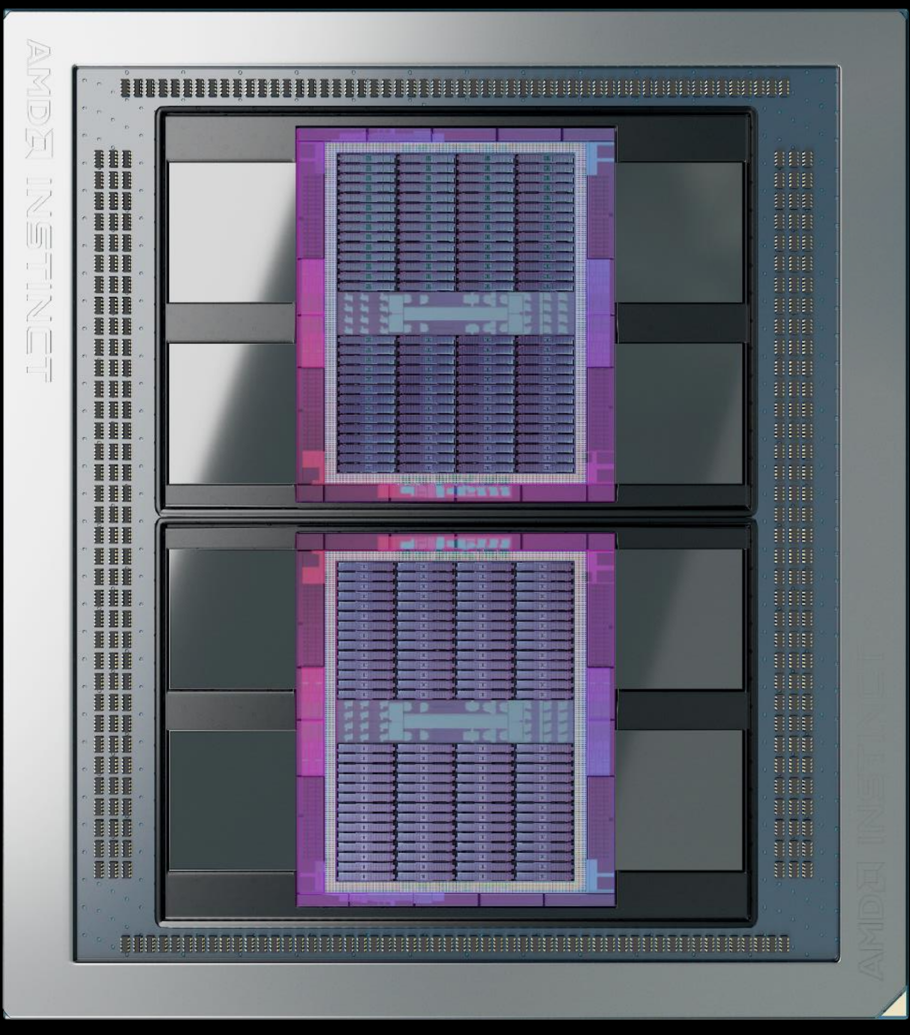
Introduction to AMD ROCm™ Ecosystem

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, Bob Robey

Comprehensive General LUMI Course
16/02/2023

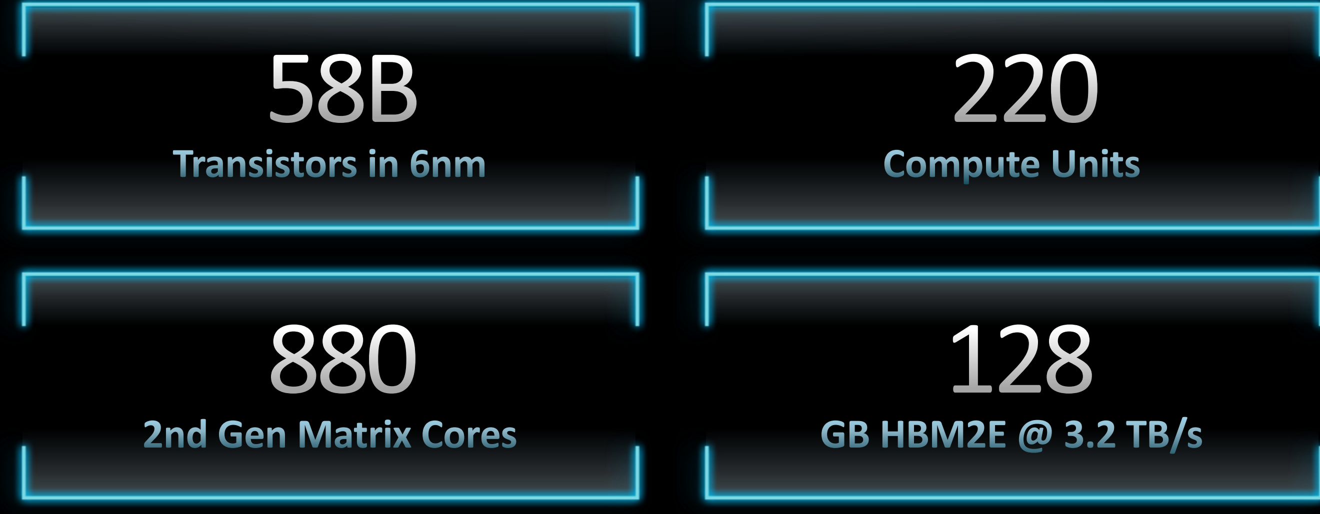
Introduction to the Architecture

[Public]



AMD INSTINCT™ MI250X

WORLD'S MOST ADVANCED DATA CENTER ACCELERATOR



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

AMD
together we advance_

[Public]

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

[Public]

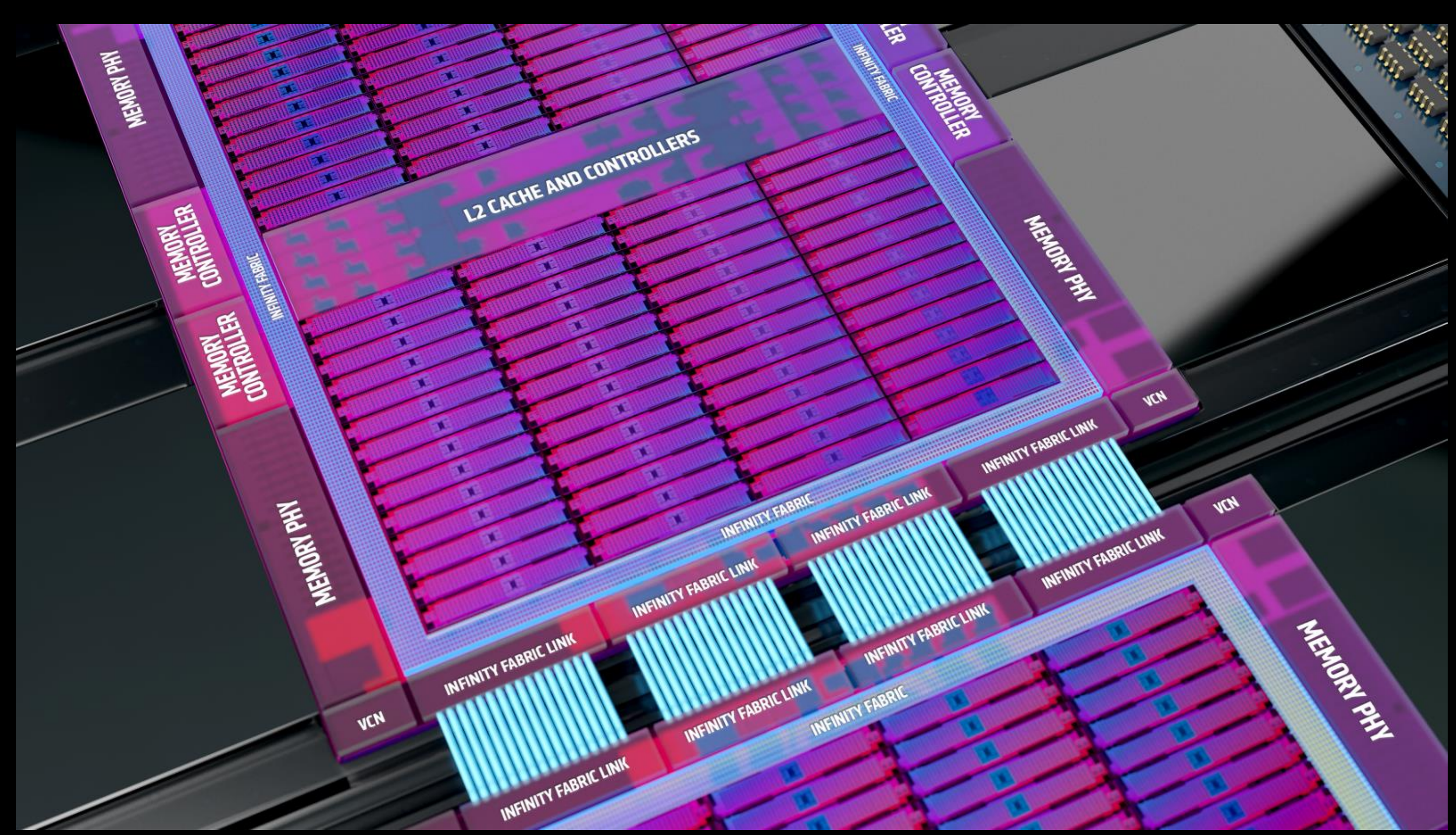
MULTI-CHIP DESIGN

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-DIRECTIONALBANDWIDTH

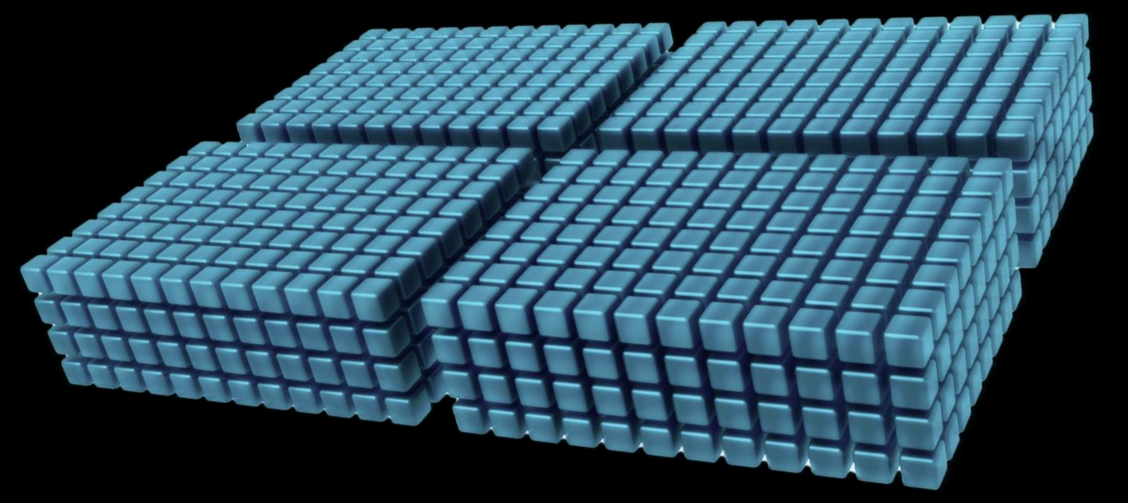


AMD
together we advance_

[Public]

2nd GENERATION MATRIX CORES

OPTIMIZED COMPUTE UNITS FOR SCIENTIFIC COMPUTING

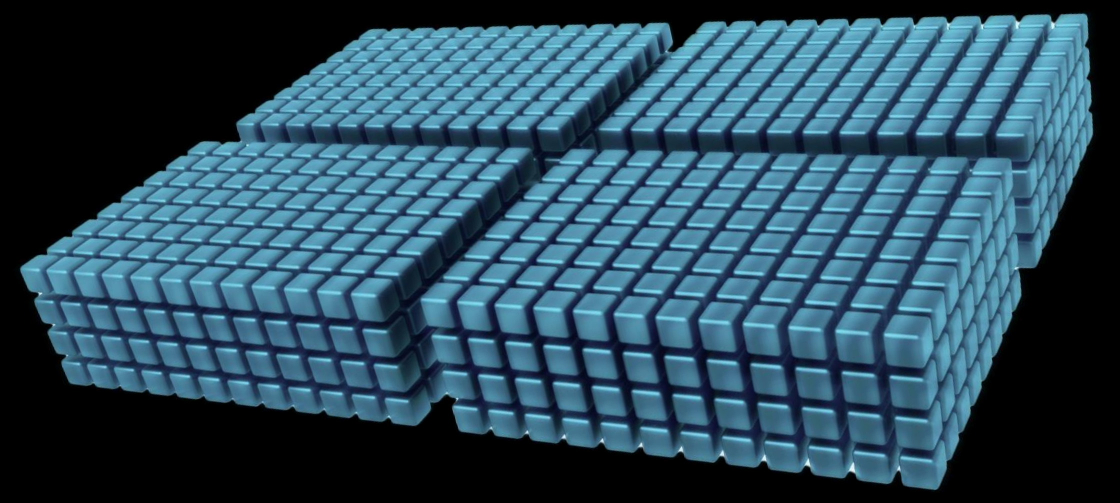


DOUBLE PRECISON (FP64) MATRIX CORE THROUGHPUT REPRESENTATION

MI100 MATRIX CORES OPS/CLOCK/COMPUTE UNIT	MI250X MATRIX CORES OPS/CLOCK/COMPUTE UNIT
No FP64 Matrix Core	256 FP64
256 FP32	256 FP32
1024 FP16	1024 FP16
512 BF16	1024 BF16
512 INT8	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

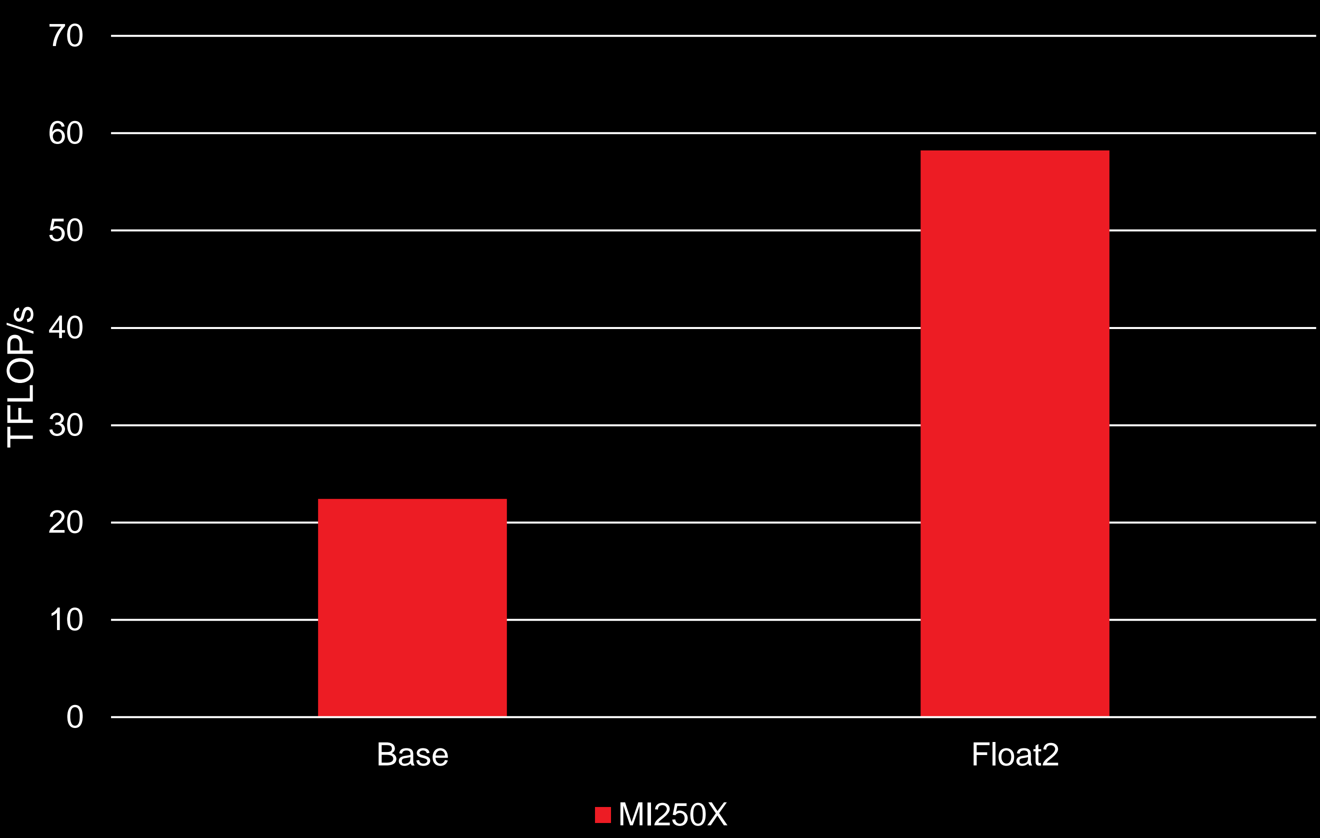
[Public]

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

AMD
together we advance_

From AMD MI100 to AMD MI250X

MI100

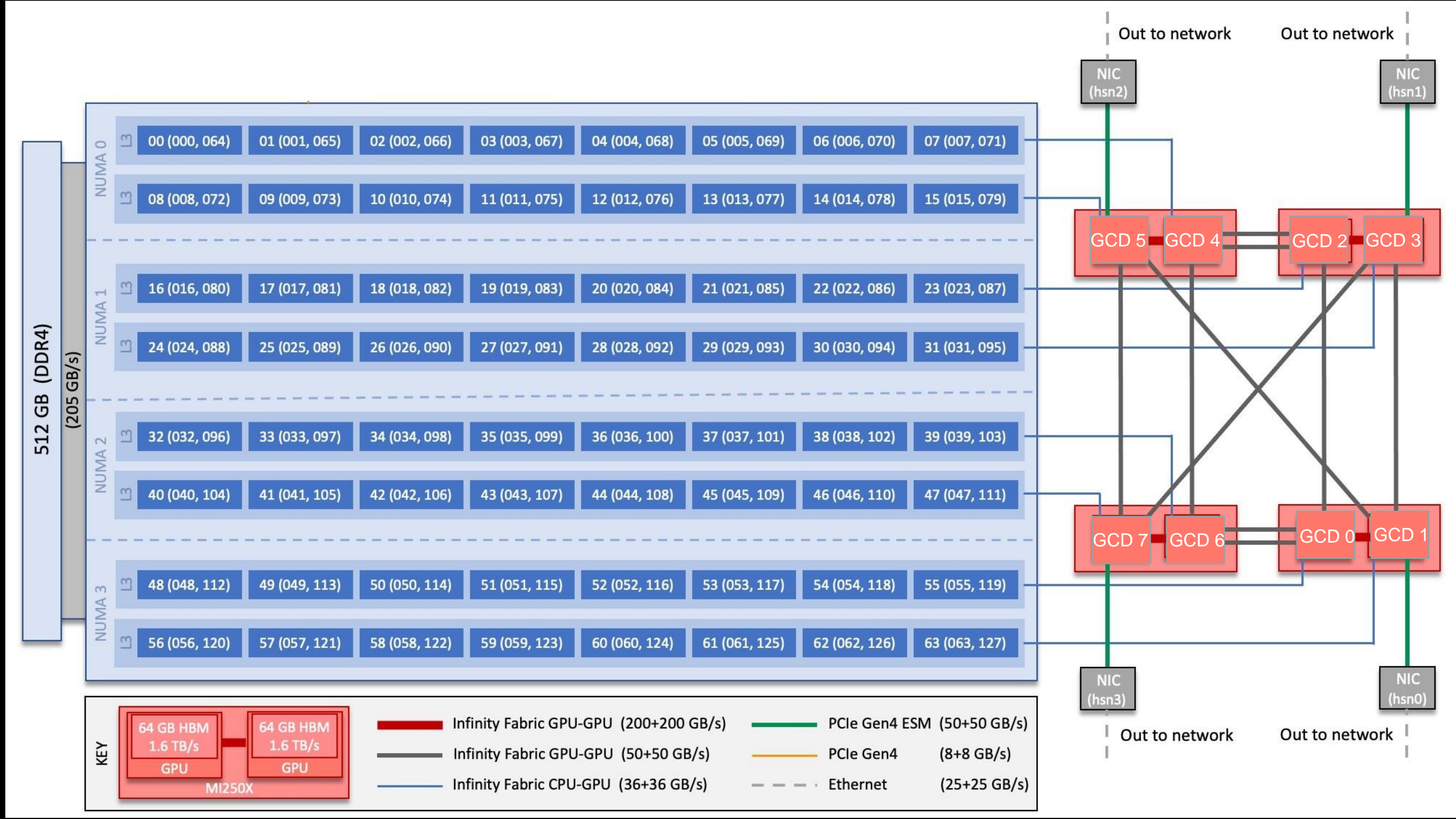
- One graphic compute die (GCD)
- 32GB of HBM2 memory
- 11.5 TFLOPS peak performance per GCD
- 1.2 TB/s peak memory bandwidth per GCD
- 120 CU per GPU
- The interconnection is attached on the CPU

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

64-core AMD "Optimized 3rd Gen EPYC" CPU Core Chiplet Die connected to GCD via Infinity Fabric CPU-GPU

AMD together we advance_

[Public]

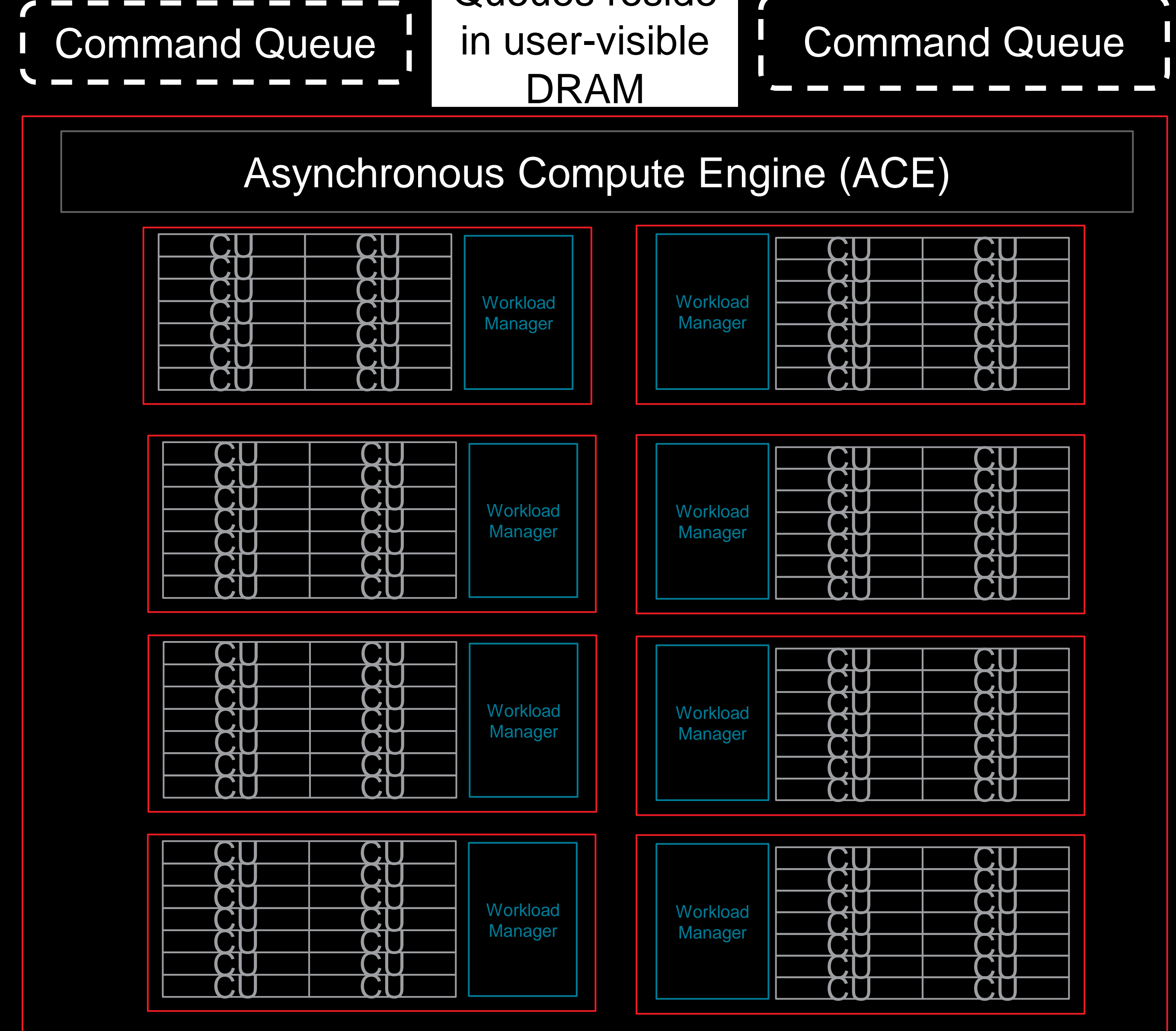
AMD GCN GPU Hardware Layout (MI250X one GCD)

Asynchronous Compute Engine (ACE)

Shader Engine (SE0)	Shader Engine (SE1)
Shader Engine (SE2)	Shader Engine (SE3)
Shader Engine (SE4)	Shader Engine (SE5)
Shader Engine (SE6)	Shader Engine (SE7)

AMD
together we advance_

AMD GCN GPU Hardware Layout (MI250X one GCD)



AMD
together we advance_

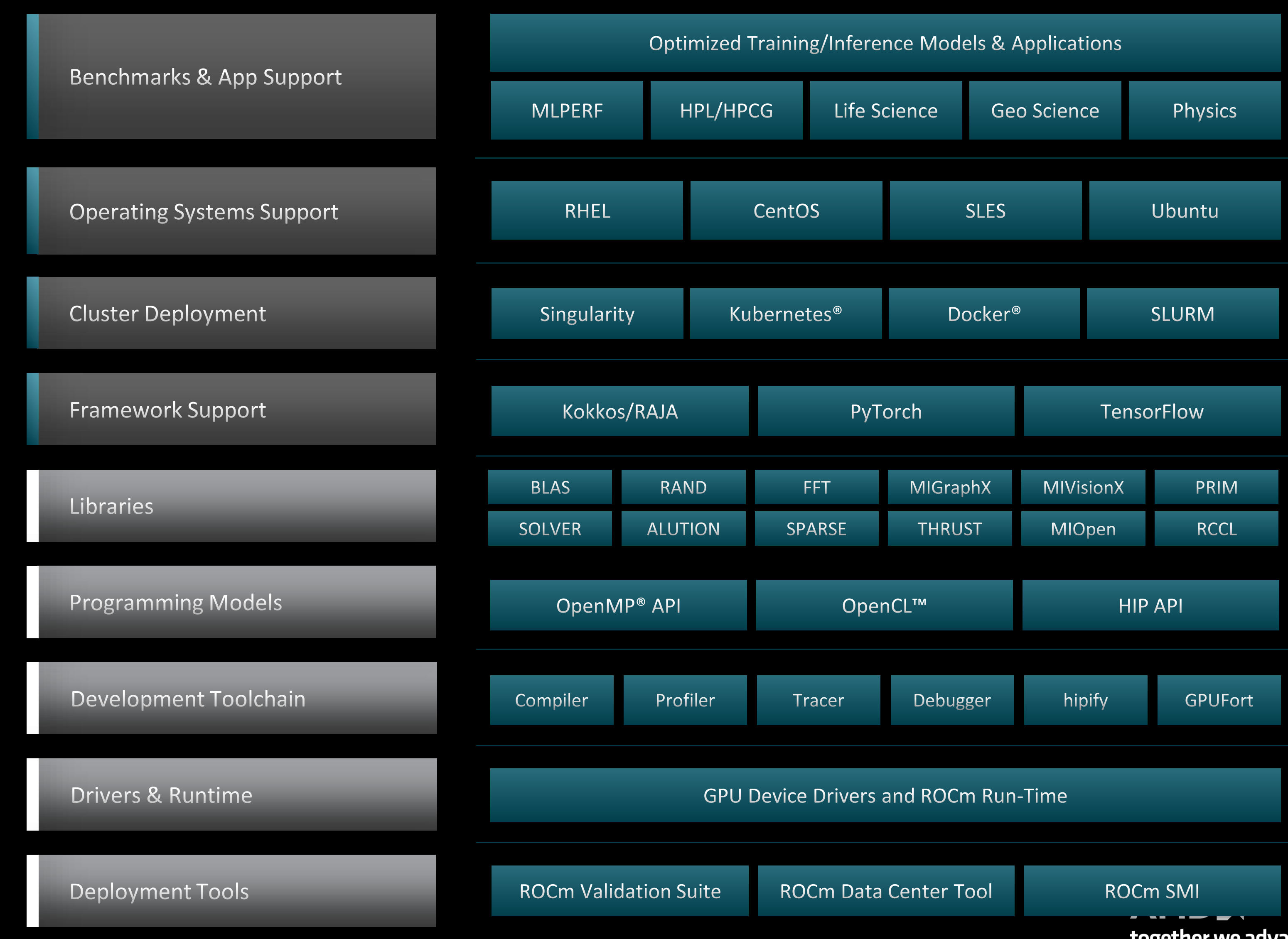
ROCm and HIP

[Public]

Open Software Platform For GPU Compute

AMD ROCm

- Unlocked GPU Power To Accelerate Computational Tasks
- Optimized for HPC and Deep Learning Workloads at Scale
- Open Source Enabling Innovation, Differentiation, and Collaboration



AMD ROCm 5.0

DEMOCRATIZING EXASCALE FOR ALL

EXPANDING SUPPORT & ACCESS

- Support for Radeon Pro W6800 Workstation GPUs
- Remote access through the AMD Accelerator Cloud

OPTIMIZING PERFORMANCE

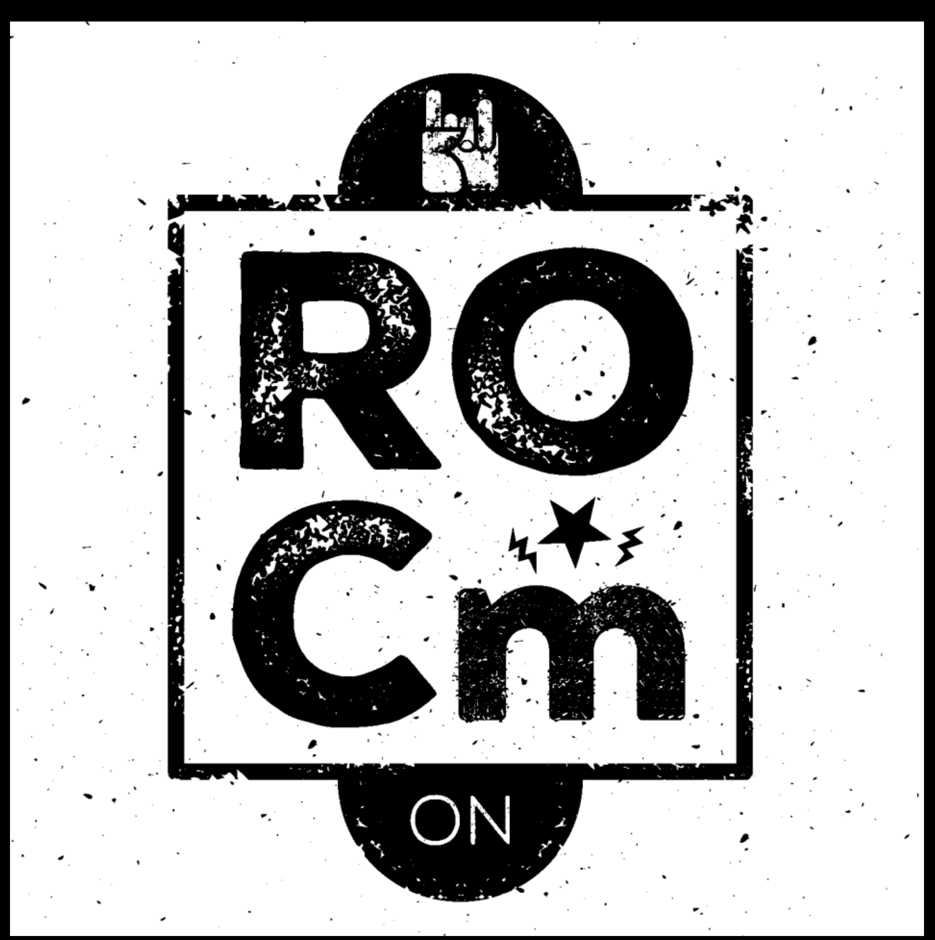
- MI200 Optimizations: FP64 Matrix ops, Improved Cache
- Improved launch latency and kernel performance

ENABLING DEVELOPER SUCCESS

- HPC Apps & ML Frameworks on AMD InfinityHub
- Streamlined and improved tools increasing productivity

ROCm - Radeon Open Compute Platform

- **H**eterogeneous-compute **I**nterface for **P**ortability (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



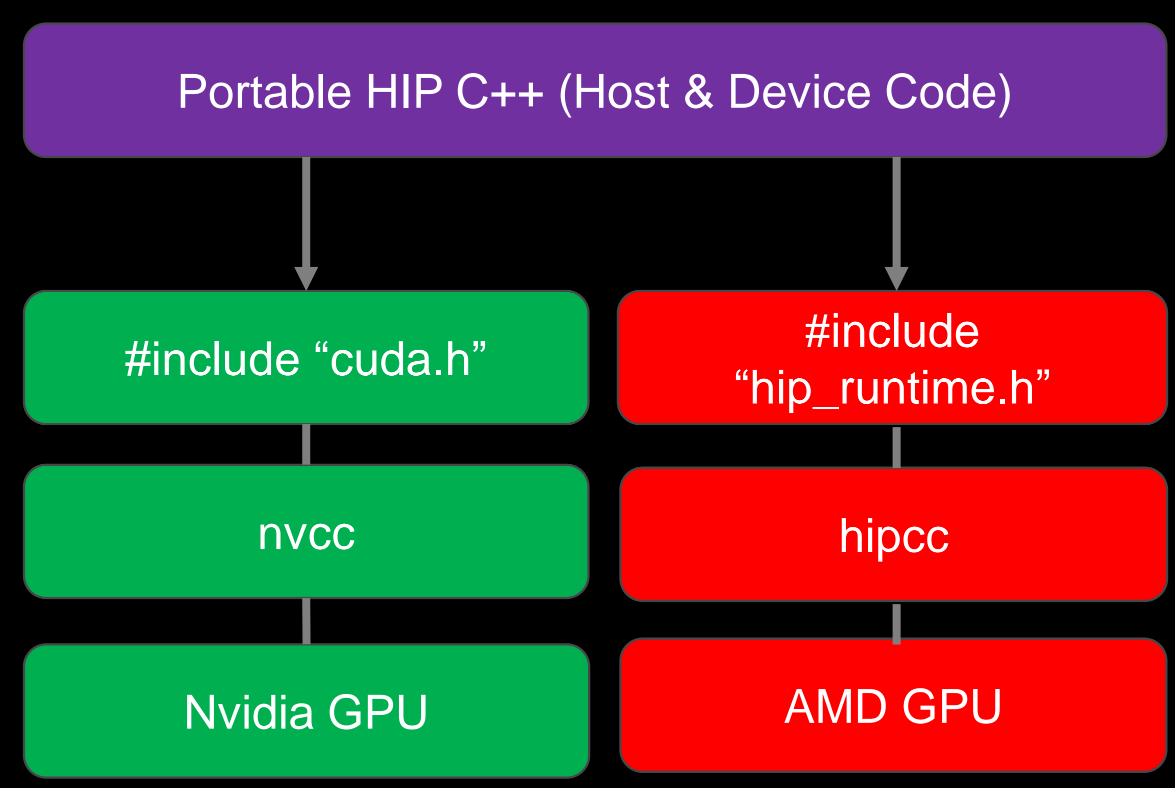
AMD
together we advance_

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



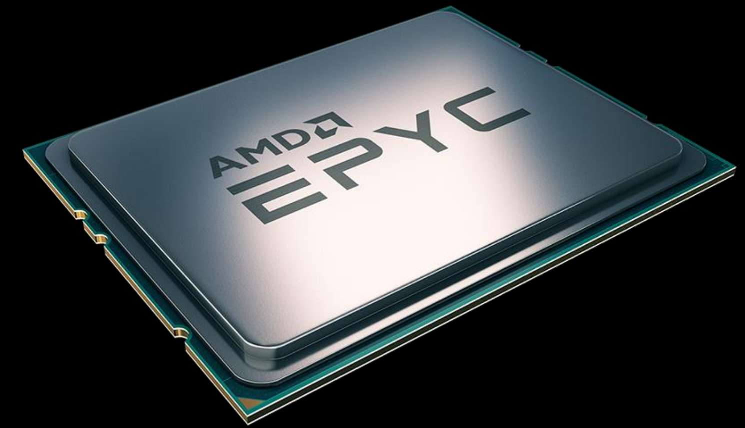
AMD
together we advance_

[Public]

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”

AMD
together we advance_

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

AMD
together we advance_

[Public]

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();

AMD
together we advance_

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, ...);
```


AMD
together we advance_

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

AMD
together we advance_

The Grid: blocks of threads in 1D

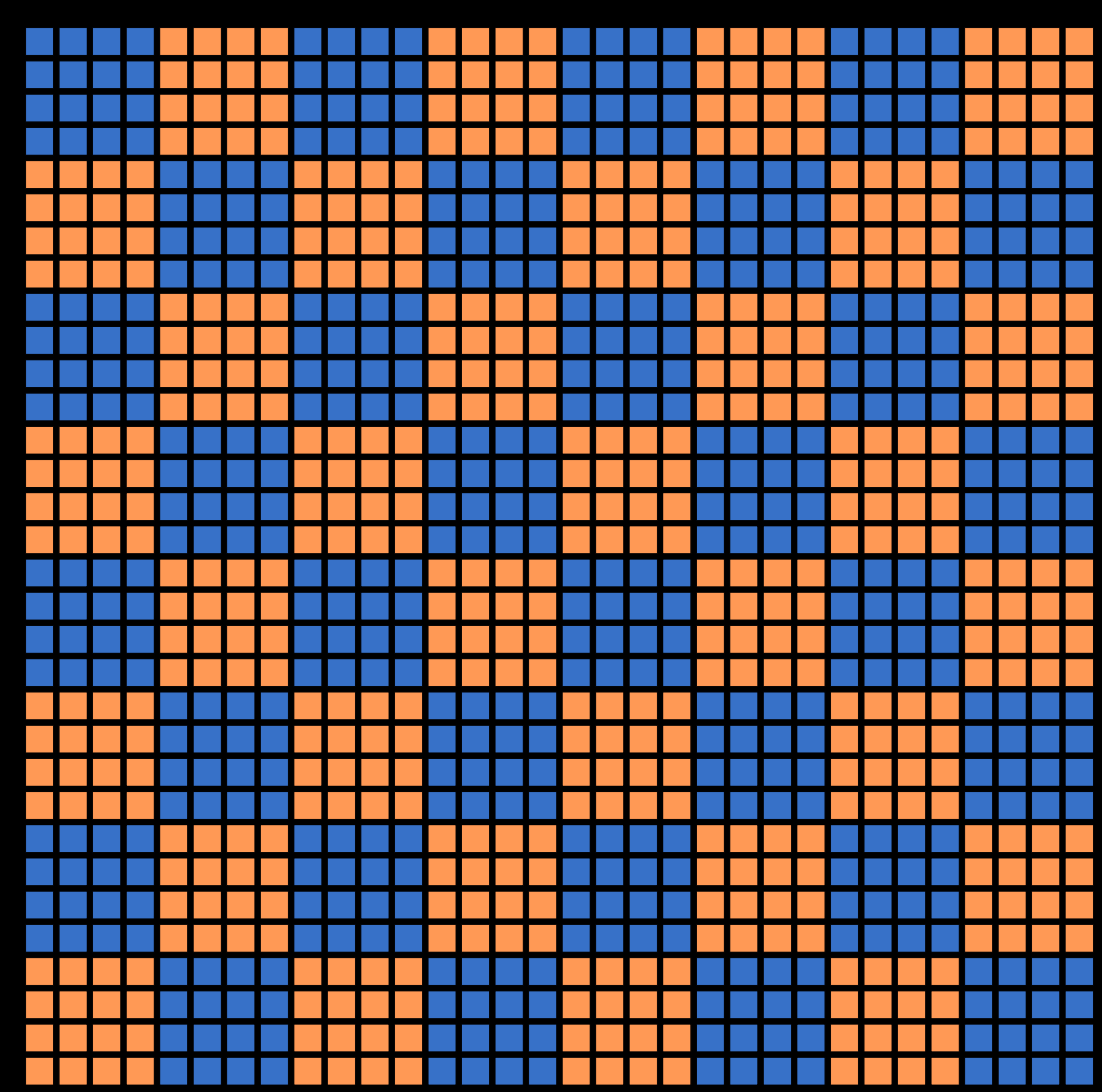
Threads in grid have access to:

- 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

AMD
together we advance_

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.

AMD
together we advance_

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

AMD
together we advance_

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,            //Kernel name (__global__ void function)
                   blocks,              //Grid dimensions
                   threads,             //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);
```


AMD
together we advance_

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)

AMD
together we advance_

[Public]

Device Memory

The host instructs the device to allocate memory in VRAM and records a pointer to device memory:

```
int main() {
  …
  int N = 1000;
  size_t Nbytes = N*sizeof(double);
  double *h_a = (double*) malloc(Nbytes);                //Host memory

  double *d_a = NULL;
  hipMalloc(&d_a, Nbytes);                               //Allocate Nbytes on device


  …

  free(h_a);                                             //free host memory
  hipFree(d_a);                                          //free device memory
}
```


AMD
together we advance_

Device Memory

The host queues memory transfers:

```
//copy data from host to device
hipMemcpy(d_a, h_a, Nbytes, hipMemcpyHostToDevice);

//copy data from device to host
hipMemcpy(h_a, d_a, Nbytes, hipMemcpyDeviceToHost);

//copy data from one device buffer to another
hipMemcpy(d_b, d_a, Nbytes, hipMemcpyDeviceToDevice);
```

Device Memory

Can copy strided sections of arrays:

```
hipMemcpy2D(d_a,          //pointer to destination
            DLDAbytes,    //pitch of destination array
            h_a,          //pointer to source
            LDAbytes,     //pitch of source array
            Nbytes,       //number of bytes in each row
            Nrows,        //number of rows to copy
            hipMemcpyHostToDevice);
```

[Public]

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 hipGetErrorString(status). Helpful for debugging, e.g.,

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


AMD
together we advance_

Putting it all together

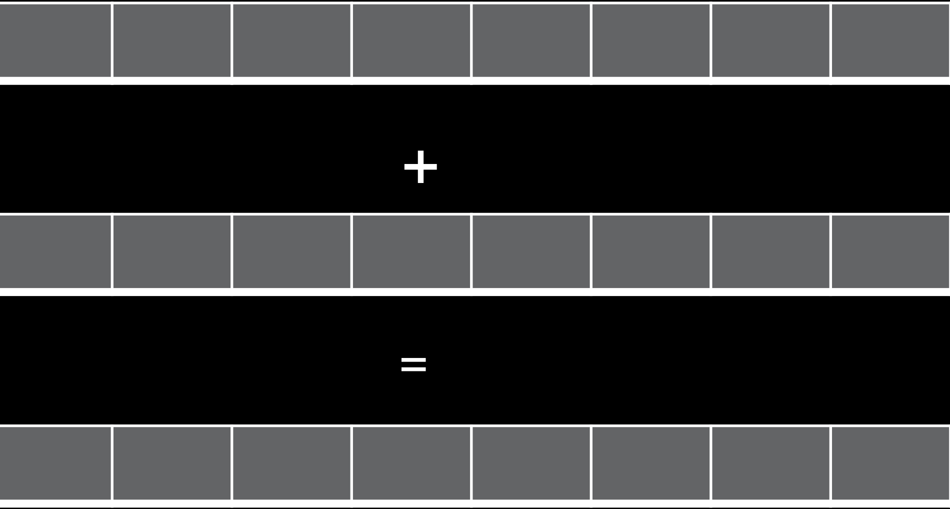
```
#include "hip/hip_runtime.h"
int main() {
  int N = 1000;
  size_t Nbytes = N*sizeof(double);
  double *h_a = (double*) malloc(Nbytes);  //host memory
  double *d_a = NULL;
  HIP_CHECK(hipMalloc(&d_a, Nbytes));
  …
  HIP_CHECK(hipMemcpy(d_a, h_a, Nbytes, hipMemcpyHostToDevice));   //copy data to device

  hipLaunchKernelGGL(myKernel, dim3((N+256-1)/256,1,1), dim3(256,1,1), 0, 0, N, d_a); //Launch kernel
  HIP_CHECK(hipGetLastError());

  HIP_CHECK(hipMemcpy(h_a, d_a, Nbytes, hipMemcpyDeviceToHost));
  …
  free(h_a);                    //free host memory
  HIP_CHECK(hipFree(d_a));   //free device memory
}
```

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

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

[Public]

Vector Addition



Let's discuss an example with:

- Dimension of 16384*16384
- 16 blocks for X and Y dimensions and 1 for Z dimension

AMD
together we advance_

[Public]

Vector Addition (example code)

```
…
hostA = (float*)malloc(NUM * sizeof(float));
hostB = (float*)malloc(NUM * sizeof(float));
hostC = (float*)malloc(NUM * sizeof(float));
//initialize
…
hipMalloc((void**)&deviceA, NUM * sizeof(float));
hipMalloc((void**)&deviceB, NUM * sizeof(float));
hipMalloc((void**)&deviceC, NUM * sizeof(float));

hipMemcpy(deviceB, hostB, NUM*sizeof(float), hipMemcpyHostToDevice);
hipMemcpy(deviceC, hostC, NUM*sizeof(float), hipMemcpyHostToDevice);
…
```


AMD
together we advance_

Vector Addition (example code)

```
…
vectoradd_float<<<dim3(WIDTH/THREADS_PER_BLOCK_X, HEIGHT/THREADS_PER_BLOCK_Y),
                  dim3(THREADS_PER_BLOCK_X, THREADS_PER_BLOCK_Y), 0, 0>>>
                  (deviceA ,deviceB ,deviceC ,WIDTH ,HEIGHT);

hipMemcpy(hostA, deviceA, NUM*sizeof(float), hipMemcpyDeviceToHost);

// verify the results
…
hipFree(deviceA);
hipFree(deviceB);
hipFree(deviceC);
```


AMD
together we advance_

[Public]

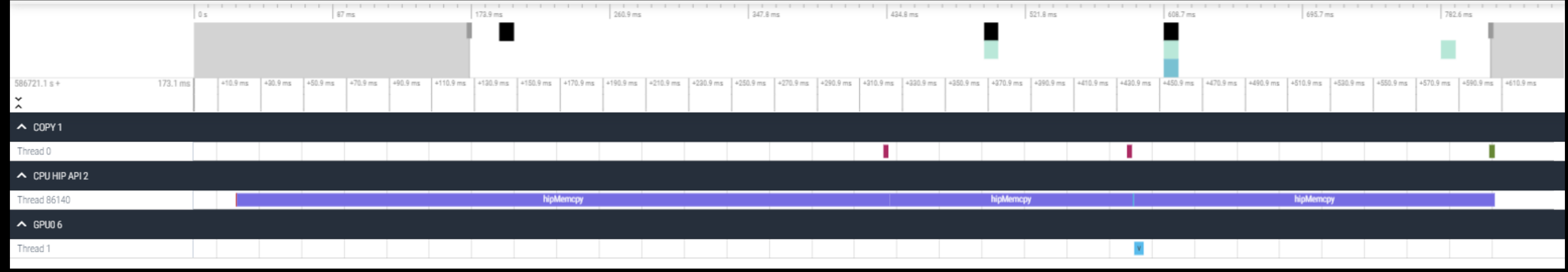
Vector addition - Profiling

rocprof --stats --hip-trace vectoradd_hip.exe

File: results.hip_stats.csv:

```
"Name",                          "Calls", "TotalDurationNs", "AverageNs", "Percentage"
"hipMemcpy",                       3,     591195337,         197065112,    99.78088892497593
"hipLaunchKernel",                 1,        637889,            637889,     0.10766176164116796
"hipMalloc",                       3,        452560,            150853,     0.07638226532880638
"hipFree",                         3,        202860,             67620,     0.03423834705807332
"hipGetDeviceProperties",          1,          2600,              2600,     0.00043882333380212493
"__hipPushCallConfiguration",      1,          1860,              1860,     0.0003139274648921245
"__hipPopCallConfiguration",       1,           450,               450,     7.595019311906238e-05
```

Perfetto - visualization



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.

AMD
together we advance_

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++ -stdc=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

AMD
together we advance_

[Public]

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

AMD
together we advance_

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 enqueued**
  - 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);
```

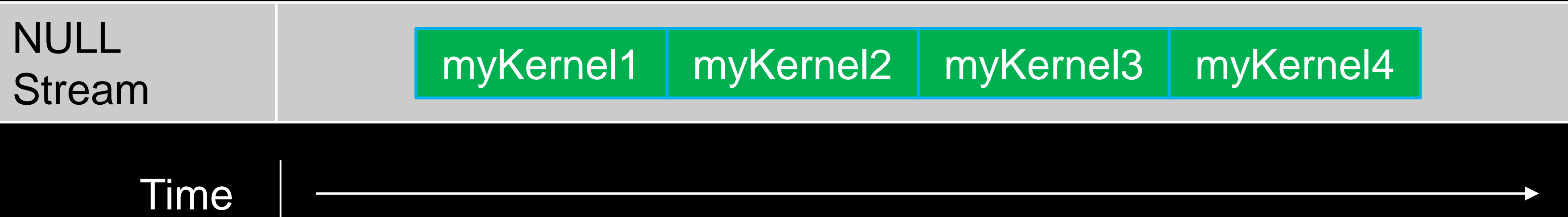
- Passing 0 or NULL as the hipStream_t argument to a function instructs the function to execute on a stream called the 'NULL Stream':
  - No task on the NULL stream will begin until **all previously enqueued tasks in all other streams have completed**
  - Blocking calls like hipMemcpy run on the NULL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



AMD
together we advance_

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.

AMD
together we advance_

[Public]

Streams

- There is another use for streams besides concurrent kernels:
  - Overlapping kernels with data movement.

- AMD GPUs have separate engines for:
  - Host->Device memcpys
  - Device->Host memcpys
  - Compute kernels.

- These three different operations can overlap without dividing the GPU's resources.
  - The overlapping operations should be in separate, non-NULL, streams.
  - The host memory should be **pinned.**

[Public]

Pinned Memory

Host data allocations are pageable by default. The GPU can directly access Host data if it is pinned instead.

- Allocating pinned host memory:

```
double *h_a = NULL;
hipHostMalloc(&h_a, Nbytes);
```

- Free pinned host memory:

```
hipHostFree(h_a);
```

- Host<->Device memcpy **bandwidth increases significantly when host memory is pinned**.
  - It is good practice to allocate host memory that is frequently transferred to/from the device as pinned memory.

AMD
together we advance_

Streams

Suppose we have 3 kernels which require moving data to and from the device:

```
hipMemcpy(d_a1, h_a1, Nbytes, hipMemcpyHostToDevice));
hipMemcpy(d_a2, h_a2, Nbytes, hipMemcpyHostToDevice));
hipMemcpy(d_a3, h_a3, Nbytes, hipMemcpyHostToDevice));

hipLaunchKernelGGL(myKernel1, blocks, threads, 0, 0, N, d_a1);
hipLaunchKernelGGL(myKernel2, blocks, threads, 0, 0, N, d_a2);
hipLaunchKernelGGL(myKernel3, blocks, threads, 0, 0, N, d_a3);

hipMemcpy(h_a1, d_a1, Nbytes, hipMemcpyDeviceToHost);
hipMemcpy(h_a2, d_a2, Nbytes, hipMemcpyDeviceToHost);
hipMemcpy(h_a3, d_a3, Nbytes, hipMemcpyDeviceToHost);
```

NULL Stream	HToD1	HToD2	HToD3	myKernel 1	myKernel 2	myKernel 3	DToH1	DToH2	DToH3

AMD
together we advance_

[Public]

Streams

Changing to asynchronous memcpys and using streams:

```
hipMemcpyAsync(d_a1, h_a1, Nbytes, hipMemcpyHostToDevice, stream1);
hipMemcpyAsync(d_a2, h_a2, Nbytes, hipMemcpyHostToDevice, stream2);
hipMemcpyAsync(d_a3, h_a3, Nbytes, hipMemcpyHostToDevice, stream3);

hipLaunchKernelGGL(myKernel1, blocks, threads, 0, stream1, N, d_a1);
hipLaunchKernelGGL(myKernel2, blocks, threads, 0, stream2, N, d_a2);
hipLaunchKernelGGL(myKernel3, blocks, threads, 0, stream3, N, d_a3);

hipMemcpyAsync(h_a1, d_a1, Nbytes, hipMemcpyDeviceToHost, stream1);
hipMemcpyAsync(h_a2, d_a2, Nbytes, hipMemcpyDeviceToHost, stream2);
hipMemcpyAsync(h_a3, d_a3, Nbytes, hipMemcpyDeviceToHost, stream3);
```

NULL Stream					
Stream1	HToD1	myKernel 1	DToH1		
Stream2		HToD2	myKernel 2	DToH2	
Stream3			HToD3	myKernel 3	DToH3

AMD
together we advance_

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

AMD
together we advance_

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

AMD
together we advance_

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

AMD
together we advance_

[Public]

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.

AMD
together we advance_

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

[Public]

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

AMD
together we advance_

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

AMD
together we advance_

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

AMD
together we advance_

[Public]

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

AMD
together we advance_

Understanding the hardware options

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

```
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)
```

```
#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

AMD
together we advance_

RADEON INSTINCT
ROCm Libraries

ROCm GPU Libraries

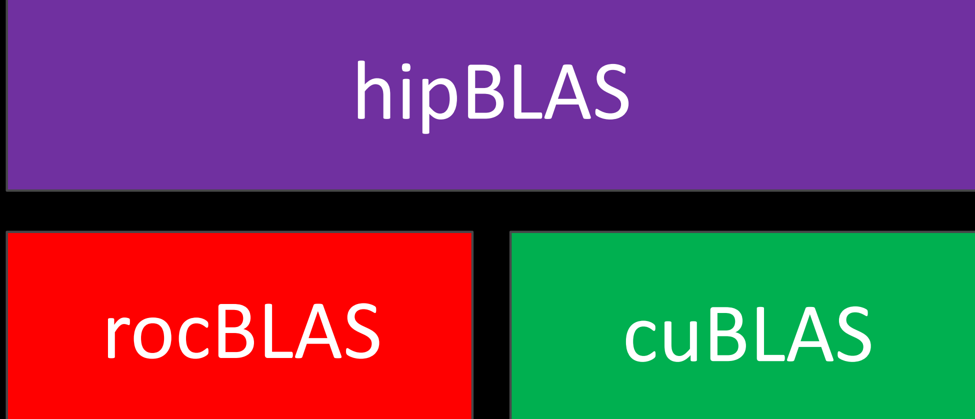
ROCm provides several GPU math libraries

- 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



AMD
together we advance_

AMD Math Library Equivalents: “Decoder Ring”

CUBLAS	ROCBLAS	Basic Linear Algebra Subroutines
CUFFT	ROCFFT	Fast Fourier Transforms
CURAND	ROCRAND	Random Number Generation
THRUST	ROCTHRUST	C++ Parallel Algorithms
CUB	ROCPRIM	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

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
together we advance_

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