Introduction to ROCm and HIP for AMD GPUs

Tom Papatheodore
AMD University Program
Research & Advanced Development

CARLA 2024
September 30, 2024

AMD
together we advance_

AMD University Program Vision

Empower academics with AMD technology to enhance teaching and learning experiences and advance state-of-the-art research.

https://www.amd.com/en/corporate/university-program.html

Session Overview

9:30 AM – 10:40 AM	**Resource Access**	Log in to Cluster and Clone Tutorial Repository
	Background	Parallel Programming Models, AMD GPUs, ROCm, HIP
	Basics w/ Vector Addition	Basic structure of HIP code, API calls, grid hierarchy, kernels
	HIP Error Checking	Understanding HIP API errors and kernel errors
	Timing w/ hipEvents	Using HIP events to time GPU operations
10:45 AM – 11:00 AM	**Hands-On Session 1**	Participants work on exercises on content covered to this point
11:00 AM – 11:30 AM	**BREAK**	
11:30 AM – 12:10 PM	**2D Grids**	Matrix addition example
	Shared Memory	Vector addition example
	Concurrency	Streams, overlap kernels, overlap data transfers w/ compute
	CUDA-to-HIP Translations	hipify and hipifly
12:10 PM – 12:40 PM	**Hands-On Session 2**	Participants work on exercises on content covered to this point
12:40 PM – 1:00 PM	**AI on ROCm**	Supported frameworks and ecosystem partners
	Wrap Up	Additional resources and where to go from here

Resource Access

Log In to Cluster and Clone Repository

AMD AI & HPC Cluster

40-node cluster consisting of AMD EPYC CPUs (7V13, 7763) and AMD Instinct GPUs (MI100, MI210, MI250) connected with a high-speed GPU-aware interconnect.

You will be given a temporary username + password to access the cluster during the tutorial. To log in, open a terminal and issue the following command, then enter your password when prompted:

```
$ ssh <username>@hpcfund.amd.com
```

Once logged in, please clone the tutorial repository by issuing the following command and move into the directory:

```
$ git clone https://github.com/AMDResearch/introduction_to_hip.git
$ cd introduction_to_hip
```

Background

Parallel Programming Models

Distributed-Memory Model

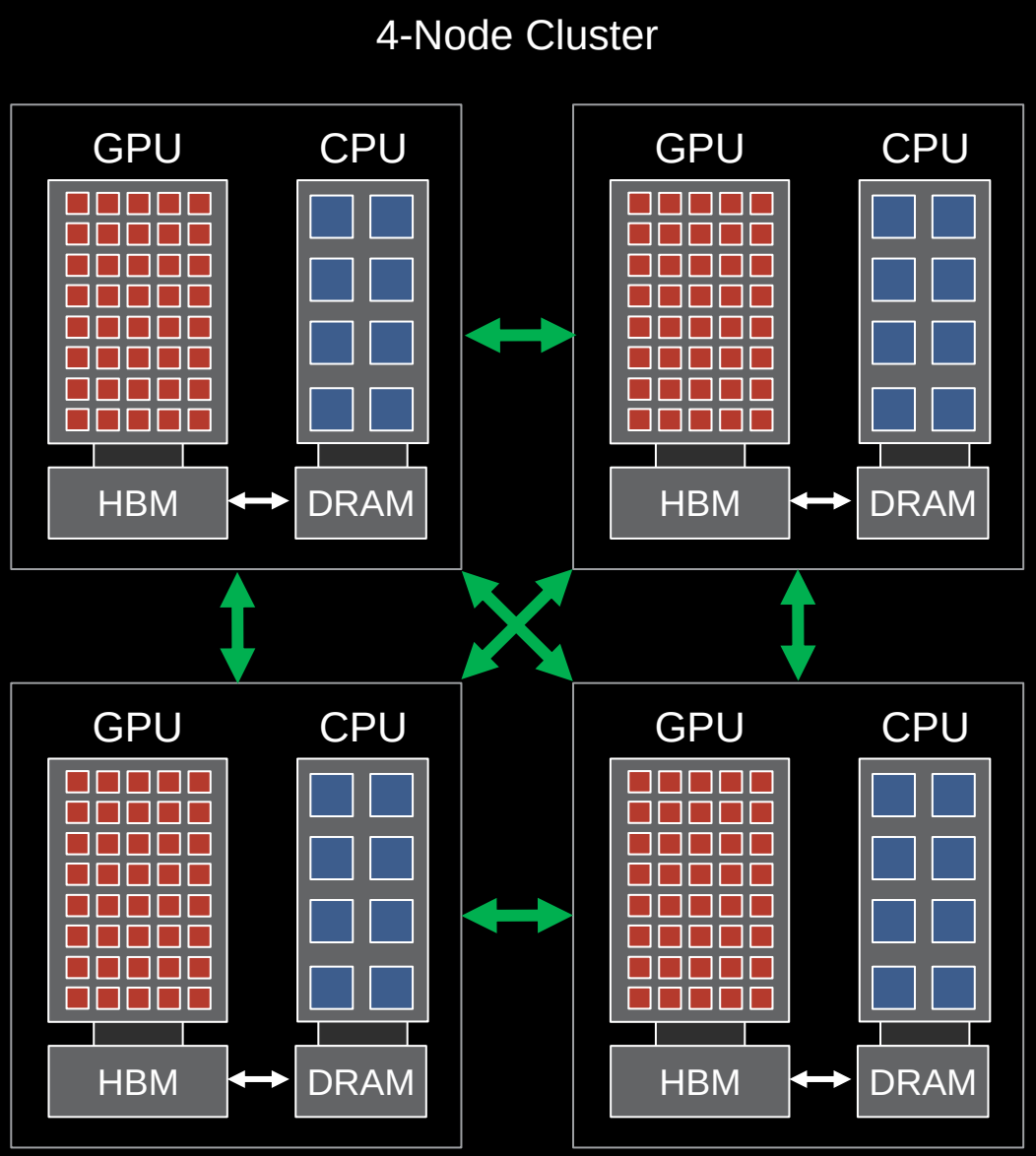
- For example, message passing interface (MPI)
- Coarse-grained parallelism across nodes

Shared-Memory Model

- For example, OpenMP®
- Fine-grained parallelism within node

GPU Programming

- For example, HIP, CUDA
- Fine-grained parallelism within node



AMD ROCm

An Open-Source SW Stack for GPU Computation in AI & HPC

	FRAMEWORKS		PyTorch, TensorFlow, ONNX-RT, JAX
ROCm	LIBRARIES	Machine Learning & Computer Vision	Composable Kernel, MIGraphX, MIOpen, MIVisionX, RPP, rocDecode, rocAL
		Communication	RCCL
		Math	half, hipBLAS, hipBLASLt, hipFFT, hipfort, hipRAND, hipSOLVER, hipSPARSE, hipSPARSELt, rocALUTION, rocBLAS, rocFFT, rocRAND, rocSOLVER, rocSPARSE, rocWMMA, Tensile
		Primitives	hipCUB, hipTensor, rocPRIM, rocThrust
	TOOLS		HIPIFY, ROCm Bandwidth Test, ROCm CMake, ROCm Data Center Tool, ROCm Info, ROCm Validation Suite, ROCdbgapi, ROC Debug Agent, ROCgdb, ROCProfiler, ROCTracer, AMD SMI, ROCm SMI, TransferBench
	COMPILERS		hipCC, LLVM (amdclang, amdflang, OpenMP)
	RUNTIMES		AMD Common Language Runtime (CLR), HIP, ROCr
	OPERATING SYSTEMS		RHEL, SLES, Ubuntu, Windows
	ACCELERATORS & GPUS		AMD Instinct™, AMD Radeon™, AMD Radeon™ PRO

- Consists of a collection of drivers, development tools, and APIs that enable GPU programming from low-level kernel to end-user applications
- Powered by the Heterogeneous-Compute Interface for Portability (HIP)

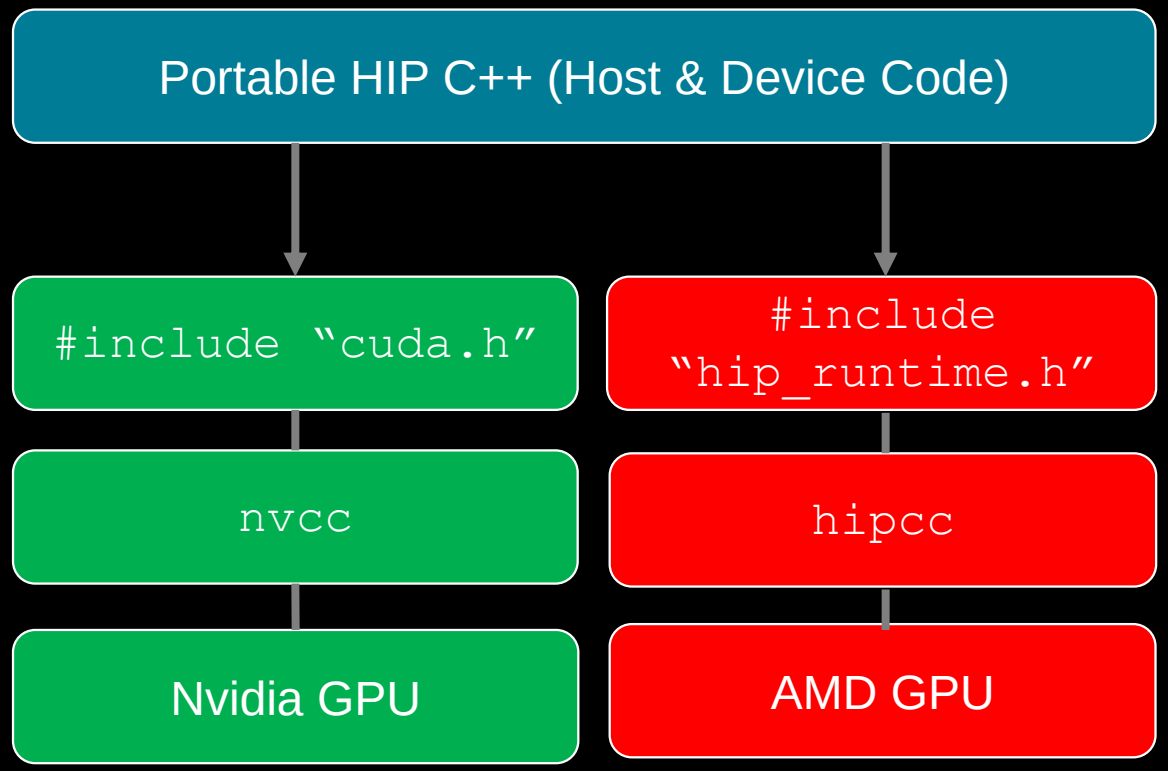
https://rocm.docs.amd.com/en/latest/

Heterogeneous-Compute Interface for Portability (HIP)

The AMD Heterogeneous-compute Interface for Portability (HIP) is a C++ runtime API and kernel language that allows developers to create portable applications that can run on AMD 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



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__`, `hipLaunchKernelGGL()`, `<<< >>>`

Device code

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

Error handling

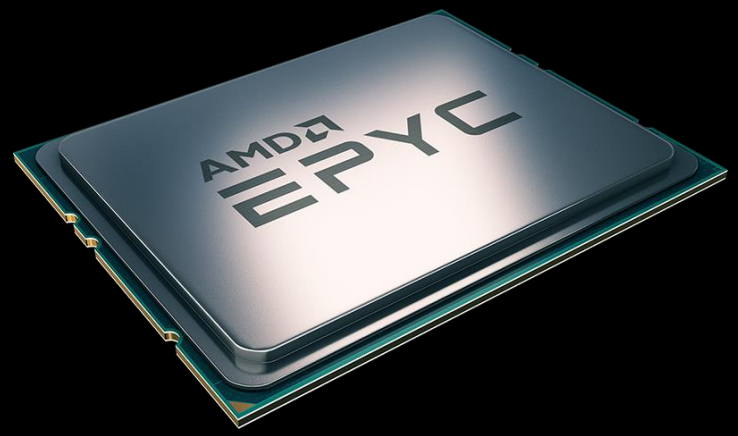
- `hipGetLastError()`, `hipGetErrorString()`

A Tale of Host and Device

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

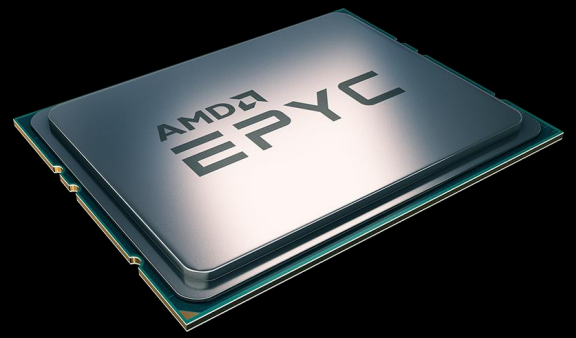
- Host is the CPU
- Host code runs here
- Usual C++ syntax and features
- Entry point is the ‘main’ function
- HIP API can be used to create device buffers, move between host and device, and launch device code

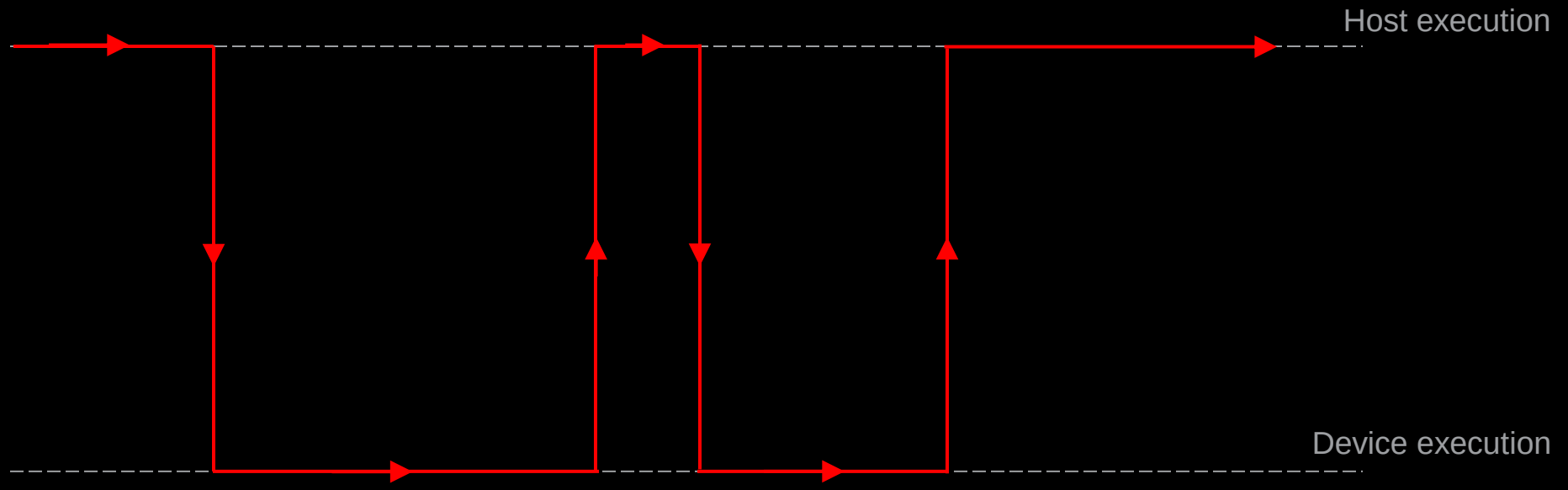
- Device is the GPU
- Device code runs here
- C-like syntax
- Device codes are launched via “kernels”
- Instructions from the Host are enqueued into “streams”



Host-Device Programming Model

- Execution is passed back and forth between host and device as a program runs.
- The host and device each have their own physical memories so data must also be copied back and forth between their memories.





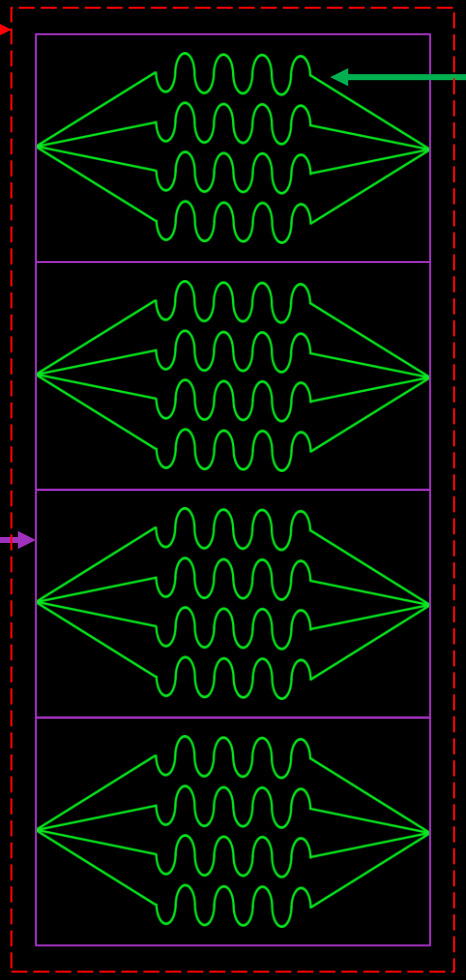
Grid Hierarchy – Threads & Thread Blocks

Grid of thread blocks

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

Thread blocks

- Coarse-grained parallelism

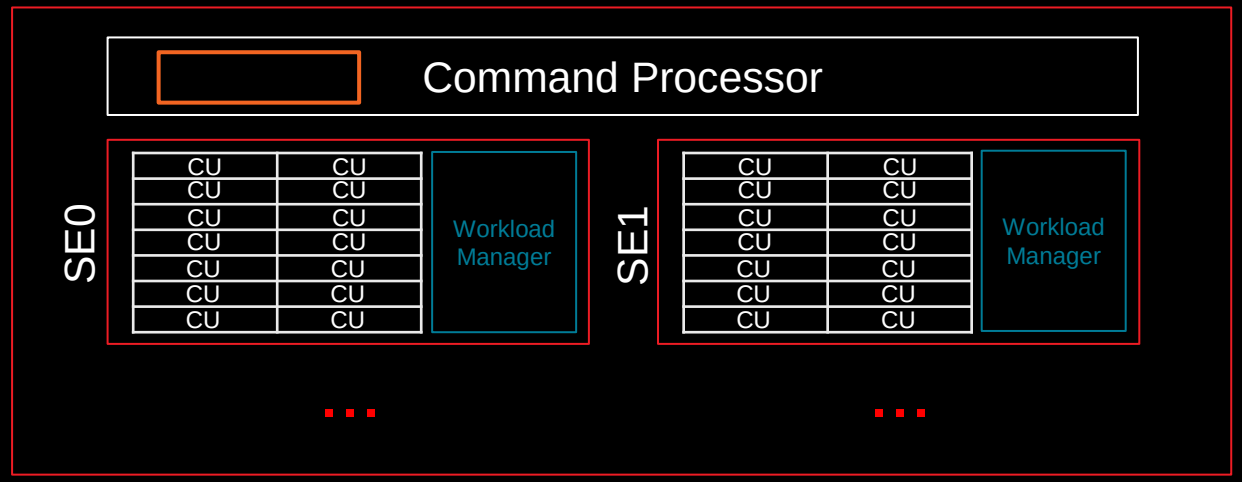


Threads

- Fine-grained parallelism
- Threads within a block can cooperate (e.g., access same shared memory, block-level synchs)

AMD	NVIDIA
Grid	Grid
Workgroup	Thread Block
Thread	Thread
Wavefront	Warp

SIMD operations



Why blocks and threads?

Allows 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:

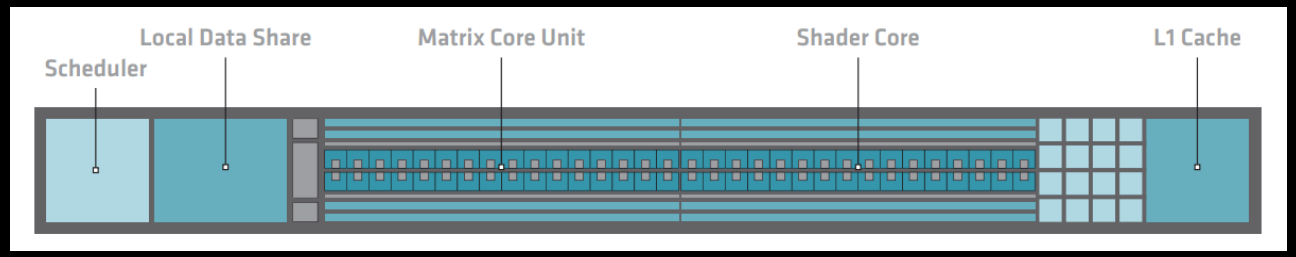
- Execute on the same CU, share LDS memory and L1 cache, can synchronize

Threads in a block are executed in 64-wide chunks called "wavefronts“ (or 32-wide for Radeon)

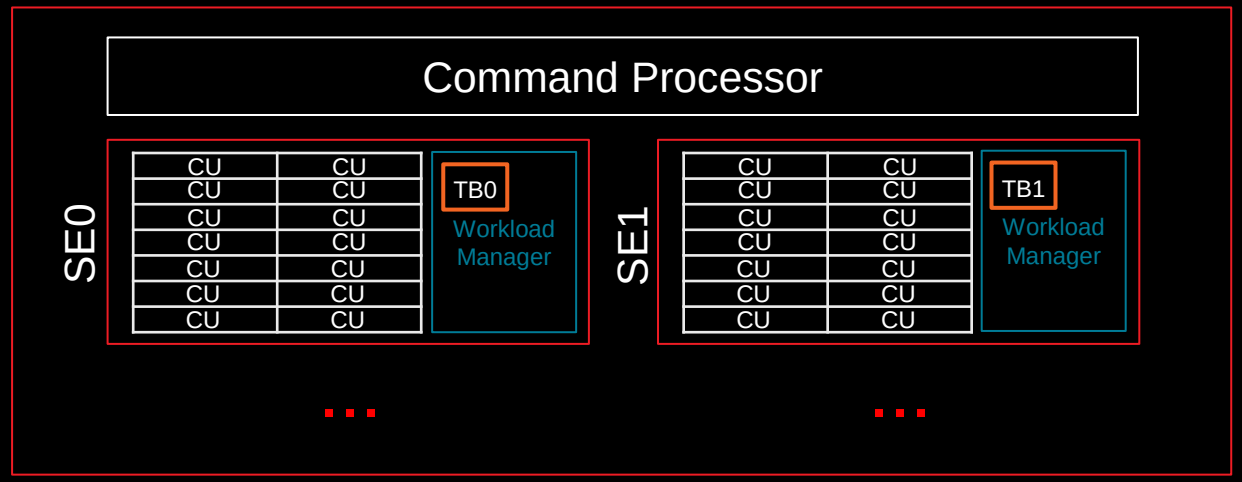
- Wavefronts execute on SIMD units (Single Instruction Multiple Data)
- If a wavefront stalls (e.g., data dependency) CUs can quickly context switch to another wavefront

Good practice to make block size a multiple of 64 to have several wavefronts (e.g., 256 threads)

CDNA3 CU



SIMD operations



Why blocks and threads?

Allows 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:

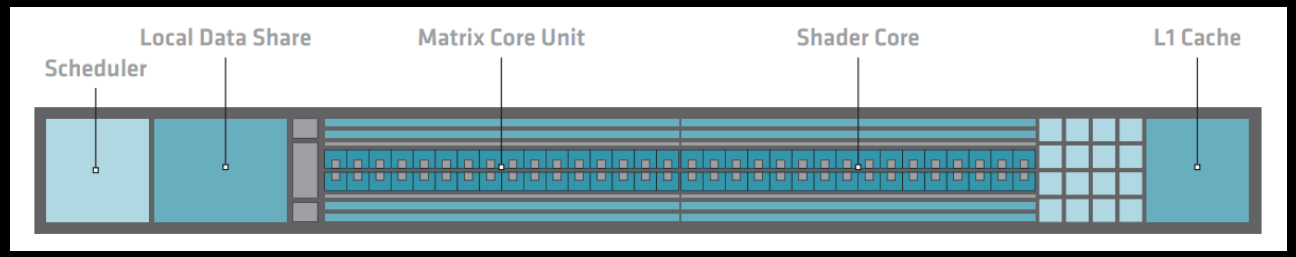
- Execute on the same CU, share LDS memory and L1 cache, can synchronize

Threads in a block are executed in 64-wide chunks called "wavefronts“ (or 32-wide for Radeon)

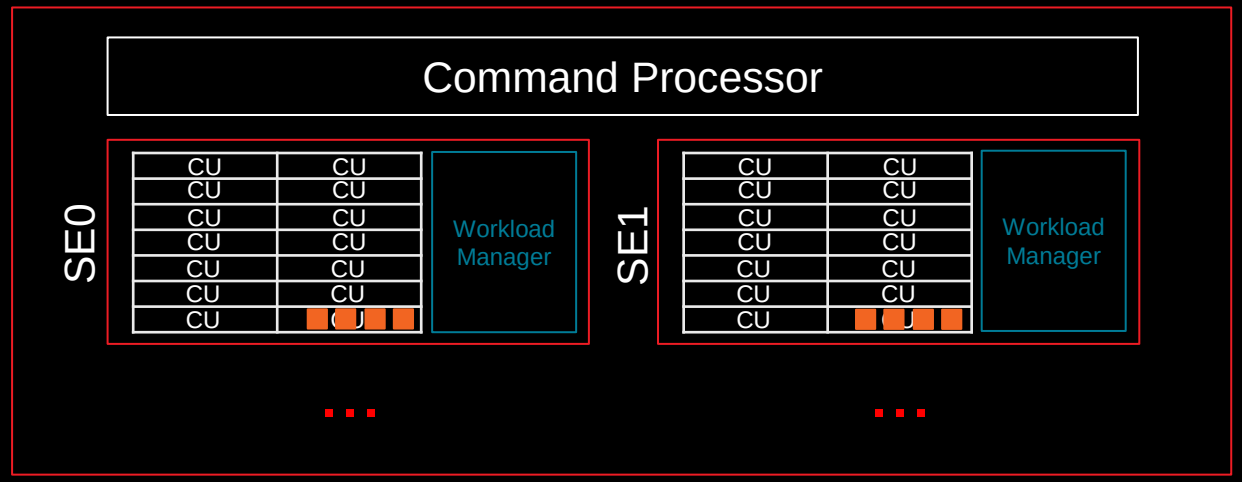
- Wavefronts execute on SIMD units (Single Instruction Multiple Data)
- If a wavefront stalls (e.g., data dependency) CUs can quickly context switch to another wavefront

Good practice to make block size a multiple of 64 to have several wavefronts (e.g., 256 threads)

CDNA3 CU



SIMD operations



Why blocks and threads?

Allows 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:

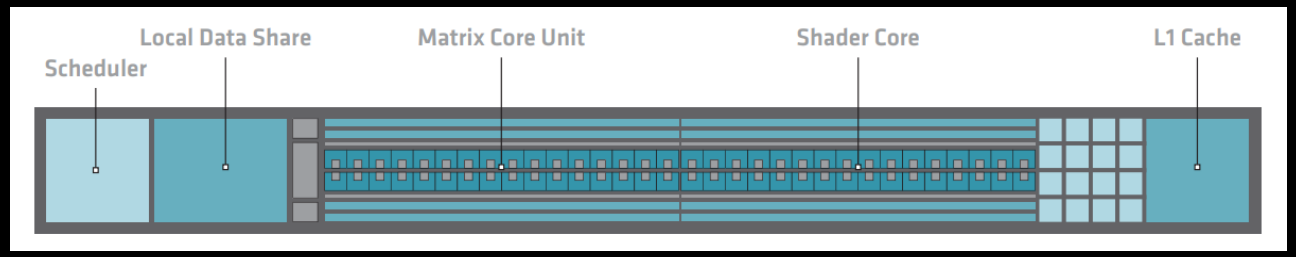
- Execute on the same CU, share LDS memory and L1 cache, can synchronize

Threads in a block are executed in 64-wide chunks called "wavefronts“ (or 32-wide for Radeon)

- 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

Good practice to make block size a multiple of 64 to have several wavefronts (e.g., 256 threads)

CDNA3 CU



Learning the Basics through Vector Addition

Concepts demonstrated by:
`introduction_to_hip/examples/01_vector_addition`

Vector Addition

Vector addition is simply the element-wise addition of 2 vectors

A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}

+

B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}

=

C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}	C_{12}	C_{13}	C_{14}	C_{15}

HIP API Calls for Simple Vector Addition Code

GPU Memory Allocation

HIP return status — Pointer to GPU buffer — Size of GPU buffer (bytes)

```
hipError_t hipMalloc(void **ptr_to_buf, size_t size_in_bytes)
```

CPU-GPU Data Transfers

HIP return status — Destination buffer — Source buffer — Size of buffer (bytes) — Direction of transfer

```
hipError_t hipMemcpy(void *dst, void *src, size_t size_in_bytes, hipMemcpyKind kind)
```

GPU Memory Deallocation

HIP return status — GPU buffer

```
hipError_t hipFree(void *buf)
```

Simple Vector Addition Example

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

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

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

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

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

    for(int i=0; i<N; i++){
        h_A[i] = sin(i) * sin(i);
        h_B[i] = cos(i) * cos(i);
        h_C[i] = 0.0;
    }

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

```
    hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_B, h_B, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_C, h_C, bytes, hipMemcpyHostToDevice);

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

    vec_add<<<blk_in_grid,thr_per_blk>>>(d_A, d_B, d_C, N);

    hipMemcpy(h_C, d_C, bytes, hipMemcpyDeviceToHost);

    double sum       = 0.0;
    double tolerance = 1.0e-14;

    for(int i=0; i<N; i++){
        sum = sum + h_C[i];
    }

    if( fabs( (sum / N) - 1.0 ) > tolerance ){
        printf("Error: Sum/N = %0.2f instead of ~1.0\n", sum / N);
        exit(1);
    }

    free(h_A);
    free(h_B);
    free(h_C);

    hipFree(d_A);
    hipFree(d_B);
    hipFree(d_C);

    printf("__SUCCESS__\n");

    return 0;
}
```

Simple Vector Addition Example

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

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

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

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

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

    for(int i=0; i<N; i++){
        h_A[i] = sin(i) * sin(i);
        h_B[i] = cos(i) * cos(i);
        h_C[i] = 0.0;
    }

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

GPU Kernel Function

```
    hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_B, h_B, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_C, h_C, bytes, hipMemcpyHostToDevice);

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

    vec_add<<<blk_in_grid,thr_per_blk>>>(d_A, d_B, d_C, N);

    hipMemcpy(h_C, d_C, bytes, hipMemcpyDeviceToHost);

    double sum       = 0.0;
    double tolerance = 1.0e-14;

    for(int i=0; i<N; i++){
        sum = sum + h_C[i];
    }

    if( fabs( (sum / N) - 1.0 ) > tolerance ){
        printf("Error: Sum/N = %0.2f instead of ~1.0\n", sum / N);
        exit(1);
    }

    free(h_A);
    free(h_B);
    free(h_C);

    hipFree(d_A);
    hipFree(d_B);
    hipFree(d_C);

    printf("__SUCCESS__\n");

    return 0;
}
```

Simple Vector Addition Example

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

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

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

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

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

    for(int i=0; i<N; i++){
        h_A[i] = sin(i) * sin(i);
        h_B[i] = cos(i) * cos(i);
        h_C[i] = 0.0;
    }

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

Beginning of main

```
    hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_B, h_B, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_C, h_C, bytes, hipMemcpyHostToDevice);

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

    vec_add<<<blk_in_grid,thr_per_blk>>>(d_A, d_B, d_C, N);

    hipMemcpy(h_C, d_C, bytes, hipMemcpyDeviceToHost);

    double sum       = 0.0;
    double tolerance = 1.0e-14;

    for(int i=0; i<N; i++){
        sum = sum + h_C[i];
    }

    if( fabs( (sum / N) - 1.0 ) > tolerance ){
        printf("Error: Sum/N = %0.2f instead of ~1.0\n", sum / N);
        exit(1);
    }

    free(h_A);
    free(h_B);
    free(h_C);

    hipFree(d_A);
    hipFree(d_B);
    hipFree(d_C);

    printf("__SUCCESS__\n");

    return 0;
}
```

End of main

Simple Vector Addition Example

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

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

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

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

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

    for(int i=0; i<N; i++){
        h_A[i] = sin(i) * sin(i);
        h_B[i] = cos(i) * cos(i);
        h_C[i] = 0.0;
    }

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

Number of elements and bytes in array

```
    hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_B, h_B, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_C, h_C, bytes, hipMemcpyHostToDevice);

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

    vec_add<<<blk_in_grid,thr_per_blk>>>(d_A, d_B, d_C, N);

    hipMemcpy(h_C, d_C, bytes, hipMemcpyDeviceToHost);

    double sum       = 0.0;
    double tolerance = 1.0e-14;

    for(int i=0; i<N; i++){
        sum = sum + h_C[i];
    }

    if( fabs( (sum / N) - 1.0 ) > tolerance ){
        printf("Error: Sum/N = %0.2f instead of ~1.0\n", sum / N);
        exit(1);
    }

    free(h_A);
    free(h_B);
    free(h_C);

    hipFree(d_A);
    hipFree(d_B);
    hipFree(d_C);

    printf("__SUCCESS__\n");

    return 0;
}
```

Simple Vector Addition Example

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

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

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

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

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

    for(int i=0; i<N; i++){
        h_A[i] = sin(i) * sin(i);
        h_B[i] = cos(i) * cos(i);
        h_C[i] = 0.0;
    }

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

Allocate memory for host arrays

```
    hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_B, h_B, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_C, h_C, bytes, hipMemcpyHostToDevice);

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

    vec_add<<<blk_in_grid,thr_per_blk>>>(d_A, d_B, d_C, N);

    hipMemcpy(h_C, d_C, bytes, hipMemcpyDeviceToHost);

    double sum       = 0.0;
    double tolerance = 1.0e-14;

    for(int i=0; i<N; i++){
        sum = sum + h_C[i];
    }

    if( fabs( (sum / N) - 1.0 ) > tolerance ){
        printf("Error: Sum/N = %0.2f instead of ~1.0\n", sum / N);
        exit(1);
    }

    free(h_A);
    free(h_B);
    free(h_C);

    hipFree(d_A);
    hipFree(d_B);
    hipFree(d_C);

    printf("__SUCCESS__\n");

    return 0;
}
```

Simple Vector Addition Example

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

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

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

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

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

    for(int i=0; i<N; i++){
        h_A[i] = sin(i) * sin(i);
        h_B[i] = cos(i) * cos(i);
        h_C[i] = 0.0;
    }

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

Initialize host arrays

```
    hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_B, h_B, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_C, h_C, bytes, hipMemcpyHostToDevice);

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

    vec_add<<<blk_in_grid,thr_per_blk>>>(d_A, d_B, d_C, N);

    hipMemcpy(h_C, d_C, bytes, hipMemcpyDeviceToHost);

    double sum       = 0.0;
    double tolerance = 1.0e-14;

    for(int i=0; i<N; i++){
        sum = sum + h_C[i];
    }

    if( fabs( (sum / N) - 1.0 ) > tolerance ){
        printf("Error: Sum/N = %0.2f instead of ~1.0\n", sum / N);
        exit(1);
    }

    free(h_A);
    free(h_B);
    free(h_C);

    hipFree(d_A);
    hipFree(d_B);
    hipFree(d_C);

    printf("__SUCCESS__\n");

    return 0;
}
```

Simple Vector Addition Example

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

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

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

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

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

    for(int i=0; i<N; i++){
        h_A[i] = sin(i) * sin(i);
        h_B[i] = cos(i) * cos(i);
        h_C[i] = 0.0;
    }

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

Allocate memory for device arrays

```
    hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_B, h_B, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_C, h_C, bytes, hipMemcpyHostToDevice);

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

    vec_add<<<blk_in_grid,thr_per_blk>>>(d_A, d_B, d_C, N);

    hipMemcpy(h_C, d_C, bytes, hipMemcpyDeviceToHost);

    double sum       = 0.0;
    double tolerance = 1.0e-14;

    for(int i=0; i<N; i++){
        sum = sum + h_C[i];
    }

    if( fabs( (sum / N) - 1.0 ) > tolerance ){
        printf("Error: Sum/N = %0.2f instead of ~1.0\n", sum / N);
        exit(1);
    }

    free(h_A);
    free(h_B);
    free(h_C);

    hipFree(d_A);
    hipFree(d_B);
    hipFree(d_C);

    printf("__SUCCESS__\n");

    return 0;
}
```

Simple Vector Addition Example

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

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

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

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

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

    for(int i=0; i<N; i++){
        h_A[i] = sin(i) * sin(i);
        h_B[i] = cos(i) * cos(i);
        h_C[i] = 0.0;
    }

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

Copy data from host arrays to device arrays

```
    hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_B, h_B, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_C, h_C, bytes, hipMemcpyHostToDevice);

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

    vec_add<<<blk_in_grid,thr_per_blk>>>(d_A, d_B, d_C, N);

    hipMemcpy(h_C, d_C, bytes, hipMemcpyDeviceToHost);

    double sum       = 0.0;
    double tolerance = 1.0e-14;

    for(int i=0; i<N; i++){
        sum = sum + h_C[i];
    }

    if( fabs( (sum / N) - 1.0 ) > tolerance ){
        printf("Error: Sum/N = %0.2f instead of ~1.0\n", sum / N);
        exit(1);
    }

    free(h_A);
    free(h_B);
    free(h_C);

    hipFree(d_A);
    hipFree(d_B);
    hipFree(d_C);

    printf("__SUCCESS__\n");

    return 0;
}
```

Simple Vector Addition Example

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

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

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

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

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

    for(int i=0; i<N; i++){
        h_A[i] = sin(i) * sin(i);
        h_B[i] = cos(i) * cos(i);
        h_C[i] = 0.0;
    }

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

Launch kernel function

```
    hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_B, h_B, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_C, h_C, bytes, hipMemcpyHostToDevice);

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

    vec_add<<<blk_in_grid,thr_per_blk>>>(d_A, d_B, d_C, N);

    hipMemcpy(h_C, d_C, bytes, hipMemcpyDeviceToHost);

    double sum       = 0.0;
    double tolerance = 1.0e-14;

    for(int i=0; i<N; i++){
        sum = sum + h_C[i];
    }

    if( fabs( (sum / N) - 1.0 ) > tolerance ){
        printf("Error: Sum/N = %0.2f instead of ~1.0\n", sum / N);
        exit(1);
    }

    free(h_A);
    free(h_B);
    free(h_C);

    hipFree(d_A);
    hipFree(d_B);
    hipFree(d_C);

    printf("__SUCCESS__\n");

    return 0;
}
```

Simple Vector Addition Example

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

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

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

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

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

    for(int i=0; i<N; i++){
        h_A[i] = sin(i) * sin(i);
        h_B[i] = cos(i) * cos(i);
        h_C[i] = 0.0;
    }

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

Send data from device array to host array

```
    hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_B, h_B, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_C, h_C, bytes, hipMemcpyHostToDevice);

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

    vec_add<<<blk_in_grid,thr_per_blk>>>(d_A, d_B, d_C, N);

    hipMemcpy(h_C, d_C, bytes, hipMemcpyDeviceToHost);

    double sum       = 0.0;
    double tolerance = 1.0e-14;

    for(int i=0; i<N; i++){
        sum = sum + h_C[i];
    }

    if( fabs( (sum / N) - 1.0 ) > tolerance ){
        printf("Error: Sum/N = %0.2f instead of ~1.0\n", sum / N);
        exit(1);
    }

    free(h_A);
    free(h_B);
    free(h_C);

    hipFree(d_A);
    hipFree(d_B);
    hipFree(d_C);

    printf("__SUCCESS__\n");

    return 0;
}
```

Simple Vector Addition Example

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

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

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

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

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

    for(int i=0; i<N; i++){
        h_A[i] = sin(i) * sin(i);
        h_B[i] = cos(i) * cos(i);
        h_C[i] = 0.0;
    }

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

Check for correctness

```
    hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_B, h_B, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_C, h_C, bytes, hipMemcpyHostToDevice);

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

    vec_add<<<blk_in_grid,thr_per_blk>>>(d_A, d_B, d_C, N);

    hipMemcpy(h_C, d_C, bytes, hipMemcpyDeviceToHost);

    double sum       = 0.0;
    double tolerance = 1.0e-14;

    for(int i=0; i<N; i++){
        sum = sum + h_C[i];
    }

    if( fabs( (sum / N) - 1.0 ) > tolerance ){
        printf("Error: Sum/N = %0.2f instead of ~1.0\n", sum / N);
        exit(1);
    }

    free(h_A);
    free(h_B);
    free(h_C);

    hipFree(d_A);
    hipFree(d_B);
    hipFree(d_C);

    printf("__SUCCESS__\n");

    return 0;
}
```

Simple Vector Addition Example

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

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

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

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

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

    for(int i=0; i<N; i++){
        h_A[i] = sin(i) * sin(i);
        h_B[i] = cos(i) * cos(i);
        h_C[i] = 0.0;
    }

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

```
    hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_B, h_B, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_C, h_C, bytes, hipMemcpyHostToDevice);

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

    vec_add<<<blk_in_grid,thr_per_blk>>>(d_A, d_B, d_C, N);

    hipMemcpy(h_C, d_C, bytes, hipMemcpyDeviceToHost);

    double sum       = 0.0;
    double tolerance = 1.0e-14;

    for(int i=0; i<N; i++){
        sum = sum + h_C[i];
    }

    if( fabs( (sum / N) - 1.0 ) > tolerance ){
        printf("Error: Sum/N = %0.2f instead of ~1.0\n", sum / N);
        exit(1);
    }

    free(h_A);
    free(h_B);
    free(h_C);

    hipFree(d_A);
    hipFree(d_B);
    hipFree(d_C);

    printf("__SUCCESS__\n");

    return 0;
}
```

Deallocate host memory

Simple Vector Addition Example

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

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

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

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

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

    for(int i=0; i<N; i++){
        h_A[i] = sin(i) * sin(i);
        h_B[i] = cos(i) * cos(i);
        h_C[i] = 0.0;
    }

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

```
    hipMemcpy(d_A, h_A, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_B, h_B, bytes, hipMemcpyHostToDevice);
    hipMemcpy(d_C, h_C, bytes, hipMemcpyHostToDevice);

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

    vec_add<<<blk_in_grid,thr_per_blk>>>(d_A, d_B, d_C, N);

    hipMemcpy(h_C, d_C, bytes, hipMemcpyDeviceToHost);

    double sum       = 0.0;
    double tolerance = 1.0e-14;

    for(int i=0; i<N; i++){
        sum = sum + h_C[i];
    }

    if( fabs( (sum / N) - 1.0 ) > tolerance ){
        printf("Error: Sum/N = %0.2f instead of ~1.0\n", sum / N);
        exit(1);
    }

    free(h_A);
    free(h_B);
    free(h_C);

    hipFree(d_A);
    hipFree(d_B);
    hipFree(d_C);

    printf("__SUCCESS__\n");

    return 0;
}
```

Deallocate device memory

Anatomy of a HIP Vector Addition Kernel

HIP kernel arguments
- GPU buffers allocated w/ `hipMalloc`
- `n` is an `int` passed by value

Indicates this is a HIP kernel function

HIP kernels do not return anything

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

Define global thread ID

Ensure we do not access memory that does not belong to us

https://rocm.docs.amd.com/projects/HIP/en/latest/reference/kernel_language.html

HIP Kernels – Global Thread IDs

```
blockDim.x  = Number of threads in block
blockIdx.x  = Block ID within the grid
threadIdx.x = Thread ID within a block
```

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

How does this define a global ID?

Ex: 16-element array

```
int N           = 16;
int thr_per_blk = 4;
int blk_in_grid = ceil( float(N) / thr_per_blk ); (= 4)
```

A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}	A_{15}

C/C++ array with 16 elements

0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
Block 0				Block 1				Block 2				Block 3			

1D HIP grid with four blocks, each with four threads

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

e.g., if I am local thread ID 3 in block 2, my global ID is:

```
int id = blockDim.x * blockIdx.x + threadIdx.x;
       = 4          * 2          + 3
       = 11
```

HIP Kernels – Global Thread IDs

```
blockDim.x  = Number of threads in block
blockIdx.x  = Block ID within the grid
threadIdx.x = Thread ID within a block
```

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

Why is conditional needed?

Ex: 15-element array

```
int N           = 15;
int thr_per_blk = 4;
int blk_in_grid = ceil( float(N) / thr_per_blk ); (= 4)
```

A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	A_{14}

C/C++ array with 15 elements

0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
Block 0				Block 1				Block 2				Block 3			

1D HIP grid with four blocks, each with four threads
(still 16 threads needed to cover the 15 elements)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Conditional needed so thread 15 does not access memory beyond end of array.

How to Launch a HIP Kernel

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

Kernel function arguments

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

/* Launch vector addition kernel */
vector_addition<<<blk_in_grid, thr_per_blk>>>(d_A, d_B, d_C, N);
```

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

Compiling and Running Basic HIP Code

HIP is a C/C++ extension which requires a compiler to convert high-level code into machine code.

`hipcc` is a compiler driver that calls `clang` or `nvcc` (depending on the target) and passes appropriate include and library options for the target architecture.

```
$ ls
vector_addition.cpp

$ hipcc vector_addition.cpp -o vector_addition

$ ls
vector_addition.cpp vector_addition

$ ./vector_addition
__SUCCESS__
```

Or use a `Makefile`

Or use batch script + job launcher

HIP Error Checking

Concepts demonstrated by:
`introduction_to_hip/examples/02_vector_addition_error_check`

Additional HIP API Calls for Error Checking

Check for last HIP error

HIP return status

No arguments

```
hipError_t hipGetLastError(void)
```

Sync across entire device

HIP return status

No arguments

```
hipError_t hipDeviceSynchronize(void)
```

HIP 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…

HIP 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.

```
/* 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) );

    ...
}
```

HIP Error Checking – Kernel 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 */
gpuCheck ( hipDeviceSynchronize() );

...
```

Why are kernel errors handled differently?

- HIP kernels do not have a return value.
- In addition, 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) are "known" immediately – since there either was or was not a kernel launch error.
  - → To handle these errors, we can 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
  - → To handle these errors, we must synchronize the device to make sure we catch these errors (`hipDeviceSychronize()`).
  - → These are known as asynchronous errors

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

hipEvents for Timing

Concepts demonstrated by:
`introduction_to_hip/examples/03_vector_addition_timers`

Timing HIP Codes with hipEvents

HIP events can be placed into a HIP stream (sequence of GPU operations carried out in-order). For example, timing operations on the GPU.

```
/* TIMER: Create start & stop event objects for timing */
hipEvent_t start, stop;
gpuCheck( hipEventCreate(&start) );
gpuCheck( hipEventCreate(&stop) );

/* TIMER: Start event timer for vector addition kernel */
gpuCheck( hipEventRecord(start, NULL) );

/* Launch vector addition kernel */
vector_addition<<<blk_in_grid, thr_per_blk>>>(d_A, d_B, d_C, N);

/* TIMER: Stop event timer for vector addition kernel */
gpuCheck( hipEventRecord(stop, NULL) );

/* TIMER: Calculate time (in ms) for vector addition kernel */
float kernel_time;
gpuCheck( hipEventSynchronize(stop) );
gpuCheck( hipEventElapsedTime(&kernel_time, start, stop) );

printf("Kernel time (ms) : %0.2f\n", kernel_time);
```

Demo 1

Show basic workflow to compile a code, submit to the batch queue, and view the results.

Motivate the need for error checking and show its benefits.

Show basic HIP event timer.

Codes used:

`introduction_to_hip/examples/01_vector_addition`
`introduction_to_hip/examples/02_vector_addition_error_check`
`introduction_to_hip/examples/03_vector_addition_timers`

Hands-On Session 1
(Example 01 + Exercises 01-03)

Follow example on next slide.

Then follow the README.md in exercise directories 01-03.

NOTE: This hands-on session will lead into the break.

Example: Vector Addition

If you have not cloned the `introduction_to_hip` repository, please see Slide 5 and clone it now.

For the `01_vector_addition` example, please issue the commands below to understand the steps needed to i) compile the code, ii) submit the job to the batch queue, and iii) view the results.

```
[~]$ cd ~/introduction_to_hip/examples/01_vector_addition

[01_vector_addition]$ make
hipcc --offload-arch=gfx90a -c vector_addition.cpp
hipcc --offload-arch=gfx90a vector_addition.o -o vector_addition

[01_vector_addition]$ sbatch submit.sh

[01_vector_addition]$ cat slurm-<jobid>.out
```

2D Grids

Concepts demonstrated by:
`introduction_to_hip/examples/08_matrix_addition`

1D Indexing of 2D Array

A is N×N matrix (or 2D array), where N = 12

```
for (int row=0; row<N; row++){
    for (int col=0; col<N; col++){
        int index = N * row + col;
    }
}
```

```
row = 0,  col = 0,  index = 12 * 0 + 0   = 0
row = 0,  col = 1,  index = 12 * 0 + 1   = 1
row = 0, col = 11,  index = 12 * 0 + 11  = 11
row = 1,  col = 0,  index = 12 * 1 + 0   = 12
row = 1, col = 11,  index = 12 * 1 + 11  = 23
row = 11, col = 0,  index = 12 * 11 + 0  = 132
row = 11, col = 11, index = 12 * 11 + 11 = 143
```

$A_{row,col}$

columns →

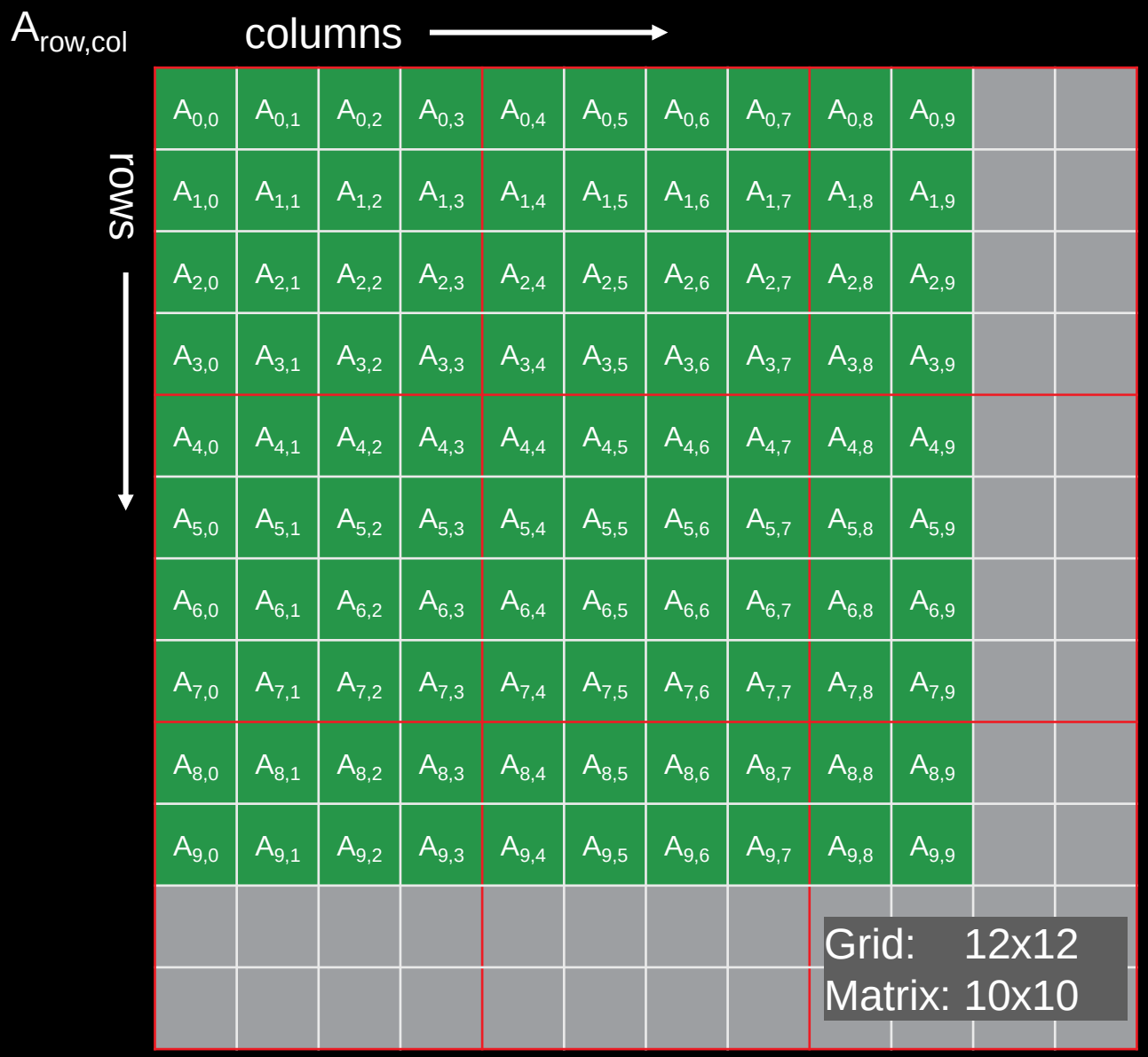
rows ↓

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$	$A_{0,5}$	$A_{0,6}$	$A_{0,7}$	$A_{0,8}$	$A_{0,9}$	$A_{0,10}$	$A_{0,11}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	$A_{1,5}$	$A_{1,6}$	$A_{1,7}$	$A_{1,8}$	$A_{1,9}$	$A_{1,10}$	$A_{1,11}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	$A_{2,5}$	$A_{2,6}$	$A_{2,7}$	$A_{2,8}$	$A_{2,9}$	$A_{2,10}$	$A_{2,11}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$	$A_{3,5}$	$A_{3,6}$	$A_{3,7}$	$A_{3,8}$	$A_{3,9}$	$A_{3,10}$	$A_{3,11}$
$A_{4,0}$	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$	$A_{4,5}$	$A_{4,6}$	$A_{4,7}$	$A_{4,8}$	$A_{4,9}$	$A_{4,10}$	$A_{4,11}$
$A_{5,0}$	$A_{5,1}$	$A_{5,2}$	$A_{5,3}$	$A_{5,4}$	$A_{5,5}$	$A_{5,6}$	$A_{5,7}$	$A_{5,8}$	$A_{5,9}$	$A_{5,10}$	$A_{5,11}$
$A_{6,0}$	$A_{6,1}$	$A_{6,2}$	$A_{6,3}$	$A_{6,4}$	$A_{6,5}$	$A_{6,6}$	$A_{6,7}$	$A_{6,8}$	$A_{6,9}$	$A_{6,10}$	$A_{6,11}$
$A_{7,0}$	$A_{7,1}$	$A_{7,2}$	$A_{7,3}$	$A_{7,4}$	$A_{7,5}$	$A_{7,6}$	$A_{7,7}$	$A_{7,8}$	$A_{7,9}$	$A_{7,10}$	$A_{7,11}$
$A_{8,0}$	$A_{8,1}$	$A_{8,2}$	$A_{8,3}$	$A_{8,4}$	$A_{8,5}$	$A_{8,6}$	$A_{8,7}$	$A_{8,8}$	$A_{8,9}$	$A_{8,10}$	$A_{8,11}$
$A_{9,0}$	$A_{9,1}$	$A_{9,2}$	$A_{9,3}$	$A_{9,4}$	$A_{9,5}$	$A_{9,6}$	$A_{9,7}$	$A_{9,8}$	$A_{9,9}$	$A_{9,10}$	$A_{9,11}$
$A_{10,0}$	$A_{10,1}$	$A_{10,2}$	$A_{10,3}$	$A_{10,4}$	$A_{10,5}$	$A_{10,6}$	$A_{10,7}$	$A_{10,8}$	$A_{10,9}$	$A_{10,10}$	$A_{10,11}$
$A_{11,0}$	$A_{11,1}$	$A_{11,2}$	$A_{11,3}$	$A_{11,4}$	$A_{11,5}$	$A_{11,6}$	$A_{11,7}$	$A_{11,8}$	$A_{11,9}$	$A_{11,10}$	$A_{11,11}$

Matrix Addition

```
n = 10

/* dim3 is a c struct with member variables x, y, z */
dim3 thr_per_blk( 4, 4, 1 );
dim3 blk_in_grid( ceil( float(N) / thr_per_blk.x),
                  ceil(float(N) / thr_per_blk.y),
                  1 );

        → blk_in_grid( 3, 3, 1 )
```

```
__global__ void matrix_addition(double *A,
                                double *B,
                                double *C,
                                int n)
{
    int col = blockDim.x * blockIdx.x + threadIdx.x;
    int row = blockDim.y * blockIdx.y + threadIdx.y;

    if (col < n && row < n){
        int index = n * row + col;
        C[index] = A[index] + B[index];
    }
}
```



`index` is the 1D index of our 2D array, where `col, row < n`

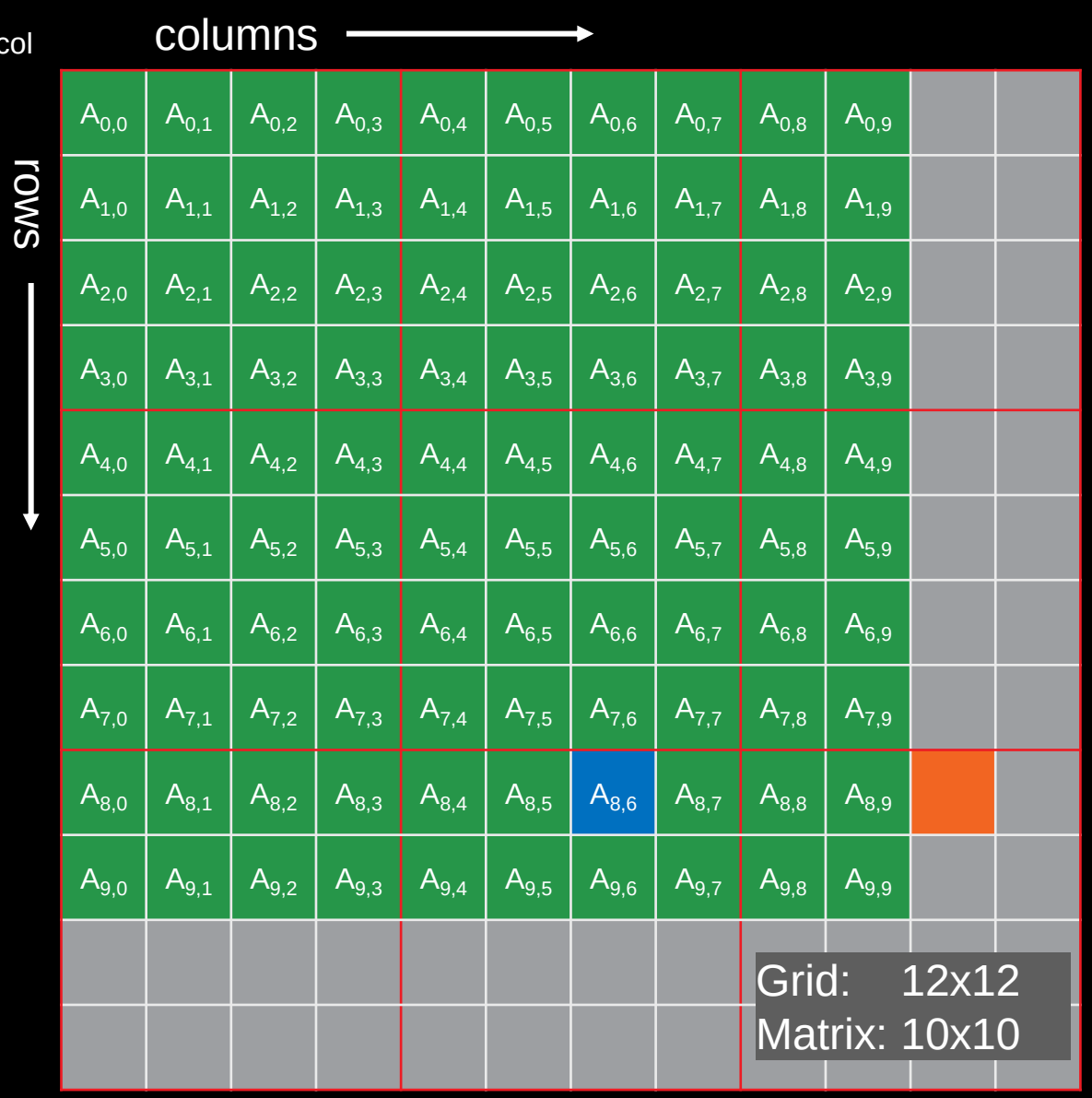
```
blockDim.x  = blockDim.y = 4
blockIdx.x  = {0-2}, blockIdx.y  = {0-2}
threadIdx.x = {0-3}, threadIdx.y = {0-3}
```

Matrix Addition

```
n = 10
/* dim3 is a c struct with member variables x, y, z */
dim3 thr_per_blk( 4, 4, 1 );
dim3 blk_in_grid( 3, 3, 1 )
```

```
__global__ void matrix_addition(double *A,
                                double *B,
                                double *C,
                                int n)
{
    int col = blockDim.x * blockIdx.x + threadIdx.x;
    int row = blockDim.y * blockIdx.y + threadIdx.y;

    if (col < n && row < n){
        int index = n * row + col;
        C[index] = A[index] + B[index];
    }
}
```

```
blockIdx.x = 1, threadIdx.x = 2, col = 4 * 1 + 2 = 6
blockIdx.y = 2, threadIdx.y = 0, row = 4 * 2 + 0 = 8
index      = 10 * 8 + 6 = 86 ✓

blockIdx.x = 2, threadIdx.x = 2, col = 4 * 2 + 2 = 10
blockIdx.y = 2, threadIdx.y = 0, row = 4 * 2 + 0 = 8
index      = 10 * 8 + 10 = 90 ✗
```




```
blockDim.x  = blockDim.y = 4
blockIdx.x  = {0-2}, blockIdx.y  = {0-2}
threadIdx.x = {0-3}, threadIdx.y = {0-3}
```

Shared Memory

Concepts demonstrated by:
`introduction_to_hip/examples/09_vector_addition_shared`

Vector Addition with Shared Memory

Shared memory is allocated to thread blocks, so all threads within a block can access the same shared memory.

```
/* ---------------------------------------------------
Vector addition kernel
--------------------------------------------------- */
__global__ void vector_addition(double *A, double *B, double *C)
{
    __shared__ double s_A[THREADS_PER_BLOCK];
    __shared__ double s_B[THREADS_PER_BLOCK];

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

    if (id < N){
        s_A[lid] = A[id];
        s_B[lid] = B[id];
    }

    __syncthreads();

    if (id < N){
        double temp = s_A[lid] + s_B[lid];
        C[id] = temp;
    }
}
```

Allocate two shared memory arrays, one to hold a block's worth of the A array and one to hold a block's worth of the B array.

Vector Addition with Shared Memory

Shared memory is allocated to thread blocks, so all threads within a block can access the same shared memory.

```
/* ------------------------------------------------
Vector addition kernel
------------------------------------------------- */
__global__ void vector_addition(double *A, double *B, double *C)
{
    __shared__ double s_A[THREADS_PER_BLOCK];
    __shared__ double s_B[THREADS_PER_BLOCK];

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

    if (id < N){
        s_A[lid] = A[id];
        s_B[lid] = B[id];
    }

    __syncthreads();

    if (id < N){
        double temp = s_A[lid] + s_B[lid];
        C[id] = temp;
    }
}
```

Global thread ID

Block-local thread ID

Vector Addition with Shared Memory

Shared memory is allocated to thread blocks, so all threads within a block can access the same shared memory.

```
/* ---------------------------------------------------
Vector addition kernel
--------------------------------------------------- */
__global__ void vector_addition(double *A, double *B, double *C)
{
    __shared__ double s_A[THREADS_PER_BLOCK];
    __shared__ double s_B[THREADS_PER_BLOCK];

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

    if (id < N){
        s_A[lid] = A[id];
        s_B[lid] = B[id];
    }

    __syncthreads();

    if (id < N){
        double temp = s_A[lid] + s_B[lid];
        C[id] = temp;
    }
}
```

If we are in the array, copy data from global GPU memory to block-local shared memory.

Vector Addition with Shared Memory

Shared memory is allocated to thread blocks, so all threads within a block can access the same shared memory.

```
/* ---------------------------------------------------
Vector addition kernel
--------------------------------------------------- */
__global__ void vector_addition(double *A, double *B, double *C)
{
    __shared__ double s_A[THREADS_PER_BLOCK];
    __shared__ double s_B[THREADS_PER_BLOCK];

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

    if (id < N){
        s_A[lid] = A[id];
        s_B[lid] = B[id];
    }

    __syncthreads();

    if (id < N){
        double temp = s_A[lid] + s_B[lid];
        C[id] = temp;
    }
}
```

Sync to ensure all threads have finished copying their data.

Vector Addition with Shared Memory

Shared memory is allocated to thread blocks, so all threads within a block can access the same shared memory.

```
/* ---------------------------------------------------
Vector addition kernel
--------------------------------------------------- */
__global__ void vector_addition(double *A, double *B, double *C)
{
    __shared__ double s_A[THREADS_PER_BLOCK];
    __shared__ double s_B[THREADS_PER_BLOCK];

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

    if (id < N){
        s_A[lid] = A[id];
        s_B[lid] = B[id];
    }

    __syncthreads();

    if (id < N){
        double temp = s_A[lid] + s_B[lid];
        C[id] = temp;
    }
}
```

If we are within the array, calculate the element-wise addition from shared memory.

Concurrent Kernels & Overlapping Data Transfers with GPU Compute

Concepts demonstrated by:
`introduction_to_hip/examples/13_concurrent_kernels_async_data`

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);
```

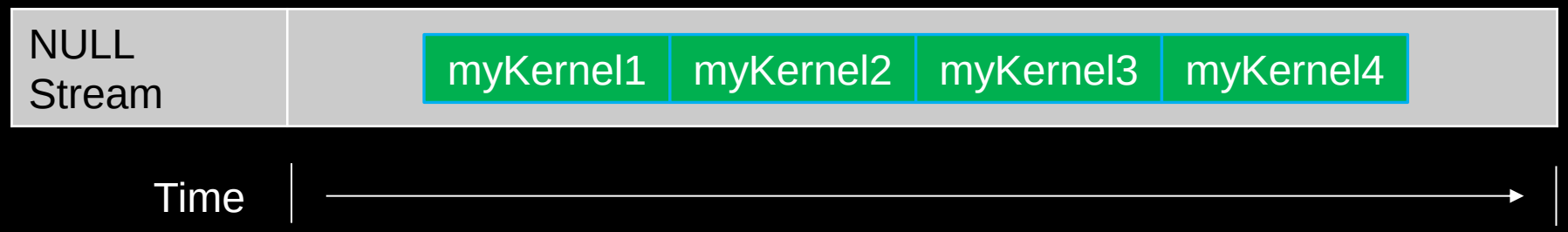
- Passing `0` or `NULL` as the `hipStream_t` argument to a function instructs the function to execute on a stream called the 'NULL Stream':
  - No task on the NULL stream will begin until **all previously enqueued tasks in all other streams have completed**.
  - Blocking calls like `hipMemcpy` run on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

Streams

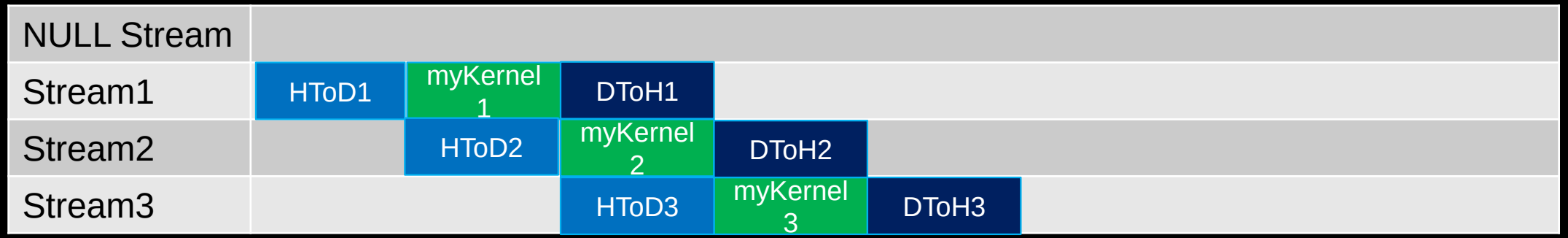
Suppose we have 3 kernels which require moving data to and from the device:

```
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’:

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

Demo 2

Show asynchronous behavior using rocprof (introduced as needed).

Rocprof Command used:

```
rocprof --stats --hip-trace --hsa-trace --sys-trace ./matrix_multiply
```

Trace viewer used to view `results.json`:

https://ui.perfetto.dev/

Code Used:

`introduction_to_hip/examples/13_concurrent_kernels_async_data`

CUDA-to-HIP Translations

Concepts demonstrated by:

`introduction_to_hip/examples/10_vector_addition_hipify`

`introduction_to_hip/examples/11_vector_addition_header_file`

hipify-ing a CUDA Code

HIP provides a hipify tool, which can be used to convert an existing CUDA code into HIP.

Use the `-examine` flag to see what *would* happen if you ran `hipify-perl` on the CUDA file.

```
$ hipify-perl -examine vector_addition.cu

[HIPIFY] info: file 'vector_addition.cu' statistics:
  CONVERTED refs count: 23
  TOTAL lines of code: 107
  WARNINGS: 0
[HIPIFY] info: CONVERTED refs by names:
  cudaDeviceSynchronize => hipDeviceSynchronize: 1
  cudaError_t => hipError_t: 1
  cudaFree => hipFree: 3
  cudaGetErrorString => hipGetErrorString: 1
  cudaGetLastError => hipGetLastError: 1
  cudaMalloc => hipMalloc: 3
  cudaMemcpy => hipMemcpy: 4
  cudaMemcpyDeviceToHost => hipMemcpyDeviceToHost: 1
  cudaMemcpyHostToDevice => hipMemcpyHostToDevice: 3
  cudaSuccess => hipSuccess: 1
```

If you are satisfied, pass the results of hipify-ing the CUDA file into a `.cpp` file

```
$ hipify-perl vector_addition.cu > vector_addition.cpp
```

NOTE:

- Only recognized CUDA calls will be translated to HIP.
- Comments and user-defined variables/macros will not be translated.

Supported CUDA APIs can be found here:
https://github.com/ROCm-Developer-Tools/HIPIFY/blob/amd-staging/docs/supported_apis.md#supported-cuda-apis

hipify documentation:
https://github.com/ROCm-Developer-Tools/HIPIFY

Running CUDA Code with a HIP Header File

Some researchers do not want to translate their code into HIP because they run on multiple systems, and some don’t have HIP installed.

① Create a header file to translate from CUDA to HIP at runtime.

```
$ cat vector_addition.h

#include "hip/hip_runtime.h"

#define cudaMalloc hipMalloc
#define cudaMemcpy hipMemcpy
#define cudaError_t hipError_t
#define cudaGetErrorString hipGetErrorString
#define cudaSuccess hipSuccess
#define cudaMemcpyHostToDevice hipMemcpyHostToDevice
#define cudaMemcpyDeviceToHost hipMemcpyDeviceToHost
#define cudaGetLastError hipGetLastError
#define cudaDeviceSynchronize hipDeviceSynchronize
#define cudaFree hipFree
```

② Then add an include statement for the header file in your CUDA code.

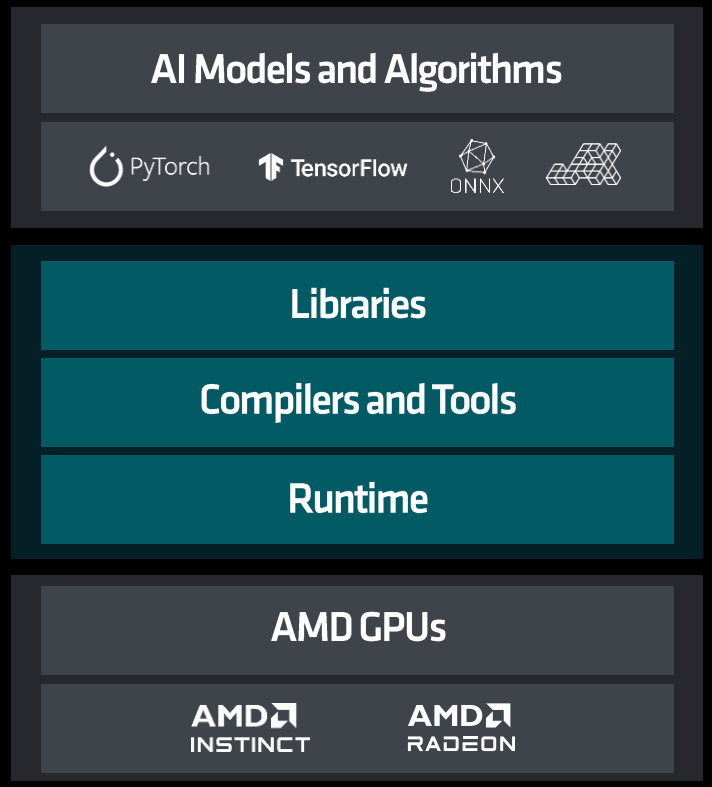
```
$ cat vector_addition.cu | head -n3

#include <stdio.h>
#include <math.h>
#include "vector_addition.h"
```

This allows you to keep your CUDA source code with only the small modification of the header include.

Hands-On Session 2
(Exercises 04-06)

AI on ROCm

AMD ROCm Software & AI Ecosystem



- ROCm offers a suite of optimizations for AI workloads and supports the broader AI software ecosystem including open frameworks, models, and tools
  - Offers dedicated libraries for machine learning, including MIOpen and MIVisionX
  - Provides upstream support for leading AI frameworks including PyTorch and TensorFlow
  - Supports a wide range of models (62K+ models running nightly) on HuggingFace that can be leveraged to develop user-specific solutions
  - Supports leading containerization tools including Docker, Singularity, Kubernetes, and Slurm to enable deployment at scale
  - Instinct powers the 1st & 5th fastest supercomputers in the world according to the latest Top500 list: https://top500.org/lists/top500/list/2024/06/
  - El Capitan at LLNL is expected to be the world's first 2 EFLOP system

AI Ecosystem Partners

databricks DeepSpeed Hugging Face ONNX OpenXLA PyTorch TensorFlow

Demo 4

Show how to use a local GPU to run an LLM locally

Ollama Demo

```
# Download Ollama binary and make it executable
$ wget https://ollama.com/download/ollama-linux-amd64 ./ollama
$ chmod +x ./ollama

# Create and activate Python environment (or however you prefer)
$ python -m venv <name-of-venv>
$ source <name-of-venv>/bin/activate

# Update pip
$ pip install --upgrade pip

# Install OpenAI package
$ pip install openai

# Start Ollama Server
OLLAMA_MODELS=<path-to-store-models> ./ollama serve 2>&1 | tee log > /dev/null &

$ python test.py
```

Ollama Demo

```
import time
from openai import OpenAI

client = OpenAI(base_url="http://localhost:11434/v1", api_key="none")

start_time = time.time()

response = client.chat.completions.create(
    model="llama3.1",
    messages=[{"role": "user", "content": "When was pizza invented?"}],
)

end_time      = time.time()
response_time = end_time - start_time

model             = response.model
message           = response.choices[0].message.content
usage             = response.usage
total_tokens      = response.usage.total_tokens
tokens_per_second = total_tokens / response_time

print(f"==============================================")
print(f"\n{message}\n")
print(f"----------------------------------------------")
print(f"Model            : {model}")
print(f"Usage            : {usage}")
print(f"Tokens per Second: {tokens_per_second}")
print(f"==============================================")
```

Additional Resources

https://rocm.docs.amd.com/projects/install-on-linux/en/latest/install/quick-start.html

Installing ROCm

Installation

```
# Update package list
sudo apt update

# Install necessary kernel headers and extra modules
sudo apt install "linux-headers-$(uname -r)" "linux-modules-extra-$(uname -r)"

# Add current user to render and video groups
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and video groups

# Download the AMD GPU installation package
wget https://repo.radeon.com/amdgpu-install/6.2/ubuntu/noble/amdgpu-install_6.2.60200-1_all.deb

# Install the AMD GPU installation package
sudo apt install ./amdgpu-install_6.2.60200-1_all.deb

# Update the package list again to include repo added by package
sudo apt update

# Install the AMD GPU drivers, ROCm, and HIP SDK
sudo apt install amdgpu-dkms rocm hiplibsdk
```

Verification

```
# Verify GPUs are visible
$ rocm-smi
========================================= ROCm System Management Interface =========================================
Device  Node  IDs              Temp    Power  Partitions          SCLK    MCLK     Fan  Perf  PwrCap  VRAM%  GPU%
              (DID,     GUID)  (Edge)  (Avg)  (Mem, Compute, ID)
====================================================================================================================
0       3     0x740f,   36799  52.0°C  44.0W  N/A, N/A, 0         800Mhz  1600Mhz  0%   auto  300.0W  0%     0%
================================================= End of ROCm SMI Log ==================================================
```

Installing ROCm-Enabled PyTorch

Run your existing PyTorch applications on top of ROCm-enabled PyTorch.

```
# Create and activate Python environment (or however you prefer)
$ python -m venv <name-of-venv>
$ source <name-of-venv>/bin/activate

# Update pip
pip install --upgrade pip

# Intall ROCm-enabled PyTorch
pip install --pre torch torchvision torchaudio –index url https://download.pytorch.org/whl/nightly/rocm6.2

# Install other packages on top of it

# Start running!
```

https://rocm.docs.amd.com/en/latest/how-to/rocm-for-ai/install.html

Things We Did Not Cover

- HIP Libraries (hipBLAS, hipSolver, etc.)
- Fortran Interfaces (hipFort)
- Profilers (rocprof, omniperf, omnitrace)
- Managed Memory
- Cooperative groups
- Writing Custom kernels for PyTorch

HW & SW Resources

Hardware Resources:

- CDNA3: https://www.amd.com/en/technologies/cdna.html
  - White Paper: https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf
- RDNA3: https://www.amd.com/en/technologies/rdna.html
- AMD Instinct & Radeon Architecture: https://rocm.docs.amd.com/en/latest/conceptual/gpu-arch.html
- Data Type Support: https://rocm.docs.amd.com/en/latest/compatibility/precision-support.html

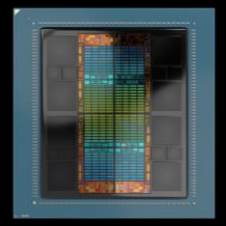
Software Resources:

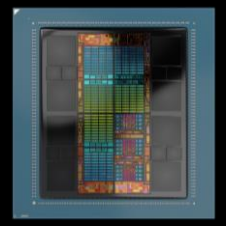
- HIP
  - Documentation: https://rocm.docs.amd.com/projects/HIP/en/latest/
  - GitHub: https://github.com/ROCm-Developer-Tools/HIP
  - AMD Lab Notes: https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-readme/
- AI
  - Webpage: https://www.amd.com/en/products/software/rocm/ai.html
  - ROCm Blogs: https://rocm.blogs.amd.com/
  - PyTorch: https://pytorch.org/docs/stable/notes/hip.html, https://pytorch.org/get-started/locally/
  - Hugging Face: https://huggingface.co/docs/optimum/en/amd/amdgpu/overview

AMD Instinct™ MI300X GPU & MI300A APU

AMD CDNA 3

AMD INSTINCT





		AMD Instinct MI300X (Up to)	MI300A (Up to)
Hardware Specifications	Memory Capacity	192GB HBM3	128GB HBM3
	Memory Bandwidth (Peak Theoretical)	~5.3 TB/s	~5.3 TB/s
	Scale-Out (Back-end) Network Bandwidth	400Gb/s Ethernet/IB	400Gb/s Ethernet/IB
	Max TDP/TBP	760W	760W
HPC Performance (Peak Theoretical)	FP64 Vector (TFLOPS)	81.7	61.3
	FP32 Vector (TFLOPS)	163.4	122.6
	FP64 Matrix (TFLOPS)	163.4	122.6
	FP32 Matrix (TFLOPS)	163.4	122.6
AI Performance (Peak Theoretical)	TF32 | TF32 Sparsity (Matrix)	653.7 | 1307.4	490.3 | 980.6
	FP16 | FP16 Sparsity (TFLOPS)	1307.4 | 2614.9	980.6 | 1961.2
	BFLOAT16 | BFLOAT16 Sparsity (TFLOPS)	1307.4 | 2614.9	980.6 | 1961.2
	FP8 | FP8 Sparsity (TFLOPS)	2614.9 | 5229.8	1961.2 | 3922.3
	INT8 | INT8 Sparsity (TOPS)	2614.9 | 5229.8	1961.2 | 3922.3

AMD Radeon™ RX 7900 XTX & Pro W7900

AMD RDNA 3

AMD RADEON

		AMD Radeon RX 7900 XTX (Up to)	AMD Radeon Pro W7900 (Up to)
Hardware Specs	Memory Capacity	24GB GDDR6	48GB GDDR6
	Memory Bandwidth (Peak Theoretical)	960 GB/s	864 GB/s
	Max TDP/TBP	355W	295W
Performance (Peak Theoretical)	FP32 (TFLOPS)	61	61
	FP16 (TFLOPS)	123	123

AI & HPC Solutions

Open
software approach

Proven
HPC & AI capabilities

Ready
broad support for
HPC & AI workflows

ENCUESTA:
COMPLETA Y PARTICIPA PARA GANAR UNA DE LAS 2 GIFT CARDS AMAZON USD 150

ENCUESTA

DISCLAIMER

AMD