Introduction to Programming GPUs with HIP

Bob Robey
Global Training Lead
Data Center GPU
AMD

HIP Lecture Series
August 14, 2023

AMD
together we advance_

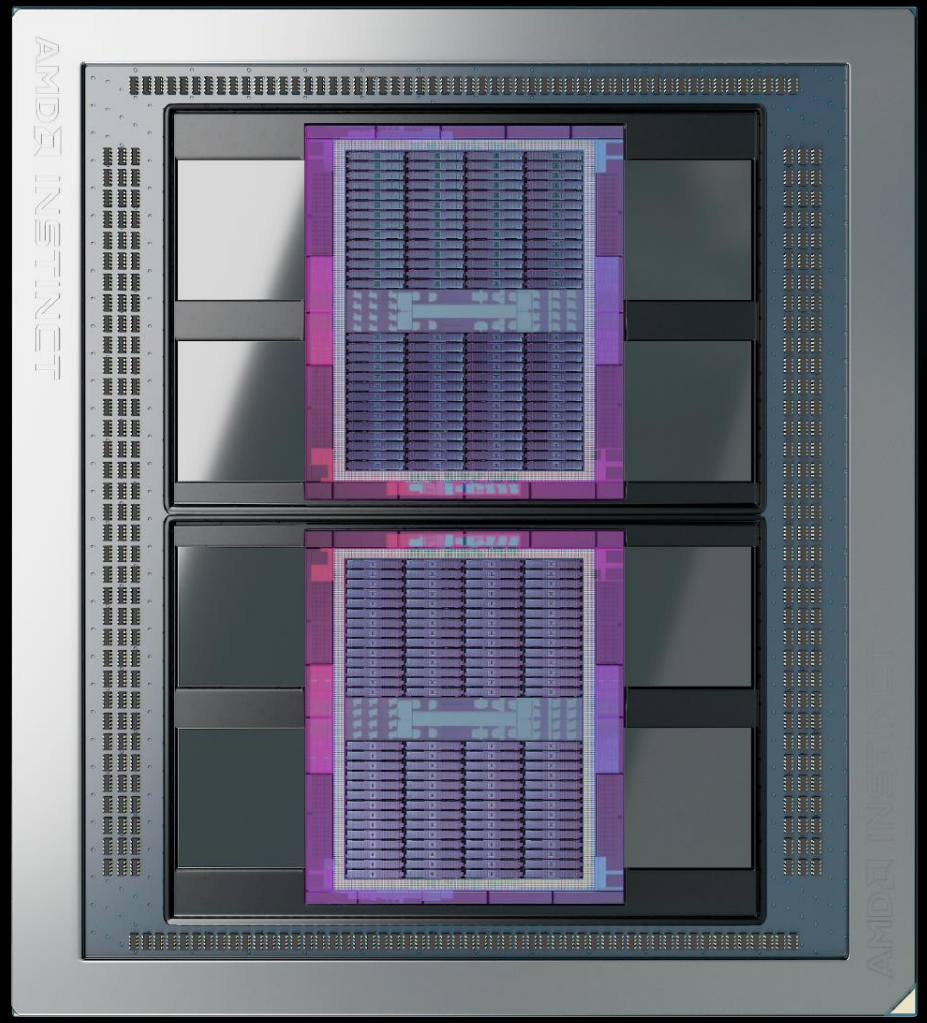
Thanks to all the AMD staff for their contributions

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
Gina Sitaraman

Agenda

1. AMD CDNA™ 2 Architecture
2. ROCm Software Ecosystem
3. AMD GPU Programming Concepts
4. Kernels, memory, and structure of host code
5. Portable Build System
6. Profiling HIP Application
7. Device management and asynchronous computing
8. Device code, shared memory, and thread synchronization
9. GPU Software
10. Shared Memory, Atomics

1. Introduction to the AMD CDNA™ 2 Architecture

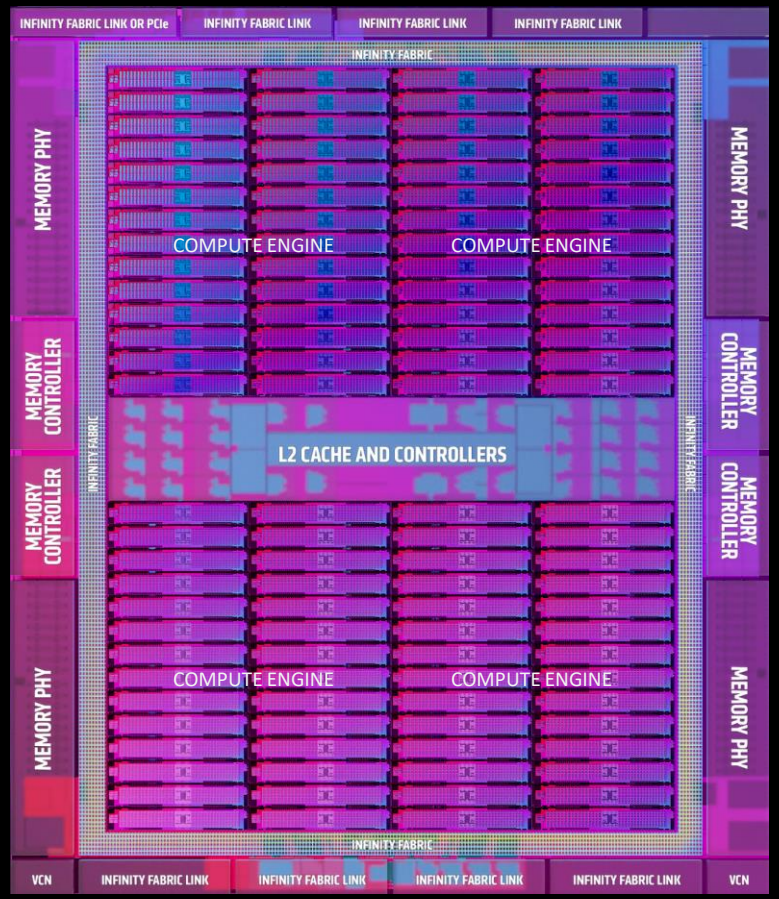


AMD INSTINCT™ MI250X

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

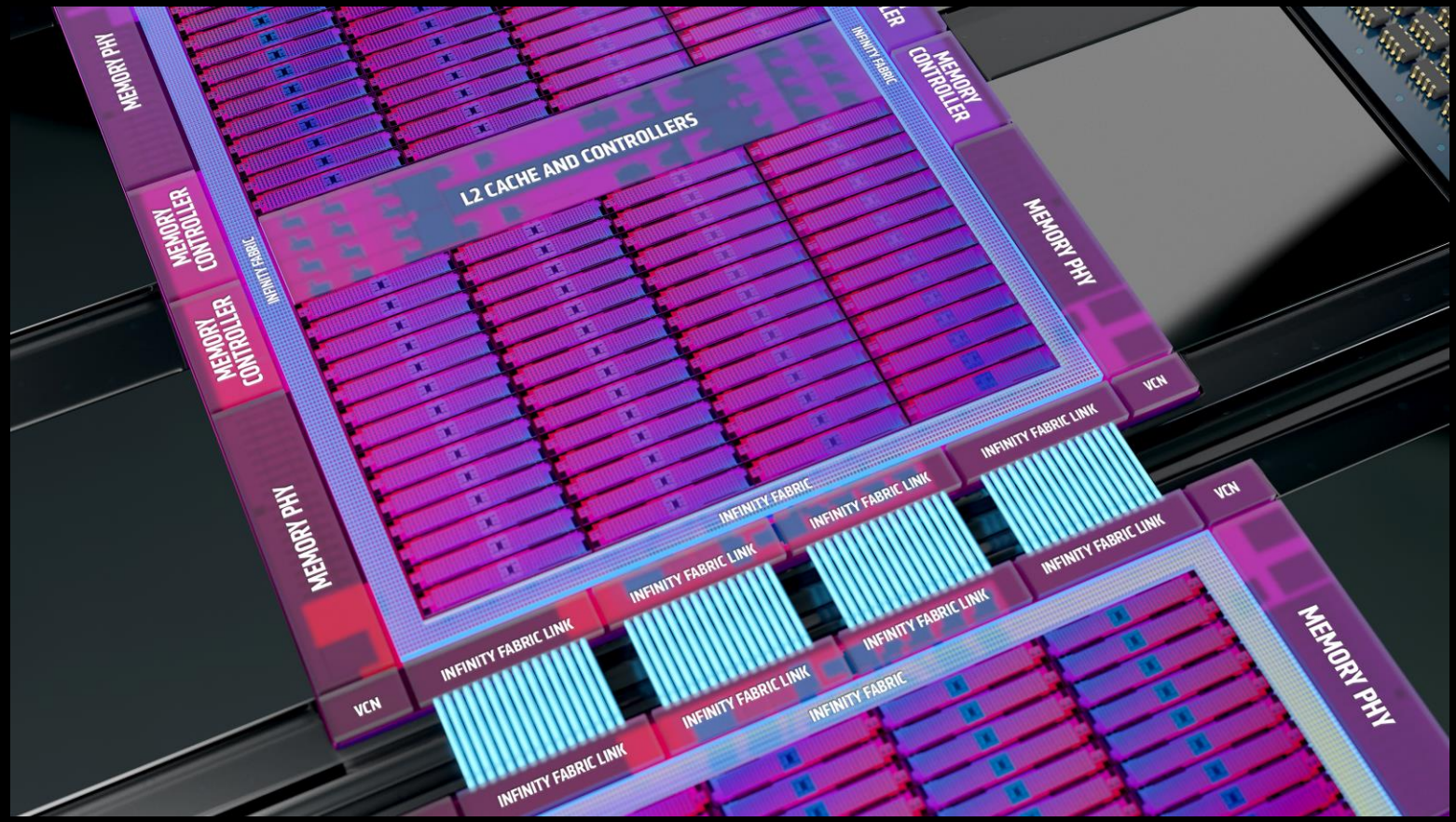
TAILORED-BUILT FOR HPC & AI

- 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

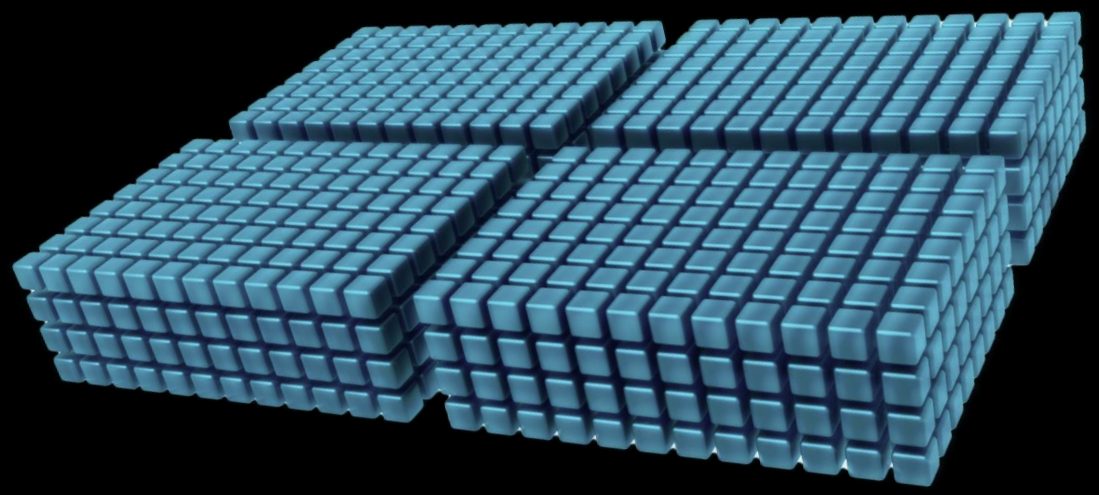
TWO GPU DIES IN PACKAGE TO MAXIMIZE COMPUTE & DATA THROUGHPUT

- INFINITY FABRIC FOR CROSS-DIE CONNECTIVITY
- 4 LINKS RUNNING AT 25GBPS
- 400GB/S OF BI-DIRECTIONALBANDWIDTH



2nd GENERATION MATRIX CORES

OPTIMIZED COMPUTE UNITS FOR SCIENTIFIC COMPUTING

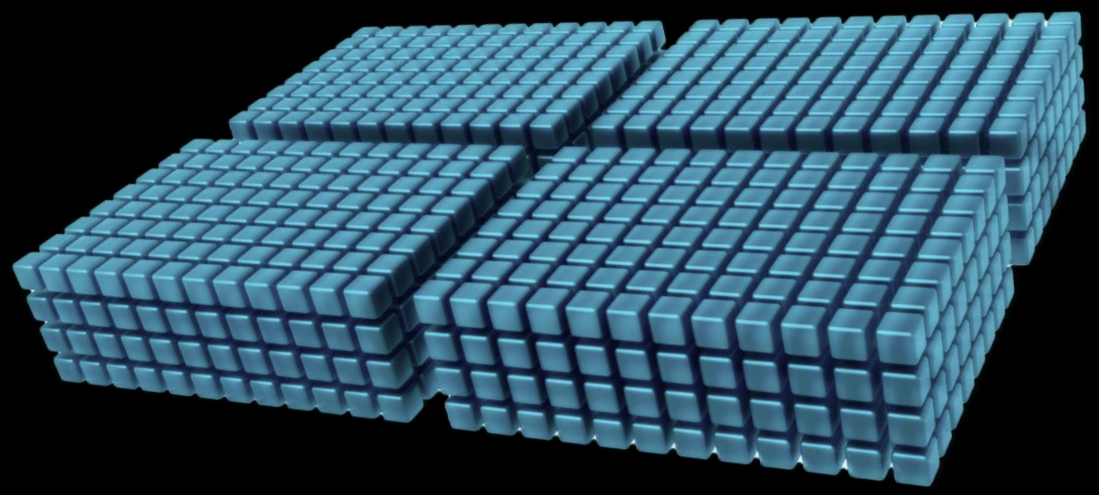


DOUBLE PRECISON (FP64)
MATRIX CORE THROUGHPUT
REPRESENTATION

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

AMD Matrix Cores Blog Post: https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-matrix-cores-readme/

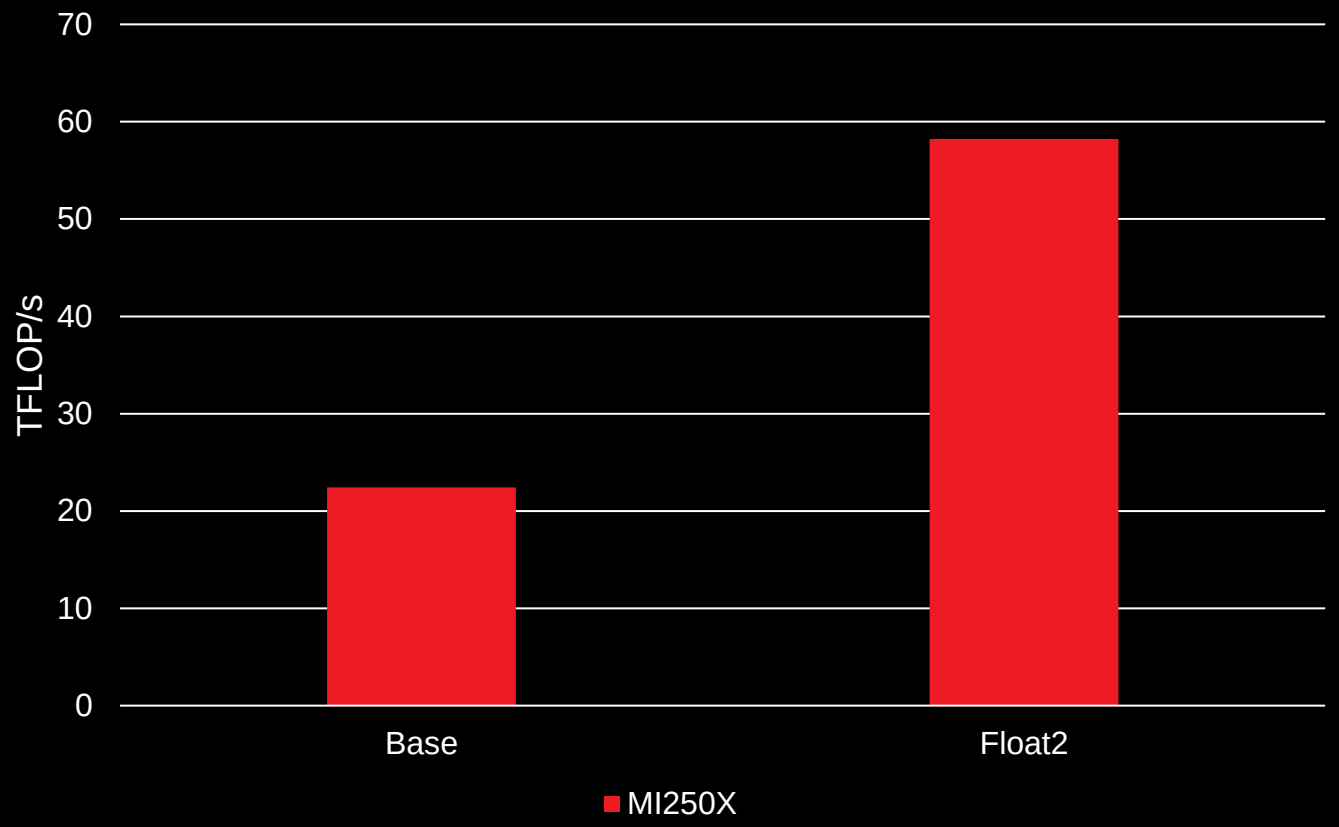
NEW IN AMD INSTINCT MI250X

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

From AMD MI100 to AMD MI250X

MI100

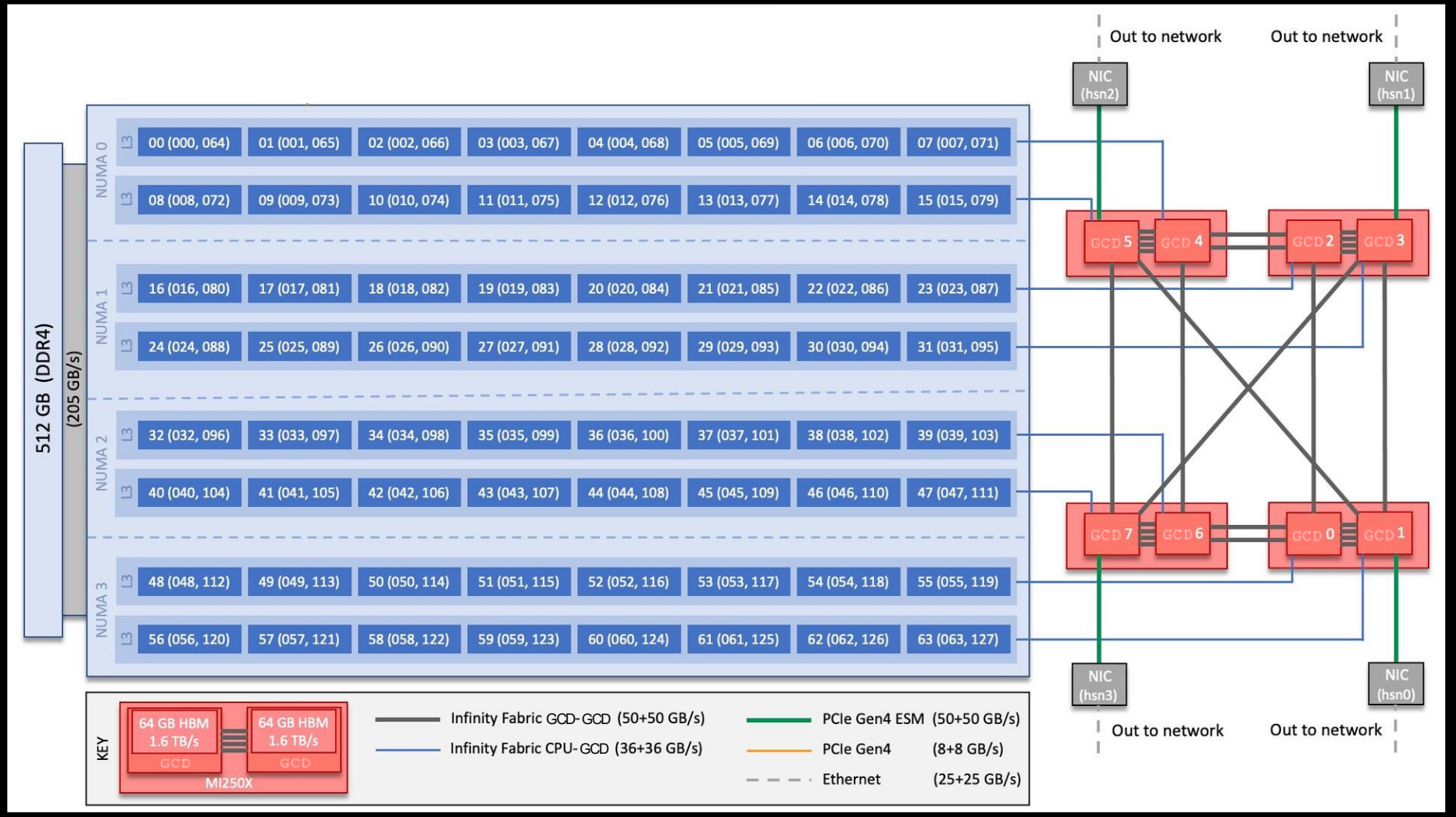
- One graphic compute die (GCD)
- 32GB of HBM2 memory
- 11.5 TFLOPS peak performance per GCD
- 1.2 TB/s peak memory bandwidth per GCD
- 120 CU per GPU
- The interconnection is attached on the CPU

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

MI250X Node Architecture



- 64 cores on a single socket CPU
- 4 MI250X GPUs, each with 2 GCDs
  - Each GCD is presented as a GPU device to `rocm-smi`
- 512 GB of DDR4 RAM
- Infinity Fabric™ links between GCDs and between GCDs and CPU cores
- 4 NICs attached to odd numbered GCDs

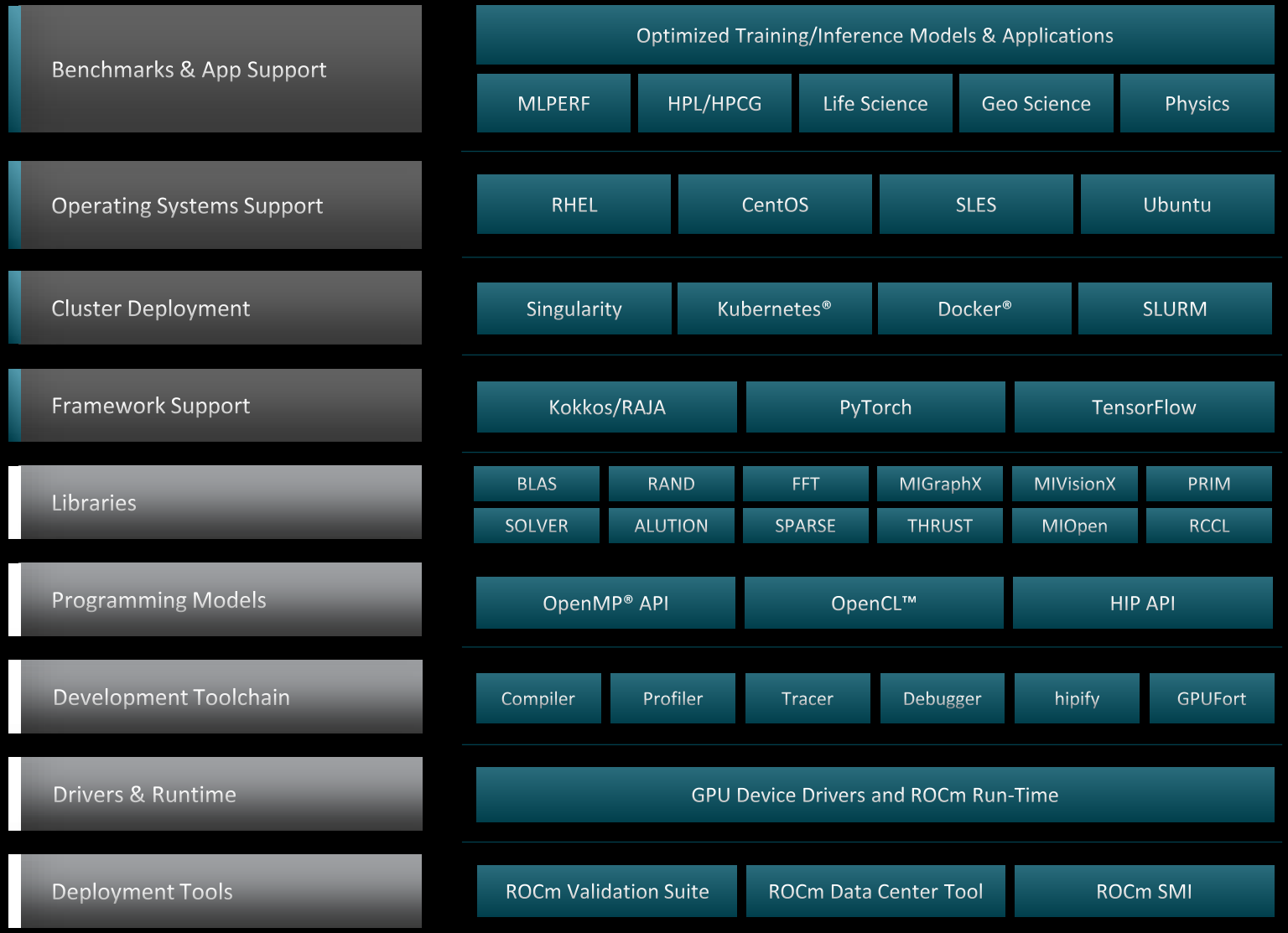
Courtesy: https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#frontier-compute-nodes

2. ROCm Software Ecosystem

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

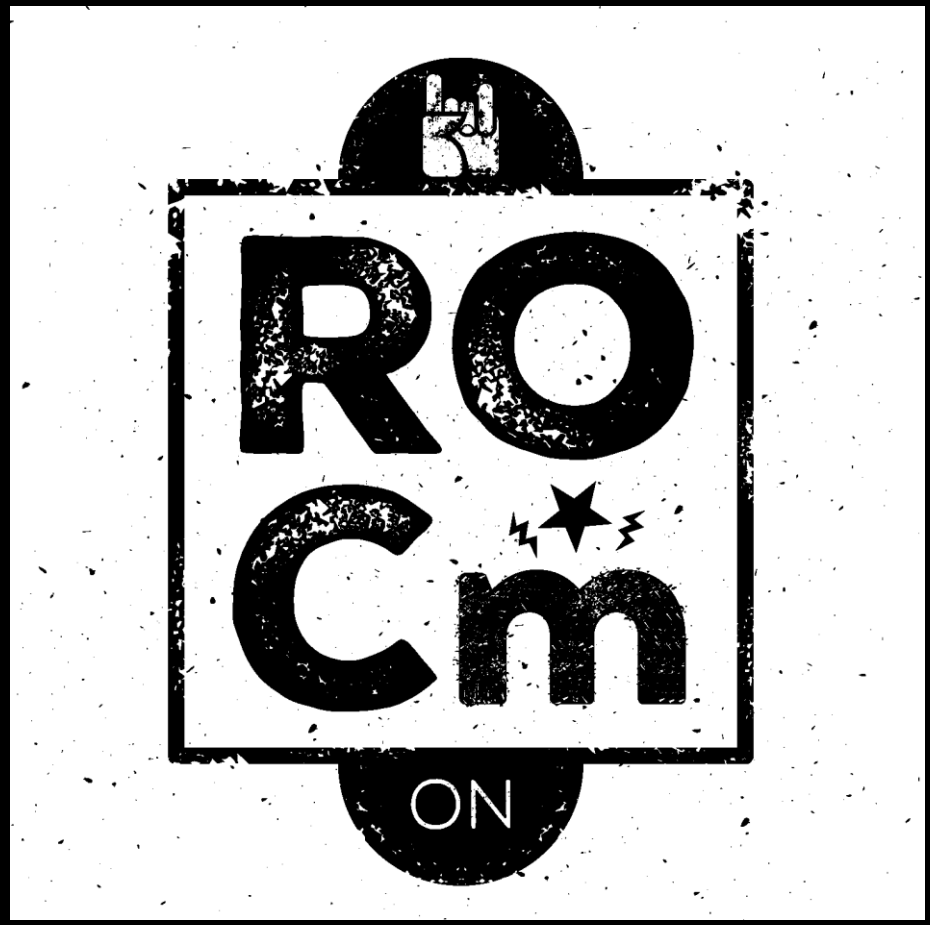
- MI200 Optimizations: FP64 Matrix ops, Improved Cache
- Improved launch latency and kernel performance

ENABLING DEVELOPER SUCCESS

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

ROCm Software Ecosystem

- **H**eterogeneous-compute **I**nterface for **P**ortability (HIP) is part of a larger software distribution called ROCm
- Install instructions and documentation:
  - https://rocm.docs.amd.com/en/latest/deploy/linux/quick_start.html
  - https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-rocm-installation-readme/
- 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



3. AMD GPU Programming Concepts

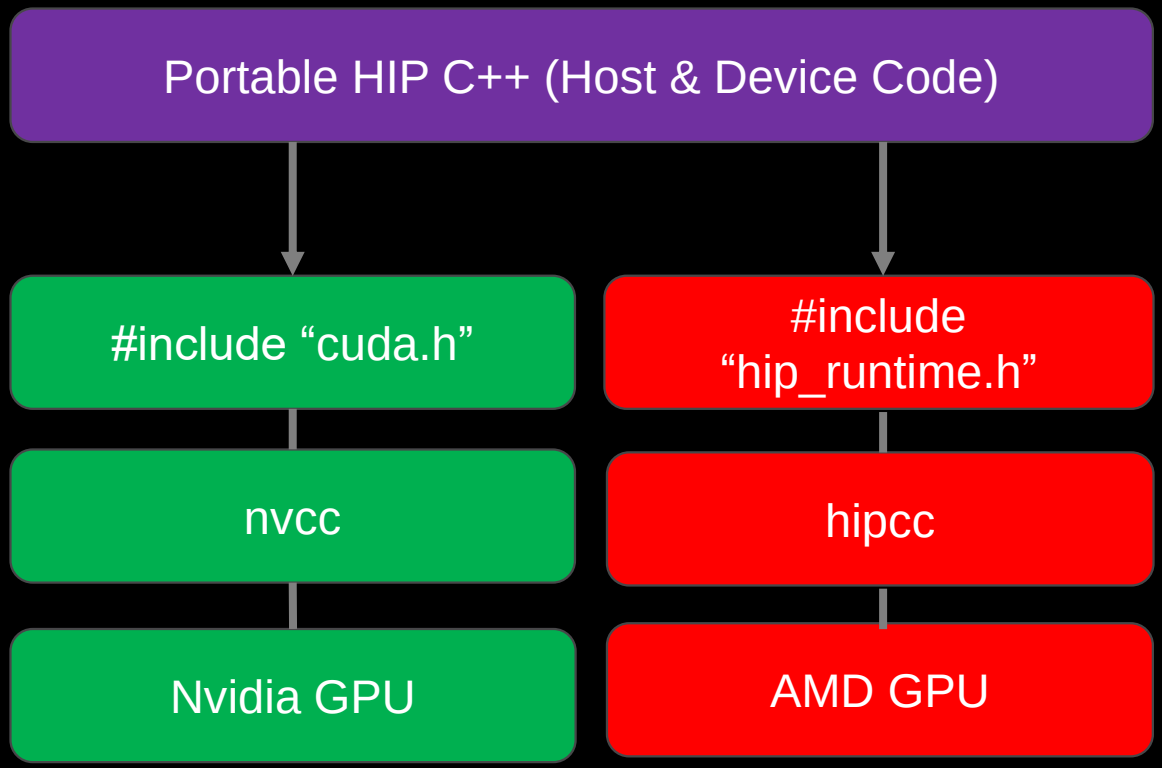
Programming with HIP: Kernels, blocks, threads, and more

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:

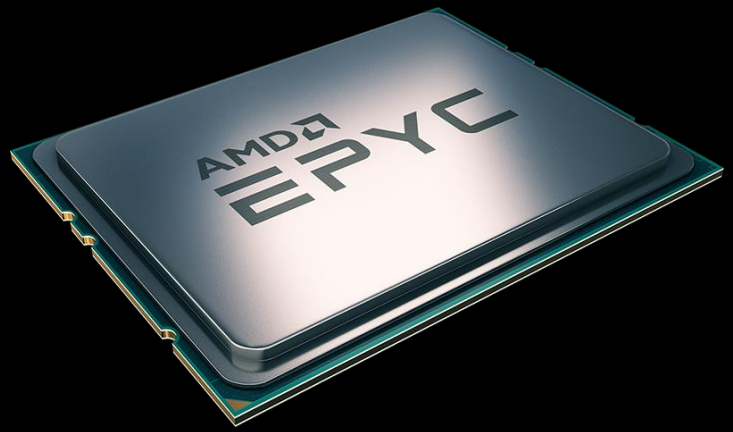
- Is open-source
- Provides an API for an application to leverage GPU acceleration for both AMD and CUDA devices
- Syntactically similar to CUDA. Most CUDA API calls can be converted in place: cuda -> hip
- Supports a strong subset of CUDA runtime functionality



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”

HIP API

- Device Management:
  - `hipSetDevice()`, `hipGetDevice()`, `hipGetDeviceProperties()`
- Memory Management
  - `hipMalloc()`, `hipMemcpy()`, `hipMemcpyAsync()`, `hipFree()`
- Streams
  - `hipStreamCreate()`, `hipDeviceSynchronize()`, `hipStreamSynchronize()`, `hipStreamDestroy()`
- 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()`

4. Kernels, memory, and structure of host code

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:

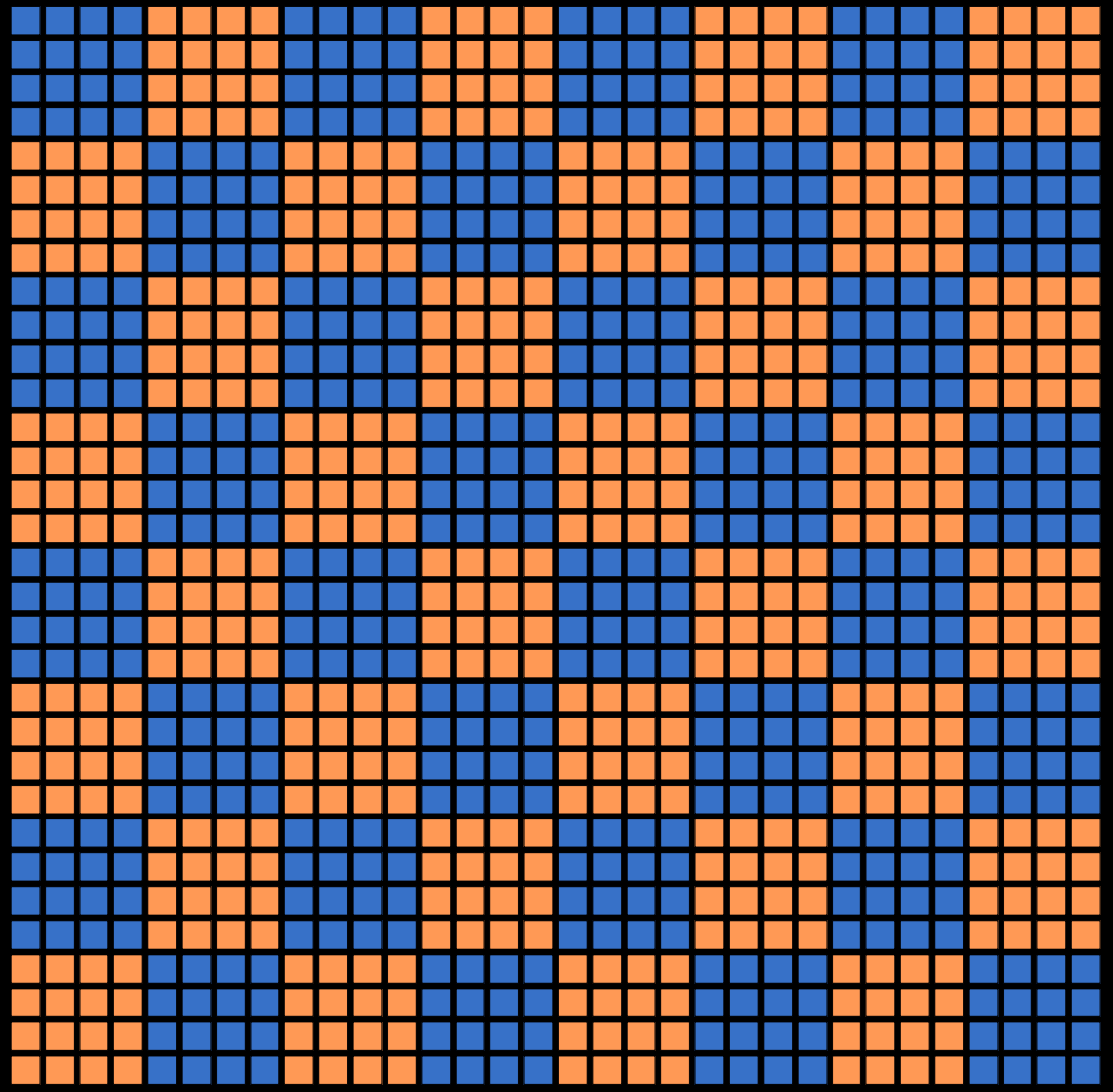
- Their respective block: blockIdx.x
- Their respective thread ID in a block: threadIdx.x
- Their block’s dimension: blockDim.x
- The number of blocks in the grid: gridDim.x

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

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

Older approach:
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
```

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)

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
}
```

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);
```

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(); } }
```

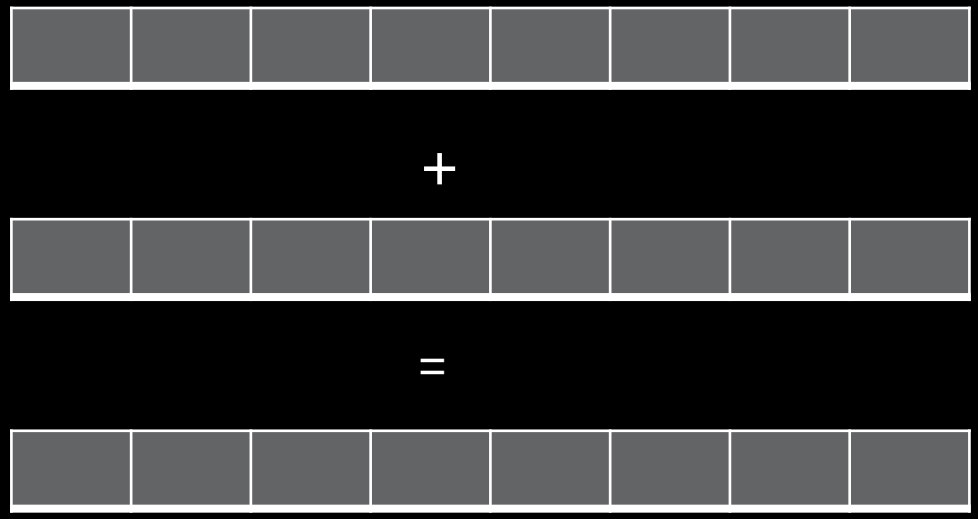
Putting it all together

```
#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

  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(); } }
```

Vector Addition



Let’s discuss an example with:

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

Vector Addition (example code)

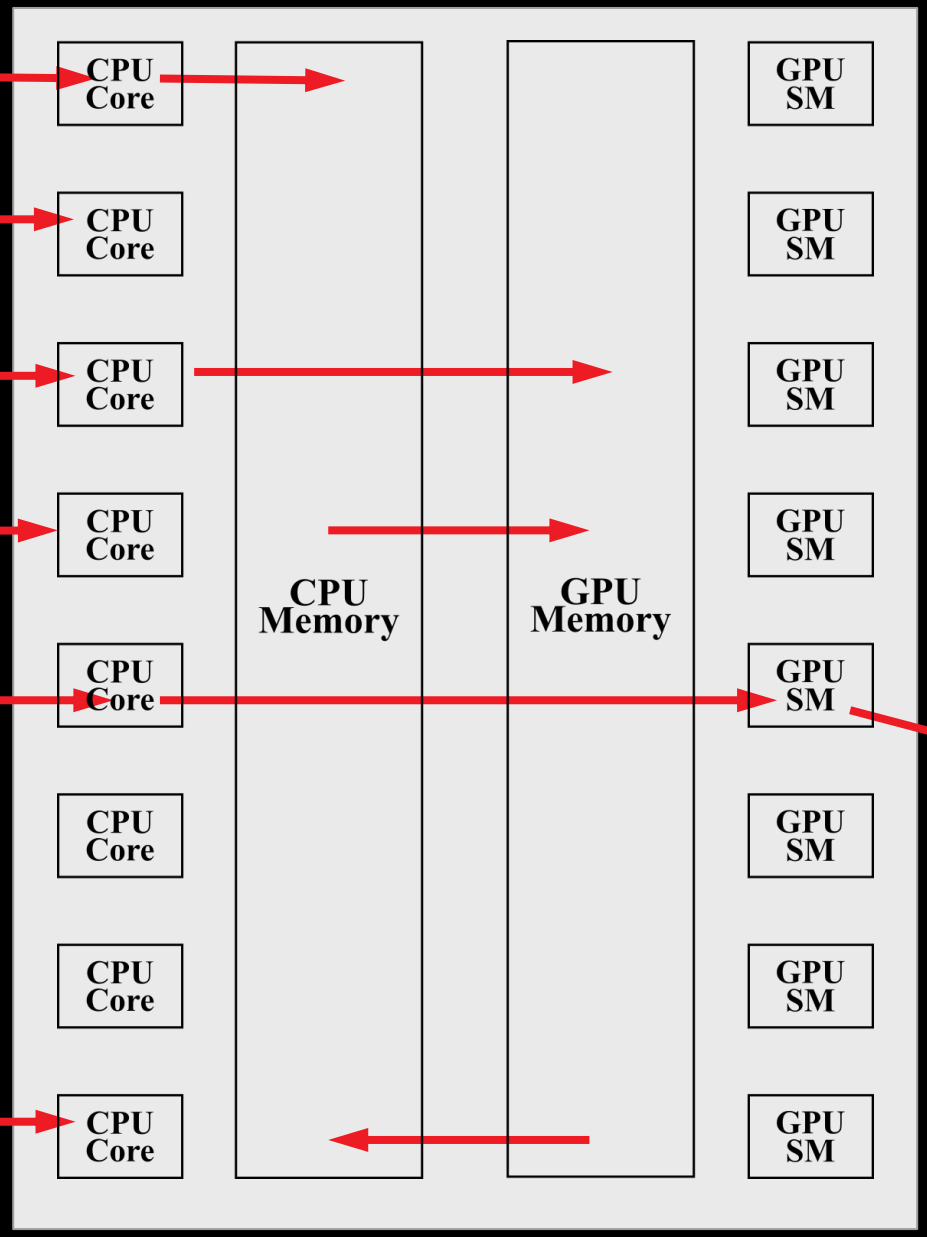
```
…
hostA = (float*)malloc(NUM * sizeof(float));
hostB = (float*)malloc(NUM * sizeof(float));
hostC = (float*)malloc(NUM * sizeof(float));
//initialize
…
hipMalloc((void**)&deviceA, NUM * sizeof(float));
hipMalloc((void**)&deviceB, NUM * sizeof(float));
hipMalloc((void**)&deviceC, NUM * sizeof(float));

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

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);
```

CPU Code

```
hostA = (float*)malloc(NUM * sizeof(float));
hostB = (float*)malloc(NUM * sizeof(float));
hostC = (float*)malloc(NUM * sizeof(float));

// initialize the input data
for (i = 0; i < NUM; i++) {
  hostB[i]
  hostC[i] = (float)i*100.0f;= (float)i;
}

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);

hipLaunchKernelGGL(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);



     ==== Executed on Device ====


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

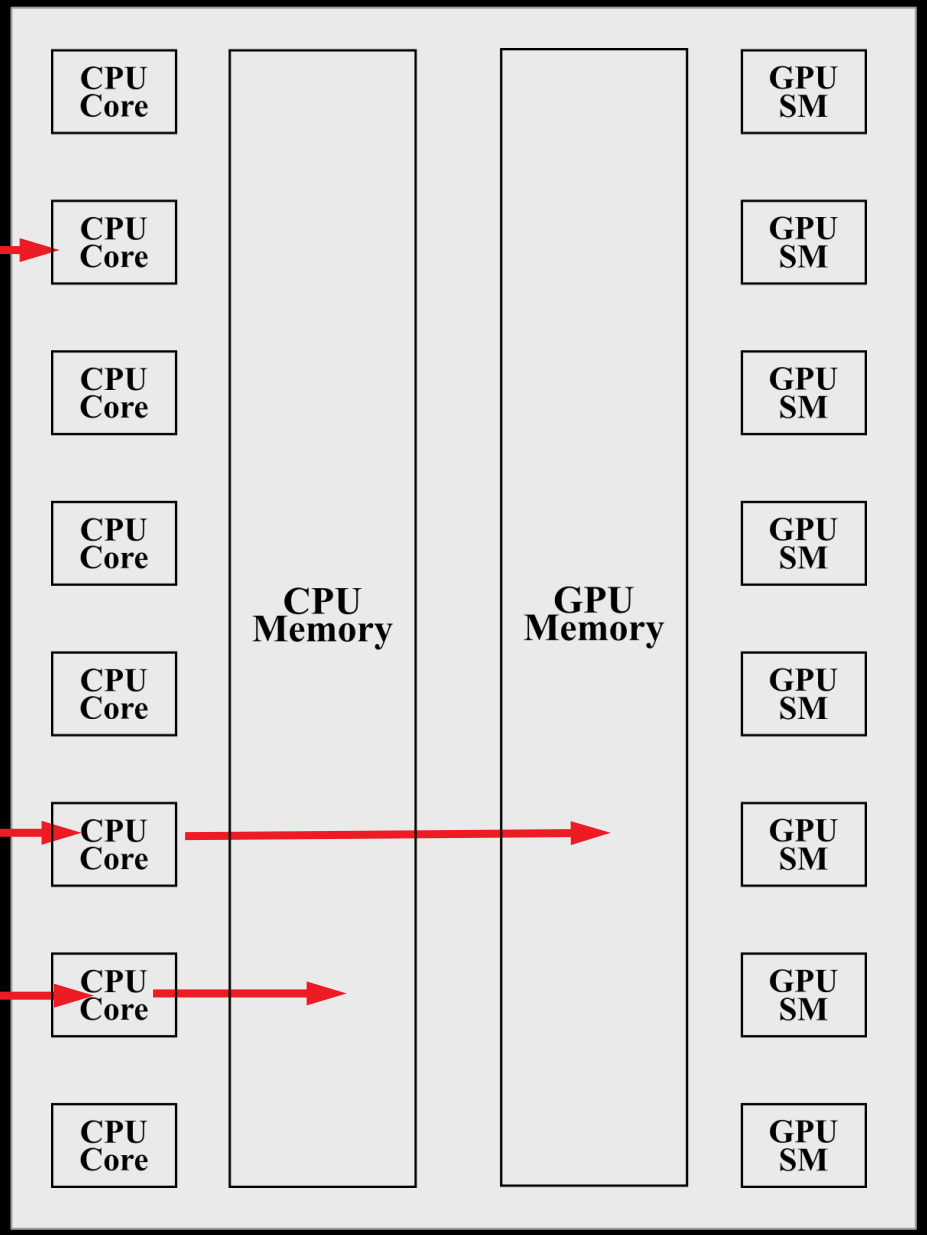


GPU Code

```
create deviceA memory
create deviceB memory
create deviceC memory


receive deviceB data
receive deviceC data

Launch kernel

__global__ void vectoradd_float(
   float* __restrict__ a,
   const float* __restrict__ b,
   const float* __restrict__ c,
   int width, int height) {

   int x = hipBlockDim_x * hipBlockIdx_x +
           hipThreadIdx_x;
   int y = hipBlockDim_y * hipBlockIdx_y +
           hipThreadIdx_y;

   int i = y * width + x;
   if ( i < (width * height)) {
      a[i] = b[i] + c[i];
   }
}

send deviceA data
```

CPU Code

```
// verify the results
errors = 0;
for (i = 0; i < NUM; i++) {
   if (hostA[i] != (hostB[i] + hostC[i]))
{
      errors++;
   }
}
if (errors!=0) {
   printf("FAILED: %d errors\n",errors);
} else {
   printf ("PASSED!\n");
}

hipFree(deviceA);
hipFree(deviceB);
hipFree(deviceC);

free(hostA);
free(hostB);
free(hostC);
```



GPU Code

```
delete deviceA memory
delete deviceB memory
delete deviceC memory
```

5. Portable HIP Build System

Exploiting the Power of HIP: Portable Build Systems

- One of the attractive features of HIP is that it can run on both AMD and Nvidia GPUs
- The HIP language has been developed with this in mind
  - Select ROCm and it will run on AMD GPUs
  - Select CUDA and it will run on Nvidia GPUs
- But it can be difficult to support this with a portable build system that switches between these two

- We'll demonstrate two of the most common build systems that can support portable builds
  - make
  - cmake

Portable Build Systems -- Makefile

```
EXECUTABLE = ./vectoradd
all: $(EXECUTABLE) test
.PHONY: test

OBJECTS = vectoradd.o

CXXFLAGS = -g -O2 –DNDEBUG -fPIC
HIPCC_FLAGS = -O2 -g –DNDEBUG

HIP_PLATFORM ?= amd

ifeq ($(HIP_PLATFORM), nvidia)
  HIP_PATH ?= $(shell hipconfig --path)
  HIPCC_FLAGS += -x cu -I${HIP_PATH}/include/
endif
ifeq ($(HIP_PLATFORM), amd)
  HIPCC_FLAGS += -x hip -munsafe-fp-atomics
endif
```

Setting default device compiler

Setting compile flags for different GPUs

```
%.o: %.hip
        hipcc $(HIPCC_FLAGS) -c $^ -o $@

$(EXECUTABLE): $(OBJECTS)
        hipcc $< $(LDFLAGS) -o $@

test: $(EXECUTABLE)
        $(EXECUTABLE)

clean:
        rm -f $(EXECUTABLE)  $(OBJECTS) build
```

Pattern rule for HIP source

Using a portable Makefile

- For ROCm

```
module load rocm
module load cmake
export CXX=${ROCM_PATH}/llvm/bin/clang++
```

- To build and run

```
make vectoradd
./srun
```

- For CUDA

```
module load rocm
module load cuda
module load cmake
```

We still need HIP for the portability layer

- To build and run

```
HIP_PLATFORM=nvidia make vectoradd
./srun
```

Overriding default to compile with Nvidia

For Frontier

- For AMD programming environment

```
module load PrgEnv-amd
module load amd
module load cmake
export CXX=${ROCM_PATH}/llvm/bin/clang++
```

- To build and run

```
make vectoradd
srun ./vectoradd
```

- For Cray programming environment
  - `module load PrgEnv-cray`
  - `module load amd-mixed`
  - `module load cmake`

- To build and run
  - `CXX=CC CRAY_CPU_TARGET=x86-64 make vectoradd`
  - `srun ./vectoradd`

For Perlmutter

- For Perlmutter

```
module load PrgEnv-gnu/8.3.3
Module load hip/5.4.3
module load PrgEnv-nvidia/8.3.3
module load cmake
```

We still need HIP for the portability layer

Overriding default to compile with Nvidia

- To build and run

```
HIP_PLATFORM=nvidia make vectoradd
srun ./vectoradd
```

Portable Build Systems – CMakeLists.text

```
cmake_minimum_required(VERSION 3.21 FATAL_ERROR)
project(Vectoradd LANGUAGES CXX)

set (CMAKE_CXX_STANDARD 14)
if (NOT CMAKE_BUILD_TYPE)
   set(CMAKE_BUILD_TYPE RelWithDebInfo)
endif(NOT CMAKE_BUILD_TYPE)

string(REPLACE -O2 -O3 CMAKE_CXX_FLAGS_RELWITHDEBINFO ${CMAKE_CXX_FLAGS_RELWITHDEBINFO})

if (NOT CMAKE_GPU_RUNTIME)
   set(GPU_RUNTIME "ROCM" CACHE STRING "Switches between ROCM and CUDA")
else (NOT CMAKE_GPU_RUNTIME)
   set(GPU_RUNTIME "${CMAKE_GPU_RUNTIME}" CACHE STRING "Switches between ROCM and CUDA")
endif (NOT CMAKE_GPU_RUNTIME)
# Really should only be ROCM or CUDA, but allowing HIP because it is the currently built-in option
set(GPU_RUNTIMES "ROCM" "CUDA" "HIP")
if(NOT "${GPU_RUNTIME}" IN_LIST GPU_RUNTIMES)
    set(ERROR_MESSAGE "GPU_RUNTIME is set to \"${GPU_RUNTIME}\".\nGPU_RUNTIME must be either HIP, ROCM, or CUDA.")
    message(FATAL_ERROR ${ERROR_MESSAGE})endif()# GPU_RUNTIME for AMD GPUs should really be ROCM, if selecting AMD
GPUs
# so manually resetting to HIP if ROCM is selected
if (${GPU_RUNTIME} MATCHES "ROCM")
   set(GPU_RUNTIME "HIP")
endif (${GPU_RUNTIME} MATCHES "ROCM")
set_property(CACHE GPU_RUNTIME PROPERTY STRINGS ${GPU_RUNTIMES})
```

Setting GPU_RUNTIME

Defining GPU_RUNTIME will select ROCM or CUDA (e.g. -DGPU_RUNTIME=ROCM)

Portable Build Systems – CMakeLists.text

```
enable_language(${GPU_RUNTIME})
set(CMAKE_${GPU_RUNTIME}_EXTENSIONS OFF)
set(CMAKE_${GPU_RUNTIME}_STANDARD_REQUIRED ON)

set(VECTORADD_CXX_SRCS "")
set(VECTORADD_HIP_SRCS vectoradd.hip)

add_executable(vectoradd ${VECTORADD_CXX_SRCS} ${VECTORADD_HIP_SRCS} )

set(ROCMCC_FLAGS "${ROCMCC_FLAGS} -munsafe-fp-atomics")
set(CUDACC_FLAGS "${CUDACC_FLAGS} ")

if (${GPU_RUNTIME} MATCHES "HIP")
   set(HIPCC_FLAGS "${ROCMCC_FLAGS}")
else (${GPU_RUNTIME} MATCHES "HIP")
   set(HIPCC_FLAGS "${CUDACC_FLAGS}")
endif (${GPU_RUNTIME} MATCHES "HIP")

set_source_files_properties(${VECTORADD_HIP_SRCS} PROPERTIES LANGUAGE ${GPU_RUNTIME})
set_source_files_properties(vectoradd.hip PROPERTIES COMPILE_FLAGS ${HIPCC_FLAGS})

install(TARGETS vectoradd)
```

Enabling either CUDA or HIP(ROCM)

Setting different flags for each GPU type

Setting language type for HIP source files

Setting device compile flags

Using a portable CMakeLists.txt

- For ROCm

```
module load rocm
module load cmake
export CXX=${ROCM_PATH}/llvm/bin/clang++
```

- To Build

```
mkdir build && cd build
cmake ..
make VERBOSE=1
./vectoradd
```

- For CUDA

```
module load rocm
module load cuda
module load cmake
```

- To Build

```
mkdir build && cd build
cmake –DCMAKE_GPU_RUNTIME=CUDA ..
make VERBOSE=1
./vectoradd
```

Overrides default GPU runtime to specify CUDA

Frontier and Perlmutter

- For Frontier

```
module load rocm
module load cmake
export CXX=${ROCM_PATH}/llvm/bin/clang++
```

- To build and run

```
mkdir build && cd build
cmake ..
make VERBOSE=1
./vectoradd
```

- For Perlmutter

```
module load PrgEnv-gnu/8.3.3
Module load hip/5.4.3
module load PrgEnv-nvidia/8.3.3
module load cmake
```

- To build and run

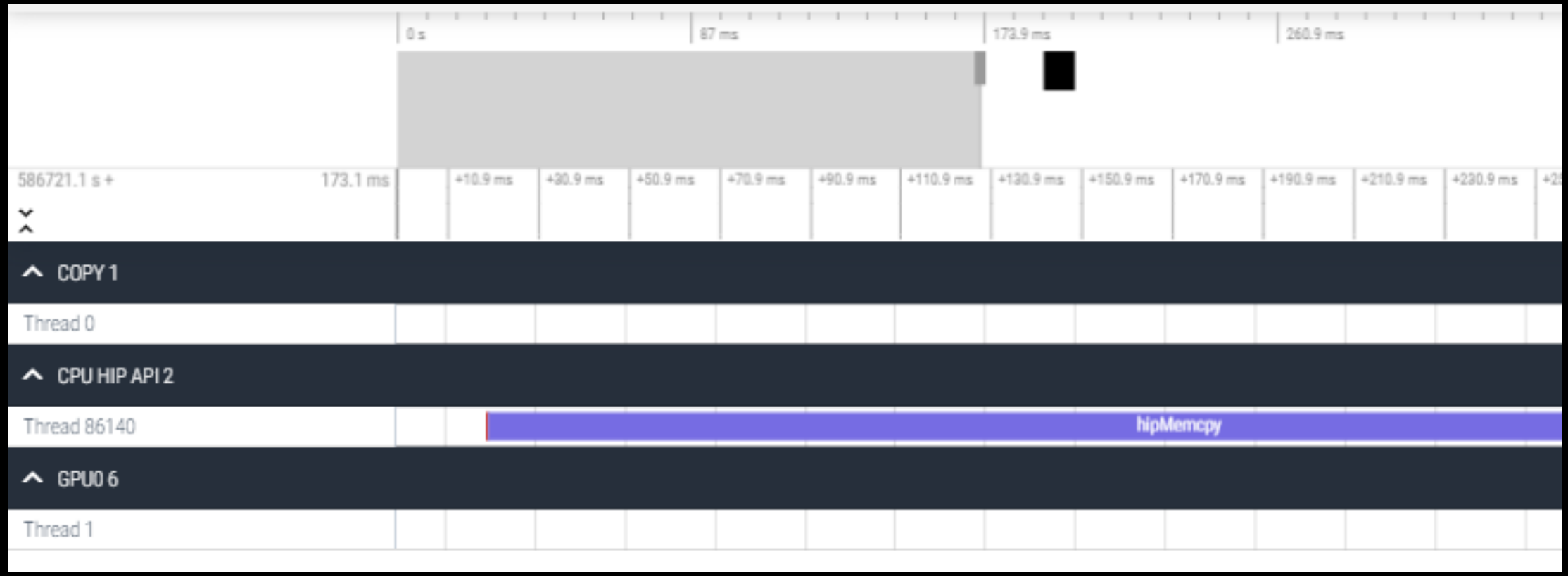
```
mkdir build && cd build
cmake –DCMAKE_GPU_RUNTIME=CUDA ..
make VERBOSE=1
./vectoradd
```

6. Profiling HIP application

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,       0.034238347058073 32
"hipGetDeviceProperties",      1,         2600,            2600,       0.0004388233380212493
"__hipPushCallConfiguration",  1,         1860,            1860,       0.0003139274648921245
"__hipPopCallConfiguration",   1,          450,             450,       7.595019311906238e-05
```

Perfetto - visualization



7. Device management and asynchronous computing

Device Management

Multiple GPUs in system? Multiple host threads/MPI ranks? What device are we running on?

- Host can query number of devices visible to system:

```
int numDevices = 0;
hipGetDeviceCount(&numDevices);
```

- Host tells the runtime to issue instructions to a particular device:

```
int deviceID = 0;
hipSetDevice(deviceID);
```

- Host can query what device is currently selected:

```
hipGetDevice(&deviceID);
```

- The host can manage several devices by swapping the currently selected device during runtime.
- Different processes can use different devices or over-subscribe (share) the same device.

Device Properties

The host can also query a device's properties:

```
hipDeviceProp_t props;
hipGetDeviceProperties(&props, deviceID);
```

- `hipDeviceProp_t` is a struct that contains useful fields like the device's name, total VRAM, clock speed, and GCN architecture.
  - See "`hip/hip_runtime_api.h`" for full list of fields.

Blocking vs Nonblocking API functions

- Launching a kernel is **non-blocking**
  - After sending instructions/data, the host continues to do more work while the device executes the kernel
  - Multiple kernels launched on different streams can run concurrently on the same device
- However, `hipMemcpy` is **blocking**
  - The data pointed to in the arguments can be accessed/modified after the function returns

- To make asynchronous copies, we need to allocate non-pageable host memory using hipHostMalloc and copy using hipMemcpyAsync

```
hipHostMalloc(h_a, Nbytes, hipHostMallocDefault);
hipMemcpyAsync(d_a, h_a, Nbytes, hipMemcpyHostToDevice, stream);
```

- It is not safe to access/modify the arguments of `hipMemcpyAsync` without some sort of synchronization.

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

  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));   //copy results back to host
  …
  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;
  }
}
```

The host waits for the kernel to finish here

Streams

- A stream in HIP is a queue of tasks (e.g. kernels, memcpys, events).
  - Tasks enqueued in a stream **complete in order on that stream**.
  - 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);
```

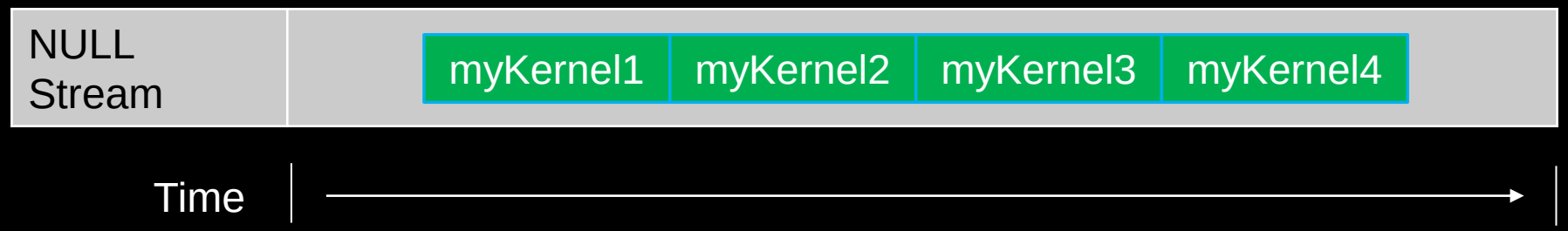
- Passing `0` or `NULL` as the `hipStream_t` argument to a function instructs the function to execute on a stream called the ‘NULL Stream’:
  - No task on the NULL stream will begin until **all previously enqueued tasks in all other streams have completed**.
  - Blocking calls like `hipMemcpy` run on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

Streams

- Suppose we have 4 small kernels to execute:

```
myKernel1<<<dim3(1), dim3(256), 0, 0>>>(256, d_a1);
myKernel2<<<dim3(1), dim3(256), 0, 0>>>(256, d_a2);
myKernel3<<<dim3(1), dim3(256), 0, 0>>>(256, d_a3);
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:

```
myKernel1<<<dim3(1), dim3(256), 0, stream1>>>(256, d_a1);
myKernel2<<<dim3(1), dim3(256), 0, stream2>>>(256, d_a2);
myKernel3<<<dim3(1), dim3(256), 0, stream3>>>(256, d_a3);
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.

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.**

Pinned Memory

Host data allocations (malloc, new) 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 effective data transfer rate **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.

Streams

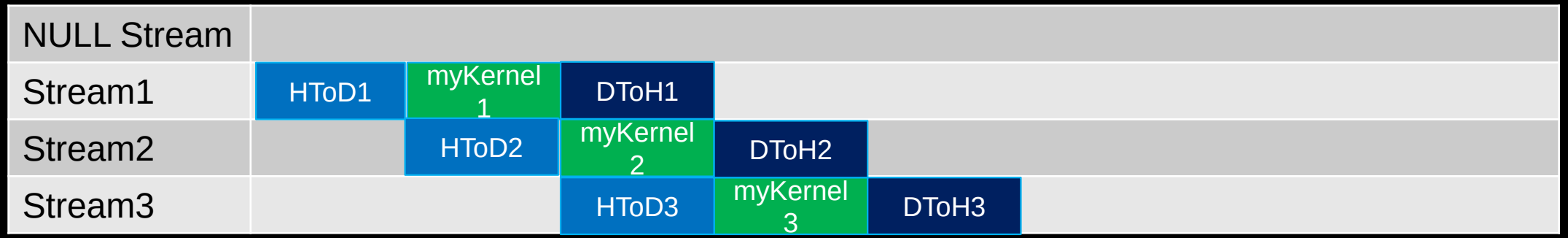
Suppose we have 3 kernels which require moving data to and from the device:

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

myKernel1<<<blocks, threads, 0, 0>>>(N, d_a1);
myKernel2<<<blocks, threads, 0, 0>>>(N, d_a2);
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);
```

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);

myKernel1<<<blocks, threads, 0, stream1>>>(N, d_a1);
myKernel2<<<blocks, threads, 0, stream2>>>(N, d_a2);
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);
```



Synchronization

How do we coordinate execution on device streams with host execution? Need some synchronization points.

- `hipDeviceSynchronize();`
  - Heavy-duty sync point.
  - Blocks host until **all work** in **all device streams** has reported complete.

- `hipStreamSynchronize(stream);`
  - Blocks host until all work in stream has reported complete.

Can a stream synchronize with another stream? For that we need ‘Events’.

Events

A `hipEvent_t` object is created on a device via:

```
hipEvent_t event;
hipEventCreate(&event);
```

We queue an event into a stream:

```
hipEventRecord(event, stream);
```

- The event records what work is currently enqueued in the stream.
- When the stream's execution reaches the event, the event is considered 'complete'.

At the end of the application, event objects should be destroyed:

```
hipEventDestroy(event);
```

Events

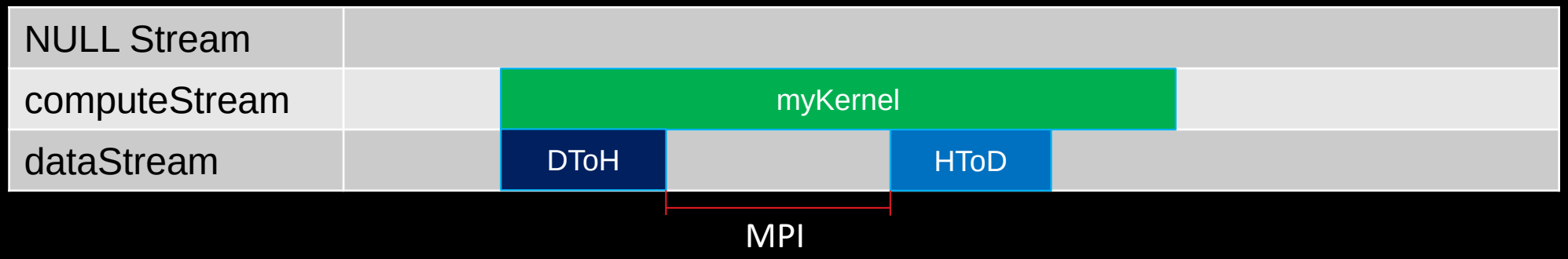
What can we do with queued events?

- `hipEventSynchronize(event);`
  - Block host until event reports complete.
  - Only a synchronization point with respect to the stream where event was enqueued.
- `hipEventElapsedTime(&time, startEvent, endEvent);`
  - Returns the time in ms between when two events, startEvent and endEvent, completed
  - Can be very useful for timing kernels/memcpys
- `hipStreamWaitEvent(stream, event);`
  - Non-blocking for host.
  - Instructs all future work submitted to stream to wait until event reports complete.
  - Primary way we enforce an ‘ordering’ between tasks in separate streams.

Streams

A common use-case for streams is MPI traffic:

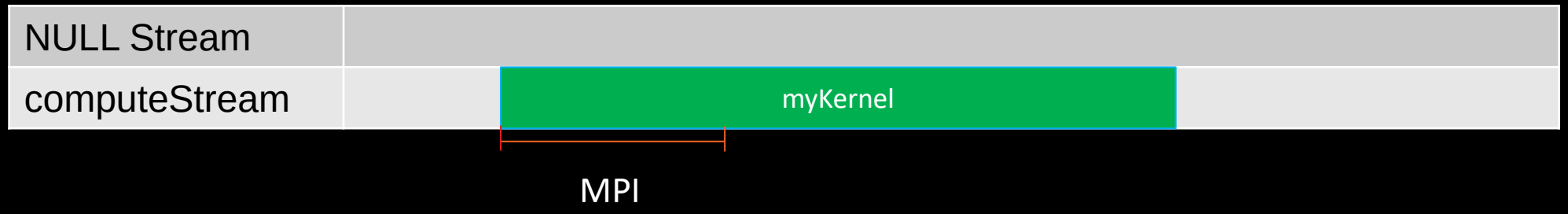
```
//Queue local compute kernel
myKernel<<<blocks, threads, 0, computeStream>>>(N, d_a);

//Copy halo data to host
hipMemcpyAsync(h_commBuffer, d_commBuffer, Nbytes, hipMemcpyDeviceToHost, dataStream);
hipStreamSynchronize(dataStream); //Wait for data to arrive

//Exchange data with MPI
MPI_Data_Exchange(h_commBuffer);

//Send new data back to device
hipMemcpyAsync(d_commBuffer, h_commBuffer, Nbytes, hipMemcpyHostToDevice, dataStream);
```



Streams

With a GPU-aware MPI stack, the Host<->Device traffic can be omitted:

```
//Some synchronization so that data on GPU and local compute are ready
hipDeviceSynchronize();

//Exchange data with MPI (with device pointer)
MPI_Data_Exchange(d_commBuffer, &request);

//Queue local compute kernel
myKernel<<<blocks, threads, 0, computeStream>>>(N, d_a);

 //Wait for MPI request to complete
MPI_Wait(&request, &status);
```



8. Device code, shared memory, and thread synchronization

Function Qualifiers

hipcc makes two compilation passes through source code. One to compile host code, and one to compile device code.

- __global__ functions:
  - These are entry points to device code, called from the host
  - Code in these regions will execute on SIMD units

- __device__ functions:
  - Can be called from __global__ and other __device__ functions.
  - Cannot be called from host code.
  - Not compiled into host code – essentially ignored during host compilation pass

- __host__ __device__ functions:
  - Can be called from __global__, __device__, and host functions.
  - Will execute on SIMD units when called from device code!

Single Instruction Multiple Data (SIMD) Execution

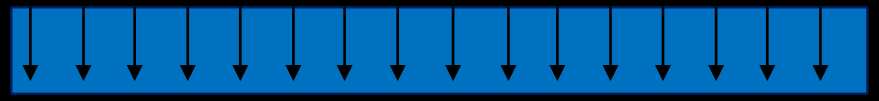
On SIMD units, be aware of divergence.

- Branching logic (if – else) can be costly:
  - Wavefront encounters an if statement
  - Evaluates conditional
    - If true, continues to statement body
    - If false, **also continues to statement body** with all instructions replaced with NoOps.
  - Known as ‘thread divergence’

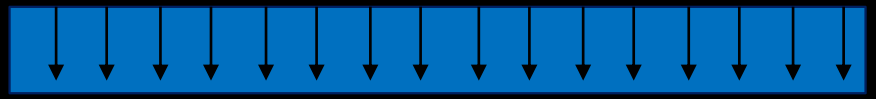
- Generally, wavefronts diverging from each other is okay.
- Thread divergence within a wavefront can impact performance.
  - Ok if divergence is short

SIMD Execution

```
if (threadIdx.x % 2) {
  a *= 2.0;
} else {
  a *= 3.14;
}
```



```
//if (threadIdx.x % 2) {
  NoOp;
//} else {
  a *= 3.14;
//}
```



```
//if (threadIdx.x % 2) {
  a *= 2.0;
//} else {
  NoOp;
//}
```

Memory declarations in Device Code

- Malloc/free not supported in device code.
- Variables/arrays can be declared on the stack.
- Stack variables declared in device code are allocated in registers and are private to each thread.
- Threads can all access common memory via device pointers, but otherwise do not share memory.
  - Important exception: __shared__ memory
- Stack variables declared as __shared__:
  - Allocated once per block in LDS memory
  - Shared and accessible by all threads in the same block
  - Access is faster than device global memory (but slower than register)
  - Must have size known at compile time

Shared Memory

```
__global__ void reverse(double *d_a) {
  __shared__ double s_a[256]; //array of doubles, shared in this block

  int tid = threadIdx.x;
  s_a[tid] = d_a[tid];    //each thread fills one entry

  //all wavefronts must reach this point before any wavefront is allowed to continue.
                          //something is missing here…

__syncthreads();

  d_a[tid] = s_a[255-tid]; //write out array in reverse order
}

int main() {
  …
  reverse<<<dim3(1), dim3(256), 0, 0>>>(d_a); //Launch kernel
  …
}
```

Thread Synchronization

- __syncthreads():
  - Blocks a wavefront from continuing execution until all wavefronts have reached __syncthreads()
  - Memory transactions made by a thread before __syncthreads() are visible to all other threads in the block after __syncthreads()
  - Can have a noticeable overhead if called repeatedly

- **Best practice:** Avoid deadlocks by checking that **all** threads in a block execute **the same** __syncthreads() instruction.

- *Note 1*: So long as at least one thread in the wavefront encounters __syncthreads(), the whole wavefront is considered to have encountered __syncthreads().

- *Note 2*: Wavefronts can synchronize at different __syncthreads() instructions, and if a wavefront exits a kernel completely, other wavefronts waiting at a __syncthreads() may be allowed to continue.

9. GPU Software

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
- -Rpass-analysis=kernel-resource-usage will report kernel resource usage numbers
- With the command *hipconfig* you can see many information about environment variables declaration

Inspecting the AMD GCN ISA

- You can inspect the AMD CDNA assembly that was emitted by the compiler by using compiler options "-g –ggdb --save-temps"
- This outputs files into the current directory and the assembly can be found in:
  - `vectoradd_hip-hip-amdgcn-amd-amdhsa-gfx908.s`
- Also found are the compile-time estimates of #SGPRs, #VGPRs, ScratchSize (Spills), Occupancy
- If kernel is templated, then assembly is generated for the various instantiations of the kernel
- -g –ggdb flags help annotate assembly with source code line numbers
- The CDNA and CDNA2 ISA guides are publicly available:
  - https://developer.amd.com/wp-content/resources/CDNA1_Shader_ISA_14December2020.pdf
  - https://developer.amd.com/wp-content/resources/CDNA2_Shader_ISA_18November2021.pdf

```
$ hipcc -O3 -g -ggdb --save-temps vectoradd_hip.cpp
$ grep v_add vectoradd_hip-hip-amdgcn-amd-amdhsa-gfx908.s
    v_add_u32_e32 v1, s9, v1
    v_add3_u32 v0, s8, v0, v1
    v_add_co_u32_e32 v2, vcc, s0, v0
    v_addc_co_u32_e32 v3, vcc, v3, v1, vcc
    v_add_co_u32_e32 v4, vcc, s4, v0
    v_addc_co_u32_e32 v5, vcc, v5, v1, vcc
    v_add_co_u32_e32 v0, vcc, s2, v0
    v_addc_co_u32_e32 v1, vcc, v6, v1, vcc
    v_add_f32_e32 v0, v6, v7                                    // 0000000011CC: 02040702
```

Querying System

- rocminfo: Queries and displays information on the system's hardware
  - More info at: https://github.com/RadeonOpenCompute/rocminfo
- Querying ROCm version:
  - If you install ROCm in the standard location (/opt/rocm) version info is at: /opt/rocm/.info/version-dev
  - Can also run the command 'dkms status' and the ROCm version will be displayed
- rocm-smi: Queries and sets AMD GPU frequencies, power usage, and fan speeds
  - sudo privileges are needed to set frequencies and power limits
  - sudo privileges are not needed to query information
  - Get more info by running 'rocm-smi -h' or looking at: https://github.com/RadeonOpenCompute/ROC-smi

```
$ /opt/rocm/bin/rocm-smi
========================ROCm System Management Interface========================
================================================================================
GPU  Temp   AvgPwr  SCLK     MCLK    Fan   Perf    PwrCap  VRAM%  GPU%
1    38.0c  18.0W   1440Mhz  945Mhz  0.0%  manual  220.0W    0%   0%
================================================================================
==============================End of ROCm SMI Log ===============================
```

ROCm GPU Libraries

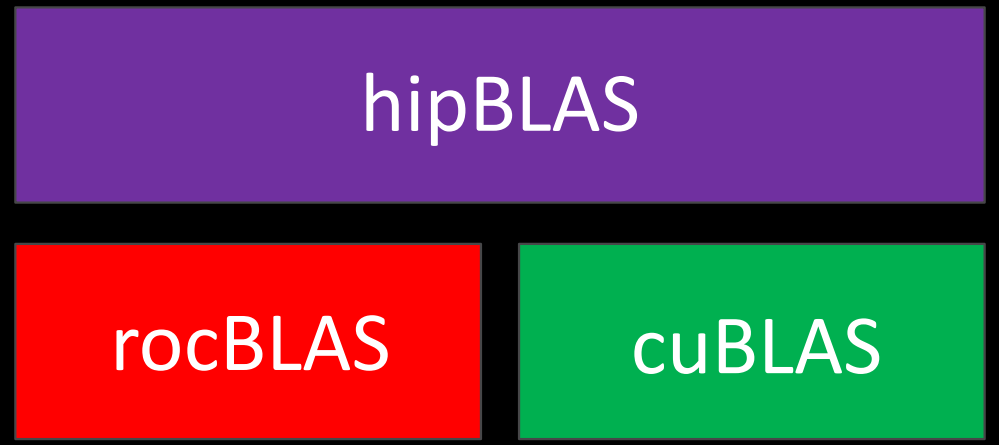
ROCm provides several GPU math libraries

- 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 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

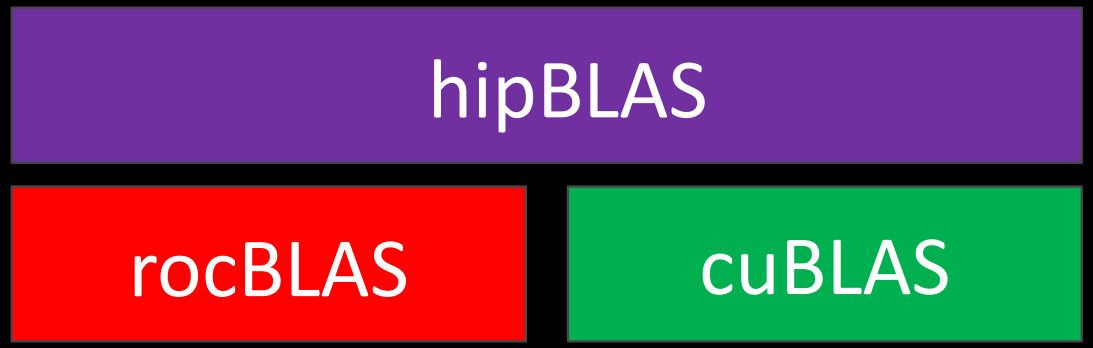
AMD Math Library Equivalents: “Decoder Ring”

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

AMD GPU Libraries: BLAS

- rocBLAS – `sudo apt install rocblas`
  - Source code: https://github.com/ROCmSoftwarePlatform/rocBLAS
  - Documentation: https://rocblas.readthedocs.io/en/latest/
  - Basic linear algebra functionality
    - axpy, gemv, trsm, etc
  - Use hipBLAS if you need portability between AMD and NVIDIA devices
- hipBLAS - `sudo apt install hipblas`
  - Documentation: https://github.com/ROCmSoftwarePlatform/hipBLAS/wiki/Exported-functions
  - Use this if you need portability between AMD and NVIDIA
  - It is just a thin wrapper:
    - It can dispatch calls to rocBLAS for AMD devices
    - It can dispatch calls to cuBLAS for NVIDIA devices



AMD GPU Libraries: rocBLAS example

- rocBLAS
  - Documentation: https://rocblas.readthedocs.io/en/latest/
  - Level 1, 2, and 3 functionality
    - axpy, gemv, trsm, etc
  - Note: rocBLAS syntax matches BLAS closer than hipBLAS or cuBLAS
    - Use hipBLAS only if you need portability between AMD and NVIDIA devices
  - Link with: `-lrocblas`

```
#include <rocblas.h>

int main(int argc, char ** argv) {
  rocblas_int N = 500000;

  // Allocate device memory
  double * dx, * dy;
  hipMalloc(&dx, sizeof(double) * N);
  hipMalloc(&dy, sizeof(double) * N);

  // Allocate host memory (and fill up the arrays) here
  std::vector<double> hx(N), hy(N);

  // Copy host arrays to device
  hipMemcpy(dx, hx.data(), sizeof(double) * N, hipMemcpyHostToDevice);
  hipMemcpy(dy, hy.data(), sizeof(double) * N, hipMemcpyHostToDevice);

  const double alpha = 1.0;
  rocblas_handle handle;
  rocblas_create_handle(&handle);
  rocblas_status status;
  status = rocblas_daxpy(handle, N, &alpha, dx, 1, dy, 1);
  rocblas_destroy_handle(handle);

  // Copy result back to host
  hipMemcpy(hy.data(), dy, sizeof(double) * N, hipMemcpyDeviceToHost);
  hipFree(dx);
  hipFree(dy);
  return 0;
}
```

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

10. Shared Memory, Atomics

Dynamic Shared Memory

- Can actually use __shared__ arrays when sizes aren’t known at compile time
  - Called dynamic shared memory
  - Declare one array using HIP_DYNAMIC_SHARED macro, use for all dynamic LDS space
  - Use during the kernel call, we haven’t discussed yet

Dynamic Shared Memory

```
__global__ void reverse(double *d_a, int N) {
  HIP_DYNAMIC_SHARED(double, s_a); //dynamic array of doubles, shared in this block

  int tid = threadIdx.x;
  s_a[tid] = d_a[tid];    //each thread fills one entry

  //all wavefronts should reach this point before any wavefront is allowed to continue.
  __syncthreads();

  d_a[tid] = s_a[N-1-tid]; //write out array in reverse order
}

int main() {
  …
  size_t NsharedBytes = N*sizeof(double);
  reverse<<<dim3(1), dim3(N), NsharedBytes, 0>>>(d_a, N); //Launch kernel
  …
}
```

Atomic Operations

Atomic functions:

- Perform a read+write of a single 32 or 64-bit word in device global or LDS memory
- Can be called by multiple threads in device code
- Performed in a conflict-free manner

- AMD GPUs support atomic operations on 32-bit integers in hardware
  - Float /double atomics implemented as atomicCAS (Compare And Swap) loops, may have poor performance

- Can check at compile time if 32 or 64-bit atomic instructions are supported on target device
  - `#ifdef __HIP_ARCH_HAS_GLOBAL_INT32_ATOMICS__`
  - `#ifdef __HIP_ARCH_HAS_GLOBAL_INT64_ATOMICS__`

Atomic Operations

Supported atomic operations in HIP:

Operation	Type, T	Notes
`T atomicAdd(T* address, T val)`	int, long long int, float, double	Adds val to *address
`T atomicExch(T* address, T val)`	int, long long int, float	Replace *address with val and return old value
`T atomicMin(T* address, T val)`	int, long long int	Replaces *address if val is smaller
`T atomicMax(T* address, T val)`	int, long long int	Replaces *address if val is larger
`T atomicAnd(T* address, T val)`	int, long long int	Bitwise AND between *address and val
`T atomicOr(T* address, T val)`	int, long long int	Bitwise OR between *address and val
`T atomicXor(T* address, T val)`	int, long long int	Bitwise XOR between *address and val

AMD GPU programming resources

- ROCm platform: https://github.com/RadeonOpenCompute/ROCm/
  - With instructions for installing from Debian/CentOS/RHEL binary repositories
  - Has links to source repositories for all components, including HIP
- HIP porting guide: https://github.com/ROCm-Developer-Tools/HIP/blob/master/docs/markdown/hip_porting_guide.md
- ROCm/HIP libraries: https://github.com/ROCmSoftwarePlatform
- ROC-profiler: https://github.com/ROCm-Developer-Tools/rocprofiler
  - Collects application traces and performance counters
  - Trace timeline can be visualized with https://ui.perfetto.dev/
- AMD GPU ISA docs and more: https://developer.amd.com/resources/developer-guides-manuals/

Summary

- HIP is an extensive API that covers a lot of GPU programming requirements
- It is under continuous development, and it is open-source
- It can be executed on AMD and NVIDIA GPUs
- We have profiling tools that we can identify bottlenecks
- It is quite easy to use especially with GPU programming knowledge

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.

Questions?


AMD