



HIP and ROCm

Presenter: Giacomo Capodaglio
AMD @ CASTIEL HPC
Oct 28-30, 2025

AMD 
together we advance_

Agenda

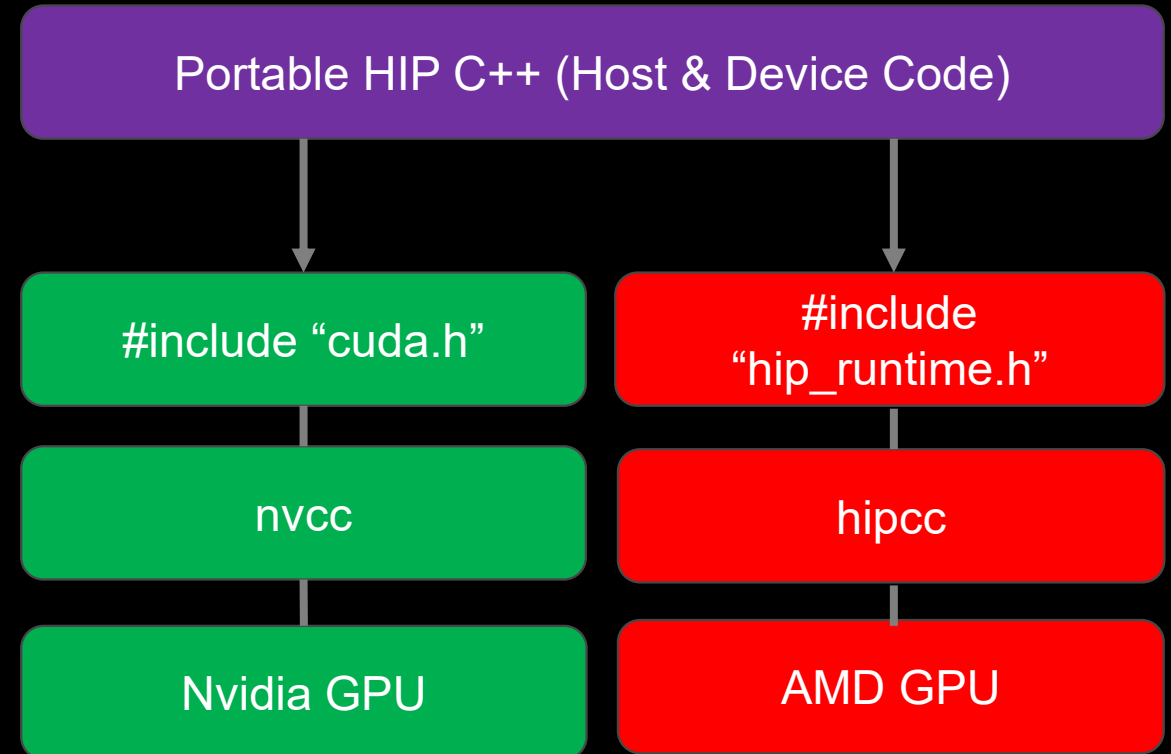
-
1. AMD GPU programming concepts
 2. HIP API calls and GPU kernel code
 3. Error checking, device management, and asynchronous computing
 4. Shared memory and thread synchronization
 5. ROCm and ROCm libraries

1. AMD GPU programming concepts

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

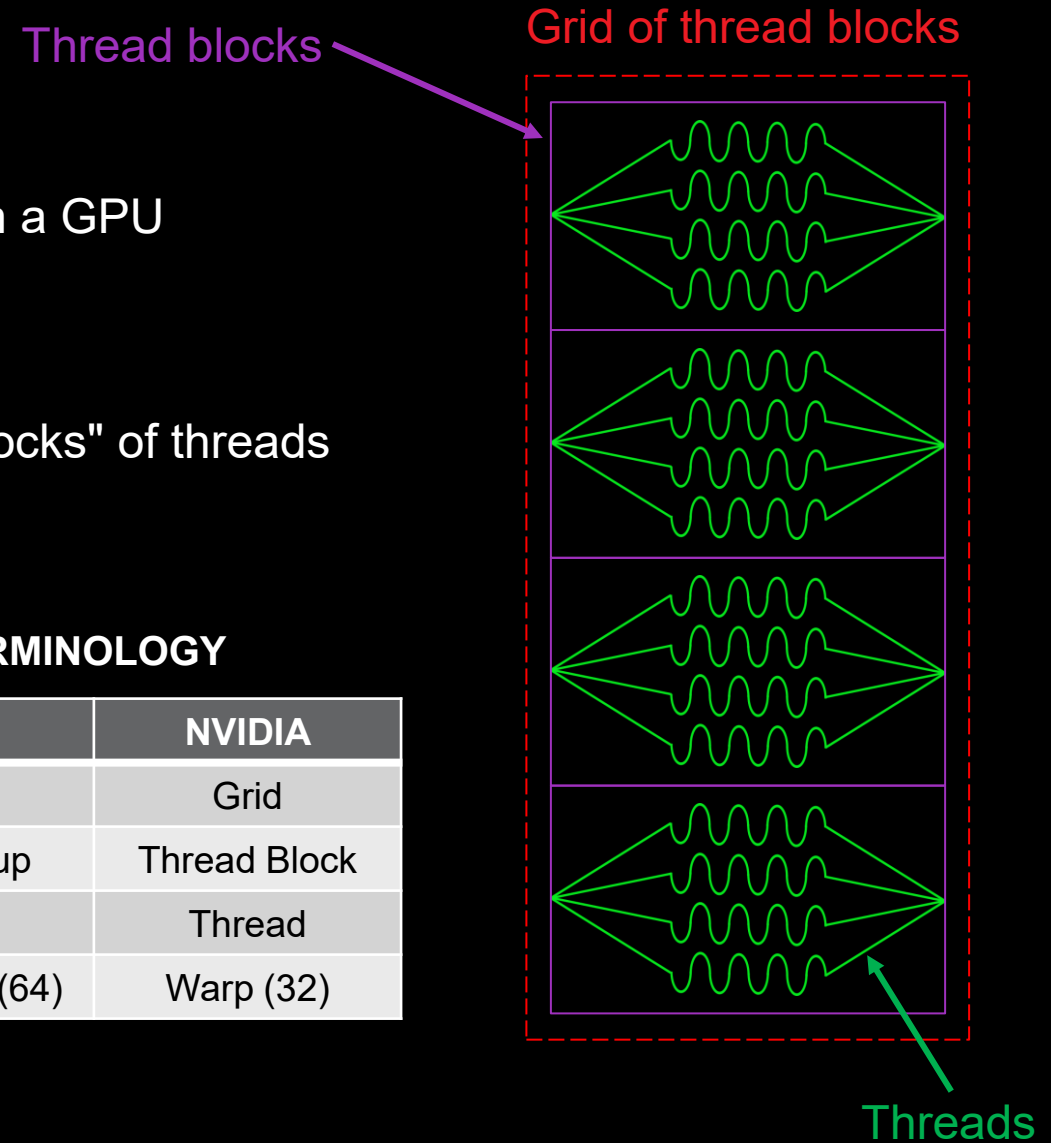


Device Kernels: Grid Hierarchy

- 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

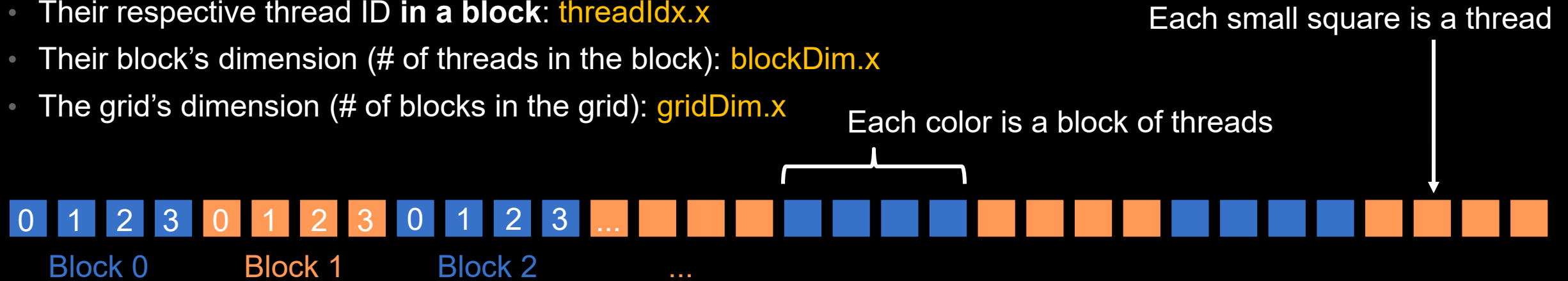
AMD	NVIDIA
Grid	Grid
Workgroup	Thread Block
Thread	Thread
Wavefront (64)	Warp (32)



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, the fourth thread of
block 2 would have a global thread
ID of 11

$$= 4 \quad * \quad 2 \quad + \quad 3$$

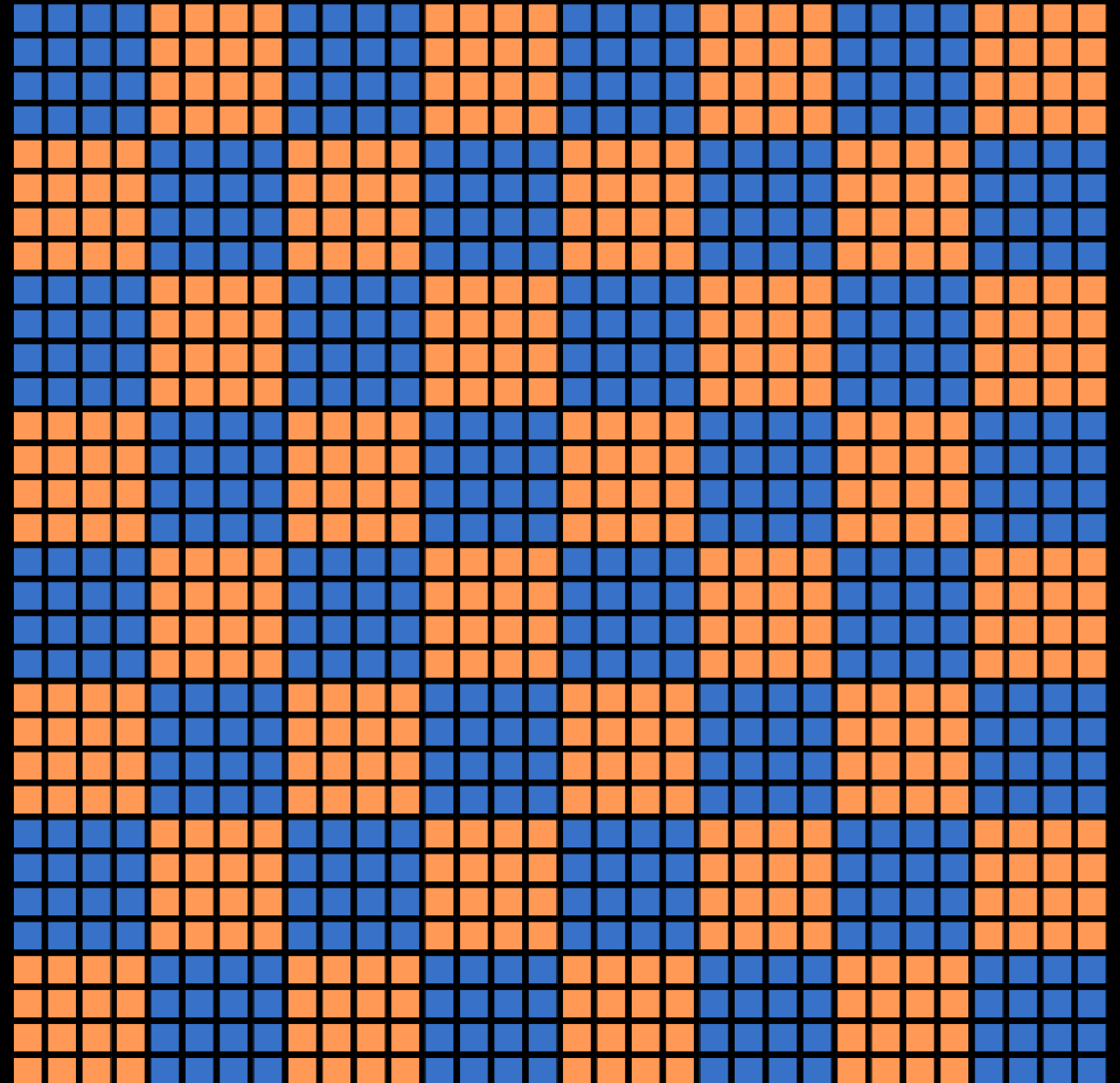
$$= 11$$

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

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;
}
```

Example: simple discrete GPU multiply

➤ Launching the kernel

Type dim3

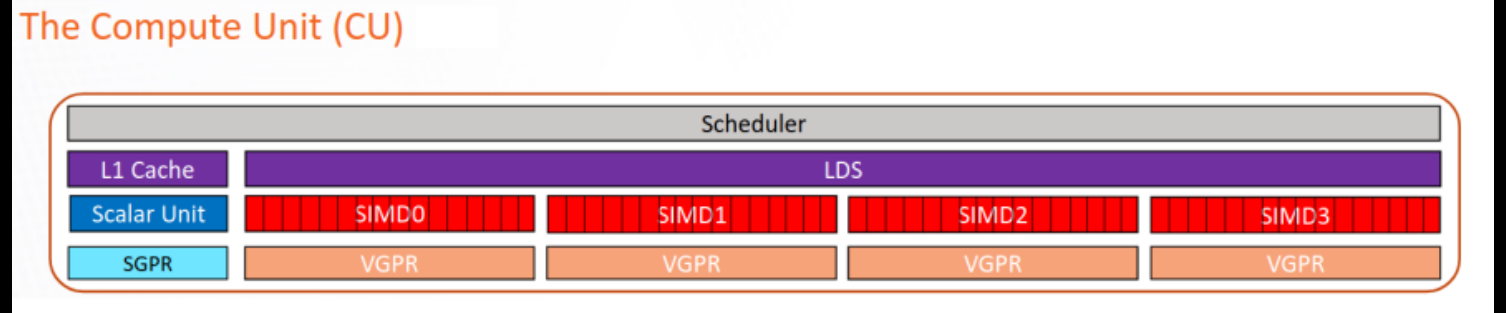
Ex: BLOCKS_IN_GRID(<nblocksx>,
<nblocksy>,
<nblocksz>)

```
kernel_name<<< BLOCKS_IN_GRID, THREADS_PER_BLOCK,  
[OPTIONAL] BYTES_OF_SHARED_MEMORY, [OPTIONAL] STREAM_ID >>>  
(ARG1, ARG2, ...);
```

```
int thr_per_blk = 256;  
int blk_in_grid = ceil( float(N) / thr_per_blk );  
  
/* Launch multiply kernel */  
multiply<<<blk_in_grid, thr_per_blk>>>(d_A, N);
```

NOTE: GPU kernel launches are asynchronous with respect to the host.

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. Error checking, device management, and asynchronous computing

Error Checking

There are two main types of HIP errors to check for:

- **Errors returned from HIP API calls**
 - HIP API calls return a `hipError_t` status
- **Errors from HIP kernels**
 - Synchronous errors: related to kernel launch
 - Asynchronous errors: related to kernel execution

Let's look at how to check for these errors...

Error checking – API errors

The `hipError_t` value should be checked for all HIP API calls!

The easiest method is wrapping the API calls in a **macro**, which can be reused in all your HIP codes. Recent compiler versions give a warning if the error is not checked.

```
/* Macro for checking GPU API return values */
#define gpuCheck(call)
do{
    hipError_t gpuErr = call;
    if(hipSuccess != gpuErr){
        printf("GPU API Error - %s:%d: '%s'\n", __FILE__, __LINE__, hipGetErrorString(gpuErr));
        exit(1);
    }
}while(0)

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

    gpuCheck( hipMalloc(&d_A, bytes) );

    ...
}
```

Error checking – kernel errors

Why are kernel errors handled differently?

- **HIP kernels do not have a return value.**
- When a kernel is launched, execution is immediately given back to the host process.

So how do we handle kernel errors?

- Errors related to the kernel launch (e.g., invalid execution parameters)
 - Manually check for the last error that occurred using `hipGetLastError()`
 - These are known as **synchronous** errors
- Errors related to kernel execution (e.g., invalid memory access) can happen at any time while the kernel is running
 - Must synchronize the device to make sure we catch these errors (`hipDeviceSynchronize()`).
 - These are known as **asynchronous** errors

```

...

/* Launch multiply kernel */
multiply<<<blk_in_grid, thr_per_blk>>>(d_A, N);

/* Check for kernel launch errors */
gpuCheck( hipGetLastError() );

/* Check for kernel execution errors */
if (DEBUG)
    gpuCheck ( hipDeviceSynchronize() );

...

```

NOTE: Device synchronization can cause reduced performance so should be reserved for debugging.

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 (~2x) when host memory is pinned

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

Streams

- A stream in HIP is a **queue of tasks** (e.g. kernels, memcpyys, 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:

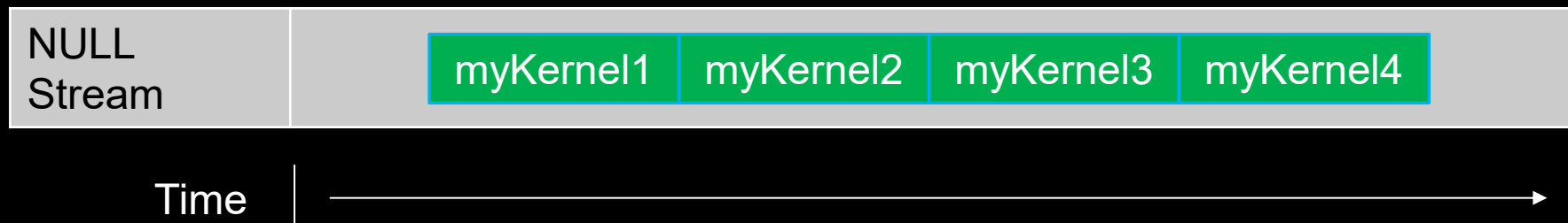
```
hipStreamDestroy(stream);
```
- Passing **0** or **NULL** as the `hipStream_t` argument to a function instructs the function to execute on a stream called the '**NULL Stream**':
 - No task on the NULL stream will begin until all previously enqueued tasks in all other streams have completed.
 - Blocking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

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);
```

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

4. Shared memory and thread synchronization

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

Thread Synchronization

`__syncthreads()`:

- Blocks a thread in a block from continuing execution until all threads in the block 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

Shared Memory Example

```
__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 threads in the block must reach this point before they are 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
    ...
}
```

5. ROCm and ROCm libraries

ROCm GPU libraries

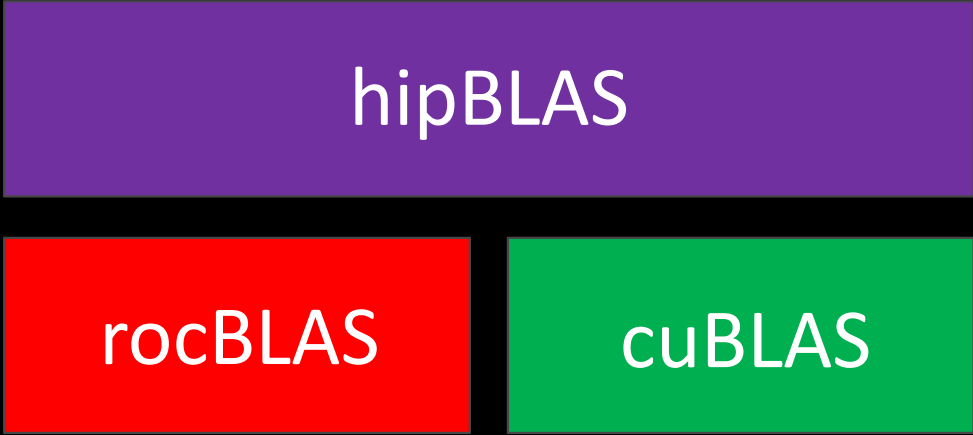
ROCm provides several GPU math libraries

- 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



Correspondence with Nvidia math libraries (1/3)

cuBLAS	hipBLAS	rocBLAS	Basic Linear Algebra Subroutines (BLAS)
cuBLASLt	hipBLASLt	N/A	Basic Linear Algebra Subroutines (BLAS), lightweight and new flexible API
cuRAND	hipRAND	rocRAND	Random Number Generator Library
Thrust	N/A	rocThrust	C++ Parallel Algorithms
CUB	hipCUB	rocPRIM	Low Level Optimized Parallel Primitives

hipBLASLt is the library in this case, not the thin portability layer

Correspondence with Nvidia math libraries (2/3)

cuSPARSE	hipSPARSE	rocSPARSE	Basic Linear Algebra Subroutines (BLAS) for sparse computation
cuSPARSELt	hipSPARSELt	rocSPARSELt	Library for ML sparsity operations
cuFFT	hipFFT	rocFFT	Fast Fourier Transform Library
cuSOLVER	hipSOLVER	rocSOLVER	Lapack Library
AmgX	N/A	rocALUTION	Sparse iterative solvers and preconditioners with algebraic multigrid



rocSPARSELt is actually part of the hipSPARSELt github repo:
https://github.com/ROCm/hipSPARSELt/tree/develop/library/src/hcc_detail/rocsparselt

Correspondence with Nvidia math libraries (3/3)

cuTENSOR

hipTENSOR

N/A

Accelerate tensor primitives using GPU matrix cores (currently a work in progress)

Correspondence with Nvidia AI and ML libraries

cuDNN

N/A

MIOpen

Deep learning Solver Library

NCCL

N/A

RCCL

Communications Primitives Library based on the MPI equivalents

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 =====
```

Hands-on exercises

Located in 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

Log into the AAC node and clone the repo:

```
ssh <username>@aac6.amd.com -p 7000 -i <path_to_ssh_key>  
git clone https://github.com/amd/HPCTrainingExamples.git
```



Porting code to HIP

Presenter: Giacomo Capodaglio
AMD @ CASTIEL HPC
Oct 28-30, 2025

AMD 
together we advance_

No one size fits all approach

Self contained GPU code

- ▲ Automatic conversion tools (hipify)
- ▲ Header file interception layer (hipiFLY)
 - Fast way to get running code
 - May break on creative use of previous device code

More complex accelerator layers

- ▲ Need combination of automatic conversion and manual rewrite
 - Abstraction layer needs to be adapted to use HIP API
 - Can later use HIP to target both ROCm and CUDA
 - Longer time to running code

Code Conversion Tools

EXTEND YOUR APPLICATION
PLATFORM SUPPORT BY
CONVERTING CUDA® CODE

Single source

Maintain portability

Maintain performance

Hipify-perl

- ▲ Easiest to use; point at a directory and it will hipify CUDA code
- ▲ Very simple string replacement technique; may require manual post-processing
- ▲ It replaces cuda with hip, `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

- ▲ More robust translation of the code
- ▲ Generates warnings and assistance for additional analysis
- ▲ High quality translation, particularly for cases where the user is familiar with the make system

Hipify-perl

It is located in `/opt/rocm/bin`

- Command line tool: `hipify-perl foo.cu > new_foo.cpp`
- Compile: `hipcc new_foo.cpp`

How does this this work in practice?

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

Hipify-clang

It is located in /opt/rocm/bin

Build from source ([needs clang compiler](#))

- 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`

More info at: <https://github.com/ROCm/HIPIFY/blob/master/README.md>

Can be used to perform build-time conversion if project can be automatically converted




Ported in an Afternoon
HACC
Cosmology



Ported in a Single Day
SPECFEM3D
Seismology



Ported in 21 Days
QUDA
Quantum Physics



Ported in a Couple of Days
CHOLLA
Astrophysics

NAMD
Scalable Molecular Dynamics

LAMMPS

kokkos

Nekbone

GROMACS
FAST. FLEXIBLE. FREE.

MILC

Chroma

TensorFlow

PYTORCH

GridTools

ALTAIR

SIRIUS

AMBER

PICongPU

CP2K

LSMS

Other Hipify tools

Individual file tools (already discussed)

- hipify-perl
- hipify-clang

Recursive directory tools (also in /opt/rocm/bin)

- hipconvertinplace.sh
- hipconvertinplace-perl.sh
- hipexamine.sh
- hipexamine-perl.sh

The Perl[®] scripts are a set and the shell/clang tools are a set. The directory-based tools basically call the base tools, hipify-perl and hipify-clang, respectively.

For example:

hipifyexamine-perl.sh recursively runs hipify-perl with the -no-output -print-stats options (-examine option is a shorthand for -no-output -print-stats options).

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 (**grep for "32" just in case**). Use `#define WARPSIZE size`.
- Certain functions may not have a correspondent hip version (e.g., `__shfl_down_sync` – hint: use `__shfl_down` instead)
- Hipifying can't handle inline PTX assembly or CUDA intrinsics
 - Can either use inline GCN ISA, or convert it to HIP
- 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.

Notes

- Hipify-perl and hipify-clang can both convert library calls (i.e. cuBLAS becomes hipBLAS)
- CMake starting with version 3.21 can be used to automatically set up basic compilation flags by using `enable_language(HIP)`, supports `CMAKE_HIP_ARCHITECTURES` for setting devices to build for

HIIFLY: Intercept API method to choose GPU backend

- Enable running existing code on different backends with single header
- Can change between targeting CUDA and ROCm in one place
- Only works if no difference between API calls
- Existing code cannot use any CUDA specific hard coded values
- Performance needs to be evaluated on a case-by-case basis



```

#ifndef CUDA_TO_HIP_H
#define CUDA_TO_HIP_H

#include <hip/hip_runtime.h>

#define WARPSIZE 64
static constexpr int maxWarpsPerBlock = 1024/WARPSIZE;

#define CUFFT_D2Z HIPFFT_D2Z
#define CUFFT_FORWARD HIPFFT_FORWARD
#define CUFFT_INVERSE HIPFFT_BACKWARD
#define CUFFT_Z2D HIPFFT_Z2D
#define CUFFT_Z2Z HIPFFT_Z2Z

#define cudaDeviceSynchronize hipDeviceSynchronize
#define cudaError hipError_t
#define cudaError_t hipError_t
#define cudaErrorInsufficientDriver hipErrorInsufficientDriver
#define cudaErrorNoDevice hipErrorNoDevice
#define cudaEvent_t hipEvent_t
#define cudaEventCreate hipEventCreate
#define cudaEventElapsedTime hipEventElapsedTime
#define cudaEventRecord hipEventRecord
#define cudaEventSynchronize hipEventSynchronize
#define cudaFree hipFree
#define cudaFreeHost hipHostFree
#define cudaGetDevice hipGetDevice
#define cudaGetDeviceCount hipGetDeviceCount
#define cudaGetErrorString hipGetErrorString
#define cudaGetLastError hipGetLastError
#define cudaHostAlloc hipHostMalloc
#define cudaHostAllocDefault hipHostMallocDefault
#define cudaMalloc hipMalloc
#define cudaMemcpy hipMemcpy
#define cudaMemcpyAsync hipMemcpyAsync
#define cudaMemcpyDeviceToHost hipMemcpyDeviceToHost
#define cudaMemcpyDeviceToDevice hipMemcpyDeviceToDevice
#define cudaMemcpyHostToDevice hipMemcpyHostToDevice
#define cudaMemGetInfo hipMemGetInfo
#define cudaMemset hipMemset
#define cudaReadModeElementType hipReadModeElementType

```

Link to the header file:

https://github.com/amd/HPCTrainingExamples/blob/main/hipifly/vector_add/src/cuda_to_hip.h

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
```

← Setting default device compiler

```
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 compile flags for different GPUs

← Pattern rule for HIP source

```
%.o: %.hip
```

```
    hipcc $(HIPCC_FLAGS) -c $^ -o $@
```

```
$(EXECUTABLE): $(OBJECTS)
```

```
    hipcc $< $(LDFLAGS) -o $@
```

```
test: $(EXECUTABLE)
```

```
    $(EXECUTABLE)
```

```
clean:
```

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

Using a portable Makefile

- For ROCm

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

To build and run:

```
make vectoradd  
./vectoradd
```

- For CUDA

```
module load rocm ← We still need HIP for the portability layer  
module load cuda
```

To build and run:

```
HIP_PLATFORM=nvidia ← Overriding default to compile with nvidia  
make vectoradd  
./vectoradd
```

Portable Build Systems – CMakeLists.txt (1 of 3)

```
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 (CMAKE_GPU_RUNTIME)
    set(GPU_RUNTIME "${CMAKE_GPU_RUNTIME}" CACHE STRING "Switches between ROCM and CUDA")
endif (NOT CMAKE_GPU_RUNTIME)
```

} Setting
GPU_RUNTIME

Portable Build Systems – CMakeLists.txt (2 of 3)

```
# Should only be ROCM or CUDA, but allowing HIP because it is the currently built-in option
# Select with e.g., -DGPU_RUNTIME=ROCM
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 should really be ROCM for 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})

enable_language(${GPU_RUNTIME}) ← Enabling either CUDA or HIP (ROCM)
set(CMAKE_${GPU_RUNTIME}_EXTENSIONS OFF)
set(CMAKE_${GPU_RUNTIME}_STANDARD_REQUIRED ON)
```

Portable Build Systems – CMakeLists.txt (3 of 3)

```
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 (${GPU_RUNTIME} MATCHES "CUDA")
    set(HIPCC_FLAGS "${CUDACC_FLAGS}")
endif (${GPU_RUNTIME} MATCHES "HIP")
```

Setting different flags for each GPU type

Setting language type for HIP source files

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

Setting device compile flags

```
install(TARGETS vectoradd)
```

Using a portable CMakeLists.txt

- For ROCm

```
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 CUDA

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

To build and run:

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

← Overrides default GPU runtime to specify CUDA

Important CMake variables

- CMAKE_HIP_ARCHITECTURES
 - CMAKE_HIP_ARCHITECTURES="gfx90a;gfx908"
 - GPU_TARGETS="gfx90a;gfx908"

List of gfx models: <https://lvm.org/docs/AMDGPUUsage.html>

Find the gfx model with rocinfo: rocinfo | grep gfx | sed -e 's/Name:/' | head -1 | sed 's/ //g'

- CMAKE_CXX_COMPILER:PATH=/opt/rocm/bin/amdclang++
- CMAKE_HIP_COMPILER_ROCM_ROOT:PATH=/opt/rocm – to help cmake find the cmake config files
- CMAKE_PREFIX_PATH=/opt/rocm
- Finding HIP packages and use results
 - find_package(rocprim)
 - target_link_libraries(MyLib PUBLIC roc::rocprim)
- Using host and device from find_package(hip)
 - target_link_libraries(MyLib PRIVATE hip::device)
 - target_link_libraries(MyApp PRIVATE hip::host)

CUDA Fortran Fortran + HIP C/C++

- There is no HIP equivalent to CUDA Fortran
- HIP functions are callable from C, using extern C, so they can be called directly from Fortran
- The strategy here is:
 - 1) **Manually port** CUDA Fortran code to HIP kernels in C-like syntax
 - 2) Wrap the kernel launch in a C function
 - 3) Call the C function from Fortran through Fortran's ISO_C_binding. It requires Fortran 2008 because of using pointers to share data.
- 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/ROCm/hipfort>

Hands-on exercises

Located in 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:

- [HIPIFY](#) directory
- [hipify](#) directory
- [HIPFort](#) directory

Link to instructions on how to run the HIPIFY tests: [HIPIFY/README.md](#)

Log into the AAC node and clone the repo:

```
ssh <username>@aac6.amd.com -p 7000 -i <path_to_ssh_key>  
git clone https://github.com/amd/HPCTrainingExamples.git
```

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.

© 2025 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, AMD CDNA, AMD ROCm, AMD Instinct, and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names are for informational purposes only and may be trademarks of their respective owners.

Git and the Git logo are either registered trademarks or trademarks of Software Freedom Conservancy, Inc., corporate home of the Git Project, in the United States and/or other countries

AMD 