HIP and ROCm

Presenter: Sam Antao
LUMI Advanced Training
Oct. 23rd, 2025

AMD
together we advance_

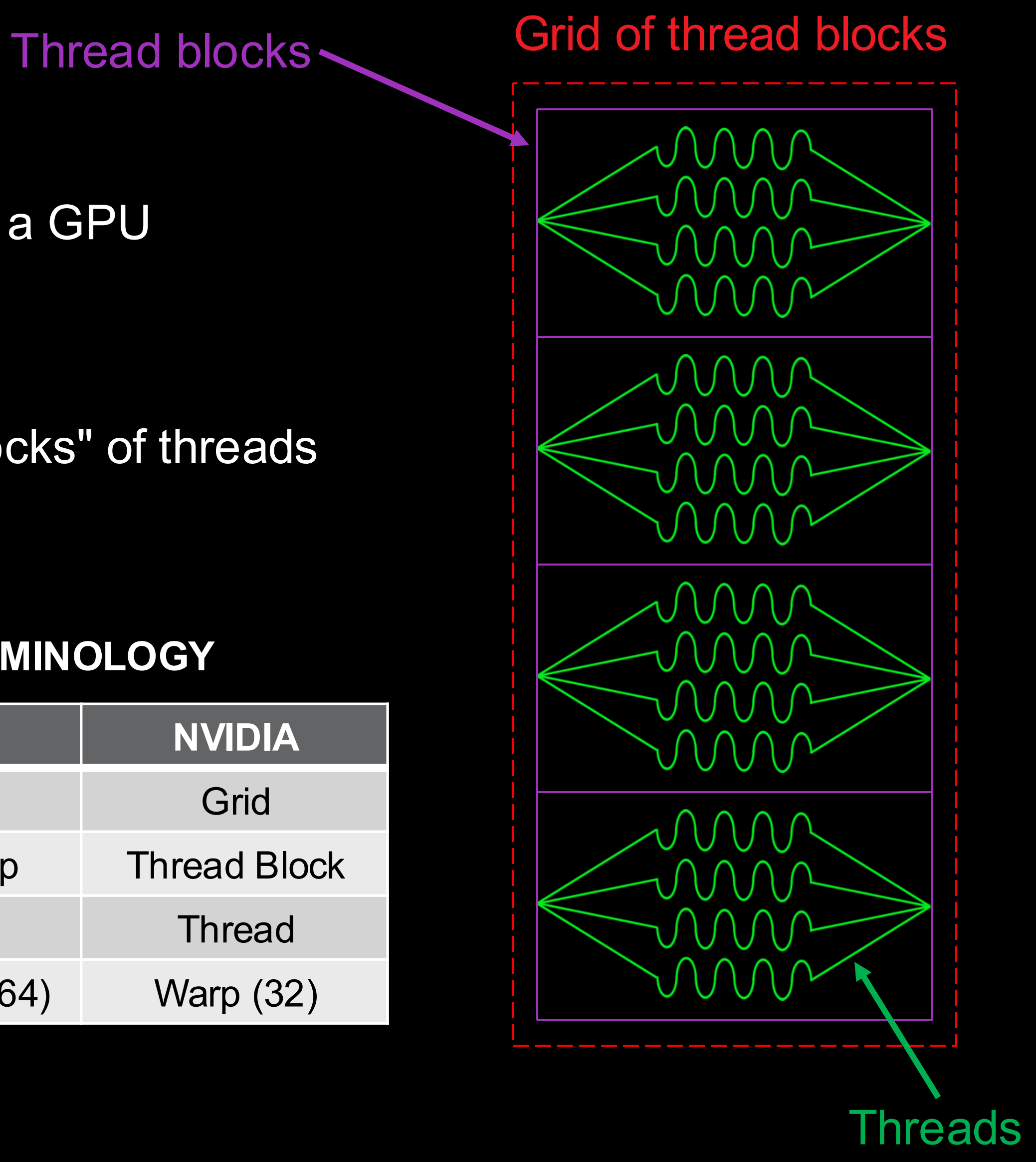
1. AMD GPU programming concepts

Device Kernels: Grid Hierarchy

- In HIP, kernels are executed on a "grid" of threads that run on a GPU
    - 1D, 2D, and 3D grids are supported, but most work maps well to 1D
    - The grid is what you map your problem to
- Each dimension of the grid is partitioned into **equal sized** "blocks" of threads
- 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

TERMINOLOGY

AMD	NVIDIA
Grid	Grid
Workgroup	Thread Block
Thread	Thread
Wavefront (64)	Warp (32)

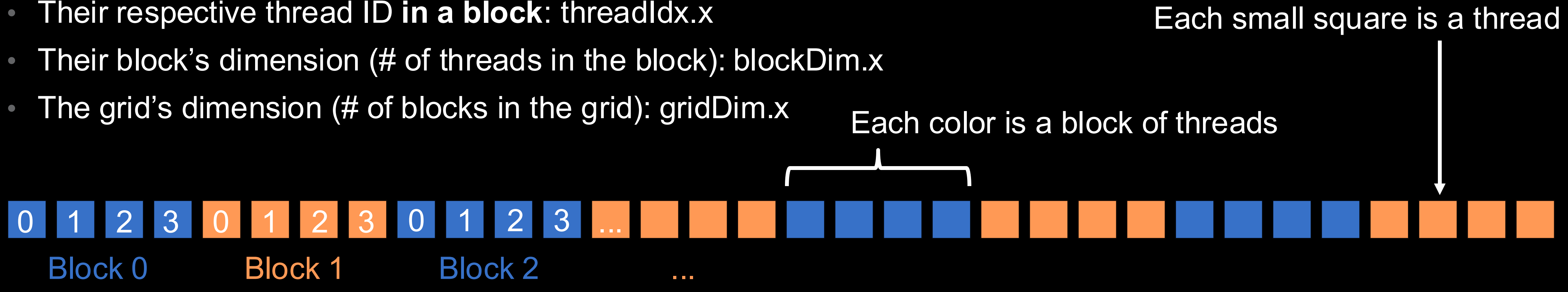


AMD
together we advance_

The Grid: blocks of threads in 1D

Threads in grid have access to:

- Their respective block (workgroup): blockIdx.x
- Their respective thread ID **in a block**: threadIdx.x
- Their block's dimension (# of threads in the block): blockDim.x
- The grid's dimension (# of blocks in the grid): gridDim.x



Global thread ID

```
int id = blockDim.x * blockIdx.x + threadIdx.x;
```

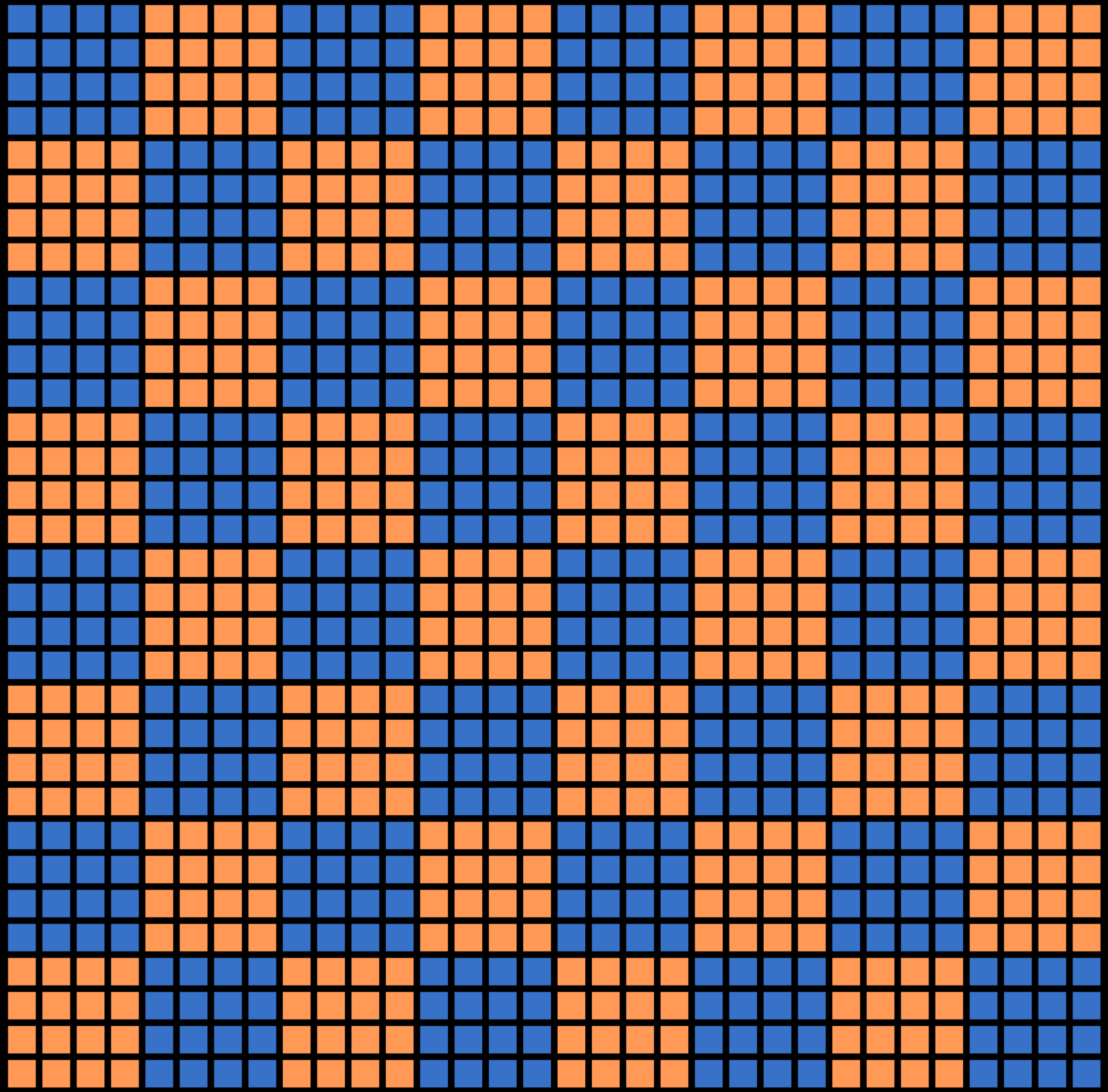
For example, thread 3 of block 2
would have a global thread ID of 11

```
       = 4         * 2          + 3
       = 11
```


AMD together we advance_

The Grid: blocks of threads in 2D

- 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:

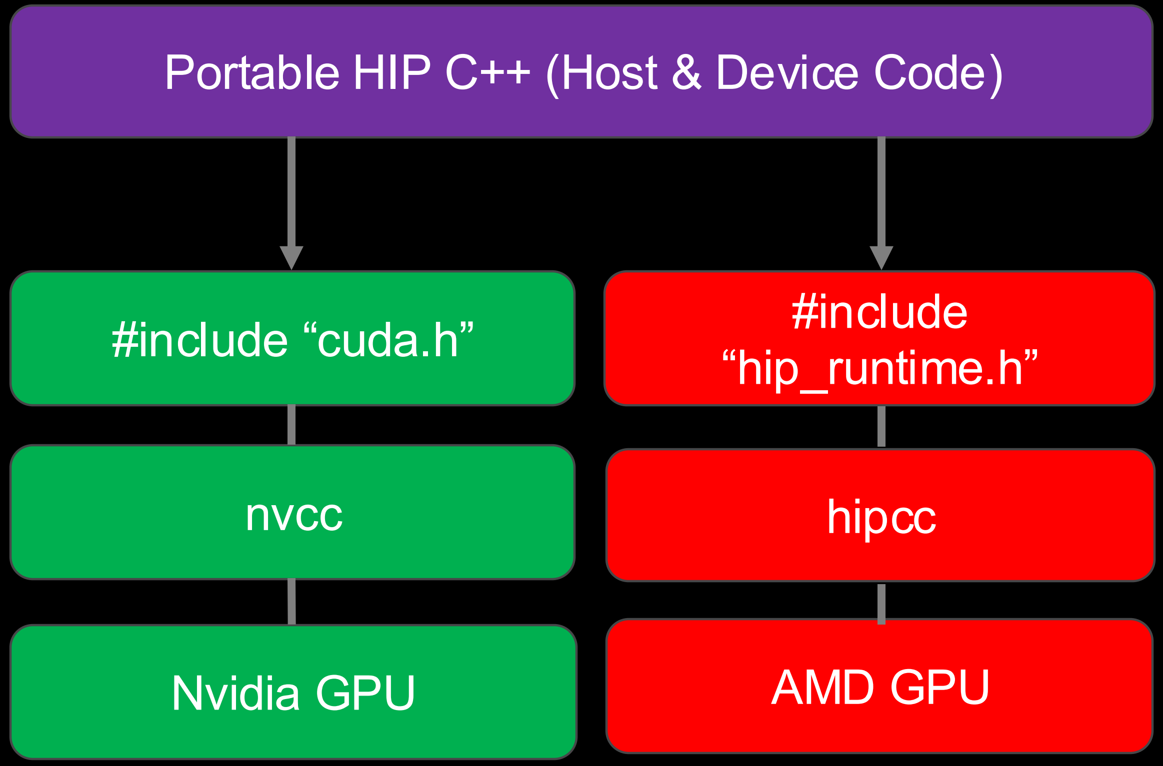
- Their respective block IDs: blockIdx.x, blockIdx.y
- Their respective thread IDs in a block: threadIdx.x, threadIdx.y
- Etc.

2. HIP API calls and GPU kernel code

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

- **Open-source**
- Syntactically similar to CUDA. Most CUDA API calls can be converted in place: cuda -> hip
- Supports a strong subset of CUDA runtime functionality



https://rocm.docs.amd.com/projects/HIP/en/latest/.doxygen/docBin/html/index.html

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__`

Device code
- `threadIdx`, `blockIdx`, `blockDim`, `__shared__`, 200+ math functions covering entire CUDA math library.

Error handling
- `hipGetLastError()`, `hipGetErrorString()`

Example: simple discrete GPU multiply

```
#include <stdio.h>
#include <math.h>
#include "hip/hip_runtime.h"

__global__ void multiply(double *A, int n)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < n) A[id] = 2.0 * A[id];
}

int main(int argc, char *argv[]){

  int N = 1024 * 1024;
  size_t bytes = N * sizeof(double);

  double *h_A = (double*)malloc(bytes);

  for(int i=0; i<N; i++){
    h_A[i] = (double)rand()/(double)RAND_MAX;
  }
```

```
  double *d_A;
  hipMalloc(&d_A, bytes);

  hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);

  int thr_per_blk = 256;
  int blk_in_grid = ceil( float(N) / thr_per_blk );

  multiply<<<blk_in_grid,thr_per_blk>>>(d_A, N);

  hipMemcpy(h_A, d_A, bytes, hipMemcpyDeviceToHost);

  free(h_A);
  hipFree(d_A);

  printf("__SUCCESS__\n");

  return 0;
}
```


AMD together we advance_

Example: simple discrete GPU multiply

Include header for HIP runtime

```
#include <stdio.h>
#include <math.h>
#include "hip/hip_runtime.h"

__global__ void multiply(double *A, int n)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < n) A[id] = 2.0 * A[id];
}

int main(int argc, char *argv[]){

  int N = 1024 * 1024;
  size_t bytes = N * sizeof(double);

  double *h_A = (double*)malloc(bytes);

  for(int i=0; i<N; i++){
    h_A[i] = (double)rand()/(double)RAND_MAX;
  }
```

```
  double *d_A;
  hipMalloc(&d_A, bytes);

  hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);

  int thr_per_blk = 256;
  int blk_in_grid = ceil( float(N) / thr_per_blk );

  multiply<<<blk_in_grid,thr_per_blk>>>(d_A, N);

  hipMemcpy(h_A, d_A, bytes, hipMemcpyDeviceToHost);

  free(h_A);
  hipFree(d_A);

  printf("__SUCCESS__\n");

  return 0;
}
```

Example: simple discrete GPU multiply

```
#include <stdio.h>
#include <math.h>
#include "hip/hip_runtime.h"
```

GPU kernel

```
__global__ void multiply(double *A, int n)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < n) A[id] = 2.0 * A[id];
}

int main(int argc, char *argv[]){

  int N = 1024 * 1024;
  size_t bytes = N * sizeof(double);

  double *h_A = (double*)malloc(bytes);

  for(int i=0; i<N; i++){
    h_A[i] = (double)rand()/(double)RAND_MAX;
  }
```

```
  double *d_A;
  hipMalloc(&d_A, bytes);

  hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);

  int thr_per_blk = 256;
  int blk_in_grid = ceil( float(N) / thr_per_blk );

  multiply<<<blk_in_grid,thr_per_blk>>>(d_A, N);

  hipMemcpy(h_A, d_A, bytes, hipMemcpyDeviceToHost);

  free(h_A);
  hipFree(d_A);

  printf("__SUCCESS__\n");

  return 0;
}
```

[Public]

Example: simple discrete GPU multiply

```
#include <stdio.h>
#include <math.h>
#include "hip/hip_runtime.h"

__global__ void multiply(double *A, int n)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < n) A[id] = 2.0 * A[id];
}
```

Allocate and initialize host memory buffer

```
int main(int argc, char *argv[]){

  int N = 1024 * 1024;
  size_t bytes = N * sizeof(double);

  double *h_A = (double*)malloc(bytes);

  for(int i=0; i<N; i++){
    h_A[i] = (double)rand()/(double)RAND_MAX;
  }
```

```
  double *d_A;
  hipMalloc(&d_A, bytes);

  hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);

  int thr_per_blk = 256;
  int blk_in_grid = ceil( float(N) / thr_per_blk );

  multiply<<<blk_in_grid,thr_per_blk>>>(d_A, N);

  hipMemcpy(h_A, d_A, bytes, hipMemcpyDeviceToHost);

  free(h_A);
  hipFree(d_A);

  printf("__SUCCESS__\n");

  return 0;
}
```


AMD together we advance_

Example: simple discrete GPU multiply

Allocate GPU buffer and copy values from CPU buffer to GPU buffer

```
#include <stdio.h>
#include <math.h>
#include "hip/hip_runtime.h"

__global__ void multiply(double *A, int n)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < n) A[id] = 2.0 * A[id];
}

int main(int argc, char *argv[]){

  int N = 1024 * 1024;
  size_t bytes = N * sizeof(double);

  double *h_A = (double*)malloc(bytes);

  for(int i=0; i<N; i++){
    h_A[i] = (double)rand()/(double)RAND_MAX;
  }
```

```
  double *d_A;
  hipMalloc(&d_A, bytes);

  hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);
```

Not needed for unified memory

```
  int thr_per_blk = 256;
  int blk_in_grid = ceil( float(N) / thr_per_blk );

  multiply<<<blk_in_grid,thr_per_blk>>>(d_A, N);

  hipMemcpy(h_A, d_A, bytes, hipMemcpyDeviceToHost);

  free(h_A);
  hipFree(d_A);

  printf("__SUCCESS__\n");

  return 0;
}
```

Example: simple discrete GPU multiply

```
#include <stdio.h>
#include <math.h>
#include "hip/hip_runtime.h"

__global__ void multiply(double *A, int n)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < n) A[id] = 2.0 * A[id];
}

int main(int argc, char *argv[]){

  int N = 1024 * 1024;
  size_t bytes = N * sizeof(double);

  double *h_A = (double*)malloc(bytes);

  for(int i=0; i<N; i++){
    h_A[i] = (double)rand()/(double)RAND_MAX;
  }
```

```
  double *d_A;
  hipMalloc(&d_A, bytes);

  hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);

  int thr_per_blk = 256;
  int blk_in_grid = ceil( float(N) / thr_per_blk );

  multiply<<<blk_in_grid,thr_per_blk>>>(d_A, N);

  hipMemcpy(h_A, d_A, bytes, hipMemcpyDeviceToHost);

  free(h_A);
  hipFree(d_A);

  printf("__SUCCESS__\n");

  return 0;
}
```

Launch GPU kernel

Example: simple discrete GPU multiply

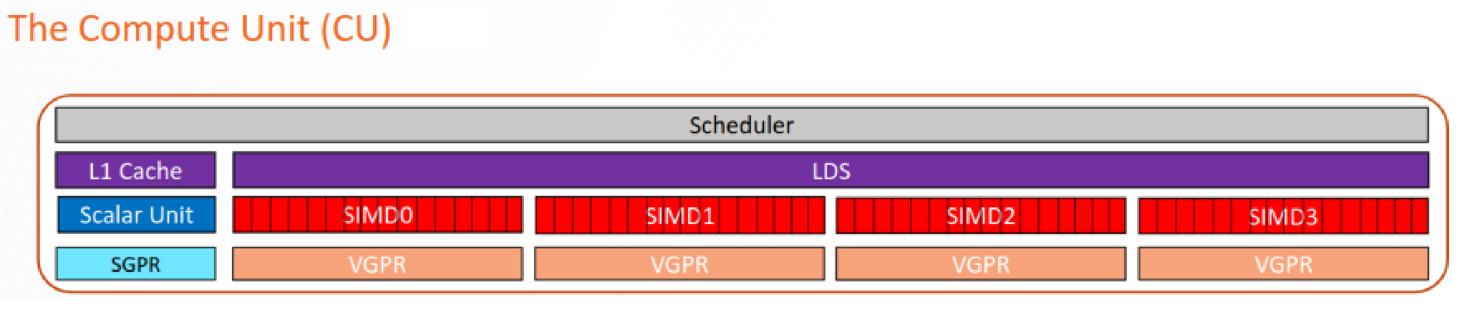
```
#include <stdio.h>
#include <math.h>
#include "hip/hip_runtime.h"

__global__ void multiply(double *A, int n)
{
  int id = blockDim.x * blockIdx.x + threadIdx.x;
  if (id < n) A[id] = 2.0 * A[id];
}

int main(int argc, char *argv[]){

  int N = 1024 * 1024;
  size_t bytes = N * sizeof(double);

  double *h_A = (double*)malloc(bytes);

  for(int i=0; i<N; i++){
    h_A[i] = (double)rand()/(double)RAND_MAX;
  }
```

```
  double *d_A;
  hipMalloc(&d_A, bytes);

  hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);

  int thr_per_blk = 256;
  int blk_in_grid = ceil( float(N) / thr_per_blk );

  multiply<<<blk_in_grid,thr_per_blk>>>(d_A, N);

  hipMemcpy(h_A, d_A, bytes, hipMemcpyDeviceToHost);

  hipFree(d_A);

  free(h_A);
  printf("__SUCCESS__\n");

  return 0;
}
```

Not needed for unified memory

Copy data from GPU buffer to CPU buffer and free memory

Software to hardware mapping



Blocks and threads allow a natural mapping of kernels to hardware:

- Upon kernel launch, a grid of thread blocks is launched to compute the kernel on the compute units (CUs)

Threads within a thread block (workgroup):

- **Execute on the same CU in chunks of 64 threads** called **wavefronts** (or waves).
- Share Local Data Share (LDS) memory and L1 cache
- Can synchronize

About wavefronts:

- Wavefronts execute on SIMD units (located inside the CU)
- 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)

3. ROCm and ROCm libraries

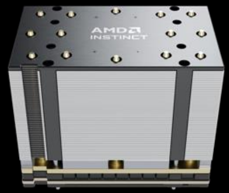
ROCm

ROCm is an open-source platform for GPU computing (including drivers, development tools, APIs, and libraries) on AMD GPUs.

- ROCm drivers allow the OS to communicate with the GPU hardware.
- ROCm libraries provide optimized routines for scientific computing and machine learning tasks, such as BLAS, FFT, etc.
- **ROCm is powered by AMD’s HIP programming environment and runtime.**

ROCm is supported on AMD INSTINCT & certain RADEON GPUs.

For the full list, please visit https://rocm.docs.amd.com/en/latest/release/gpu_os_support.html#linux-supported-gpus




Querying system

- `rocminfo`: Queries and displays information on the system's hardware
  - More info at: https://github.com/ROCm/rocminfo

Querying ROCm version:

- If you install ROCm in the standard location (`/opt/rocm`) version info is at: `/opt/rocm/.info/version-dev`

- `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/ROCm/rocm_smi_lib/tree/master/python_smi_tools

```
$ /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 on LUMI

ROCm

User level	5.7.3	**6.0.3**	6.1.3	6.2.1	6.3 ?	6.4 ?
	Dec 2023	Mar 2024	Jun 2024	Sep 2024	Nov 2024	Apr 2025
Driver		**6.0.3**			Not supported by the driver	

5.7.3 (Dec 2023)

Meant to support older version of apps and frameworks

Facilitate transition

GPU address sanitizer (beta)

Data pre-processing capabilities (MIVisionX)

~~GPU-Aware MPI~~

6.0.3 (Mar 2024)

Default version

Officially supported

Recommended for debugging

Improved sparse matrix operations

6.1.3 (Jun 2024)

Latest Pytorch and other AI frameworks require this version

Introduced many performance improvements

6.2.1 (Sep 2024)

Many stability and performance improvements for performance libraries

Improved support for lower precisions

Best tunned for AI inference workloads

Integration of profiling tools
Autocast (mixed-precision)

Native OpenXLA support

ROCm 6.2 release specific modifications

With the release of ROCm 6.2 (https://github.com/ROCm/ROCm/releases) Omnitrace and Omniperf are **included in the ROCm stack**, but they still need to be installed.

One LUMI, we are including both version of Omnitrace and Omniperf:

- The built-in versions included in the ROCm 6.2.2 software stack (installed with `sudo apt-get` as above)
  - These can be used loading the modules:
    ```
    module use /appl/local/containers/test-modules
    module load rocm/6.2.2 omnitrace/1.12.0-rocm6.2.x omniperf/2.1.0
    ```
- The latest versions from AMD Research that would be used for ROCm releases < 6.2 (install from source)
  - These can be used by loading their dedicated modules:
    ```
    module use /appl/local/containers/test-modules
    module load rocm/6.0.3 omnitrace/1.12.0-rocm6.0.x
    module load omniperf/2.1.0
    ```

ROCm GPU libraries

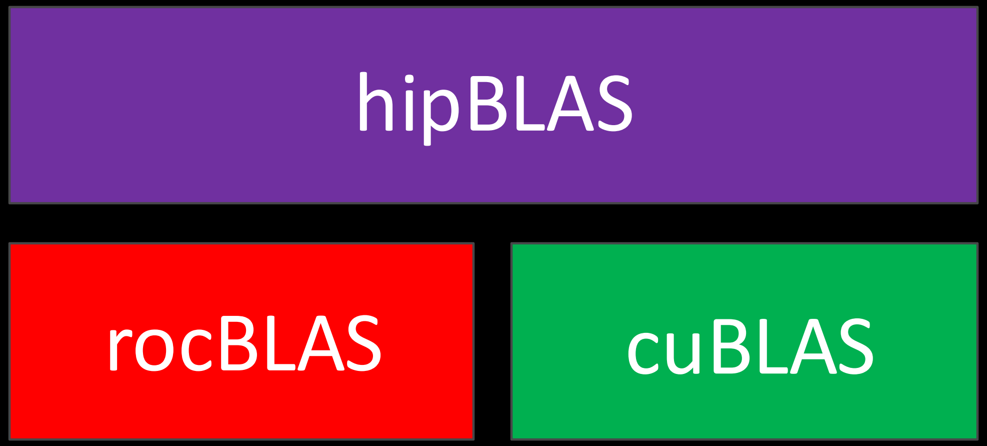
ROCm provides several GPU math libraries

- 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

See the link below for the full list:

HTTPS://GITHUB.COM/ROCM/HIP/BLOB/AMD-STAGING/DOCS/HOW-TO/HIP_PORTING_GUIDE.MD

4. Error checking, device management, and asynchronous computing

Blocking vs Nonblocking API functions

- Launching a kernel is **non-blocking for the host**
  - After sending instructions/data, the host continues to do more work while the device executes the kernel

- However, `hipMemcpy` is **blocking for the host**
  - The data pointed to in the arguments can be safely accessed/modified after the function returns

- To make asynchronous copies, we need to allocate non-pageable (pinned) 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.

Side Note: H2D/D2H bandwidth **increases significantly when host memory is pinned**

- It is good practice to use pinned host memory where data is frequently transferred to/from the device

AMD together we advance_

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:

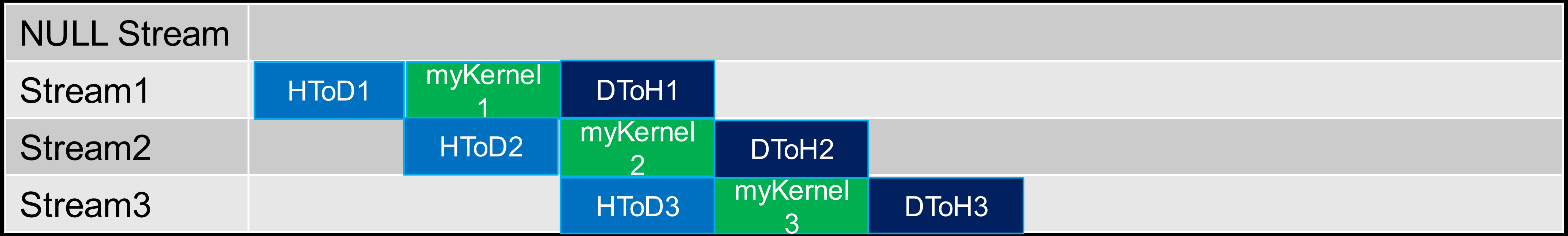
```
hipStreamDestroy(stream);
```

- 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

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);
```

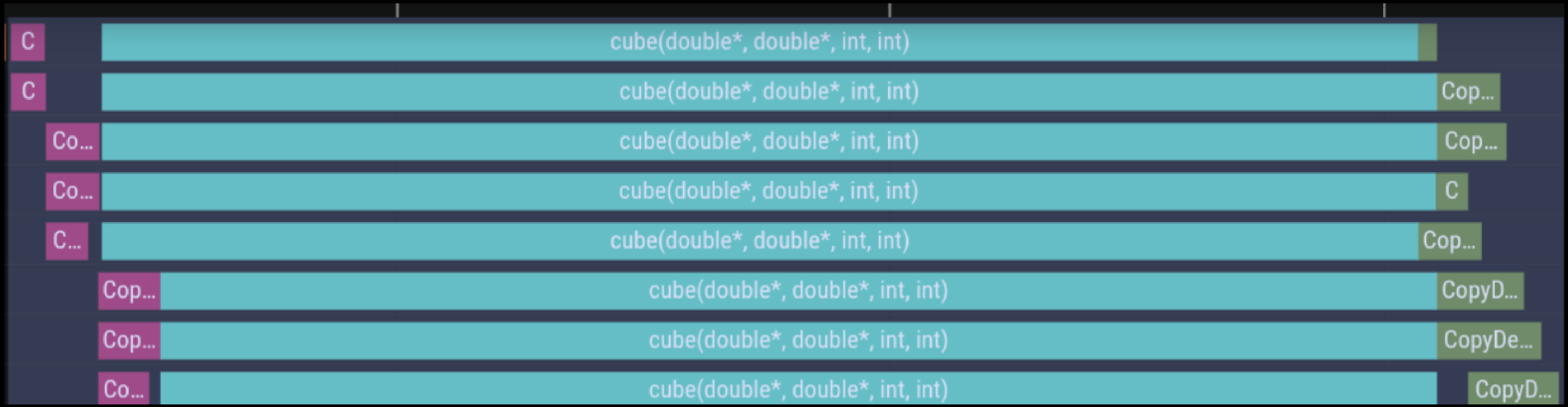


HIP stream example

In real stream overlapping, the communication and computation time will not be the same

For a real example of overlapping compute and communication in HIP

```
git clone https://github.com/AMD/HPCTrainingExamples
cd HPCTrainingExamples/HIP/Stream_Overlap
```



AMD together we advance_

5. Shared memory and thread syncronization

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':
https://rocm.docs.amd.com/projects/HIP/en/latest/.doxygen/docBin/html/group___event.html

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 and device properties:

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

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!

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.
  __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.

Hands-on exercises

https://hackmd.io/@sfantao/lumi-training-tal-2025#HIP-Exercises
https://hackmd.io/@sfantao/lumi-training-tal-2025#Hipify

We welcome you to explore our HPC Training Examples repo:
https://github.com/amd/HPCTrainingExamples

A table of contents for the READMEs if available at the top-level README in the repo

Relevant exercises for this presentation located in HIP directory.

Link to instructions on how to run the tests: HIP/README.md **and subdirectories**

Disclaimer

AMD