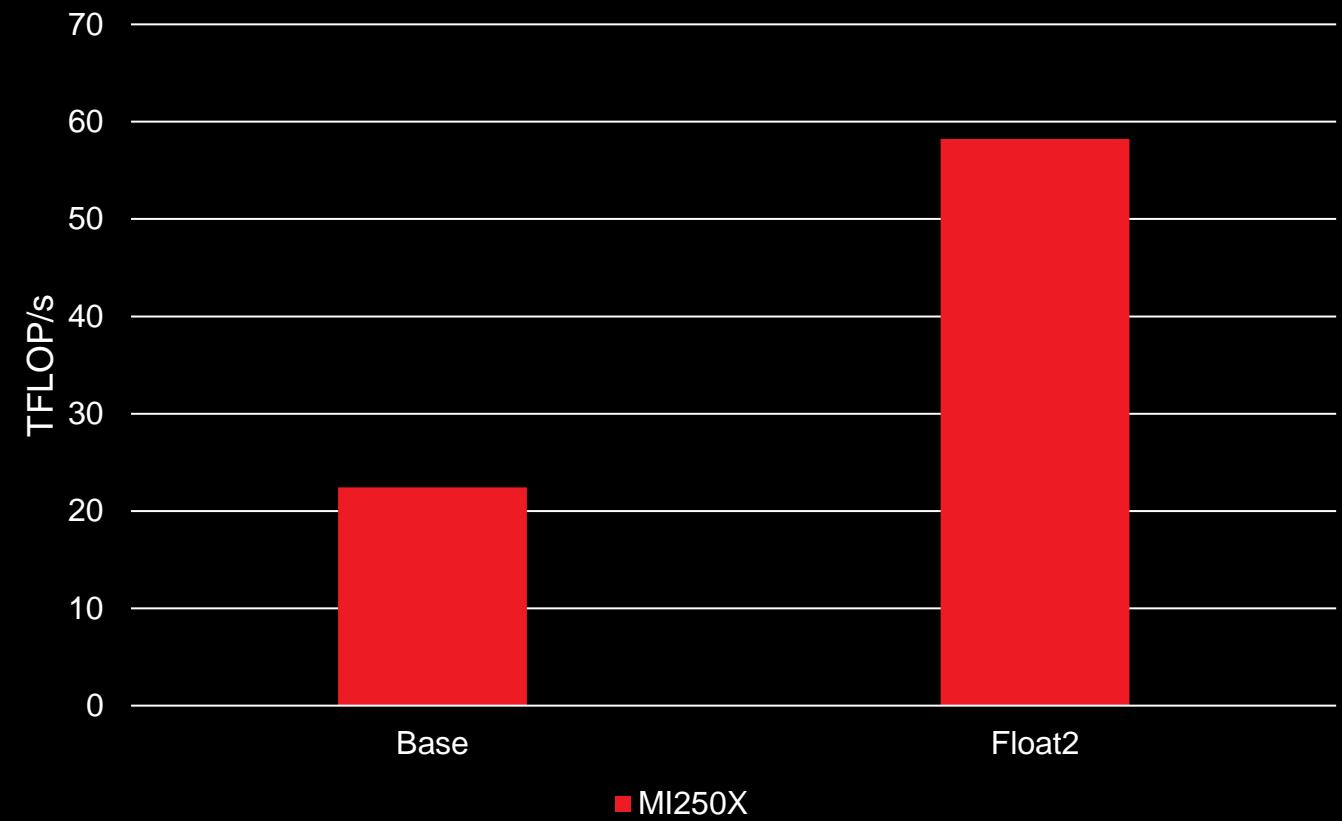
- Current support for using MFMA instructions:
 - AMD libraries: rocBLAS
 - Intrinsics
 - Inline assembly
- Not currently supported:
 - Libraries of device functions, utilizing the matrix operations, that can be called from kernels
 - Abstraction frameworks (Kokkos, Raja, OCCA)
 - These would have to use one of the other mechanisms internally

NEW IN AMD INSTINCT MI250X PACKED FP32

FP64 PATH USED TO EXECUTE
TWO COMPONENT VECTOR
INSTRUCTIONS ON FP32

DOUBLES FP32 THROUGHPUT
PER CLOCK PER COMPUTE UNIT

pk_FMA, pk_ADD, pk_MUL, pk_MOV
operations



<https://www.amd.com/en/technologies/infinity-hub/mini-hacc>

2nd phase pilot projects training on LUMI-G

From AMD MI100 to AMD MI250X

MI100

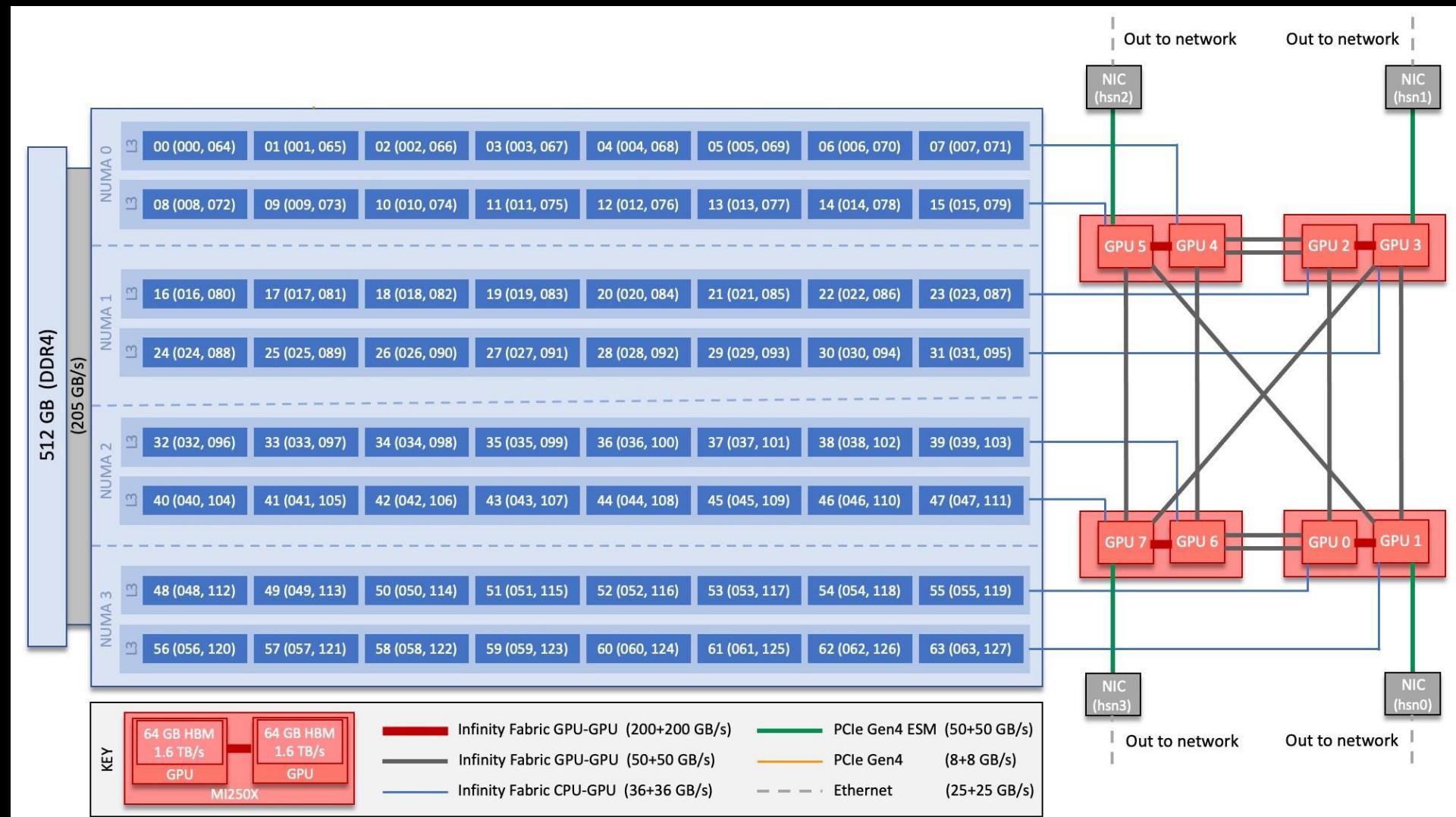
- One graphic compute die (GCD)
- 32GB of HBM2 memory
- 11.5 TFLOPS peak performance per GCD
- 1.2 TB/s peak memory bandwidth per GCD
- 120 CU per GPU
- The interconnection is attached on the CPU

AMD CDNA™ 2 white paper:
<https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf>

MI250X

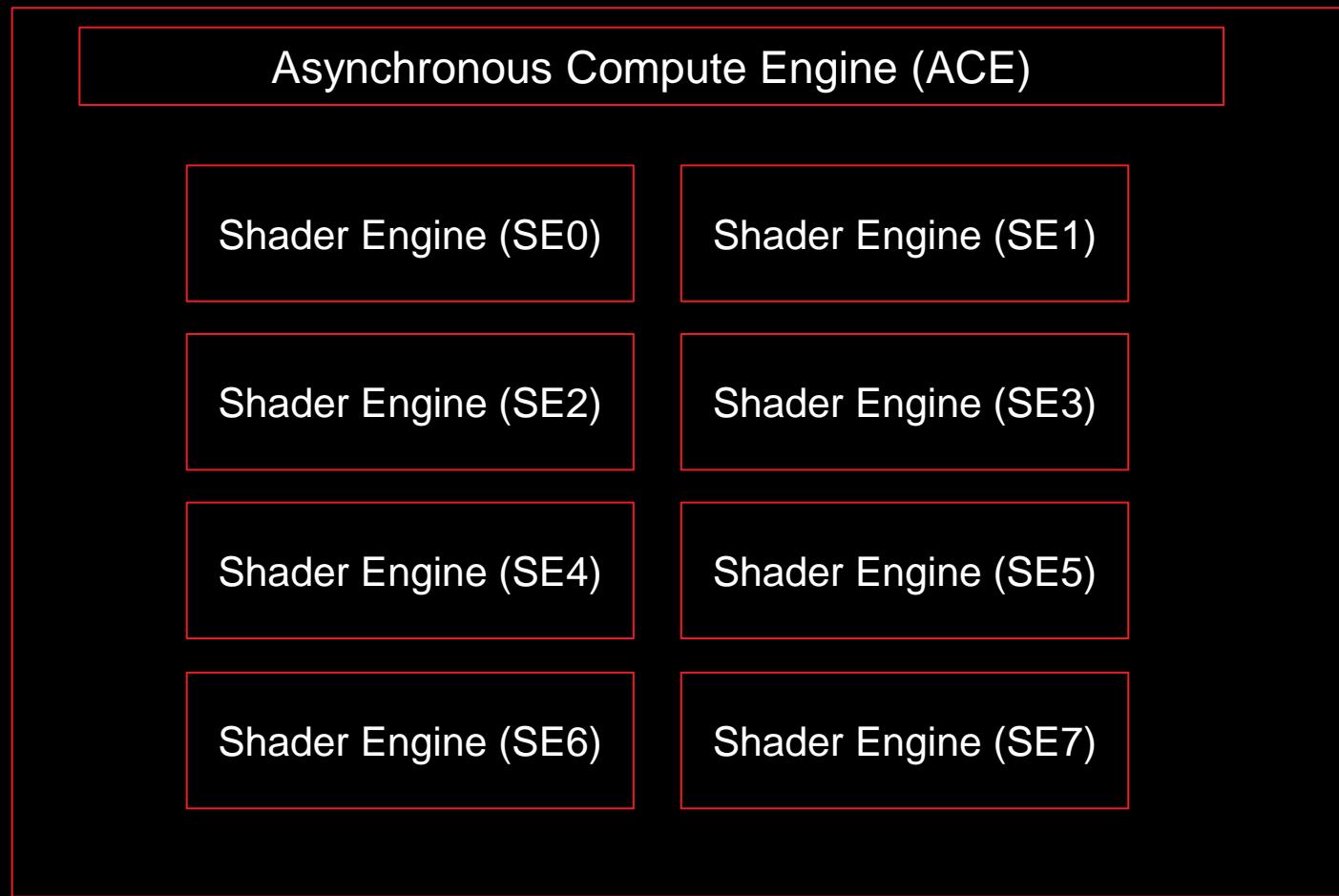
- Two graphic compute dies (GCDs)
- 64GB of HBM2e memory per GCD (total 128GB)
- 26.5 TFLOPS peak performance per GCD
- 1.6 TB/s peak memory bandwidth per GCD
- 110 CU per GCD, totally 220 CU per GPU
- The interconnection is attached on the GPU (not on the CPU)
- Both GCDs are interconnected with 200 GB/s per direction
- 128 single precision FMA operations per cycle
- AMD CDNA 2 Matrix Core supports double-precision data
- Memory coherency

LUMI – MI250X

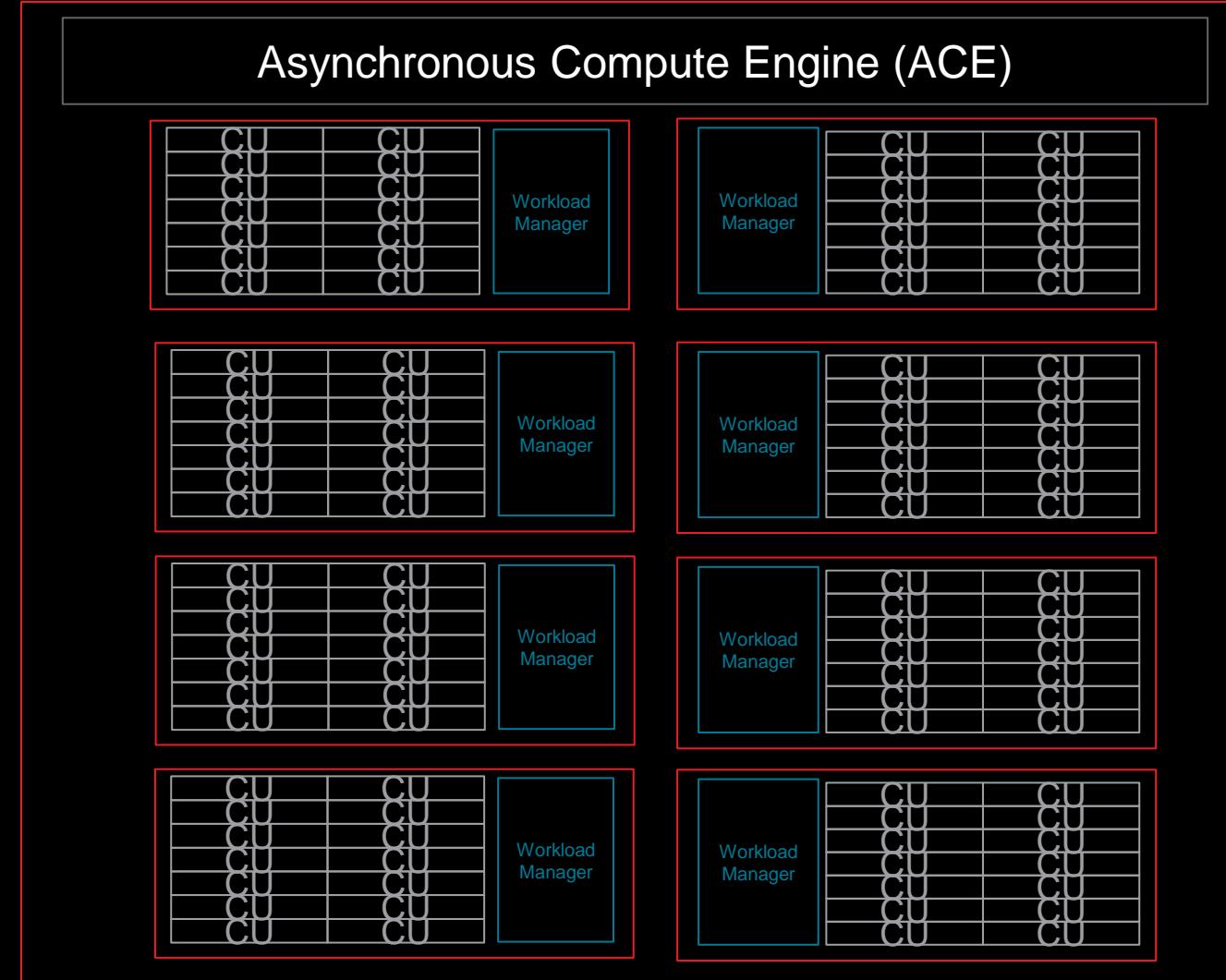
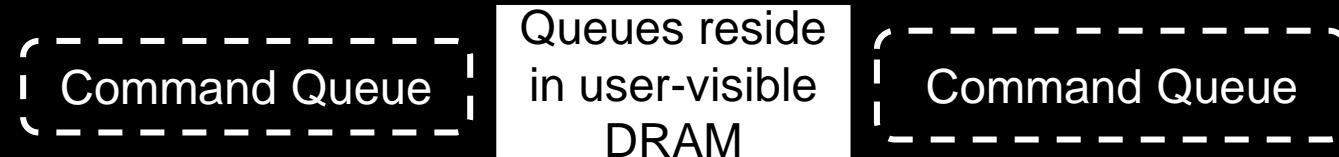


Credit: ORNL, https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html

AMD GCN GPU Hardware Layout (MI250X one GCD)



AMD GCN GPU Hardware Layout (MI250X one GCD)

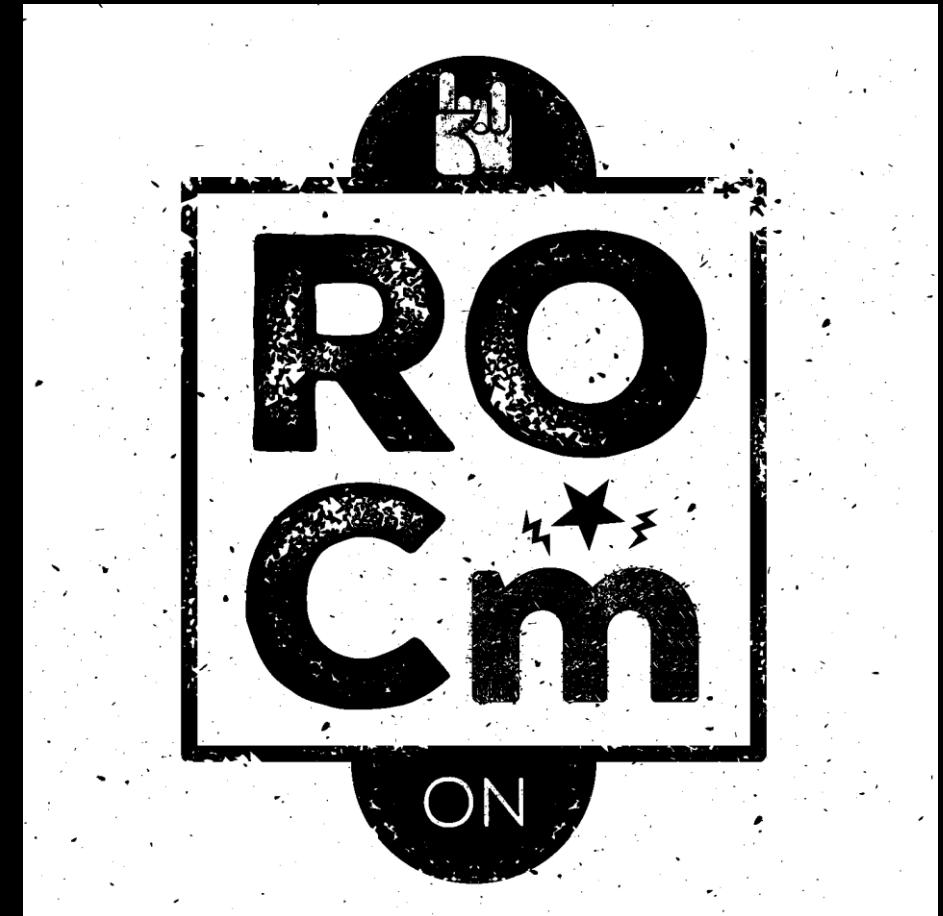




ROCM and HIP

ROCM - Radeon Open Compute Platform

- HIP is part of a larger software distribution called the Radeon Open Compute Platform, or ROCm, Package
- Install instructions and documentation can be found here:
 - https://rocmdocs.amd.com/en/latest/Installation_Guide/Installation-Guide.html
- The ROCm package provides libraries and programming tools for developing HPC and ML applications on AMD GPUs
- All the ROCm environment and the libraries are provided from the supercomputer, usually, there is no need to install something yourselves
- Heterogeneous System Architecture (HSA) runtime is an API that exposes the necessary interfaces to access and interact with the hardware driven by AMDGPU driver

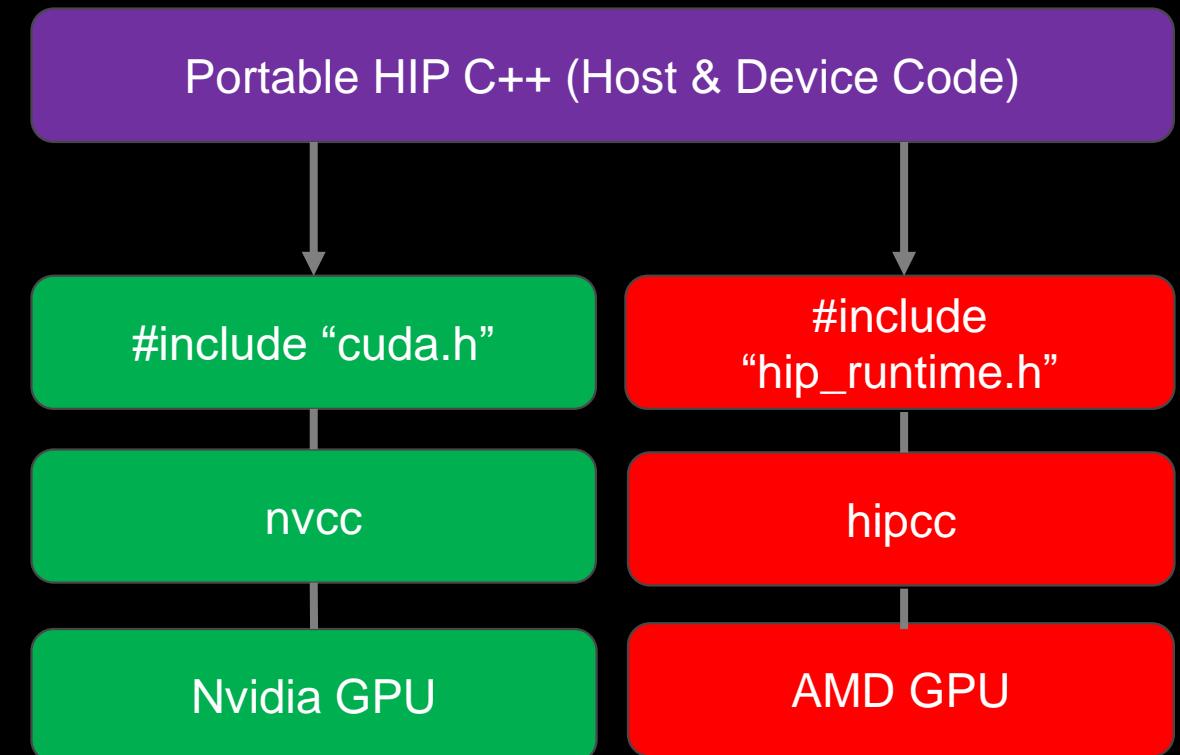


What is HIP?

AMD's **H**eterogeneous-compute **I**nterface for **P**ortability, or **HIP**, is a C++ runtime API and kernel language that allows developers to create portable applications that can run on AMD's accelerators as well as CUDA devices

HIP:

- Is open-source
- Provides an API for an application to leverage GPU acceleration for both AMD and CUDA devices
- Syntactically similar to CUDA. Most CUDA API calls can be converted in place: cuda -> hip
- Supports a strong subset of CUDA runtime functionality



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

```
cudaMalloc(&d_x, N*sizeof(double));
```

```
cudaMemcpy(d_x, x, N*sizeof(double),  
cudaMemcpyHostToDevice);
```

```
cudaDeviceSynchronize();
```

HIP

```
hipMalloc(&d_x, N*sizeof(double));
```

```
hipMemcpy(d_x, x, N*sizeof(double),  
hipMemcpyHostToDevice);
```

```
hipDeviceSynchronize();
```

Launching a kernel

CUDA KERNEL LAUNCH SYNTAX

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

HIP KERNEL LAUNCH SYNTAX

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

Or

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

Device Kernels: The Grid

- In HIP, kernels are executed on a 3D "grid"
 - You might feel comfortable thinking in terms of a mesh of points, but it's not required
- The "grid" is what you can map your problem to
 - It's not a physical thing, but it can be useful to think that way
- AMD devices (GPUs) support 1D, 2D, and 3D grids, but most work maps well to 1D
- Each dimension of the grid partitioned into equal sized "blocks"
- Each block is made up of multiple "threads"
- The grid and its associated blocks are just organizational constructs
 - The threads are the things that do the work
- If you're familiar with CUDA already, the grid+block structure is very similar in HIP

Device Kernels: The Grid

Some Terminology:

CUDA	HIP	OpenCL™
grid	grid	NDRange
block	block	work group
thread	work item / thread	work item
warp	wavefront	sub-group

The Grid: blocks of threads in 1D



Threads in grid have access to:

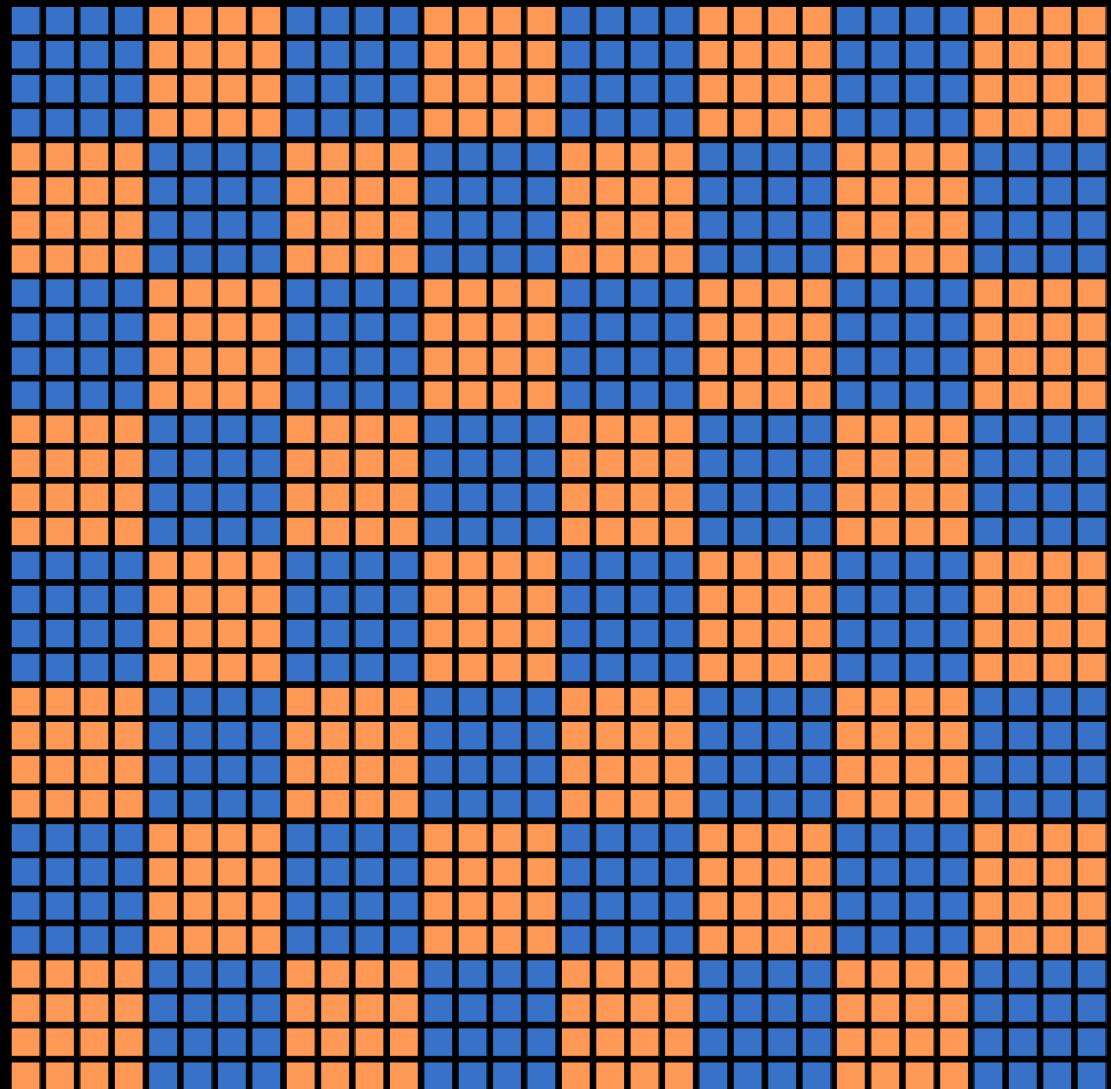
- Their respective block: `blockIdx.x`
- Their respective thread ID in a block: `threadIdx.x`
- Their block's dimension: `blockDim.x`
- The number of blocks in the grid: `gridDim.x`

The Grid: blocks of threads in 2D

- Each color is a block of threads
- Each small square is a thread
- The concept is the same in 1D and 2D
- In 2D each block and thread now has a two-dimensional index

Threads in grid have access to:

- Their respective block IDs: `blockIdx.x`, `blockIdx.y`
- Their respective thread IDs in a block: `threadIdx.x`, `threadIdx.y`
- Etc.



Kernels

A simple embarrassingly parallel loop

```
for (int i=0;i<N;i++) {  
    h_a[i] *= 2.0;  
}
```

Can be translated into a GPU kernel:

```
__global__ void myKernel(int N, double *d_a) {  
    int i = threadIdx.x + blockIdx.x*blockDim.x;  
    if (i<N) {  
        d_a[i] *= 2.0;  
    }  
}
```

- A device function that will be launched from the host program is called a kernel and is declared with the `__global__` attribute
- Kernels should be declared `void`
- All threads execute the kernel's body "simultaneously"
- Each thread uses its unique thread and block IDs to compute a global ID
- There could be more than N threads in the grid

Kernels

Kernels are launched from the host:

```
dim3 threads(256,1,1);           //3D dimensions of a block of threads
dim3 blocks((N+256-1)/256,1,1); //3D dimensions the grid of blocks

hipLaunchKernelGGL(myKernel,
                  blocks,           //Kernel name (__global__ void function)
                  threads,          //Grid dimensions
                  0,                //Block dimensions
                  0,                //Bytes of dynamic LDS space
                  0,                //Stream (0=NULL stream)
                  N, a);           //Kernel arguments
```

Also supported similar to CUDA kernel launch syntax:

```
myKernel<<<blocks, threads, 0, 0>>>(N,a);
```

Difference between HIP and CUDA

Some things to be aware of writing HIP, or porting from CUDA:

- AMD GCN hardware ‘warp’ size = 64 (warps are referred to as ‘wavefronts’ in AMD documentation)
- Device and host pointers allocated by HIP API use flat addressing
 - Unified virtual addressing is available
- Dynamic parallelism not currently supported
- CUDA 9+ thread independent scheduling not supported (e.g., no `__syncwarp`)
- Some CUDA library functions do not have AMD equivalents
- Shared memory and registers per thread can differ between AMD and Nvidia hardware
- Inline PTX or AMD GCN assembly is not portable

Despite differences, majority of CUDA code in applications can be simply translated.

Usage of hipcc

Usage is straightforward. Accepts all/any flags that clang accepts, e.g.,

```
hipcc --offload-arch=gfx90a dotprod.cpp -o dotprod
```

Set HIPCC_VERBOSE=7 to see a bunch of useful information

- Compile and link lines
- Various paths

```
$ HIPCC_VERBOSE=7 hipcc --offload-arch=gfx90a dotprod.cpp -o dotprod
HIP_PATH=/opt/rocm-5.2.0
HIP_PLATFORM=amd
HIP_COMPILER=clang
HIP_RUNTIME=rocclr
ROCM_PATH=/opt/rocm-5.2.0
...
hipcc-args: --offload-arch=gfx90a dotprod.cpp -o dotprod
hipcc-cmd: /opt/rocm-5.2.0/llvm/bin/clang++ -std=c++11 -hc -D__HIPCC__ -isystem /opt/rocm-5.2.0/llvm/lib/clang/14.0.0/include
-isystem /opt/rocm-5.2.0/has/include -isystem /opt/rocm-5.2.0/include -offload-arch=gfx90a -O3 ...
```

- You can use also *hipcc -v* ... to print some information
- With the command *hipconfig* you can see many information about environment variables declaration

HIP API

- Device Management: `hipSetDevice()`, `hipGetDevice()`, `hipGetDeviceProperties()`
- Memory Management: `hipMalloc()`, `hipMemcpy()`, `hipMemcpyAsync()`, `hipFree()`, `hipHostMalloc()`
- Streams: `hipStreamCreate()`, `hipSynchronize()`, `hipStreamSynchronize()`, `hipStreamFree()`
- Events: `hipEventCreate()`, `hipEventRecord()`, `hipStreamWaitEvent()`, `hipEventElapsedTime()`
- Device Kernels: `__global__`, `__device__`, `hipLaunchKernelGGL()`
- Device code:
 - `threadIdx`, `blockIdx`, `blockDim`, `__shared__`
 - 200+ math functions covering entire CUDA math library
- Error handling: `hipGetLastError()`, `hipGetErrorString()`
- More information: https://docs.amd.com/bundle/HIP_API_Guide/page/modules.html

Error Checking

- Most HIP API functions return error codes of type `hipError_t`

```
hipError_t status1 = hipMalloc(...);  
hipError_t status2 = hipMemcpy(...);
```

- If API function was error-free, returns `hipSuccess`, otherwise returns an error code

- Can also peek/get at last error returned with

```
hipError_t status3 = hipGetLastError();  
hipError_t status4 = hipPeekLastError();
```

- Can get a corresponding error string using `hipGetString(status)`. Helpful for debugging, e.g.,

```
#define HIP_CHECK(command) {      \  
    hipError_t status = command;  \  
    if (status!=hipSuccess) {    \  
        std::cerr << "Error: HIP reports " << hipGetString(status) << std::endl; \  
        std::abort(); } }
```

Streams

- A stream in HIP is a queue of tasks (e.g., kernels, memcpys, events)
 - Tasks enqueued in a stream are **completed in the order enqueue**d
 - Tasks being executed in different streams are allowed to overlap and share device resources
- Streams are created via:

```
hipStream_t stream;
hipStreamCreate(&stream);
```
- And destroyed via:

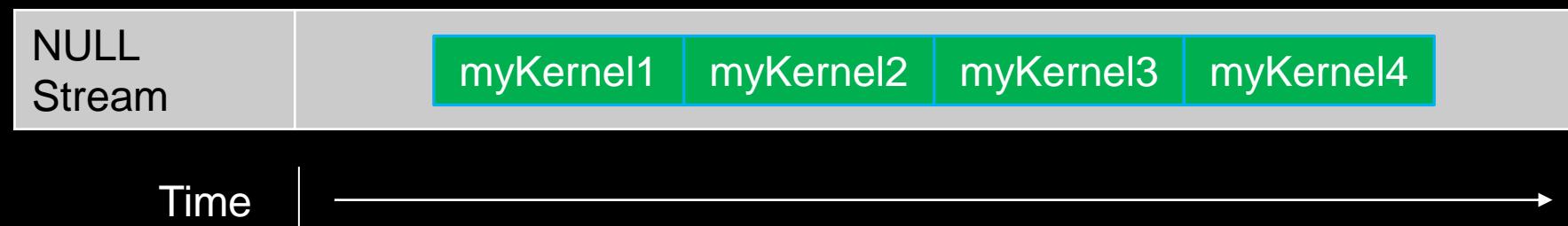
```
hipStreamDestroy(stream);
```
- Passing 0 or **NULL** as the `hipStream_t` argument to a function instructs the function to execute on a stream called the ‘NULL Stream’:
 - No task on the NULL stream will begin until **all previously enqueued tasks in all other streams have completed**
 - Blocking calls like `hipMemcpy` run on the NULL stream

Streams

- Suppose we have 4 small kernels to execute:

```
hipLaunchKernelGGL(myKernel1, dim3(1), dim3(256), 0, 0, 256, d_a1);  
hipLaunchKernelGGL(myKernel2, dim3(1), dim3(256), 0, 0, 256, d_a2);  
hipLaunchKernelGGL(myKernel3, dim3(1), dim3(256), 0, 0, 256, d_a3);  
hipLaunchKernelGGL(myKernel4, dim3(1), dim3(256), 0, 0, 256, d_a4);
```

- Even though these kernels use only one block each, they'll execute in serial on the NULL stream:



Streams

- With streams we can effectively share the GPU's compute resources:

```
hipLaunchKernelGGL(myKernel1, dim3(1), dim3(256), 0, stream1, 256, d_a1);
hipLaunchKernelGGL(myKernel2, dim3(1), dim3(256), 0, stream2, 256, d_a2);
hipLaunchKernelGGL(myKernel3, dim3(1), dim3(256), 0, stream3, 256, d_a3);
hipLaunchKernelGGL(myKernel4, dim3(1), dim3(256), 0, stream4, 256, d_a4);
```

NULL Stream	
Stream1	myKernel1
Stream2	myKernel2
Stream3	myKernel3
Stream4	myKernel4

Note 1: Kernels must modify different parts of memory to avoid data races.

Note 2: With large kernels, overlapping computations may not help performance.

SIMD operations

Why blocks and threads?

Natural mapping of kernels to hardware:

- Blocks are dynamically scheduled onto CUs
- All threads in a block execute on the same CU
- Threads in a block share LDS memory and L1 cache
- Threads in a block are executed in **64-wide** chunks called “wavefronts”
- Wavefronts execute on SIMD units (Single Instruction Multiple Data)
- If a wavefront stalls (e.g., data dependency) CUs can quickly context switch to another wavefront

A good practice is to make the block size a multiple of 64 and have several wavefronts (e.g., 256 threads)



Porting Applications to HIP

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://github.com/ROCM-Developer-Tools/HIPIFY/blob/master/README.md>
- 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
 - It will not translate if it does not recognize a CUDA call and it will report it
- 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

- It is located 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 work in practice?
 - Hipify source code
 - Check it in to your favorite version control
 - Try to build
 - Manually work on the rest

Hipify-clang

- Build from source
- hipify-clang has unit tests using LLVM lit/FileCheck (44 tests)
- 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`
- <https://github.com/ROCM-Developer-Tools/HIP/tree/master/hipify-clang>

Gotchas

- Hipify tools are not running your application, or checking correctness
- Code relying on specific Nvidia hardware aspects (e.g., warp size == 32) may need attention after conversion
- Certain functions may not have a correspondent hip version (e.g., __shfl_down_sync)
- Hipifying can't handle inline PTX assembly
 - Can either use inline GCN ISA, or convert it to HIP
- Hipify-perl and hipify-clang can both convert library calls
- None of the tools convert your build system script such as CMAKE or whatever else you use. The user is responsible to find the appropriate flags and paths to build the new converted HIP code.

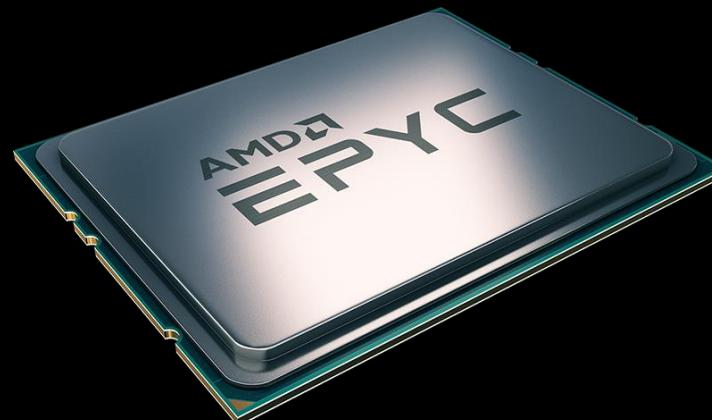
What to look for when porting:

- 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! Rely on warpSize device definition, #define WARPSIZE size, or props.warpSize from host
- Code geared toward limiting size of register file on NVIDIA hardware
- Unsupported functions

A Tale of Host and Device

Source code in HIP has two flavors: Host code and Device code

- The Host is the CPU
- Host code runs here
- Usual C++ syntax and features
- Entry point is the ‘main’ function
- HIP API can be used to create device buffers, move between host and device, and launch device code.
- The Device is the GPU
- Device code runs here
- C-like syntax
- Device codes are launched via “kernels”
- Instructions from the Host are enqueued into “streams”



Fortran

- First Scenario: Fortran + CUDA C/C++
 - Assuming there is no CUDA code in the Fortran files.
 - Hipify CUDA
 - Compile and link with hipcc
- Second Scenario: CUDA Fortran
 - There is no hipify equivalent but there is another approach...
 - HIP functions are callable from C, using `extern C`
 - See hipfort

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-like syntax
 - Wrap the kernel launch in a C function
 - Call the C function from Fortran through Fortran's ISO_C_binding. It requires Fortran 2008 because of the pointers utilization.
- 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>

Alternatives to HIP

- Can also target AMD GPUs through OpenMP 5.0 target offload
 - ROCm provides OpenMP support
 - AMD OpenMP compiler (AOMP) could integrate updated improvements regarding OpenMP offloading performance, sometimes experimental stuff to validate before ROCm integration (<https://github.com/ROCM-Developer-Tools/aomp>)
 - GCC provides OpenMP offload support.
- GCC will provide OpenACC
- Clacc from ORNL: <https://github.com/llvm-doe-org/llvm-project/tree/clacc/main> OpenACC from LLVM only for C (Fortran and C++ in the future)
 - Translate OpenACC to OpenMP Offloading

OpenMP Offload GPU Support

- ROCm and AOMP
 - ROCm supports both HIP and OpenMP
 - AOMP: the AMD OpenMP research compiler, it is used to prototype the new OpenMP features for ROCm
- HPE Compilers
 - Provides offloading support to AMD GPUs, through OpenMP, HIP, and OpenACC (only for Fortran)
- GNU compilers:
 - Provide OpenMP and OpenACC offloading support for AMD GPUs
 - GCC 11: Supports AMD GCN gfx908
 - GCC 13: Supports AMD GCN gfx90a

Understanding the hardware options

- ***rocminfo***

- 110 CUs
- Wavefront of size 64
- 4 SIMDs per CU

```
#pragma omp target teams distribute parallel for simd
```

Options for `pragma omp teams target`:

- `num_teams(220)`: Multiple number of workgroups with regards the compute units
- `thread_limit(256)`: Threads per workgroup
- Thread limit is multiple of 64
- Teams*`thread_limit` should be multiple or a divisor of the trip count of a loop

Node:	11
Device Type:	GPU
Cache Info:	
L1:	16(0x10) KB
L2:	8192(0x2000) KB
Chip ID:	29704(0x7408)
Cacheline Size:	64(0x40)
Max Clock Freq. (MHz):	1700
BDFID:	56832
Internal Node ID:	11
Compute Unit:	110
SIMDs per CU:	4
Shader Engines:	8
Shader Arrs. per Eng.:	1
WatchPts on Addr. Ranges:	4
Features:	KERNEL_DISPATCH
Fast F16 Operation:	TRUE
Wavefront Size:	64(0x40)
Workgroup Max Size:	1024(0x400)
Workgroup Max Size per Dimension:	
x	1024(0x400)
y	1024(0x400)
z	1024(0x400)
Max Waves Per CU:	32(0x20)
Max Work-item Per CU:	2048(0x800)



ROCm Libraries

ROCM GPU Libraries

ROCM provides several GPU math libraries

- Typically, two versions:
 - roc* -> AMD GPU library, usually written in HIP
 - hip* -> Thin interface between roc* and Nvidia cu* library

When developing an application meant to target both CUDA and AMD devices, use the hip* libraries (portability)

When developing an application meant to target only AMD devices, may prefer the roc* library API (performance).

- Some roc* libraries perform **better** by using addition APIs not available in the cu* equivalents



hipBLAS



rocBLAS



cuBLAS

AMD Math Library Equivalents: “Decoder Ring”

CUBLAS

ROCLAS

Basic Linear Algebra
Subroutines

CUFFT

ROCFFT

Fast Fourier Transforms

CURAND

ROCRAND

Random Number
Generation

THRUST

ROCTRUST

C++ Parallel Algorithms

CUB

ROCPIM

Optimized Parallel
Primitives

AMD Math Library Equivalents: “Decoder Ring”

CUSPARSE

ROCSPARSE

Sparse BLAS, SpMV, etc.

CUSOLVER

ROCSOLVER

Linear Solvers

AMGX

ROCALUTION

Solvers and preconditioners
for sparse linear systems

[GITHUB.COM/ROCM-DEVELOPER-TOOLS/HIP → HIP_PORTING_GUIDE.MD FOR A COMPLETE LIST](https://github.com/rocm-developer-tools/HIP/tree/main/HIP_PORTING_GUIDE.MD)

Some Links to Key Libraries

- BLAS
 - rocBLAS (<https://github.com/ROCMSoftwarePlatform/rocBLAS>)
 - hipBLAS (<https://github.com/ROCMSoftwarePlatform/hipBLAS>)
- FFTs
 - rocFFT (<https://github.com/ROCMSoftwarePlatform/rocFFT>)
 - hipFFT (<https://github.com/ROCMSoftwarePlatform/hipFFT>)
- Random number generation
 - rocRAND (<https://github.com/ROCMSoftwarePlatform/rocRAND>)
- Sparse linear algebra
 - rocSPARSE (<https://github.com/ROCMSoftwarePlatform/rocSPARSE>)
 - hipSPARSE (<https://github.com/ROCMSoftwarePlatform/hipSPARSE>)
- Iterative solvers
 - rocALUTION (<https://github.com/ROCMSoftwarePlatform/rocALUTION>)
- Parallel primitives
 - rocPRIM (<https://github.com/ROCMSoftwarePlatform/rocPRIM>)
 - hipCUB (<https://github.com/ROCMSoftwarePlatform/hipCUB>)

AMD Machine Learning Library Support

Machine Learning Frameworks:

- Tensorflow: <https://github.com/ROCmSoftwarePlatform/tensorflow-upstream>
- Pytorch: <https://github.com/ROCmSoftwarePlatform/pytorch>
- Caffe: <https://github.com/ROCmSoftwarePlatform/hipCaffe>

Machine Learning Libraries:

- MIOpen (similar to cuDNN): <https://github.com/ROCmSoftwarePlatform/MIOpen>
- Tensile (GEMM Autotuner): <https://github.com/ROCmSoftwarePlatform/Tensile>
- RCCL (ROCm analogue of NCCL): <https://github.com/ROCmSoftwarePlatform/rccl>
- Horovod (Distributed ML): <https://github.com/ROCmSoftwarePlatform/horovod>

Benchmarks:

- DeepBench: <https://github.com/ROCmSoftwarePlatform/DeepBench>
- MLPerf: <https://mlperf.org>



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 Perfetto (<https://ui.perfetto.dev/>)

rocProf: Getting started + useful flags

- To get help:
 - `$ /opt/rocm-5.2.0/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: 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

rocProf: Information about the kernels

- rocprofiler can collect kernels information
 - `$ /opt/rocm/bin/rocprof --stats --basenames on <app with arguments>`
 - This will output two csv files, one with information per each call of the kernel `results.csv` and one with statistics grouped by each kernel `results.stats.csv`.
 - Content of `results.stats.csv`:

"Name",	"Calls",	"TotalDurationNs",	"AverageNs",	"Percentage"
"LocalLaplacianKernel",	1000,	817737586,	817737,	40.908259879301134
"JacobilterationKernel",	1000,	699515425,	699515,	34.994060790890174
"NormKernel1",	1001,	454737348,	454283,	22.748756969583884
"HaloLaplacianKernel",	1000,	14561933,	14561,	0.7284773865206329
"NormKernel2",	1001,	12395374,	12382,	0.620092789636225
"__amd_rocclr_fillBufferAligned.kd",	1,	7040,	7040,	0.00035218406794656007

- This way you know directly which kernels consume most of the time, it does not mean that the performance is slow, for now.

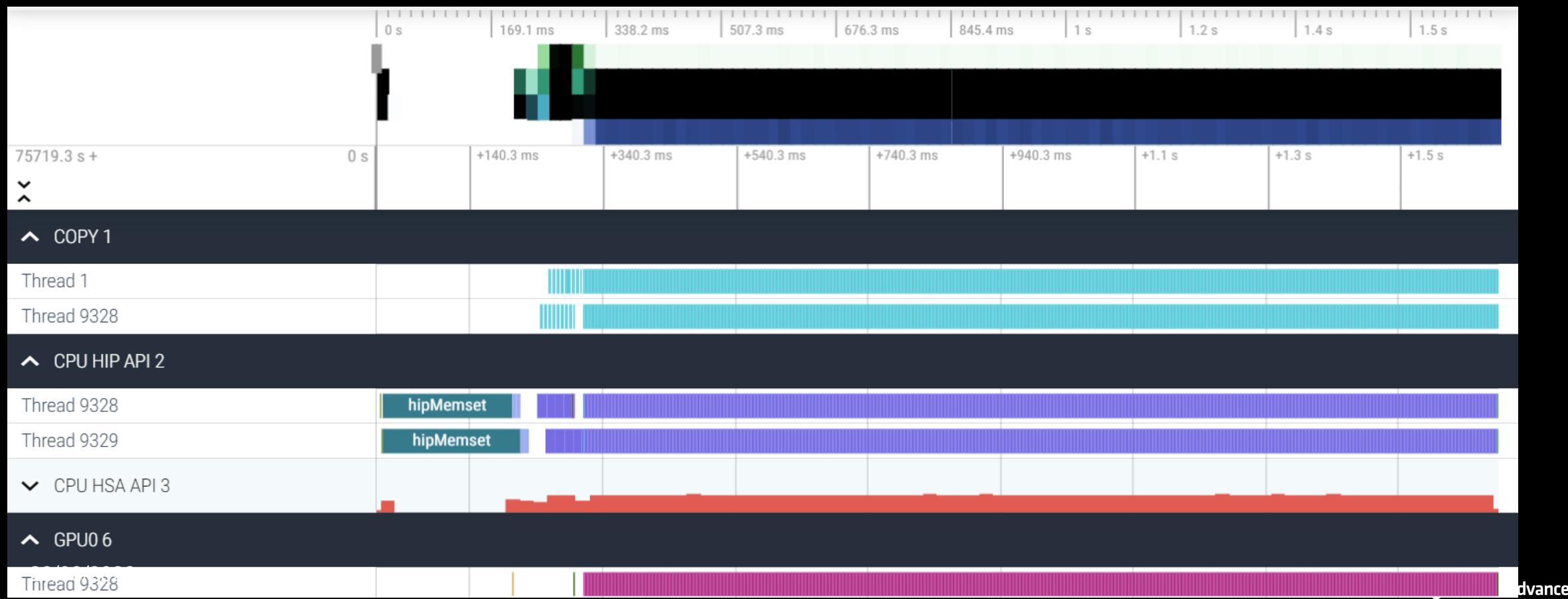
rocProf and Perfetto: Collecting and visualizing application traces

- rocprofiler can collect traces
 - `$ /opt/rocm/bin/rocprof --hip-trace --hsa-trace <app with arguments>`
 - This will output a .json file that can be visualized using the chrome browser and Perfetto (<https://ui.perfetto.dev/>)



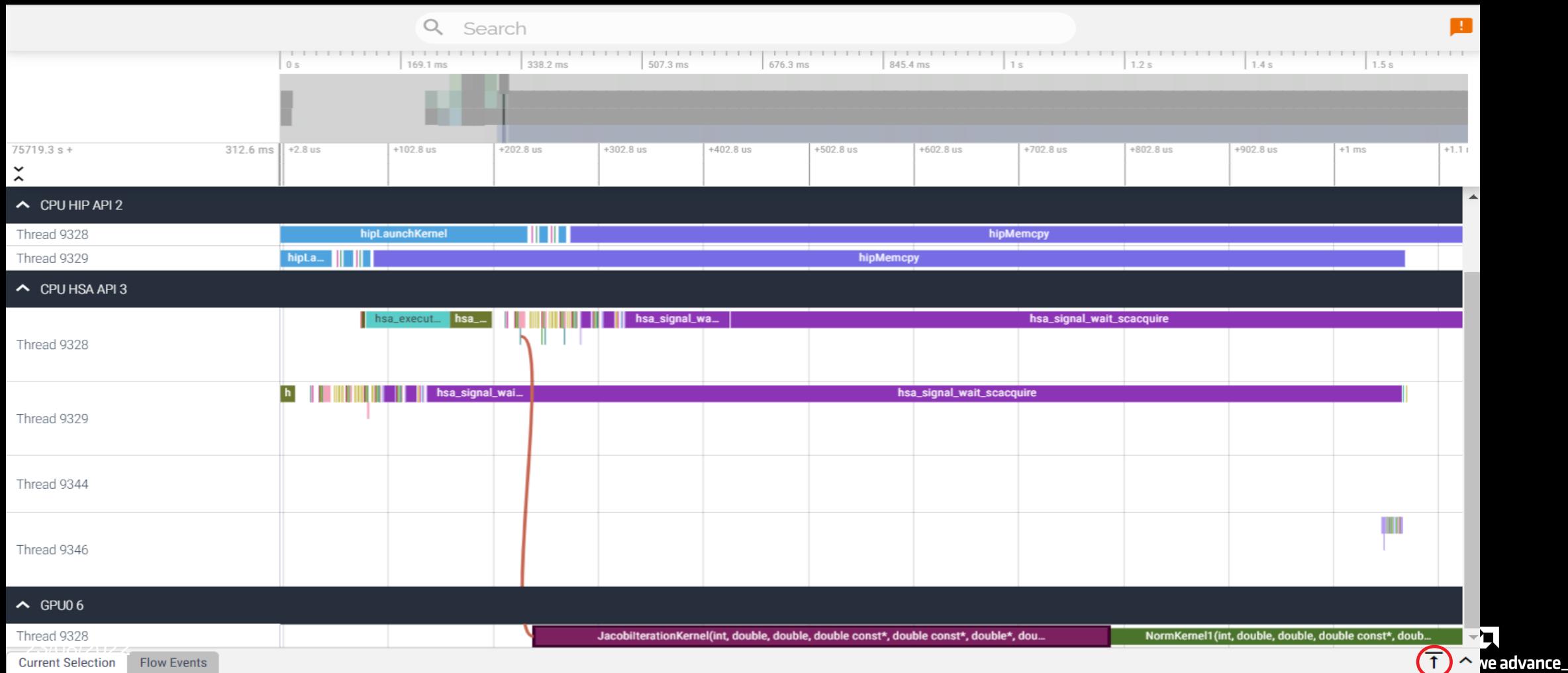
Perfetto: Visualizing application traces

- We have expanded the COPY 1, CPU HIP API 2 and GPU0 6
- X axis is time and it displays events or counters.
- Handle the zoom by keystrokes: W zoom, S zoom out, A move left, D move right

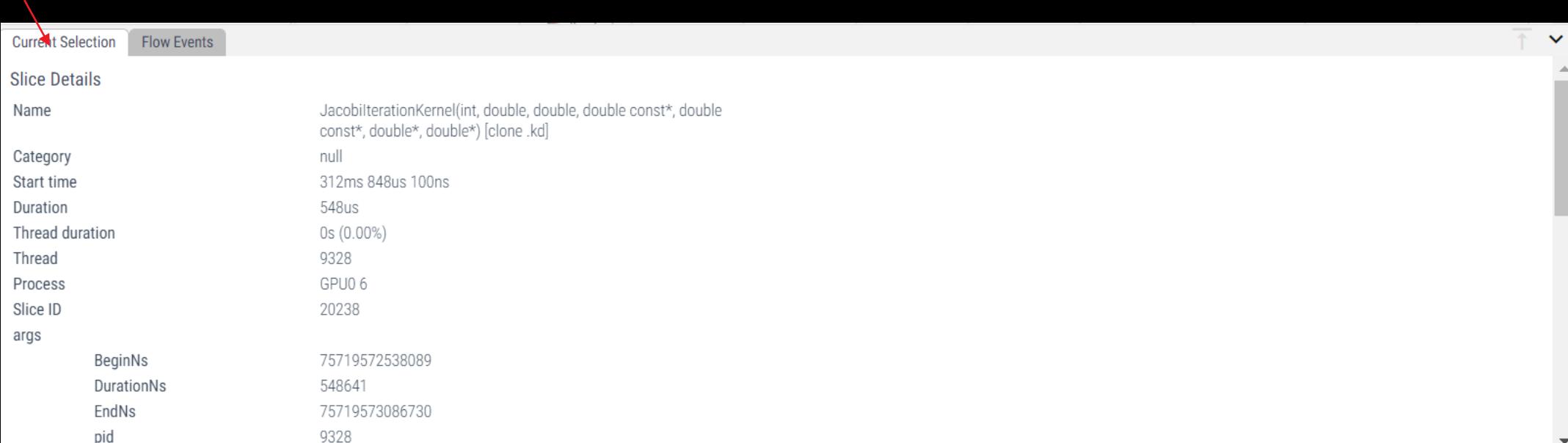


Perfetto: Kernel and flows

- Zoom and select a kernel, you can see the link to the HSA call enables the kernel
- Try to open the information for the kernel (button right down)



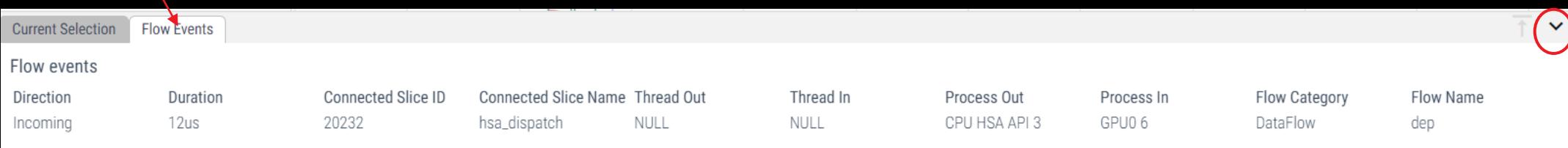
Perfetto: Information about kernels and flow events



Current Selection Flow Events

Slice Details

Name	JacobilterationKernel(int, double, double, double const*, double const*, double*, double*) [clone .kd]
Category	null
Start time	312ms 848us 100ns
Duration	548us
Thread duration	0s (0.00%)
Thread	9328
Process	GPU0 6
Slice ID	20238
args	
BeginNs	75719572538089
DurationNs	548641
EndNs	75719573086730
pid	9328



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	12us	20232	hsa_dispatch	NULL	NULL	CPU HSA API 3	GPU0 6	DataFlow	dep

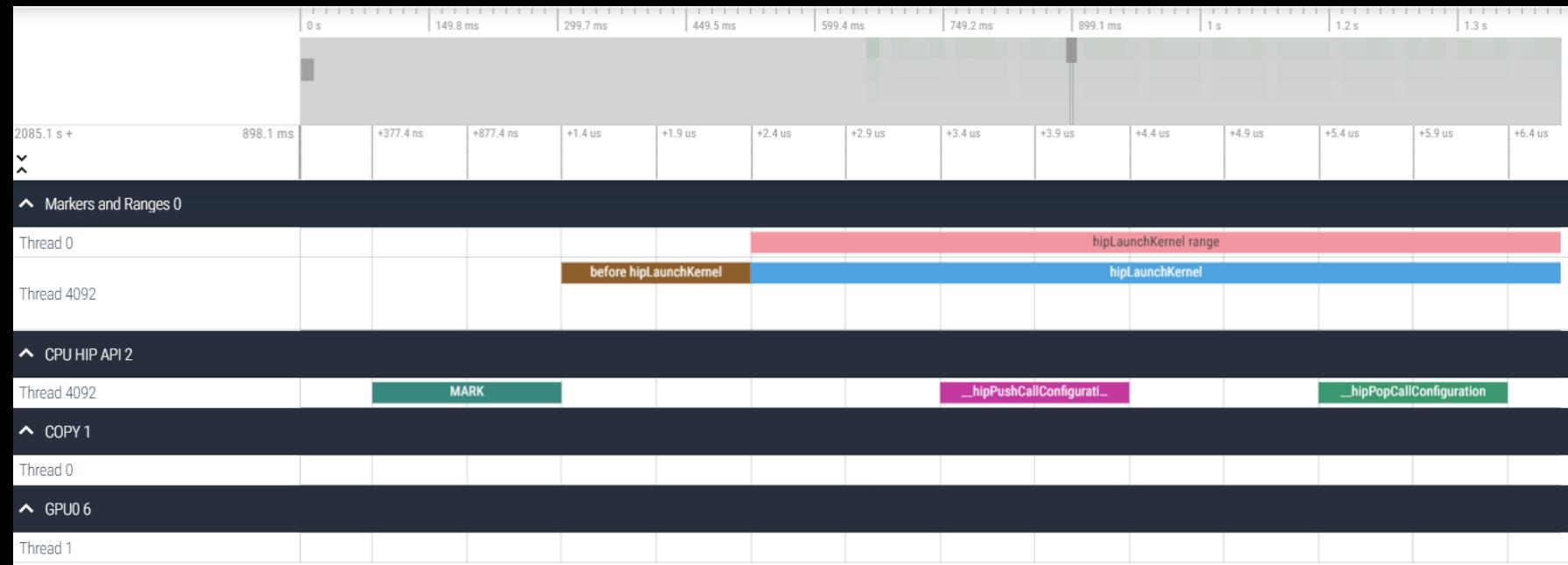
rocprof: Collecting application traces with markers

- 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>`

```
roctracer_mark("before HIP
LaunchKernel");

roctxMark("before hipLaunchKernel");
int rangeId =
roctxRangeStart("hipLaunchKernel
range");

roctxRangePush("hipLaunchKernel");
hipLaunchKernelGGL(matrixTranspose,...)
;
roctracer_mark("after HIP
LaunchKernel");
roctxMark("after hipLaunchKernel");
```



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 --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 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:

```
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 ${OMPI_COMM_WORLD_RANK+x} ]]; then
    # mpich
    export MPI_RANK=${OMPI_COMM_WORLD_RANK}
elif [[ -n ${MV2_COMM_WORLD_RANK+x} ]]; then
    # ompi
    export MPI_RANK=${MV2_COMM_WORLD_RANK}
elif [[ -n ${SLURM_PROCID+x} ]]; then
    export MPI_RANK=${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 that collects data, there is an overhead
- The percentage of the overhead depends on many aspects, for example if you try to instrument tiny tasks in a loop, this can take more time than tasks outside a loop
- If you try to collect many counters and especially ones that need more than one pass, then this could cause overhead if there a lot of related calls
- Also, if a lot of markers are added and especially in a loop then the roctx-trace can take significantly more time than the non instrumented execution time
- In general, more the data you collect, more the overhead can be, and it depends on the application.

Omnitrace: Application Profiling, Tracing, and Analysis

- It is an AMD Research tool, repository: <https://github.com/AMDRResearch/omnitrace>
- It is not part of ROCm stack
- Omnitrace is a comprehensive profiling and tracing tool for parallel applications written in C, C++, Fortran, HIP, OpenCL, and Python which execute on the CPU or CPU+GPU
- Data collection modes:
 - Dynamic instrumentation
 - Statistical sampling
 - Process-level sampling
 - Critical trace generation
- Data analysis:
 - High-level summary profiles
 - Comprehensive traces
 - Critical trace analysis
- Parallelism support: HIP, HSA, Pthreads, MPI, Kokkos, OpenMP
- GPU Metrics: GPU hardware counters, HIP/HSA API, HIP kernel tracing, HSA operation tracing, memory/power/temperature/utilization
- CPU Metrics: Hardware counters, timing metrics, memory metrics, network statistics, I/O, and more



ROCgdb

Debugging

Rocgdb

- AMD ROCm source-level debugger for Linux
- based on the GNU Debugger (GDB)
 - tracks upstream GDB master
 - standard GDB commands for both CPU and GPU debugging
- considered a prototype
 - focus on source line debugging
 - no symbolic variable debugging yet

Simple saxpy kernel

```

1  #include <hip/hip_runtime.h>
2
3  __constant__ float a = 1.0f;
4
5  __global__
6  void saxpy(int n, float const* x, int incx, float* y, int incy)
7  {
8      int i = blockDim.x*blockIdx.x + threadIdx.x;
9      if (i < n)
10         y[i] += a*x[i];
11 }
12
13 int main()
14 {
15     int n = 256;           ←
16     std::size_t size = sizeof(float)*n;
17
18     float* d_x;
19     float* d_y;
20     hipMalloc(&d_x, size);
21     hipMalloc(&d_y, size);
22
23     int num_groups = 2;    ←
24     int group_size = 128;
25     saxpy<<num_groups, group_size>>(n, d_x, 1, d_y, 1);
26     hipDeviceSynchronize();
27 }
28

```

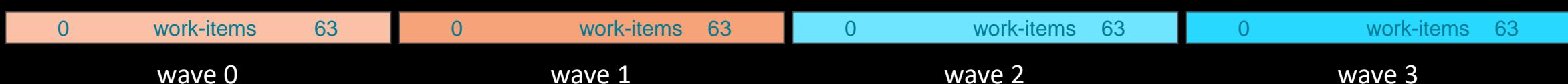
} classic saxpy operation
one array index = one work-item

size of arrays = 256

two groups
each 128 work-items

group 0

group 1



Cause a page fault

```
1 #include <hip/hip_runtime.h>
2
3 __constant__ float a = 1.0f;
4
5 __global__
6 void saxpy(int n, float const* x, int incx, float* y, int incy)
7 {
8     int i = blockDim.x*blockIdx.x + threadIdx.x;
9     if (i < n)
10         y[i] += a*x[i];
11 }
12
13 int main()
14 {
15     int n = 256;
16     std::size_t size = sizeof(float)*n;
17
18     float* d_x;
19     float* d_y;
20     // hipMalloc(&d_x, size); ←
21     // hipMalloc(&d_y, size);
22
23     int num_groups = 2;
24     int group_size = 128;
25     saxpy<<<num_groups, group_size>>>(n, d_x, 1, d_y, 1);
26     hipDeviceSynchronize();
27 }
28 }
```

Break it by commenting out the allocations.
(better to initialize the pointers to nullptr)

It's important to synchronize before exit.
Otherwise, the CPU thread may quit before the GPU gets a chance to
report the error.

Compilation with hipcc

```
1 #include <hip/hip_runtime.h>
2
3 __constant__ float a = 1.0f;
4
5 __global__
6 void saxpy(int n, float const* x, int incx, float* y, int incy)
7 {
8     int i = blockDim.x*blockIdx.x + threadIdx.x;
9     if (i < n)
10         y[i] += a*x[i];
11 }
12
13 int main()
14 {
15     int n = 256;
16     std::size_t size = sizeof(float)*n;
17
18     float* d_x;
19     float* d_y;
20     // hipMalloc(&d_x, size);
21     // hipMalloc(&d_y, size);
22
23     int num_groups = 2;
24     int group_size = 128;
25     saxpy<<<num_groups, group_size>>>(n,
26     hipDeviceSynchronize());
27 }
28 }
```

Need to set the target

- gfx906 – MI50, MI60, Radeon 7
- gfx908 – MI100
- gfx90a – MI200

```
saxpy$ hipcc --offload-arch=gfx90a -o saxpy saxpy.cpp
```

Execution

```
1 #include <hip/hip_runtime.h>
2
3 __constant__ float a = 1.0f;
4
5 __global__
6 void saxpy(int n, float const* x, int incx, float* y, int incy)
7 {
8     int i = blockDim.x*blockIdx.x + threadIdx.x;
9     if (i < n)
10         y[i] += a*x[i];
11 }
12
13 int main()
14 {
15     int n = 256;
16     std::size_t size = sizeof(float)*n;
17
18     float* d_x;
19     float* d_y;
20     // hipMalloc(&d_x, size);
21     // hipMalloc(&d_y, size);
22
23     int num_groups = 2;
24     int group_size = 128;
25     saxpy<<<num_groups, group_size>>>(n,
26     hipDeviceSynchronize());
27 }
28 }
```

```
saxpy$ hipcc --offload-arch=gfx90a -o saxpy saxpy.cpp
saxpy$ ./saxpy
```

- In this example we have already allocated a GPU with salloc

Get a page fault

```
1 #include <hip/hip_runtime.h>
2
3 __constant__ float a = 1.0f;
4
5 __global__
6 void saxpy(int n, float const* x, int incx, float* y, int incy)
7 {
8     int i = blockDim.x*blockIdx.x + threadIdx.x;
9     if (i < n)
10        y[i] += a*x[i];
11 }
12
13 int main()
14 {
15     int n = 256;
16     std::size_t size = sizeof(float)*n;
17
18     float* d_x;
19     float* d_y;
20     // hipMalloc(&d_x, size);
21     // hipMalloc(&d_y, size);
22
23     int num_groups = 2;
24     int group_size = 128;
25     saxpy<<<num_groups, group_size>>>(n, c
26     hipDeviceSynchronize();
27 }
28 }
```

```
saxpy$ hipcc --offload-arch=gfx90a -o saxpy saxpy.cpp
saxpy$ ./saxpy
Memory access fault by GPU node-2 (Agent handle: 0x2284d90) on address (nil). Reason: Unknown.
Aborted (core dumped)
saxpy$
```

Execution with rocgdb

```
1 #include <hip/hip_runtime.h>
2
3 __constant__ float a = 1.0f;
4
5 __global__
6 void saxpy(int n, float const* x, int incx, float* y, int incy)
7 {
8     int i = blockDim.x*blockIdx.x + threadIdx.x;
9     if (i < n)
10        y[i] += a*x[i];
11 }
12
13 int main()
14 {
15     int n = 256;
16     std::size_t size = sizeof(float)*n;
17
18     float* d_x;
19     float* d_y;
20     // hipMalloc(&d_x, size);
21     // hipMalloc(&d_y, size);
22
23     int num_groups = 2;
24     int group_size = 128;
25     saxpy<<<num_groups, group_size>>>(n, d_
26     hipDeviceSynchronize();
27 }
28 }
```

```
saxpy$ rocgdb saxpy
```

Get more information

```
1 #include <hip/hip_runtime.h>
2
3 __constant__ float a = 1.0f;
4
5 __global__
6 void saxpy(int n, float const* x, int incx, float* y, int incy)
7 {
8     int i = blockDim.x*blockIdx.x + threadIdx.x;
9     if (i < n)
10         y[i] += a*x[i];
11 }
12
13 int main()
14 {
15     int n = 256;
16     std::size_t size = sizeof(float)*n;
17
18     float* d_x;
19     float* d_y;
20     // hipMalloc(&d_x, size);
21     // hipMalloc(&d_y, size);
22
23     int num_groups = 2;
24     int group_size = 128;
25     saxpy<<<num_groups, group_size>>>(n,
26     hipDeviceSynchronize();
27 }
28 }
```

Reports segmentation fault in the saxpy kernel.

```
(gdb) run
Starting program: /home/gmarkoma/saxpy
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x7ffffd428700 (LWP 10456)]
Warning: precise memory violation signal reporting is not enabled, reported
location may not be accurate. See "show amdgpu precise-memory".

Thread 3 "saxpy" received signal SIGSEGV, Segmentation fault.
[Switching to thread 3, lane 0 (AMDGPU Lane 1:1:1:1/0 (0,0,0)[0,0,0])]
0x00007ffff7ec1094 in saxpy(int, float const*, int, float*, int) () from file:///home/gmarkoma/s
axy#offset=8192&size=13832
(gdb) █
```

Compile with -ggdb

```
1 #include <hip/hip_runtime.h>
2
3 __constant__ float a = 1.0f;
4
5 __global__
6 void saxpy(int n, float const* x, int incx, float* y, int incy)
7 {
8     int i = blockDim.x*blockIdx.x + threadIdx.x;
9     if (i < n)
10         y[i] += a*x[i];
11 }
12
13 int main()
14 {
15     int n = 256;
16     std::size_t size = sizeof(float)*n;
17
18     float* d_x;
19     float* d_y;
20     // hipMalloc(&d_x, size);
21     // hipMalloc(&d_y, size);
22
23     int num_groups = 2;
24     int group_size = 128;
25     saxpy<<num_groups, group_size>>(n,
26     hipDeviceSynchronize());
27 }
28
```

```
saxpy$ hipcc -ggdb --offload-arch=gfx90a -o saxpy saxpy.cpp
```

Get more details

```

1  #include <hip/hip_runtime.h>
2
3  __constant__ float a = 1.0f;
4
5  __global__
6  void saxpy(int n, float const* x, int incx, float* y, int incy)
7  {
8      int i = blockDim.x*blockIdx.x + threadIdx.x;
9      if (i < n)
10         y[i] += a*x[i];
11     }
12
13 int main()
14 {
15     int n = 256;
16     std::size_t size = sizeof(float)*n;
17
18     float* d_x;
19     float* d_y;
20     // hipMalloc(&d_x, size);
21     // hipMalloc(&d_y, size);
22
23     int num_groups = 2;
24     int group_size = 128;
25     saxpy<<num_groups, group_size>>(n, d_x, d_y);
26     hipDeviceSynchronize();
27 }
28

```

more details

- what kernel
- what file:line

```

(gdb) run
Starting program: /home/gmarkoma/saxpy
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x7fffed428700 (LWP 10637)]
Warning: precise memory violation signal reporting is not enabled, reported
location may not be accurate. See "show amdgpu precise-memory".
Thread 3 "saxpy" received signal SIGSEGV, Segmentation fault.
[Switching to thread 3, lane 0 (AMDCPU Lane 1:2:1:1/0 (0,0,0)[0,0,0])]
0x00007ffff7ec1094 in saxpy () at saxpy.cpp:10
10          y[i] += a*x[i];
(gdb) 

```

But where's my stack trace?

List threads

```

1  #include <hip/hip_runtime.h>
2
3  __constant__ float a = 1.0f;
4
5  __global__
6  void saxpy(int n, float const* x, int incx, float* y, int incy)
7  {
8      int i = blockDim.x*blockIdx.x + threadIdx.x;
9      if (i < n)
10         y[i] += a*x[i];
11 }
12
13 int main()
14 {
15     int n = 256;
16     std::size_t size = sizeof(float)*n;
17
18     float* d_x;
19     float* d_y;
20     // hipMalloc(&d_x, size);
21     // hipMalloc(&d_y, size);
22
23     int num_groups = 2;
24     int group_size = 128;
25     saxpy<<num_groups, group_size>>(n,
26     hipDeviceSynchronize());
27 }
28

```

```

(gdb) i th
Id Target Id Frame
 1 Thread 0x7ffff7fe6e80 (LWP 10633) 'saxpy' 0x00007fffee0fc499 in rocr::core::InterruptSign
al::WaitRelaxed(hsa_signal_condition_t, long, unsigned long, hsa_wait_state_t) ()
  from /opt/rocm-5.2.0/lib/libhsa-runtime64.so.1
 2 Thread 0x7ffffed428700 (LWP 10637) "saxpy" 0x00007ffff5e1972b in ioctl ()
  from /lib64/libc.so.6
* 3 AMDGPU Wave 1:2:1:1 (0,0,0)/0 "saxpy" 0x00007ffff7ec1094 in saxpy () at saxpy.cpp:10
  4 AMDGPU Wave 1:2:1:2 (0,0,0)/1 "saxpy" 0x00007ffff7ec1094 in saxpy () at saxpy.cpp:10
  5 AMDGPU Wave 1:2:1:3 (1,0,0)/0 "saxpy" 0x00007ffff7ec1094 in saxpy () at saxpy.cpp:10
  6 AMDGPU Wave 1:2:1:4 (1,0,0)/1 "saxpy" 0x00007ffff7ec1094 in saxpy () at saxpy.cpp:10
(gdb) █

```

What segfaulted is a GPU wave.
 It does not have your CPU stack.
List threads to see what's going on.

Switch to the CPU thread

```
1  #include <hip/hip_runtime.h>
2
3  __constant__ float a = 1.0f;
4
5  __global__
6  void saxpy(int n, float const* x, int incx, float* y, int incy)
7  {
8      int i = blockDim.x*blockIdx.x + threadIdx.x;
9      if (i < n)
10         y[i] += a*x[i];
11 }
12
13 int main()
14 {
15     int n = 256;
16     std::size_t size = sizeof(float)*n;
17
18     float* d_x;
19     float* d_y;
20     // hipMalloc(&d_x, size);
21     // hipMalloc(&d_y, size);
22
23     int num_groups = 2;
24     int group_size = 128;
25     saxpy<<num_groups, group_size>>(n, (gdb) | hipDeviceSynchronize();
26
27 }
28
```

t 1
(thread 1)
It's in the HSA runtime.

```
(gdb) t 1
[Switching to thread 1 (Thread 0x7ffff7fe6e80 (LWP 10633))]
#0  0x00007fff0fc499 in rocr::core::InterruptSignal::WaitRelaxed(hsa_signal_condition_t, long,
    unsigned long, hsa_wait_state_t) () from /opt/rocm-5.2.0/lib/libhsa-runtime64.so..
```

But how did it get there?

See the stack trace of the CPU thread

```

1 #include <hip/hip_runtime.h>
2
3 __constant__ float a = 1.0f;
4
5 __global__
6 void saxpy(int n, float const* x, int incx, float* y, int incy)
7 {
8     int i = blockDim.x*blockIdx.x + threadIdx.x;
9     if (i < n)
10         y[i] += a*x[i];
11 }
12
13 int main()
14 {
15     int n = 256;
16     std::size_t size = 128 * n;
17
18     float* d_x;
19     float* d_y;
20     // hipMalloc(&d_x, size);
21     // hipMalloc(&d_y, size);
22
23     int num_groups = 2;
24     int group_size = 128;
25     saxpy<<num_groups, group_size>>(n, d_x, d_y);
26     hipDeviceSynchronize();
27 }
28

```

HSA runtime

HIP runtime

where

```

(gdb) where
...
#0  0x00007ffffe0fc499 in rocr::core::InterruptSignal::WaitRelaxed(hsa_signal_condition_t, long, unsigned long, hsa_wait_state_t) () from /opt/rocm-5.2.0/lib/libhsa-runtime64.so.1
#1  0x00007ffffe0fc36a in rocr::core::InterruptSignal::WaitAcquire(hsa_signal_condition_t, long, unsigned long, hsa_wait_state_t) () from /opt/rocm-5.2.0/lib/libhsa-runtime64.so.1
#2  0x00007ffffe0f0869 in rocr::HSA::hsa_signal_wait_scacquire(hsa_signal_s, hsa_signal_condition_t, long, unsigned long, hsa_wait_state_t) () from /opt/rocm-5.2.0/lib/libhsa-runtime64.so.1
#3  0x00007fffff67bdd43 in bool roc::WaitForSignal<false>(hsa_signal_s, bool) () from /opt/rocm-5.2.0/lib/libamdhip64.so.5
#4  0x00007fffff67b5836 in roc::VirtualGPU::HwQueueTracker::CpuWaitForSignal(roc::ProfilingSignal*) () from /opt/rocm-5.2.0/lib/libamdhip64.so.5
#5  0x00007fffff67b77cf in roc::VirtualGPU::releaseGpuMemoryFence(bool) () from /opt/rocm-5.2.0/lib/libamdhip64.so.5
#6  0x00007fffff67b9523 in roc::VirtualGPU::flush(amd::Command*, bool) () from /opt/rocm-5.2.0/lib/libamdhip64.so.5
#7  0x00007fffff67b9db0 in roc::VirtualGPU::submitMarker(amd::Marker&) () from /opt/rocm-5.2.0/lib/libamdhip64.so.5
#8  0x00007fffff678ec2e in amd::Command::enqueue() () from /opt/rocm-5.2.0/lib/libamdhip64.so.5
#9  0x00007fffff678f1e0 in amd::Event::notifyCmdQueue(bool) () from /opt/rocm-5.2.0/lib/libamdhip64.so.5
#10 0x00007fffff678f28c in amd::Event::awaitCompletion() () from /opt/rocm-5.2.0/lib/libamdhip64.so.5
#11 0x00007fffff6791fdc in amd::HostQueue::finish() () from /opt/rocm-5.2.0/lib/libamdhip64.so.5
#12 0x00007fffff65c25f9 in hipDeviceSynchronize () from /opt/rocm-5.2.0/lib/libamdhip64.so.5
#13 0x000000000020d615 in main () at saxpy.cpp:25
(gdb) 

```

Quick tip

- Frontier and LUMI CPUs have 64 cores / 128 threads.
- If you're debugging an app with OpenMP threading and OMP_NUM_THREADS is not set you will see 128 CPU threads in rocgdb.
- Set OMP_NUM_THREADS=1 when debugging GPU codes.

"GUIs"

rocgdb -tui saxpy

The screenshot shows the ROCgdb -tui interface. On the left is a code editor window titled "saxpy.cpp" displaying C++ code for a SAXPY kernel. On the right is a terminal window showing the ROCgdb command-line interface. A large diagonal watermark "source" is overlaid across both windows.

```
saxpy.cpp
1 #include "hip/hip_runtime.h"
2 #include <stdio.h>
3
4 __constant__ float a = 1.0f;
5
6 __global__
7 void saxpy(int n, float const* x, int incx, float* y, int incy)
8 {
9     int i = blockIdx.x*blockDim.x + threadIdx.x;
10    if (i < n) y[i] = a*x[i] + y[i];
11 }
12
13 int main()
14 {
15     int n = 256;
16     std::size_t size = sizeof(float)*n;
17
18     float *d_x, *d_y;
19     //hipMalloc(&d_x, size);
20     //hipMalloc(&d_y, size);
21
22     int num_groups= 2;
23     int group_size=128;
24     saxpy<<n<<num_groups<<group_size>>>(n, d_x, 1, d_y, 1);
```

amd-dbgapi AMDGPU Wave 1:2:1:1 In: saxpy
Type "apropos word" to search for commands related to "word"...
Reading symbols from saxpy...
(gdb) run
Starting program: /home/gmarkoma/saxpy
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x7ffffd428700 (LWP 11074)]
Warning: precise memory violation signal reporting is not enabled, reported
location may not be accurate. See "show amdgpu precise-memory".

Thread 3 "saxpy" received signal SIGSEGV, Segmentation fault.
[Switching to thread 3, lane 0 (AMDGPU Lane 1:2:1:1/0 (0,0,0)[0,0,0])]
0x00007fff7ec1094 in saxpy () at saxpy.cpp:10
(gdb)

cgdb -d rocgdb saxpy

The screenshot shows the cgdb -d interface. On the left is a code editor window titled "saxpy : cgdb — Konsole" displaying the same C++ code as the ROCgdb -tui interface. On the right is a terminal window showing the cgdb command-line interface. A large diagonal watermark "source" is overlaid across both windows.

```
saxpy : cgdb — Konsole
File Edit View Bookmarks Settings Help
1 #include <hip/hip_runtime.h>
2
3 __constant__ float a = 1.0f;
4
5 __global__
6 void saxpy(int n, float const* x, int incx, float* y, int incy)
7 {
8     int i = blockDim.x*blockIdx.x + threadIdx.x;
9     if (i < n)
10         y[i] += a*x[i];
11 }
12
13 int main()
14 {
15     int n = 256;
16     std::size_t size = sizeof(float)*n;
17
18     float* d_x;
19     float* d_y;
20     hipMalloc(&d_x, size);
21     hipMalloc(&d_y, size);
22
23     int num_groups = 2;
24     int group_size = 128;
25     saxpy<<n<<num_groups<<group_size>>>(n, d_x, 1, d_y, 1);
26     hipDeviceSynchronize();
/mnt/shared/codes/saxpy/saxpy.hip.cpp
```

[35:1mGNU gdb (rocm-rel-4.5-56) 11.1[m
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://github.com/ROCM-Developer-Tools/ROCGdb/issues>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from [32m./saxpy[m...
[?2004h(gdb)]

Breakpoint

We try to put a breakpoint in line 22 but it is declared as line 24.

```
--Type <RET> for more, q to quit, c to continue without paging--For help, type "help".  
Type "apropos word" to search for commands related to "word"...  
Reading symbols from saxpy...  
(gdb) b saxpy.cpp:22  
Breakpoint 1 at 0x20d57f: file saxpy.cpp, line 24.  
(gdb) -
```

Default compiler optimization for hipcc is -O3, compile with -O0

```
saxpy$ hipcc -ggdb -O0 --offload-arch=gfx90a -o saxpy saxpy.cpp
```

Creating a breakpoint again and it is declared in the correct line

```
--Type <RET> for more, q to quit, c to continue without paging--For help, type "help".  
Type "apropos word" to search for commands related to "word"...  
Reading symbols from saxpy...  
(gdb) b saxpy.cpp:22  
Breakpoint 1 at 0x219dec: file saxpy.cpp, line 22.  
(gdb) -
```

Running and architecture

Running with the keystroke
r and stops at the
breakpoint

```
--Type <RET> for more, q to quit, c to continue without paging--For help, type "help".  
Type "apropos word" to search for commands related to "word"...
Reading symbols from saxpy...
(gdb) b saxpy.cpp:22
Breakpoint 1 at 0x219dec: file saxpy.cpp, line 22.
(gdb) r
Starting program: /home/gmarkoma/saxpy
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x7ffffd428700 (LWP 16916)]

Thread 1 "saxpy" hit Breakpoint 1, main () at saxpy.cpp:22
(gdb) ■
```

More information about the
thread with the command
i th

```
(gdb) i th
  Id  Target Id                               Frame
* 1   Thread 0x7ffff7fe6e80 (LWP 16912) "saxpy" main () at saxpy.cpp:22
  2   Thread 0x7ffffd428700 (LWP 16916) "saxpy" 0x00007ffff5e1972b in ioctl () from /lib64/libc.so.6
(gdb) ■
```

We can see on what
device is the thread with
the **show architecture**
command

```
(gdb) show architecture
The target architecture is set to "auto" (currently "i386:x86-64").
(gdb)
```

Breakpoint kernel and architecture

Breakpoint on the kernel called saxpy with the command **b saxpy**

```
(gdb) b saxpy
Function "saxpy" not defined.
Make breakpoint pending on future shared library load? (y or [n]) yBreakpoint 2 (saxpy) pending.
(gdb)
```

You can continue with the command **c**

```
(gdb) c
Continuing.
[New Thread 0x7fffdefff700 (LWP 16937)]
[New Thread 0x7fffecaff700 (LWP 16938)]
[Thread 0x7fffdefff700 (LWP 16937) exited]
[Switching to thread 5, lane 0 (AMDGPU Lane 1:2:1:1/0 (0,0,0)[0,0,0])]

Thread 5 "saxpy" hit Breakpoint 2, with lanes [0-63], saxpy (n=256, x=0x7fffec700000, incx=1, y=0x7fffec701000, incy=1) at saxpy.cpp:9
```

We can see on what device is the thread with the command **show architecture**

```
(gdb) show architecture
The target architecture is set to "auto" (currently "amdgcn:gfx90a").
```

rocgdb + gdbgui

breakpoint in CPU code



Load Binary /mnt/shared/codes/saxpy/saxpy

show filesystem fetch disassembly reload file jump to line /mnt/shared/codes/saxpy/saxpy.hip.cpp:22 (27 lines total)

```

1 #include <hip/hip_runtime.h>
2
3 __constant__ float a = 1.0f;
4
5 __global__
6 void saxpy(int n, float const* x, int incx, float* y, int incy)
7 {
8     int i = blockDim.x*blockIdx.x + threadIdx.x;
9     if (i < n)
10         y[i] += a*x[i];
11 }
12
13 int main()
14 {
15     int n = 256;
16     std::size_t size = sizeof(float)*n;
17
18     float* d_x;
19     float* d_y;
20     hipMalloc(&d_x, size);
21     hipMalloc(&d_y, size);
22
23     int num_groups = 2;
24     int group_size = 128;
25     saxpy<<<num_groups, group_size>>>(n, d_x, 1, d_y, 1);
26     hipDeviceSynchronize();
27 }
```

(end of file)

running command: /opt/rocm/bin/rocgdb

GNU gdb (rocm-rel-4.5-56) 11.1
 Copyright (C) 2021 Free Software Foundation, Inc.
 License GPLv3+: GNU GPL version 3 or later <<http://gnu.org/licenses/gpl.html>>
 This is free software: you are free to change and redistribute it.
 There is NO WARRANTY, to the extent permitted by law.
 Type "show copying" and "show warranty" for details.
 This GDB was configured as "x86_64-pc-linux-gnu".
 Type "show configuration" for configuration details.
 For bug reporting instructions, please see:
<<https://github.com/ROCM-Developer-Tools/ROCgdb/issues>>.
 Find the GDB manual and other documentation resources online at:
<<http://www.gnu.org/software/gdb/documentation/>>.

For help, type "help"
Type "apropos word" to search for commands related to word
New UI allocated

(gdb) █

Source
Console

Rocgdb with GUI

Execute:

rocgdb -tui saxpy

```
saxpy.cpp
14  {
15      int n = 256;
16      std::size_t size = sizeof(float)*n;
17
18      float *d_x, *d_y;
19      hipMalloc(&d_x, size);
20      hipMalloc(&d_y, size);
21
22      int num_groups= 2;
23      int group_size=128;
24      saxpy<<<num_groups,group_size>>>(n, d_x, 1, d_y, 1);
25      hipDeviceSynchronize();
26
27      hipFree(d_x);
28      hipFree(d_y);
29  }
30
```

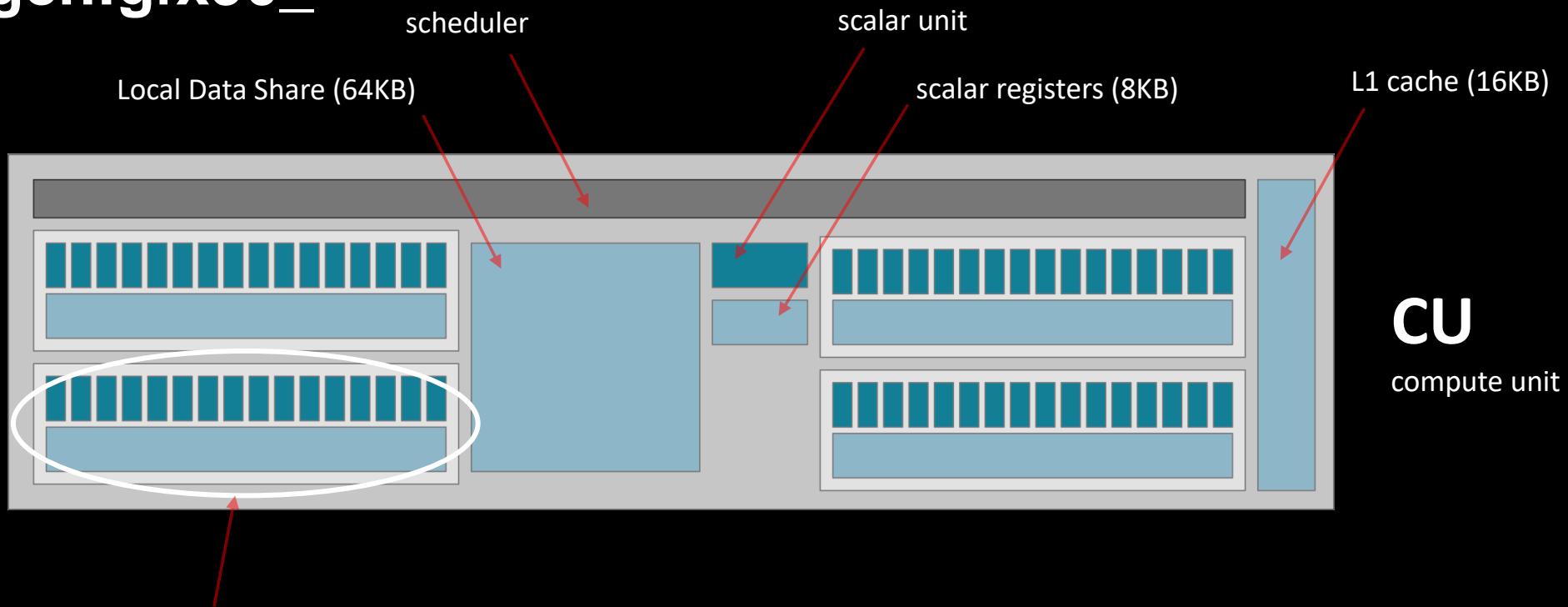
Source code

```
exec No process In:
(gdb) -
```

L?? PC: ??

Terminal

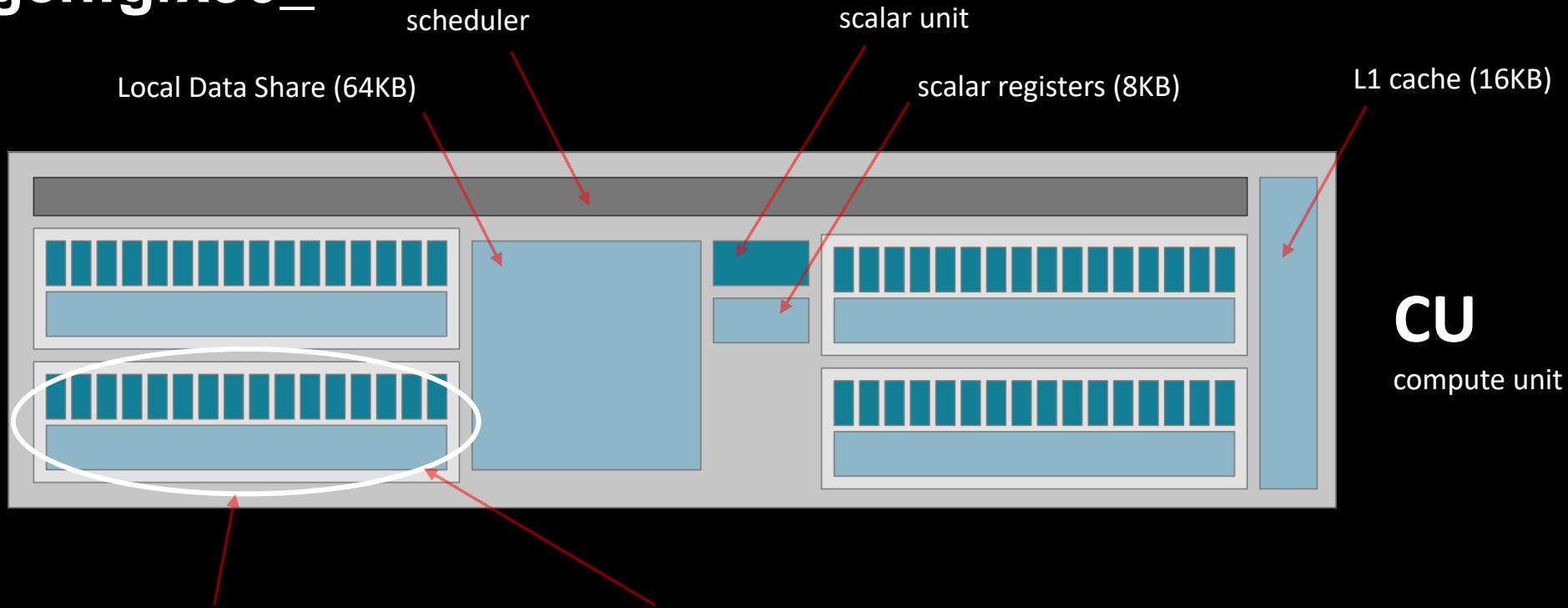
amdgcn:gfx90_



typically described as

- a 16-way SIMD unit
- with 64KB of registers

amdgcn:gfx90_



List threads / waves

i th
(info threads)

some CPU threads

4 GPU “threads” (waves)

```
(gdb) i th
      Id   Target Id          Frame
      1   Thread 0x7ffff7fe6e80 (LWP 16912) "saxpy" 0x00007ffffee0fc4c0 in rocr::core::InterruptSignal:
t) () from /opt/rocm-5.2.0/lib/libhsa-runtime64.so.1
      2   Thread 0x7ffffed428700 (LWP 16916) "saxpy" 0x00007ffff5e1972b in ioctl () from /lib64/libc.so.6
      4   Thread 0x7ffffecaff700 (LWP 16938) "saxpy" 0x00007ffffee0fc4af in rocr::core::InterruptSignal:
t) () from /opt/rocm-5.2.0/lib/libhsa-runtime64.so.1
* 5   AMDGPU Wave 1:2:1:1 (0,0,0)/0 "saxpy"    saxpy (n=256, x=0x7fffec700000, incx=1, y=0x7fffec700000)
      6   AMDGPU Wave 1:2:1:2 (0,0,0)/1 "saxpy"    saxpy (n=256, x=0x7fffec700000, incx=1, y=0x7fffec700000)
      7   AMDGPU Wave 1:2:1:3 (1,0,0)/0 "saxpy"    saxpy (n=256, x=0x7fffec700000, incx=1, y=0x7fffec700000)
      8   AMDGPU Wave 1:2:1:4 (1,0,0)/1 "saxpy"    saxpy (n=256, x=0x7fffec700000, incx=1, y=0x7fffec700000)
```

Wave details

agent-id:queue-id:dispatch-num:wave-id (work-group-x,work-group-y,work-group-z) /work-group-thread-index

(gdb) i th		Frame
Id	Target Id	
1	Thread 0x7ffff7fe6e80 (LWP 16912) "saxpy" 0x00007fffee0fc4c0 in rocr::core::InterruptSignal::t () from /opt/rocm-5.2.0/lib/libhsa-runtime64.so.1	
2	Thread 0x7ffffed428700 (LWP 16916) "saxpy" 0x00007ffff5e1972b in ioctl () from /lib64/libc.so.6	
4	Thread 0x7ffffecaff700 (LWP 16938) "saxpy" 0x00007fffee0fc4af in rocr::core::InterruptSignal::t () from /opt/rocm-5.2.0/lib/libhsa-runtime64.so.1	
*	AMDGPU Wave 1:2:1:1 (0,0,0)/0 "saxpy"	saxpy (n=256, x=0x7fffec70000, incx=1, y=0x7fffec70000)
5		saxpy (n=256, x=0x7fffec70000, incx=1, y=0x7fffec70000)
6		saxpy (n=256, x=0x7fffec70000, incx=1, y=0x7fffec70000)
7		saxpy (n=256, x=0x7fffec70000, incx=1, y=0x7fffec70000)
8	AMDGPU Wave 1:2:1:2 (0,0,0)/1 "saxpy"	saxpy (n=256, x=0x7fffec70000, incx=1, y=0x7fffec70000)

The diagram illustrates the breakdown of the string into its components:

- agent (GPU) ID
- (HSA) queue ID
- dispatch number
- wave ID
- workgroup (x, y, z)
- wave ID (within group)

Temporary breakpoints and Assembly

- Temporary breakpoint for saxpy kernel:
tbreak saxpy
- Split to see source code and assembly:
layout split
- For this example we have compiled with default -O3
- Compiling with -O0 it could give better ISA correlation

saxpy.cpp

```

14 {
15     int n = 256;
16     std::size_t size = sizeof(float)*n;
17
18     float *d_x, *d_y;
19     hipMalloc(&d_x, size);
20     hipMalloc(&d_y, size);

```

Source code

Assembly

```

0x20d550 <main()>      sub   $0x88,%rsp
0x20d557 <main()+7>    lea    0x18(%rsp),%rdi
0x20d55c <main()+12>   mov    $0x400,%esi
0x20d561 <main()+17>   call   0x20d810 <hipMalloc@plt>
0x20d566 <main()+22>   lea    0x10(%rsp),%rdi
0x20d56b <main()+27>   mov    $0x400,%esi
0x20d570 <main()+32>   call   0x20d810 <hipMalloc@plt>
0x20d575 <main()+37>   movabs $0x100000002,%rdi
0x20d57f <main()+47>   lea    0x7e(%rdi),%rdx

```

exec No process In:

(gdb) **tbreak saxpy**

Function "saxpy" not defined.

Make breakpoint pending on future shared library load? (y or [n]) yTemporary breakpoint 1 (saxpy) pending.

(gdb) **layout split**

(gdb) ■

L?? PC: ??

Terminal

List agents

info agents

➤ shows devices + properties

```
(gab) into agents
      Id State Target Id          Architecture Device Name Cores Threads Location
* 1   A     AMDGPU Agent (GPUID 31957) gfx90a       aldebaran  440    3520  29:00.0
```

gfx90a
MI200 series

SIMDs
(CUs x 4)

max waves
(SIMDs x 8)

List queues

info queues

➤ shows HSA queues

```
(gdb) info queues
  Id  Target Id          Type      Read  Write  Size   Address
  1   AMDGPU Queue 1:1 (QID 0) HSA (Multi) 4     4     4096  0x00007ffff7eb6000
* 2   AMDGPU Queue 1:2 (QID 1) HSA (Multi) 0     2     262144 0x00007ffff7e40000
(gdb)
```

agent ID

queue ID

(AQL) packets read

(AQL) packets written

Dispatch details

info dispatches

- shows kernel dispatches

```
(gdb) info dispatches
  Id      Target Id
* 1      AMDGPU Dispatch 1:2:1 (PKID 0)  Grid      Workgroup Fence   Kernel Function
                                                 [256,1,1]  [128,1,1] B|Aa|Ra saxpy(int, float const*, int, float*, int)
```

The screenshot shows a GDB command 'info dispatches' output. The first row lists columns: 'Id', 'Target Id'. The second row shows a specific dispatch entry: 'AMDGPU Dispatch 1:2:1 (PKID 0)'. To the right of this entry, several parameters are listed: 'Grid [256,1,1]', 'Workgroup [128,1,1]', 'Fence B|Aa|Ra', and 'Kernel Function saxpy(int, float const*, int, float*, int)'. Red arrows point from labels below the command to specific parts of the dispatch entry. 'agent ID' points to 'AMDGPU Dispatch'. 'queue ID' points to '1:2:1'. 'dispatch ID' points to '(PKID 0)'. 'grid dimensions' points to '[256,1,1]'. 'group dimensions' points to '[128,1,1]'. 'kernel' points to 'saxpy(int, float const*, int, float*, int)'.

More resources

- `/opt/rocm-5.2.0/share/doc/rocgdb/`
 - `rocannotate.pdf`
 - `rocgdb.pdf`
 - `rocrefcard.pdf`
 - `rocstabs.pdf`
- For LUMI: `/opt/rocm-5.0.2/share/doc/rocgdb/`

AMD_LOG_LEVEL=3

```
:3:devprogram.cpp :2978: 157529658660 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: _Z5saxpyiPKfiPfi
:3:hip_module.cpp :365 : 157529658684 us: 224178: [tid:0x7f59c7439e80] ihipModuleLaunchKernel ( 0x0x12e9720, 256, 1, 1, 128, 1, 1, 0, stream:<null>, 0x7fff94e2e07
0, char array:<null>, event:0, event:0, 0 )
:3:rocdevice.cpp :2686: 157529658695 us: 224178: [tid:0x7f59c7439e80] number of allocated hardware queues with low priority: 0, with normal priority: 0, with hig
h priority: 0, maximum per priority is: 4
:3:rocdevice.cpp :2757: 157529663975 us: 224178: [tid:0x7f59c7439e80] created hardware queue 0x7f59c72f4000 with size 4096 with priority 1, cooperative: 0
:3:devprogram.cpp :2675: 157529852150 us: 224178: [tid:0x7f59c7439e80] Using Code Object V4.
:3:devprogram.cpp :2978: 157529853058 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: __amd_rocclr_fillImage
:3:devprogram.cpp :2978: 157529853065 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: __amd_rocclr_fillBufferAligned2D
:3:devprogram.cpp :2978: 157529853070 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: __amd_rocclr_fillBufferAligned
:3:devprogram.cpp :2978: 157529853076 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: __amd_rocclr_copyImage1DA
:3:devprogram.cpp :2978: 157529853080 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: __amd_rocclr_copyBufferAligned
:3:devprogram.cpp :2978: 157529853084 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: __amd_rocclr_streamOpsWait
:3:devprogram.cpp :2978: 157529853087 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: __amd_rocclr_copyBuffer
:3:devprogram.cpp :2978: 157529853091 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: __amd_rocclr_streamOpsWrite
:3:devprogram.cpp :2978: 157529853094 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: __amd_rocclr_copyBufferRectAligned
:3:devprogram.cpp :2978: 157529853096 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: __amd_rocclr_gwsInit
:3:devprogram.cpp :2978: 157529853099 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: __amd_rocclr_copyBufferRect
:3:devprogram.cpp :2978: 157529853101 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: __amd_rocclr_copyImageToBuffer
:3:devprogram.cpp :2978: 157529853105 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: __amd_rocclr_copyBufferToImage
:3:devprogram.cpp :2978: 157529853108 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: __amd_rocclr_copyImage
:3:rocvirtual.cpp :753 : 157529853195 us: 224178: [tid:0x7f59c7439e80] Arg0: = val:256
:3:rocvirtual.cpp :679 : 157529853200 us: 224178: [tid:0x7f59c7439e80] Arg1: = ptr:0x7f59bbb00000 obj:[0x7f59bbb00000-0x7f59bbb00400]
:3:rocvirtual.cpp :753 : 157529853205 us: 224178: [tid:0x7f59c7439e80] Arg2: = val:1
:3:rocvirtual.cpp :679 : 157529853209 us: 224178: [tid:0x7f59c7439e80] Arg3: = ptr:0x7f59bbb01000 obj:[0x7f59bbb01000-0x7f59bbb01400]
:3:rocvirtual.cpp :753 : 157529853213 us: 224178: [tid:0x7f59c7439e80] Arg4: = val:1
:3:rocvirtual.cpp :2723: 157529853216 us: 224178: [tid:0x7f59c7439e80] ShaderName : _Z5saxpyiPKfiPfi
:3:hip_platform.cpp :676 : 157529853233 us: 224178: [tid:0x7f59c7439e80] ihipLaunchKernel: Returned hipSuccess :
:3:hip_module.cpp :509 : 157529853237 us: 224178: [tid:0x7f59c7439e80] hipLaunchKernel: Returned hipSuccess :
:3:hip_device_runtime.cpp :476 : 157529853243 us: 224178: [tid:0x7f59c7439e80] hipDeviceSynchronize ( )
:3:rocdevice.cpp :2636: 157529853248 us: 224178: [tid:0x7f59c7439e80] No HW event
:3:rocvirtual.hpp :62 : 157529853255 us: 224178: [tid:0x7f59c7439e80] Host active wait for Signal = (0x7f59c7442600) for -1 ns
:3:hip_device_runtime.cpp :488 : 157529853267 us: 224178: [tid:0x7f59c7439e80] hipDeviceSynchronize: Returned hipSuccess :
:3:hip_memory.cpp :536 : 157529853279 us: 224178: [tid:0x7f59c7439e80] hipFree ( 0x7f59bbb00000 )
:3:rocdevice.cpp :2093: 157529853291 us: 224178: [tid:0x7f59c7439e80] device=0x12d34f0, freeMem_ = 0xffefffc00
:3:hip_memory.cpp :538 : 157529853296 us: 224178: [tid:0x7f59c7439e80] hipFree: Returned hipSuccess :
:3:hip_memory.cpp :536 : 157529853300 us: 224178: [tid:0x7f59c7439e80] hipFree ( 0x7f59bbb01000 )
:3:rocdevice.cpp :2093: 157529853306 us: 224178: [tid:0x7f59c7439e80] device=0x12d34f0, freeMem_ = 0xffff000000
:3:hip_memory.cpp :538 : 157529853310 us: 224178: [tid:0x7f59c7439e80] hipFree: Returned hipSuccess :
:3:devprogram.cpp :2978: 157529853333 us: 224178: [tid:0x7f59c7439e80] For Init/Fini: Kernel Name: _Z5saxpyiPKfiPfi
```

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.

© 2022 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, ROCm, Radeon, Radeon Instinct and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names are for informational purposes only and may be trademarks of their respective owners.

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

Questions?

AMD