CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

CSDS 451: Designing High Performant Systems for AI

Lecture 12

10/2/2025

Sanmukh Kuppannagari

sanmukh.kuppannagari@case.edu

https://sanmukh.research.st/

Case Western Reserve University

CASE SCHOOL
OF ENGINEERING
CASE WESTERN RESERVE
UNIVERSITY

Outline

- Introduction to AMD's GPU Programming Stack
- AMD GPU Hardware Model
- AMD Programming Model
- Implementation Basics

Announcements

- WA 2 due by Friday
- PA 1 will be out by Saturday Morning
- Midterm next Thursday - 10/9
  - Will cover topics till the previous lecture

Midterm Logistics

- 60 minute exam, on paper
- 15~20 Fill in the blank/True-False questions
  - With space for explanation
  - 1 point for answer
  - 2 for explanation - wrong answer with a reasonable explanation will receive partial credits
- 2 big questions

Midterm Logistics

- Open Notes
  - You can use slides
  - I suggest you create a cheat sheet: You won't have enough time to search for concepts.
- No questions during exam (Gets too distractive)
  - If a questions is ambiguous, write your assumption and solve
  - The assumption will be taken into consideration when grading
- You should not need a calculator
  - You can use your phone if you do need it

Midterm Logistics

- Exam may be lengthy (intentionally)
  - If you know the concepts well, you should be able to solve in time
  - Searching your notes or slides will incur additional time

- Plan the questions you want to solve first accordingly
  - Suggestion: Get the big questions out of the way.
  - Don’t get stuck on a question. If you can’t solve go ahead and come back
  - If you don’t know an answer, write your explanation. We will consider that while grading

CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

PA 1 Tips

- Run the example code from canvas as early as possible, to make sure things are properly setup for you. If not, reach out to the Tas
- Please review this document for best practices and this document for more information on how to schedule jobs
- We have limited number of GPUs (6 for this course), so please do not hold them
  - Use the parameter --time=1:00:00 (1 hour, 0 minutes, 0 seconds) to set reasonable time limits. You should not be needing more than a few minutes of time to run your code
  - When you get disconnected from the terminal abruptly, the gpus still remain assigned to you until the time limit exceeds. So, it is even more important to correctly set the time limit
- There will not be any extensions because you are unable to get GPU nodes closer to the deadline (which will happen), so start early and finish it as soon as you can

Study Hard for your Midterm!

Cleveland Hydranamics	Club	3:40 PM	1	00:58.370
Pittsburgh Paddlefish	Club		2	00:58.450

- Won by 80 ms
- Comparable to CNN inference times on a gaming GPU

GATEOverflow
PCSPECIALIST AMD AM5 PC (1x RTX 4090)

Model: **retinanet**
Scenario: **MultiStream**

# of Nodes: **1**
Processor: **AMD Ryzen 9 7950X 16-Core Processor**
# of Processors per node: **1**
Accelerator: **NVIDIA GeForce RTX 4090**
# of Accelerators per node: **https://github.com/mlcommons/inference_results_v5.1/tree/main/closed/GATEOverflow/code**

Result: **111.98**
Units: **Latency (ms)**

		111.98	13.35	76.45			
							109.6(
							135.9{
		225.92	22.70	54.58			
							104.9{
0.31	88,476.40	5.49	1.69	1,766.71			
		29.96	51.00	258.35			
0.51	6,607.08	70.20	8.50	119.47	7,479.35	0.13	

https://mlcommons.org/benchmarks/inference-edge/

Outline

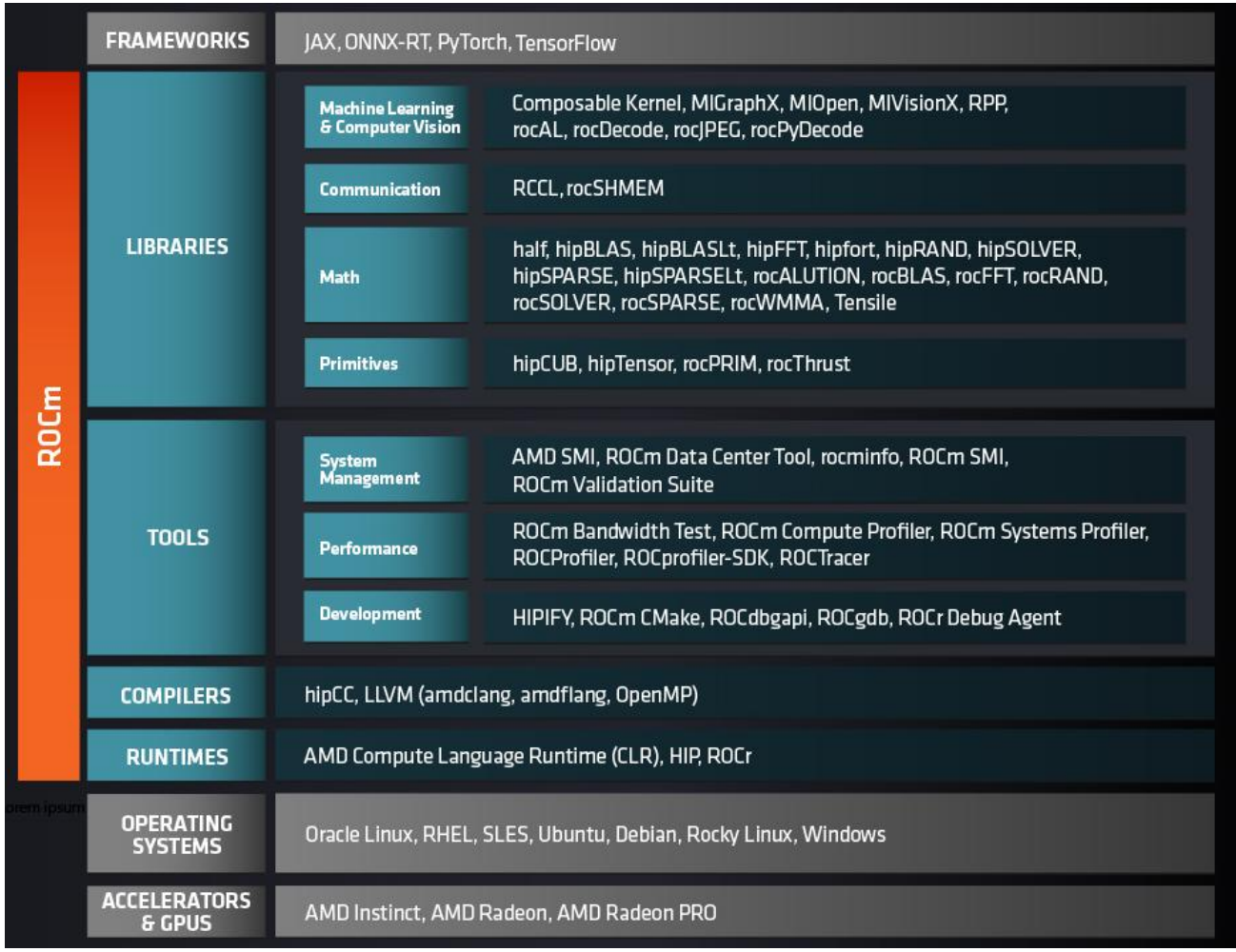
- Introduction to AMD's GPU Programming Stack
- AMD GPU Hardware Model
- AMD Programming Model
- Implementation Basics

Resources

- ROCm Documentation - https://rocm.docs.amd.com/en/latest/
- Presentation by AMD - https://www.exascaleproject.org/wp-content/uploads/2017/05/ORNL_HIP_webinar_20190606_final.pdf

ROCm (Radeon Open Compute)

- Open-source GPU computing platform by AMD, Designed for HPC, AI/ML, and scientific workloads.
- It is developed as an alternative to CUDA and supports HIP (Heterogeneous Interface for Portability).
- It includes a Compiler (LLVM-based), Runtime (HIP runtime, ROCr), Drivers & tools, Libraries (rocBLAS, MIOpen, RCCL, etc.)

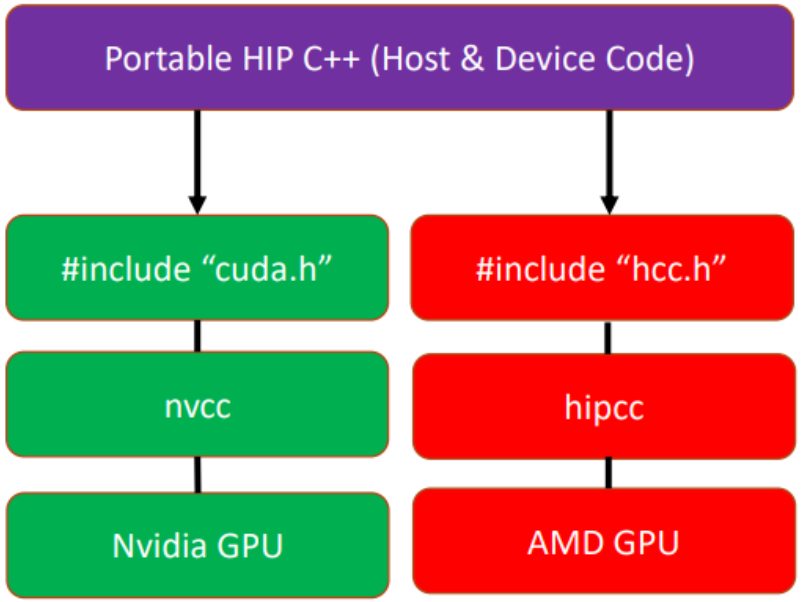


HIP (Heterogeneous-compute Interface for Portability)

- 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.
  - 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.

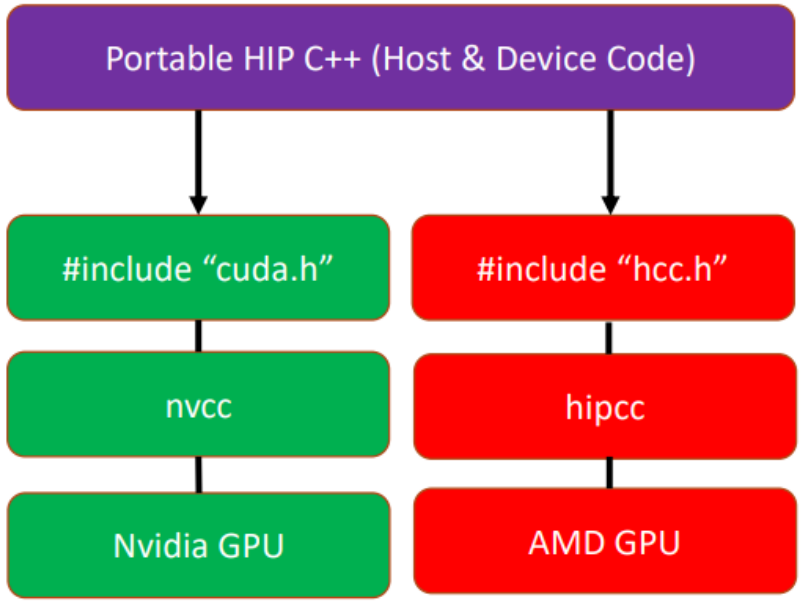
HIP - Portability & CUDA Interoperability

- Similar syntax to CUDA
  - Many CUDA constructs map directly to HIP
- Hipify tools
  - Utilities (hipify-clang, hipify-perl) to convert existing CUDA code to HIP automatically
- Single source portability
  - A single HIP codebase can run on NVIDIA (via CUDA path) or AMD GPUs depending on compilation target



HIP - Architecture & Compilation Flow

- HIP source is compiled by hipcc, which wraps Clang/LLVM or redirects to NVCC depending on target.
- On AMD target:
  HIP → LLVM → AMDGPU backend → GPU machine code.
- On NVIDIA target:
  HIP acts as a thin layer over CUDA, forwarding many calls to NVCC / CUDA runtime.
- The model sits close to hardware (few abstractions) to minimize overhead and preserve performance.



Outline

- Introduction to AMD's GPU Programming Stack
- AMD GPU Hardware Model
- AMD Programming Model
- Implementation Basics

AMD GPU Landscape

RDNA (Radeon DNA) are optimized for Gaming Applications

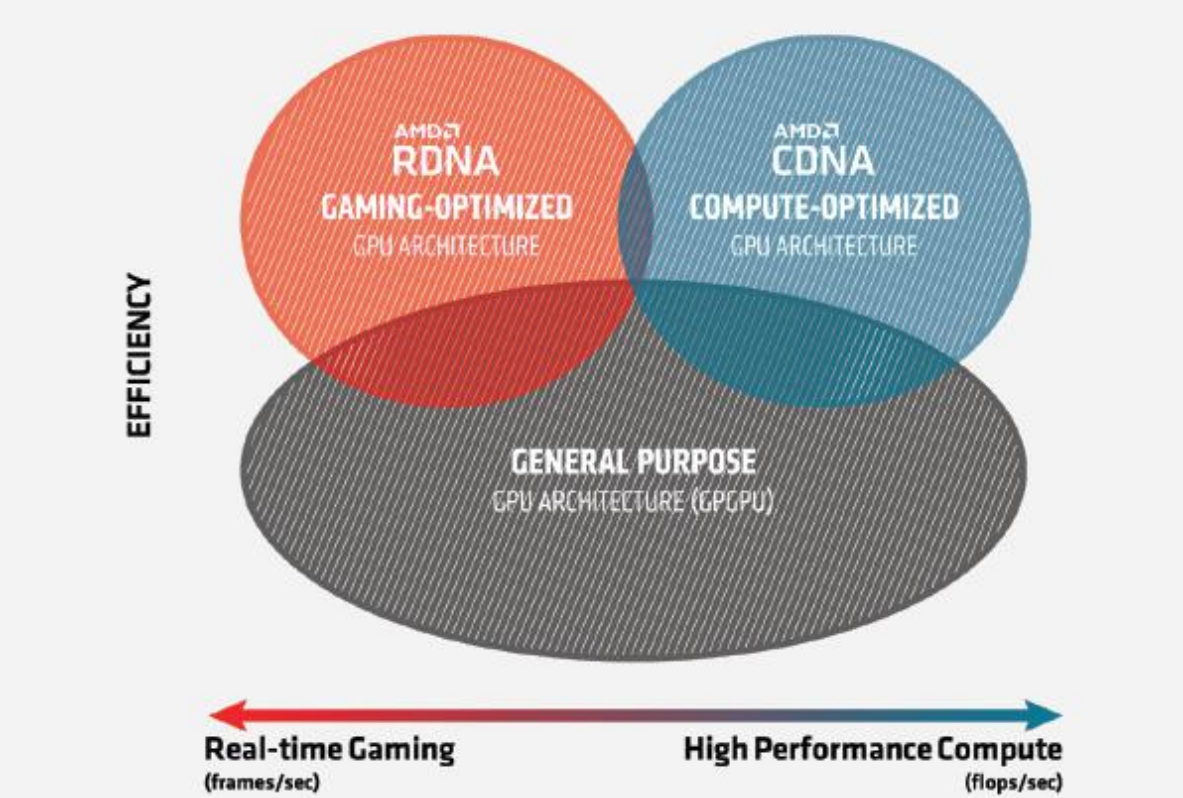
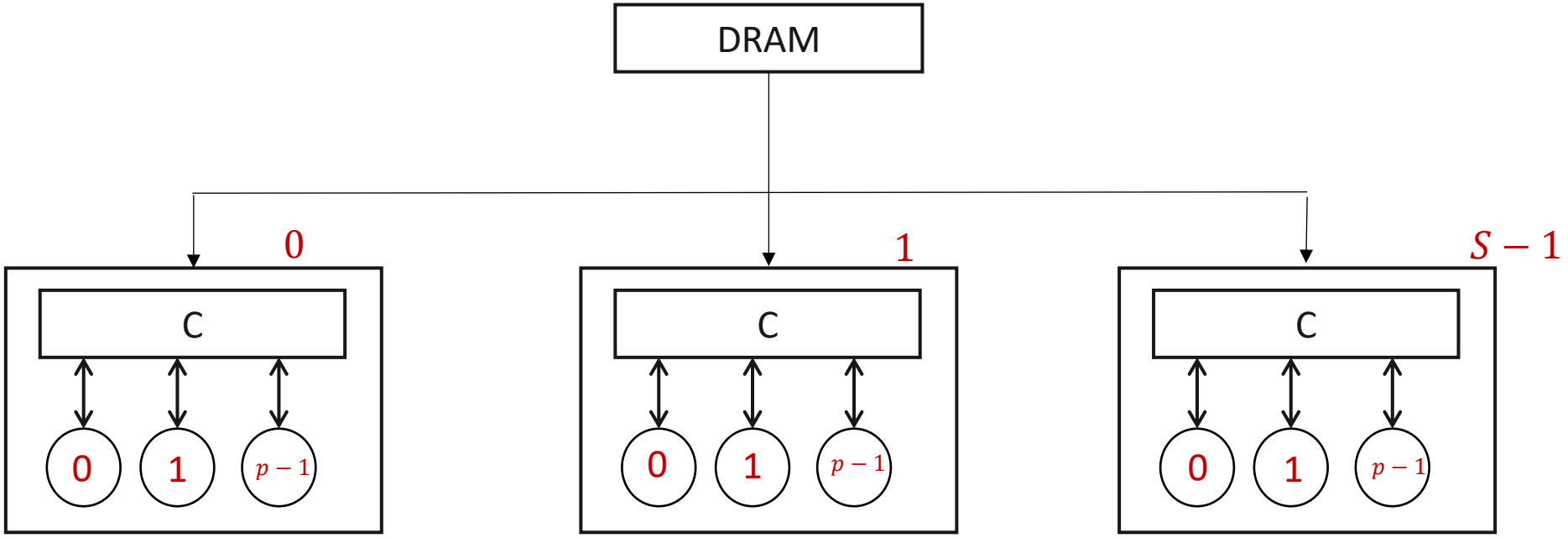


Figure 1- Domain specific architectures : optimal efficiency through domain-specific optimizations

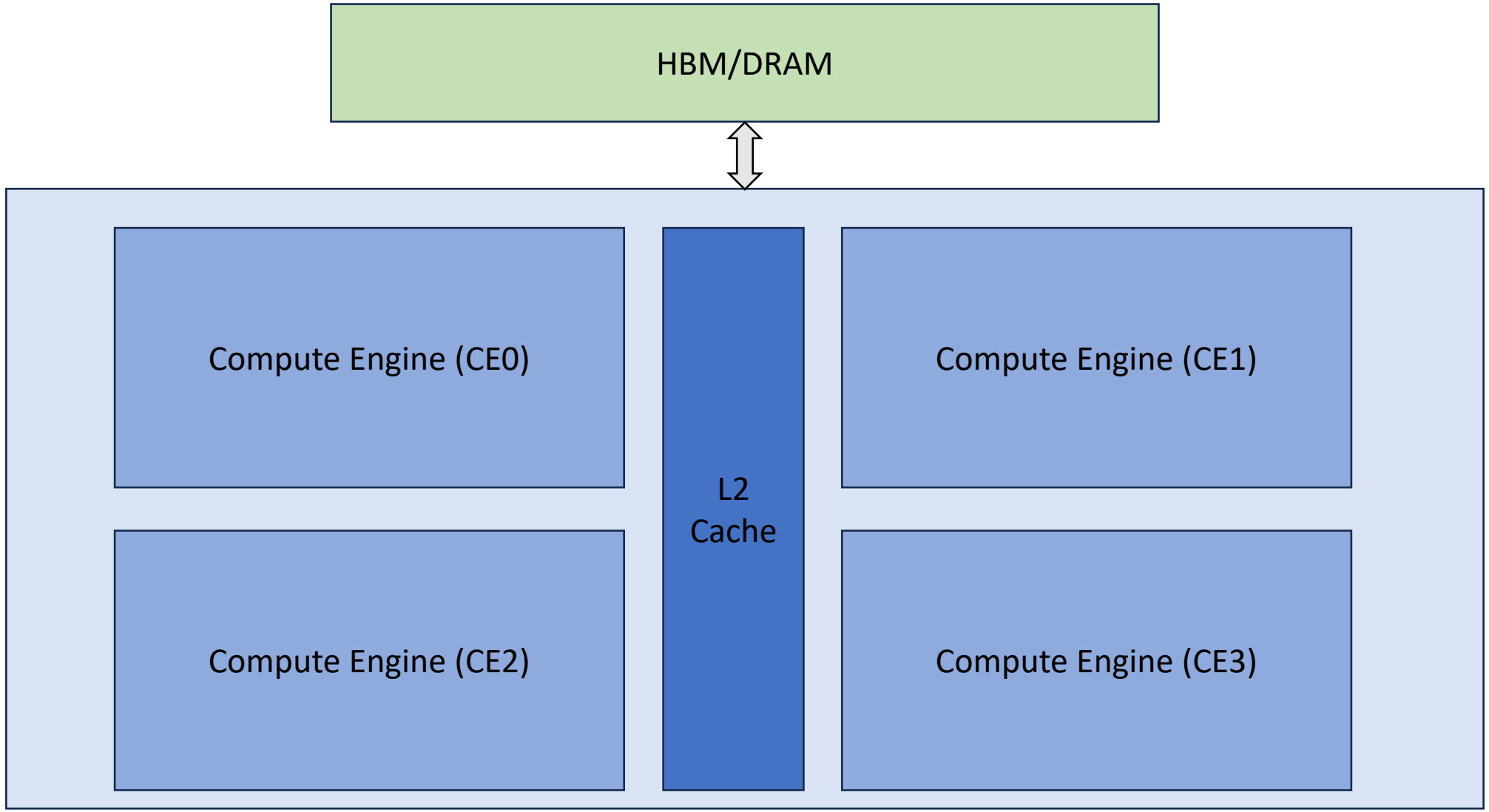
CDNA (Compute DNA) are optimized for AI/ML and Scientific Applications

Modeling GPU Architectures



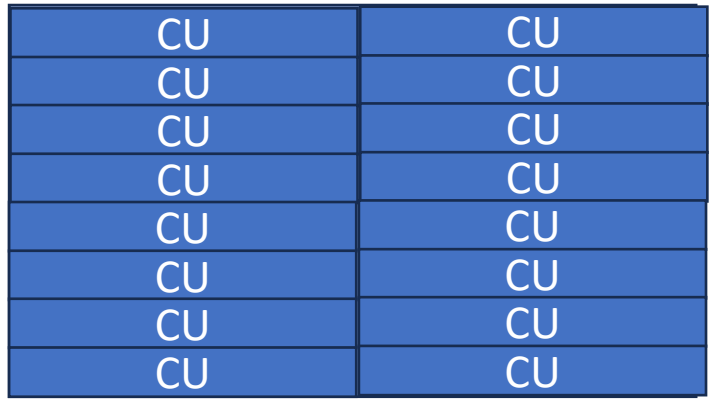
- S SMPs
- p processors per SMP

AMD CDNA GPU


HBM/DRAM
Compute Engine (CE0)
Compute Engine (CE1)
L2 Cache
Compute Engine (CE2)
Compute Engine (CE3)

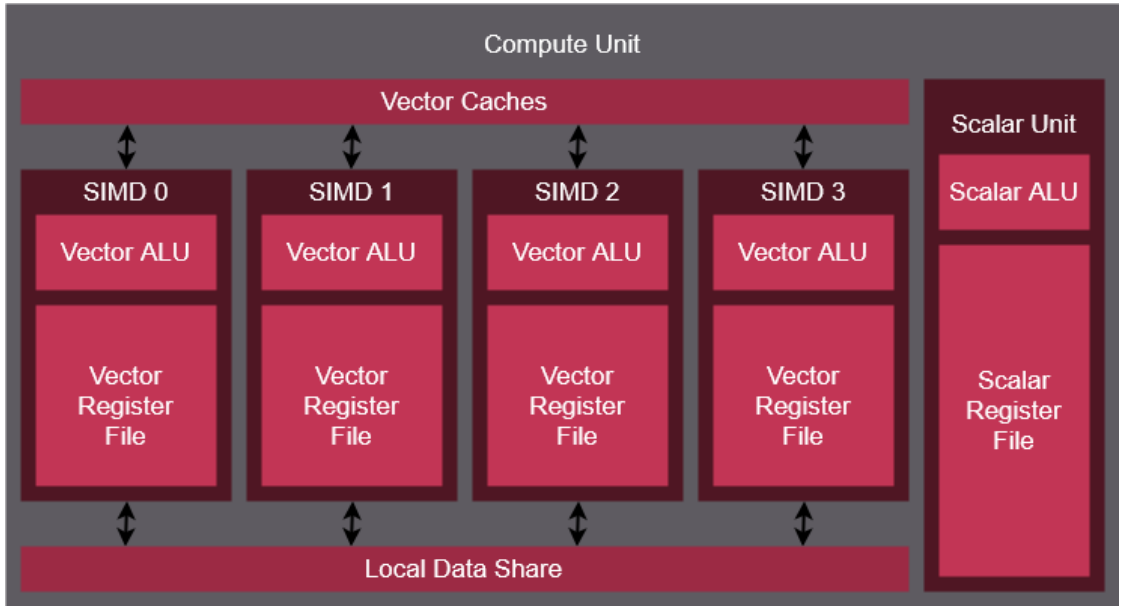
Compute Engine

- A collection of Compute Units (CU)
- We do not have any equivalent term in our GPU model as we do not require this for programming

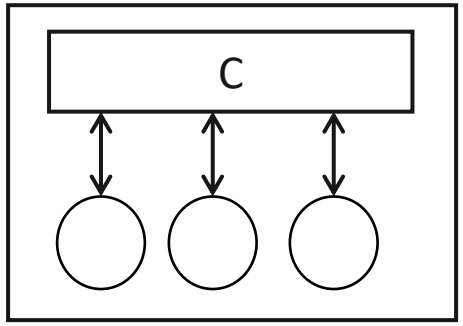


Compute Unit

- Equivalent to Blocks in our GPU model
  - Known as Streaming Multi-Processors (SMP) on Nvidia GPUs
- Consists of array of processing elements called Vector ALU (VALU)
  - Typically, 4 VALUs per CU
- AMD MI300 has 304 CUs
  - $S = 304$



=

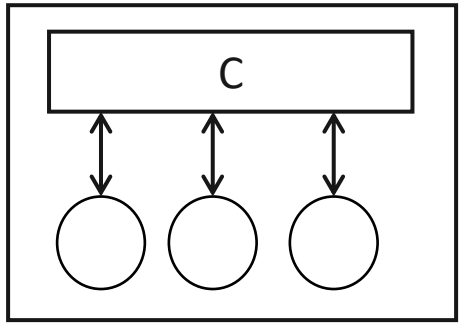


Compute Unit

- VALU: Essentially a 16-lane vector processor
  - Similar to the concept of compute cores in our GPU model
  - Each VALU contains 16 compute cores running in a lockstep manner
- So, a CU has 16 x 4 = 64 compute cores
  - $p = 64$

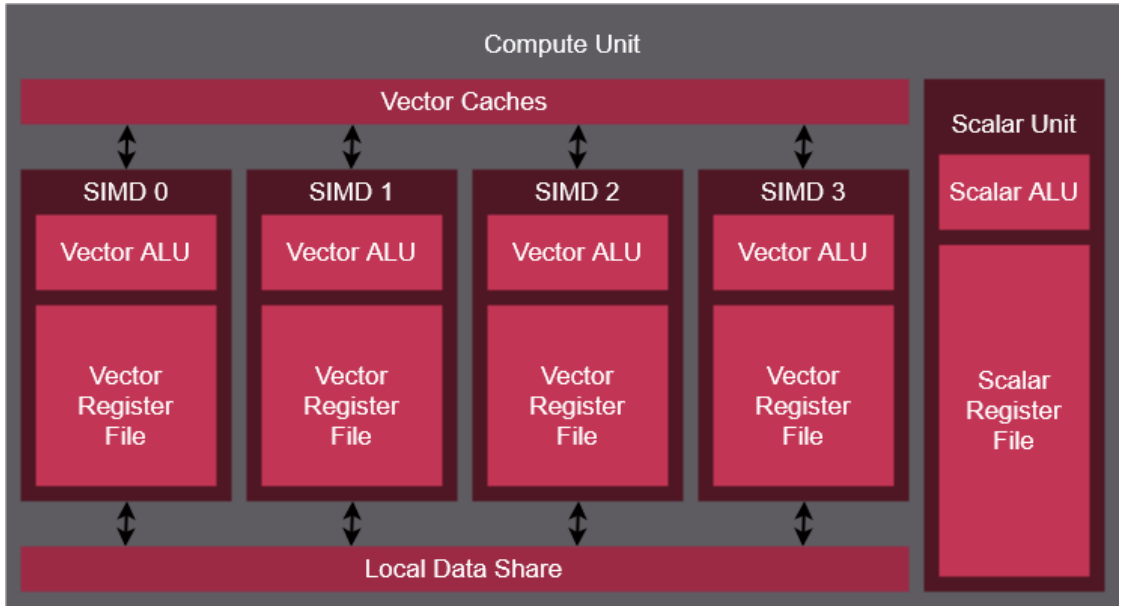
- VALU: Essentially a 16-lane vector processor
  - Similar to the concept of compute cores in our GPU model
  - Each VALU contains 16 compute cores running in a lockstep manner
- So, a CU has 16 x 4 = 64 compute cores
  - $p = 64$

=

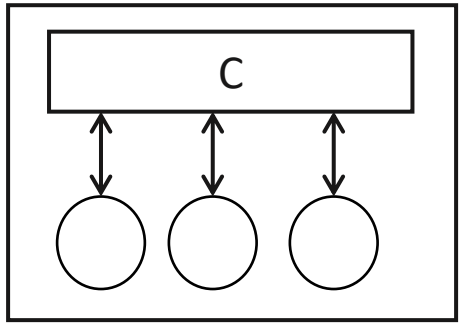


Compute Unit

- Each VALU consists of Vector Register File
  - 64KB - 256 total registers – each register is 64 4-byte-wide entries
  - Private to each compute core
- Each CU consists of Local Data Share (LDS)
  - 64 KB – Shared Cache in our GPU model
  - Can be used to share data between all threads mapped to the same CU
- A smaller Vector Cache/L1 Cache is also available, not user controllable

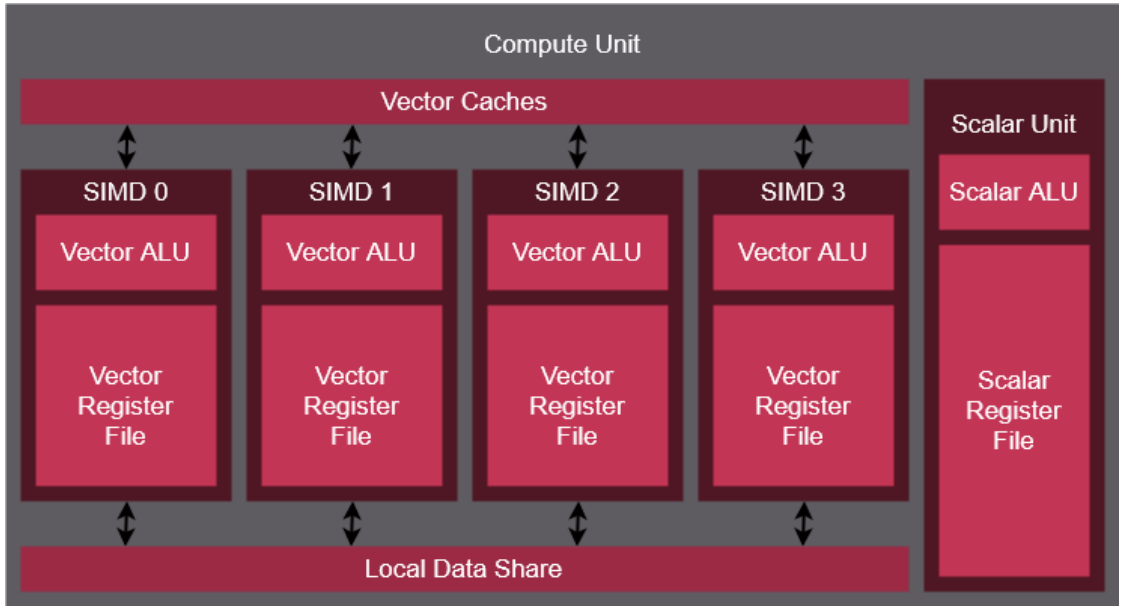


=

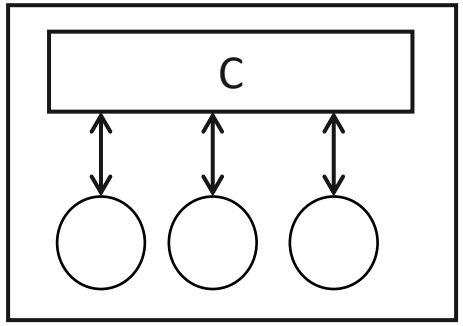


Compute Unit

- Matrix Cores – Special cores to perform matrix multiplications
- Similar to Tensor Cores in Nvidia GPUs
- Programmed using WMMA instructions
  - We will not cover, but if you are interested, reach out to me



=



Outline

- Introduction to HIP
- AMD GPU Hardware Model
- AMD Programming Model
- Implementation Basics

AMD GPU Programming Concepts

The Host (CPU)

- Host code runs here
- Usual C++ syntax and features
- Can be used to create device buffers, move data between host and device, and launch device code.

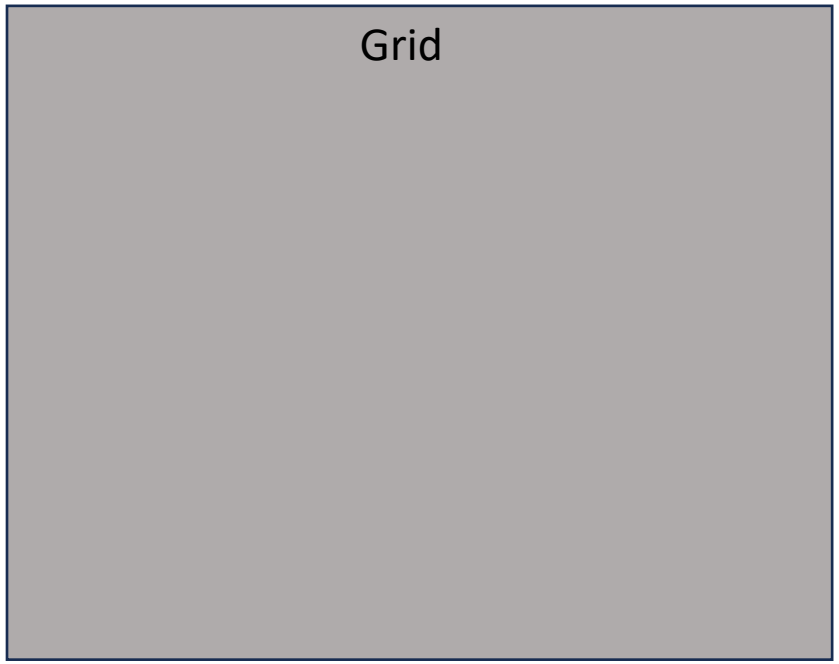
The Device (GPU)

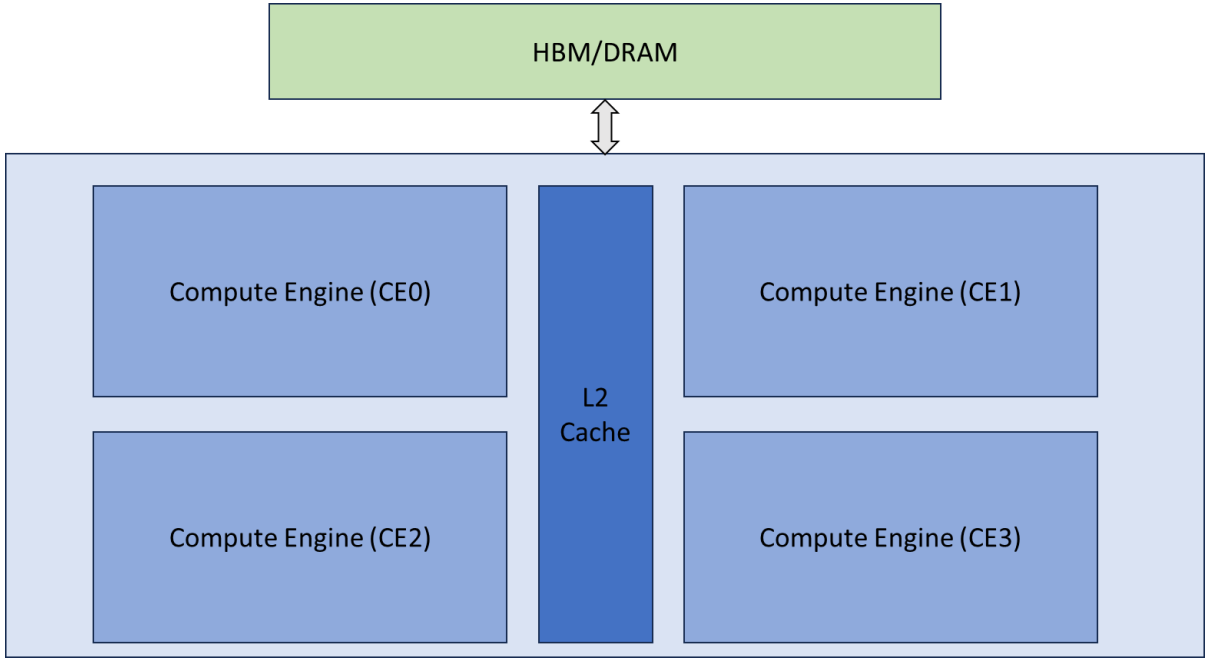
- Device code runs here
- C-like syntax
- Device codes are launched via “kernels”
- Instructions from the Host are enqueued into “streams”

AMD GPU Programming Concepts - Grid

- Piece of Code that runs on the GPU is called a kernel
- Each kernel is executed on a grid
  - Our algorithms get mapped onto the grid
- Grid can be 1D, 2D, or 3D
  - More of a programming construct, hardware simply converts into 1D grid for execution
  - Recall: We used 1D block for MV, 2D block for blocked matrix multiplication

AMD GPU Programming Concepts - Grid



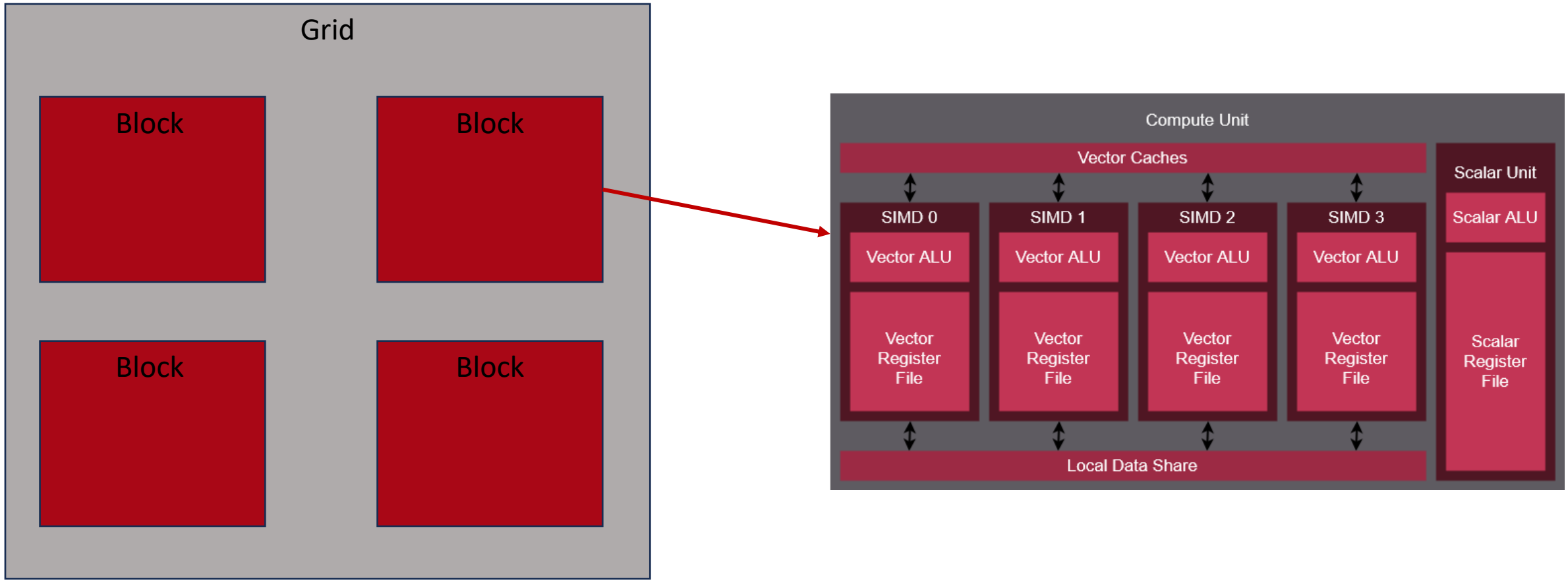


AMD GPU Programming Concepts - Block

- Grid is partitioned into equal sized blocks
  - 3D grid partitioned into 3D blocks
  - 2D grid partitioned into 2D blocks
  - 1D grid partitioned into 1D blocks

- Each block gets mapped to a CU

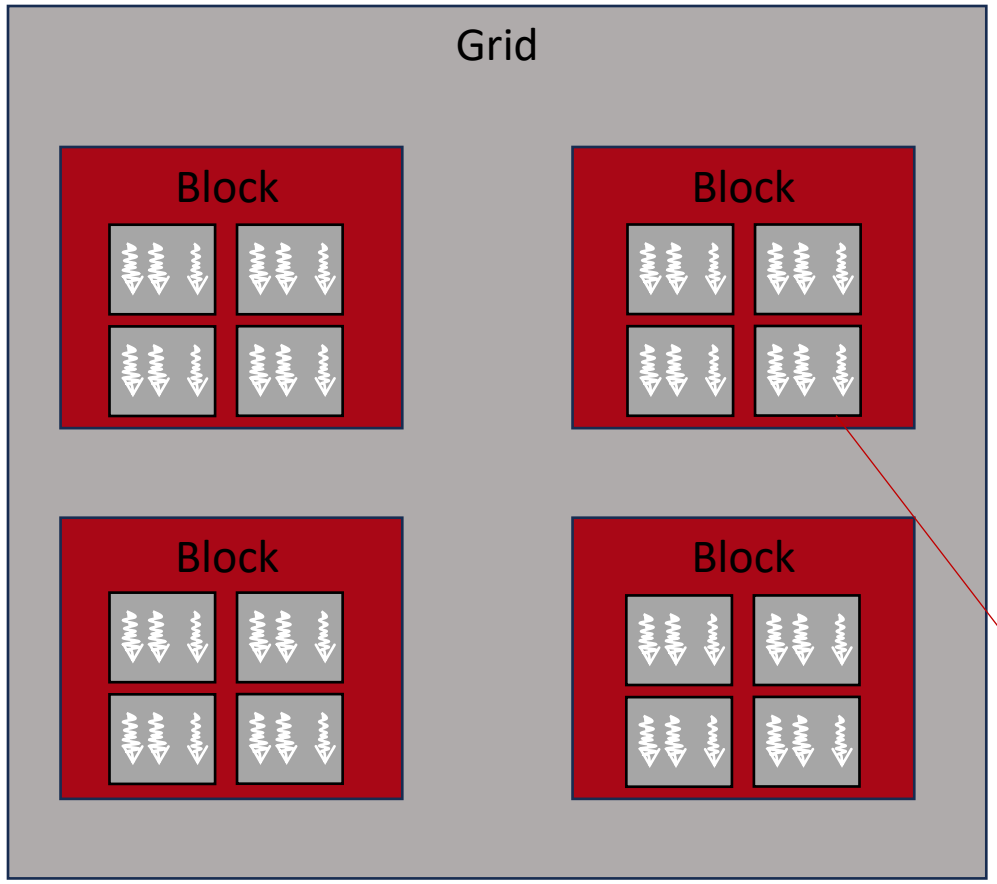
AMD GPU Programming Concepts - Block

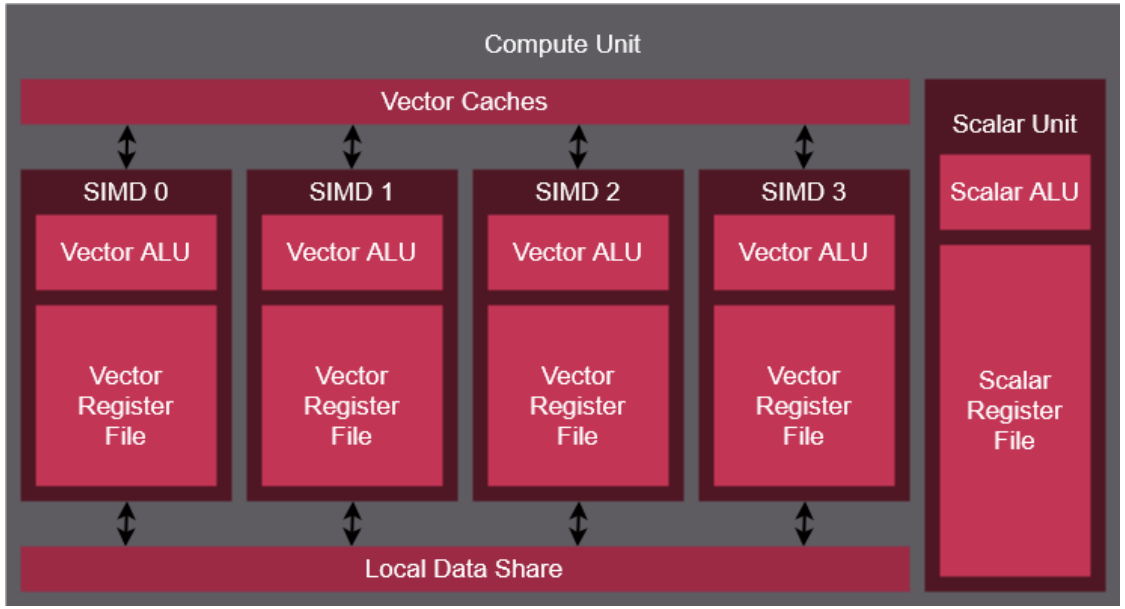


AMD GPU Programming Concepts - Threads

- Blocks are composed of threads
- Threads do the work and implement the data parallel algorithm
- Threads are grouped together into warps that get scheduled on the VALUs.
  - Warps do not appear in the programming model
  - Although useful when deciding the number of blocks per threads
- Threads in a block have access to a shared memory – Local Data Share

AMD GPU Programming Concepts - Threads



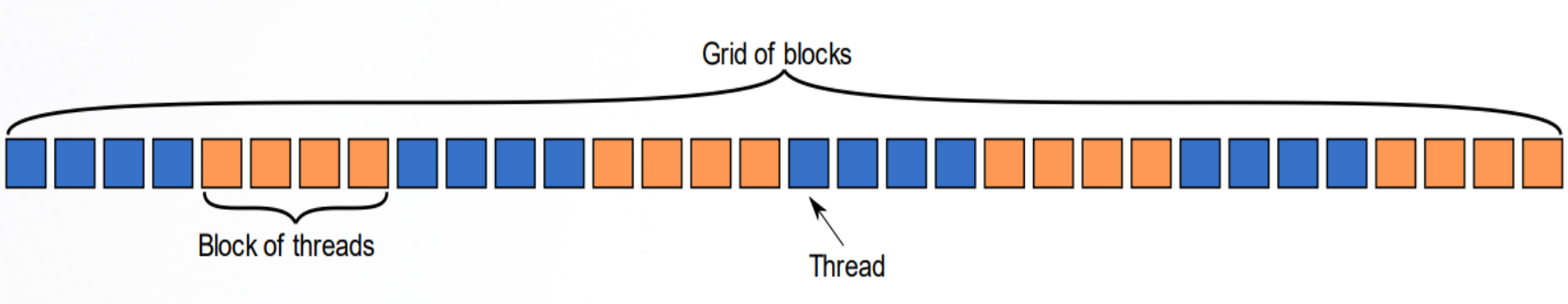


Warps and Threads

AMD GPU Programming Concepts

CUDA	HIP	OpenCL™
grid	grid	NDRange
block	block	work group
thread	thread	work item
warp	wavefront	sub-group (?)

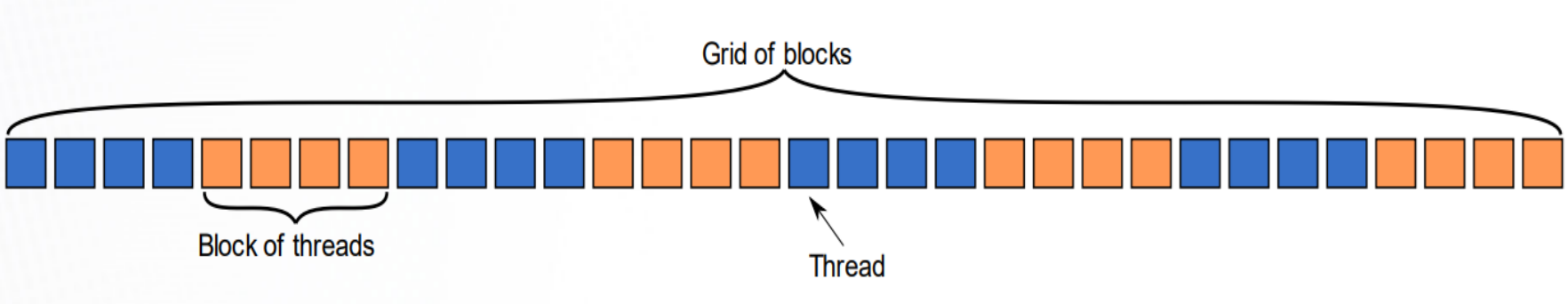
AMD GPU Programming Concepts – 1D Blocks



Each thread has access to the following configuration variables

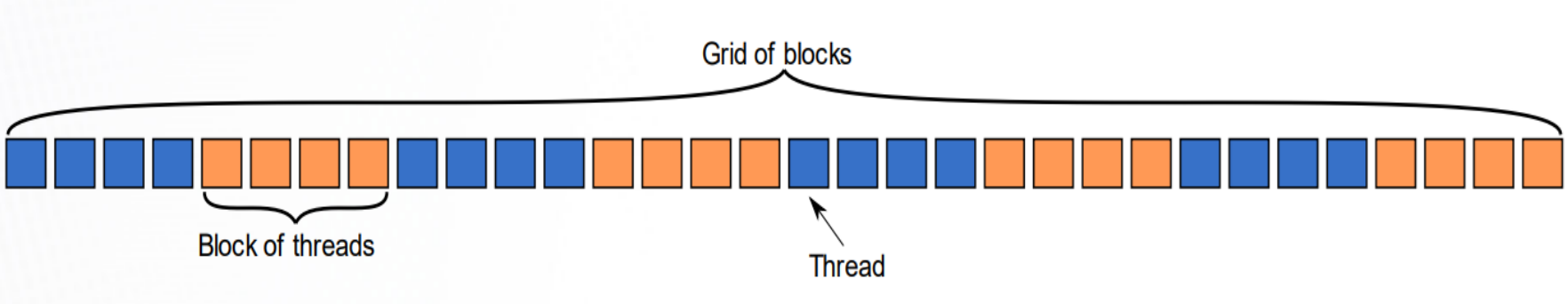
- Block ID of the block they belong to: blockIdx.x
- Thread ID within their block: threadIDx.x
- Dimension of the blocks in the kernel: blockDim.x
- Number of blocks in grid: gridDim.x

AMD GPU Programming Concepts – 1D Blocks



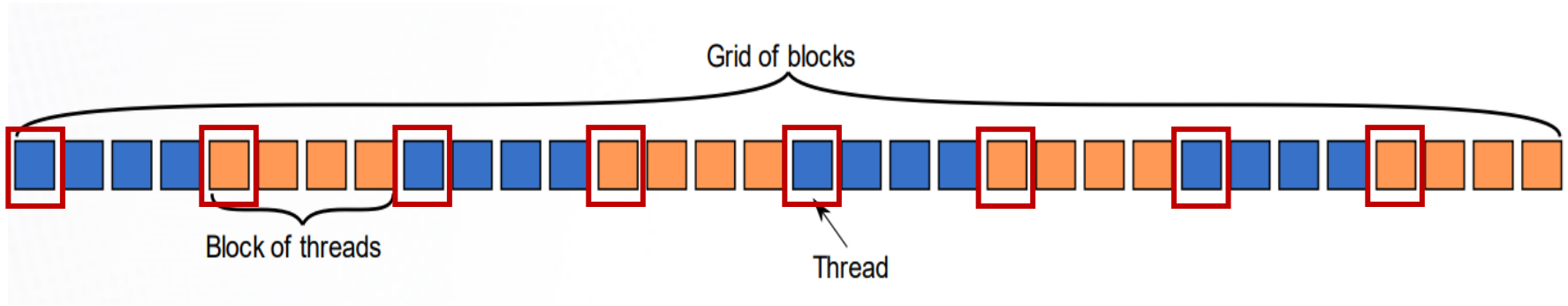
gridDim - 8
blockDim - 4

AMD GPU Programming Concepts – 1D Blocks



gridDim - 8
blockDim - 4

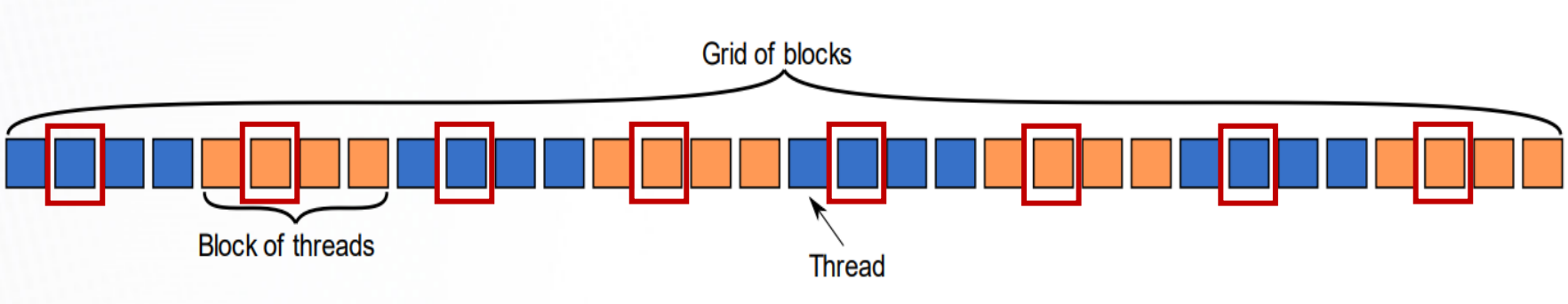
AMD GPU Programming Concepts – 1D Blocks



threadIdx.x = 0

gridDim - 8
blockDim - 4

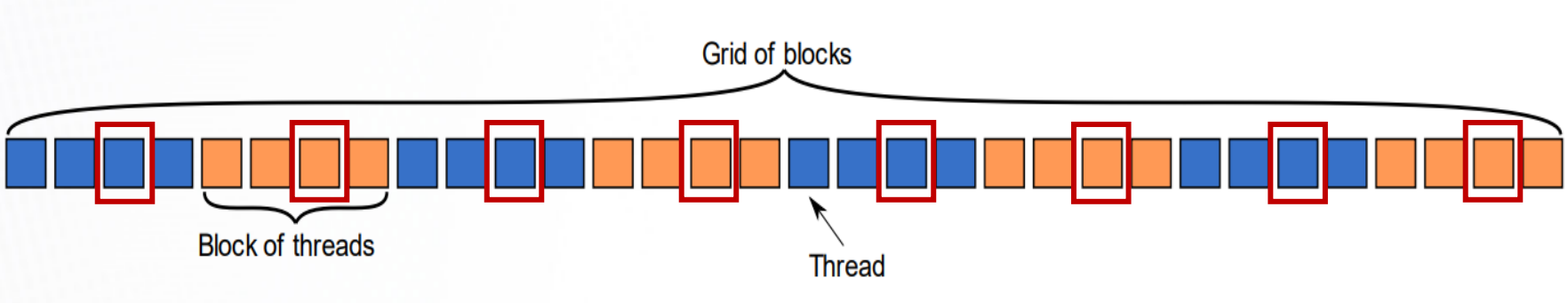
AMD GPU Programming Concepts – 1D Blocks



threadIdx.x = 1

gridDim - 8
blockDim - 4

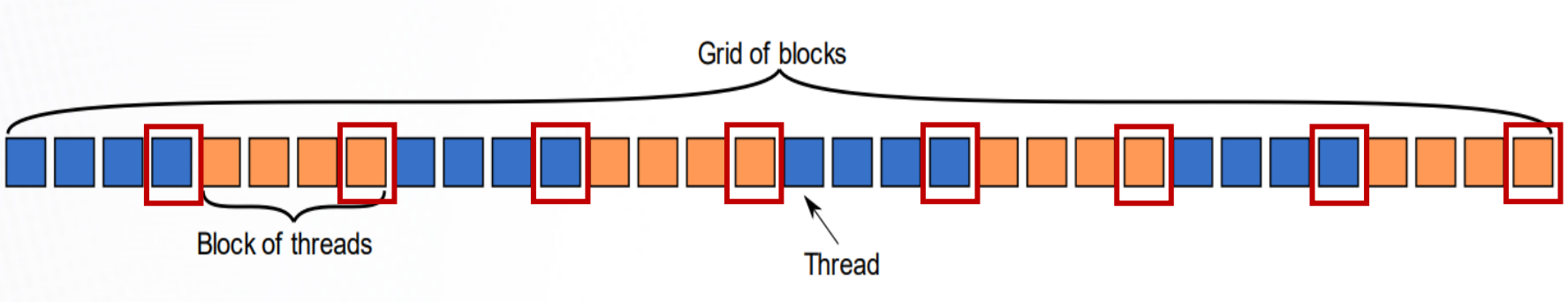
AMD GPU Programming Concepts – 1D Blocks



threadIdx.x = 2

gridDim - 8
blockDim - 4

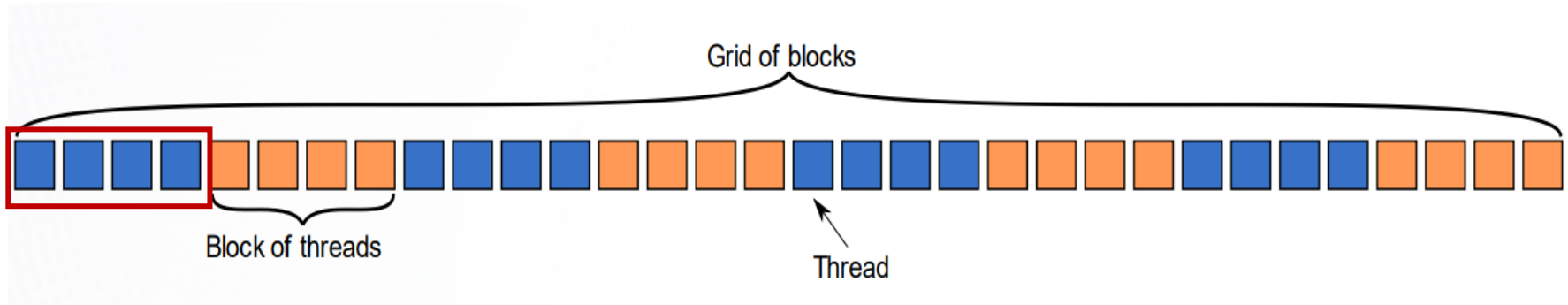
AMD GPU Programming Concepts – 1D Blocks



threadIdx.x = 3

gridDim - 8
blockDim - 4

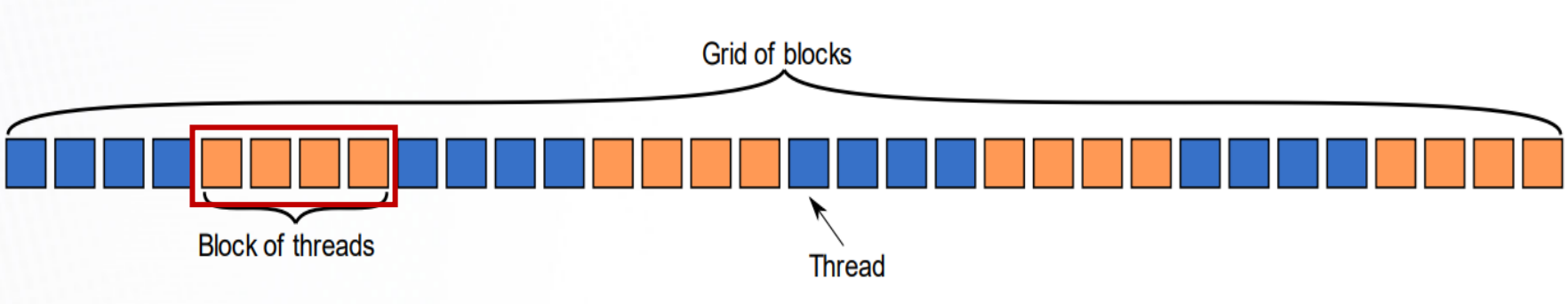
AMD GPU Programming Concepts – 1D Blocks



blockIdx.x = 0

gridDim - 8
blockDim - 4

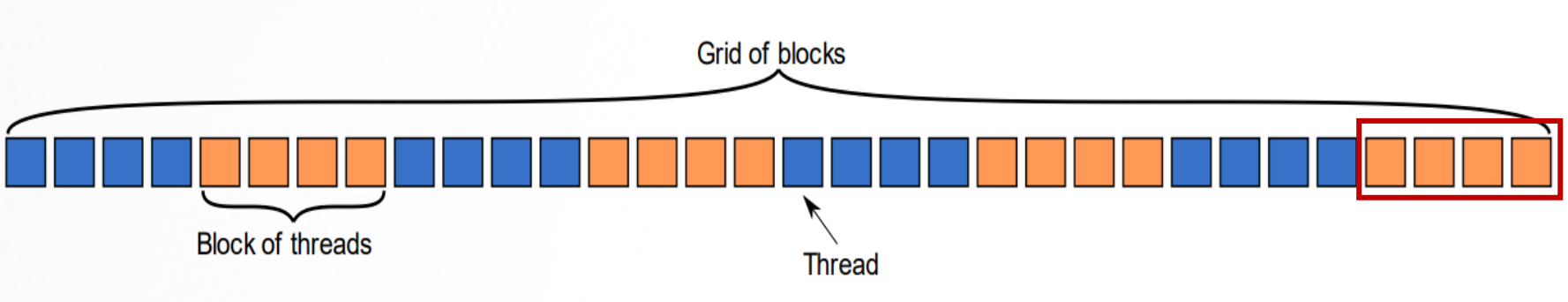
AMD GPU Programming Concepts – 1D Blocks



blockIdx.x = 1

gridDim - 8
blockDim - 4

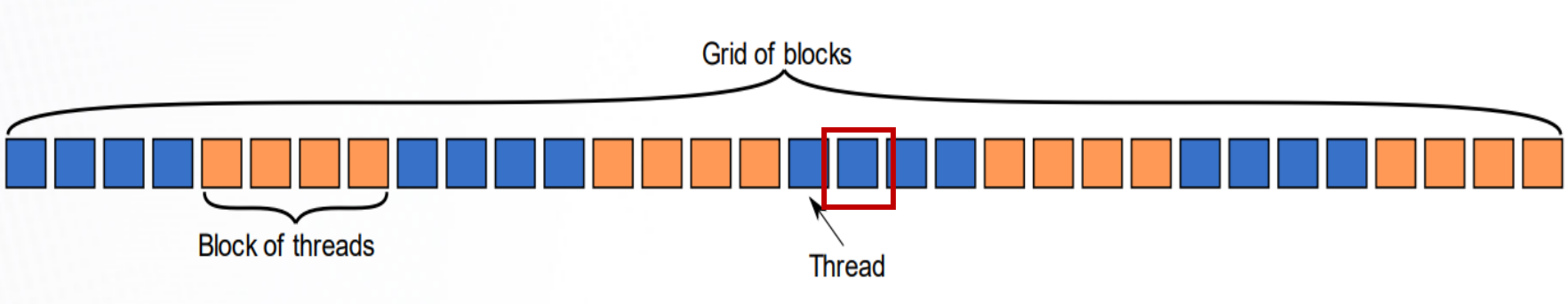
AMD GPU Programming Concepts – 1D Blocks



blockIdx.x = 7

gridDim - 8
blockDim - 4

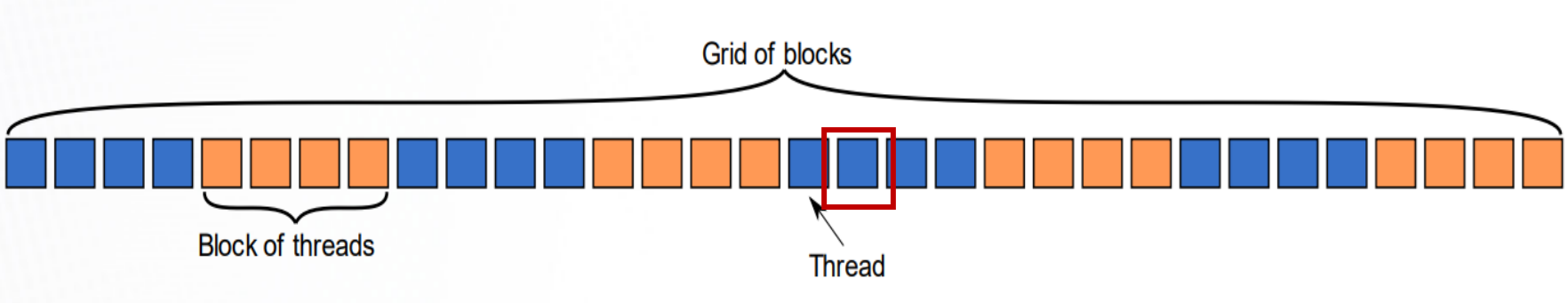
AMD GPU Programming Concepts – 1D Blocks



What is the ID of this thread?
Within the block: ??
Globally: ??

gridDim - 8
blockDim - 4

AMD GPU Programming Concepts – 1D Blocks



What is the ID of this thread?
Within the block: 1
Globally: 4*4 + 1 = 17

gridDim - 8
blockDim - 4

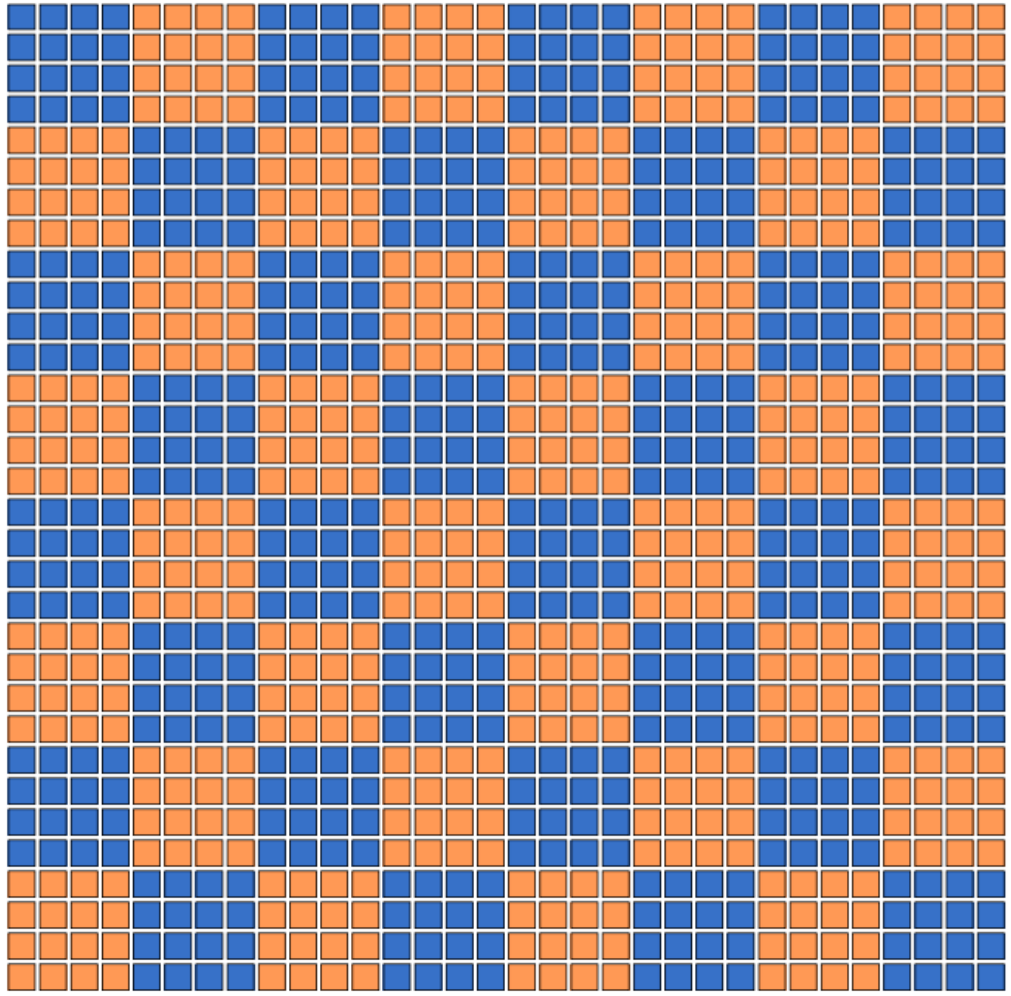
AMD GPU Programming Concepts – 2D Blocks

The Grid: blocks of threads in 2D

- Each color is a block of threads
- Each small square is a thread
- The concept is the same in 1D and 2D
- In 2D each block and thread now has a two-dimensional index

Threads in grid have access to:

- Their respective block IDs: blockIdx.x, blockIdx.y
- Their respective thread IDs in a block: threadIdx.x, threadIdx.y
- Block Dimensions are: blockDim.x, blockDim.y
- Grid Dimensions are: gridDim.x, gridDim.y



AMD GPU Programming Concepts

- How do we program a GPU?

- Similar to our GPU Model
  - Think data parallel: Same instruction, different data. Data determined by thread's ID
  - Each block has access to a shared memory, use that to avoid global memory traffic

Outline

- Introduction to AMD's GPU Programming Stack
- AMD GPU Hardware Model
- AMD Programming Model
- Implementation Basics

Creating a Kernel Function

A simple loop

```
for (int i=0;i<N;i++) {
  h_a[i] *= 2.0;
}
```

Can be translated into GPU kernel

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

- A device function that will be launched from the host program is called a kernel and is declared with the __global__ attribute
- Kernels should be declared void
- All pointers passed to kernels must point to memory on the device
- All threads execute the kernel's body "simultaneously"
- Each thread uses its unique thread and block IDs to compute a global ID
- There could be more than N threads in the grid

Launching a Kernel from Host CPU

```
dim3 threads(256,1,1);                  //3D dimensions of a block of threads
dim3 blocks((N+256-1)/256,1,1);         //3D dimensions the grid of blocks

hipLaunchKernelGGL(myKernel,            //Kernel name (__global__ void function)
                   blocks,              //Grid dimensions
                   threads,             //Block dimensions
                   0,                   //Bytes of dynamic LDS space (ignore for now)
                   0,                   //Stream (0=NULL stream)
                   N, a);               //Kernel arguments
```

Analogous to CUDA kernel launch syntax:

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

Things to Note

- Blocks get scheduled on the CUs
  - Note: number of blocks can be greater than the number of available CUs on the GPU
- All threads in a block execute on the same CU
- Threads in a block share LDS memory and L1 cache
- Threads in a block are executed in **64-wide** chunks called "wavefronts"
- A good practice is to make the block size a multiple of 64 and have several wavefronts(e.g. 256 threads)

How do we decide Number of Threads?

- Total number of threads = number of blocks x number of threads per blocks

- `dim3 blocks((N+256-1)/256, 1, 1)`
- `dim3 threads(256, 1, 1)`

- It is good to have threads as a multiple of 64 (warp size)
- Blocks can be determined by how much parallelism makes sense for a problem
  - In the case above, there is no point having more threads than the number of elements
  - Number of threads/blocks cannot be more than the maximum that can be offered by the hardware
  - More threads may also have more scheduling overheads, while offering more parallelism – sweet spot can be determined by trail and error

Memory Management

- GPU application typically will require memory allocation at:
  - Host Memory: memory allocated in CPU's RAM
  - Device Memory: memory allocated in GPU's HBM/DRAM

- Host memory allocated using malloc

- Host instructs device to allocate memory in its Global memory using **hipMalloc**

Memory Management

```
int main () {

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

        double *d_a = NULL;
        hipMalloc(&d_a, Nbytes);   //Allocate Nbytes in the device memory
        …
        free(h_a);  //Free Host memory
        hipFree(d_a); //Free Device memory



}
```

Memory Management

- Data will need to be copied from host to device
  - Model weights
  - Dataset
- And, results need to be transferred back from device to host
  - Trained model weights
  - Prediction results
- You can also copy data within the device

Memory Management

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

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

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

Memory Management

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

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

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

Notice which buffer is source and which buffer is destination

Memory Management

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

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

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

Notice the flags that decide the direction of transfer

Error Checking

- Most HIP API functions return error codes of type `hipError_t`
- `hipError_t == hipSuccess` implies the API call was successful
- `hipGetErrorString(status)` outputs the error string
- You can use the following macro in your code for ease of programming:

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

Putting it all Together

```
#include "hip/hip_runtime.h"
int main() {
int N = 1000;
size_t Nbytes= N*sizeof(double);
double *h_a= (double*) malloc(Nbytes);  //host memory
double *d_a= NULL;
HIP_CHECK(hipMalloc(&d_a, Nbytes));
…
HIP_CHECK(hipMemcpy(d_a, h_a, Nbytes, hipMemcpyHostToDevice));
//copy data to device
HIP_CHECK(hipLaunchKernelGGL(myKernel, dim3((N+256-1)/256,1,1),
dim3(256,1,1), 0, 0, N, d_a)); //Launch kernel
HIP_CHECK(hipMemcpy(h_a, d_a, Nbytes, hipMemcpyDeviceToHost));
//copy results back to host
…
free(h_a);                  //free host memory
HIP_CHECK(hipFree(d_a));    //free device memory
}
```

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

*There is an error in this code, will discuss in a few slides

Querying Device Properties

- You can query device properties to find out information such as total global memory, shared memory, limits on maximum number of blocks per grids, or threads per block, number of CUs, warpsize, etc.

- Helpful in keep our code more performance-portable
  - Example: Number of threads per block should be multiple of warpsize

Querying Device Properties

```
hipDevice Prop_tprops;
hipGetDeviceProperties(&props, deviceID);
```

- Look at https://github.com/ROCm/hip/blob/develop/include/hip/hip_runtime_api.h to find the information on the properties

- Ungraded HW Assignment: Can you find out how to get the properties listed in the previous slide?

Blocking vs Non-Blocking APIs

- In CPUs, function calls are usually Blocking
  - `retval = func();`
  - `retval` will be set only after `func` finishes the execution
- GPU related function calls can be blocking or non-blocking
- Non-blocking: We do not wait for the function execution to finish, but proceed in parallel

Blocking vs Non-Blocking APIs

- The kernel launch function, `hipLaunchKernelGGL`, is **non-blocking** for the host.
  - After sending instructions/data, the host continues immediately while the device executes the kernel
  - You can do some different work (on CPU) during this time.
- `hipMemcpy` is **blocking**.
  - Host code will proceed to the next line only after the memory transfer is complete
- Non blocking version of memory transfers is `hipMemcpyAsync`
  - `hipMemcpyAsync(d_a, h_a, Nbytes, hipMemcpyHostToDevice, stream);`

Blocking vs Non-Blocking APIs

- Sometimes we need to be sure that a non-blocking call has finished execution.

- `hipDeviceSynchronize()`

- Host waits until all pending asynchronous calls on the device finish execution

Putting it all Together

```
#include "hip/hip_runtime.h"
int main() {
int N = 1000;
size_t Nbytes= N*sizeof(double);
double *h_a= (double*) malloc(Nbytes);  //host memory
double *d_a= NULL;
HIP_CHECK(hipMalloc(&d_a, Nbytes));
…
HIP_CHECK(hipMemcpy(d_a, h_a, Nbytes, hipMemcpyHostToDevice));
//copy data to device
HIP_CHECK(hipLaunchKernelGGL(myKernel, dim3((N+256-1)/256,1,1),
dim3(256,1,1), 0, 0, N, d_a)); //Launch kernel
HIP_CHECK(hipMemcpy(h_a, d_a, Nbytes, hipMemcpyDeviceToHost));
//copy results back to host
…
free(h_a);                  //free host memory
HIP_CHECK(hipFree(d_a));    //free device memory
}
```

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

Can you tell me what the error is now?

Putting it all Together

```
#include "hip/hip_runtime.h"
int main() {
int N = 1000;
size_t Nbytes= N*sizeof(double);
double *h_a= (double*) malloc(Nbytes);  //host memory
double *d_a= NULL;
HIP_CHECK(hipMalloc(&d_a, Nbytes));
…
HIP_CHECK(hipMemcpy(d_a, h_a, Nbytes, hipMemcpyHostToDevice));
//copy data to device
HIP_CHECK(hipLaunchKernelGGL(myKernel, dim3((N+256-1)/256,1,1),
dim3(256,1,1), 0, 0, N, d_a)); //Launch kernel
HIP_CHECK(hipMemcpy(h_a, d_a, Nbytes, hipMemcpyDeviceToHost));
//copy results back to host
…
free(h_a);                  //free host memory
HIP_CHECK(hipFree(d_a));    //free device memory
}
```

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

Can you tell me what the error is now? We are not waiting for the kernel to finish execution before copying back the data

Putting it all Together

```
#include "hip/hip_runtime.h"
int main() {
int N = 1000;
size_t Nbytes= N*sizeof(double);
double *h_a= (double*) malloc(Nbytes);  //host memory
double *d_a= NULL;
HIP_CHECK(hipMalloc(&d_a, Nbytes));
…
HIP_CHECK(hipMemcpy(d_a, h_a, Nbytes, hipMemcpyHostToDevice));
//copy data to device
HIP_CHECK(hipLaunchKernelGGL(myKernel, dim3((N+256-1)/256,1,1),
dim3(256,1,1), 0, 0, N, d_a)); //Launch kernel
HIP_CHECK(hipDeviceSynchronize())
HIP_CHECK(hipMemcpy(h_a, d_a, Nbytes, hipMemcpyDeviceToHost));
//copy results back to host
…
free(h_a);                  //free host memory
HIP_CHECK(hipFree(d_a));    //free device memory
}
```

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

The host code will wait on this line till the launched kernel finishes execution.

Shared Memory and Thread Synchronization

- Recall: In matrix-matrix multiplication, we can shared memory
- In $Bid_y, Bid_x, Tid_y, Tid_x$ {
- For $k = 0$ to $\frac{N}{b} - 1$
  - Read A[$Bid_y * \frac{N}{b} + Tid_y$][k*N/b+$Tid_x$]
  - Read B[$k * \frac{N}{b} + Tid_y$][$Bid_x * \frac{N}{b} + Tid_x$]
  - Syncthreads()
  - For l = 0 to $b - 1$:
    - C[$Bid_y * \frac{N}{b} + Tid_y$][$Bid_x * \frac{N}{b} + Tid_x$] +=
    - A[$Bid_y * \frac{N}{b} + Tid_y$][k*N/b+$l$]* B[$k * \frac{N}{b} + l$][$Bid_x * \frac{N}{b} + Tid_x$]
  - Syncthreads()
- Store C[$Bid_y * \frac{N}{b} + Tid_y$][$Bid_x * \frac{N}{b} + Tid_x$] }

```
__global__ void blockMM(double *A, double *B, …)
{

//Define a shared memory array
__shared__ double A_shared[b][N];
__shared__ double B_shared[N][b];

for (k = 0; k < N/b; k++) {
        A_shared[…][…]  = A[…][…];
        B_shared[…][…]  = B[…][…];
        __syncthreads();
}


}
```

Shared Memory and Thread Synchronization

- Recall: In matrix-matrix multiplication, we can shared memory
- In $Bid_y, Bid_x, Tid_y, Tid_x$ {
- For $k = 0$ to $\frac{N}{b} - 1$
  - Read A[$Bid_y * \frac{N}{b} + Tid_y$][k*N/b+$Tid_x$]
  - Read B[$k * \frac{N}{b} + Tid_y$][$Bid_x * \frac{N}{b} + Tid_x$]
  - Syncthreads()
  - For l = 0 to $b - 1$:
    - C[$Bid_y * \frac{N}{b} + Tid_y$][$Bid_x * \frac{N}{b} + Tid_x$] +=
    - A[$Bid_y * \frac{N}{b} + Tid_y$][k*N/b+$l$]* B[$k * \frac{N}{b} + l$][$Bid_x * \frac{N}{b} + Tid_x$]
  - Syncthreads()
- Store C[$Bid_y * \frac{N}{b} + Tid_y$][$Bid_x * \frac{N}{b} + Tid_x$] }

```
__global__ void blockMM(double *A, double *B, …)
{

//Define a shared memory array
__shared__ double A_shared[b][N];
__shared__ double B_shared[N][b];

for (k = 0; k < N/b; k++) {
        A_shared[…][…]  = A[…][…];
        B_shared[…][…]  = B[…][…];
        __syncthreads();
}


}
```

Variable needs to be defined using __shared__ keyword

Shared Memory and Thread Synchronization

- Recall: In matrix-matrix multiplication, we can shared memory
- In $Bid_y, Bid_x, Tid_y, Tid_x$ {
- For $k = 0$ to $\frac{N}{b} - 1$
  - Read A[$Bid_y * \frac{N}{b} + Tid_y$][k*N/b+$Tid_x$]
  - Read B[$k * \frac{N}{b} + Tid_y$][$Bid_x * \frac{N}{b} + Tid_x$]
  - Syncthreads()
  - For l = 0 to $b - 1$:
    - C[$Bid_y * \frac{N}{b} + Tid_y$][$Bid_x * \frac{N}{b} + Tid_x$] +=
    - A[$Bid_y * \frac{N}{b} + Tid_y$][k*N/b+$l$]* B[$k * \frac{N}{b} + l$][$Bid_x * \frac{N}{b} + Tid_x$]
  - Syncthreads()
- Store C[$Bid_y * \frac{N}{b} + Tid_y$][$Bid_x * \frac{N}{b} + Tid_x$] }

```
__global__ void blockMM(double *A, double *B, …)
{

//Define a shared memory array
__shared__ double A_shared[b][N];
__shared__ double B_shared[N][b];

for (k = 0; k < N/b; k++) {
        A_shared[…][…]  = A[…][…];
        B_shared[…][…]  = B[…][…];
        __syncthreads();
}


}
```

Can you think about these indices, you will need it for your PA 1

Shared Memory and Thread Synchronization

- Recall: In matrix-matrix multiplication, we can shared memory
- In $Bid_y, Bid_x, Tid_y, Tid_x$ {
- For $k = 0$ to $\frac{N}{b} - 1$
  - Read A[$Bid_y * \frac{N}{b} + Tid_y$][k*N/b+$Tid_x$]
  - Read B[$k * \frac{N}{b} + Tid_y$][$Bid_x * \frac{N}{b} + Tid_x$]
  - Syncthreads()
  - For l = 0 to $b - 1$:
    - C[$Bid_y * \frac{N}{b} + Tid_y$][$Bid_x * \frac{N}{b} + Tid_x$] +=
    - A[$Bid_y * \frac{N}{b} + Tid_y$][k*N/b+$l$]* B[$k * \frac{N}{b} + l$][$Bid_x * \frac{N}{b} + Tid_x$]
  - Syncthreads()
- Store C[$Bid_y * \frac{N}{b} + Tid_y$][$Bid_x * \frac{N}{b} + Tid_x$] }

```
__global__ void blockMM(double *A, double *B, …)
{

//Define a shared memory array
__shared__ double A_shared[b][N];
__shared__ double B_shared[N][b];

for (k = 0; k < N/b; k++) {
        A_shared[…][…]  = A[…][…];
        B_shared[…][…]  = B[…][…];
        __syncthreads();
}


}
```

This ensures that all threads in the block have copied the data they own

Additional Concepts

- Concepts covered till now will enable you to write GPU accelerated programs
- To further improve performance, please checkout the following concepts
- Streams: https://rocmdocs.amd.com/projects/HIP/en/latest/how-to/hip_runtime_api/asynchronous.html
  - Enable concurrent kernel execution on GPU leading to improved resource utilization and overlapping of computation and communication
- HIP Graphs: https://rocmdocs.amd.com/projects/HIP/en/latest/how-to/hip_runtime_api/hipgraph.html
  - A more sophisticated version of scheduling kernels than streams – Think of task graphs

Next Class

- 10/7 Lecture 13
  - Pytorch Lightning

CASE SCHOOL OF ENGINEERING
CASE WESTERN RESERVE UNIVERSITY

Thank You

- Questions?

- Email: sanmukh.kuppannagari@case.edu