INTRODUCTION TO AMD GPU PROGRAMMING WITH HIP

Paul Bauman, Noel Chalmers, Nick Curtis, Chip Freitag, Joe Greathouse, Nicholas Malaya, Damon McDougall, Scott Moe, René van Oostrum, Noah Wolfe, Gina Sitaraman

Agenda

- A Mile-High View of GPU Acceleration (10 minutes)
- GPU Hardware (5 minutes)
- GPU Programming Concepts (45 minutes)

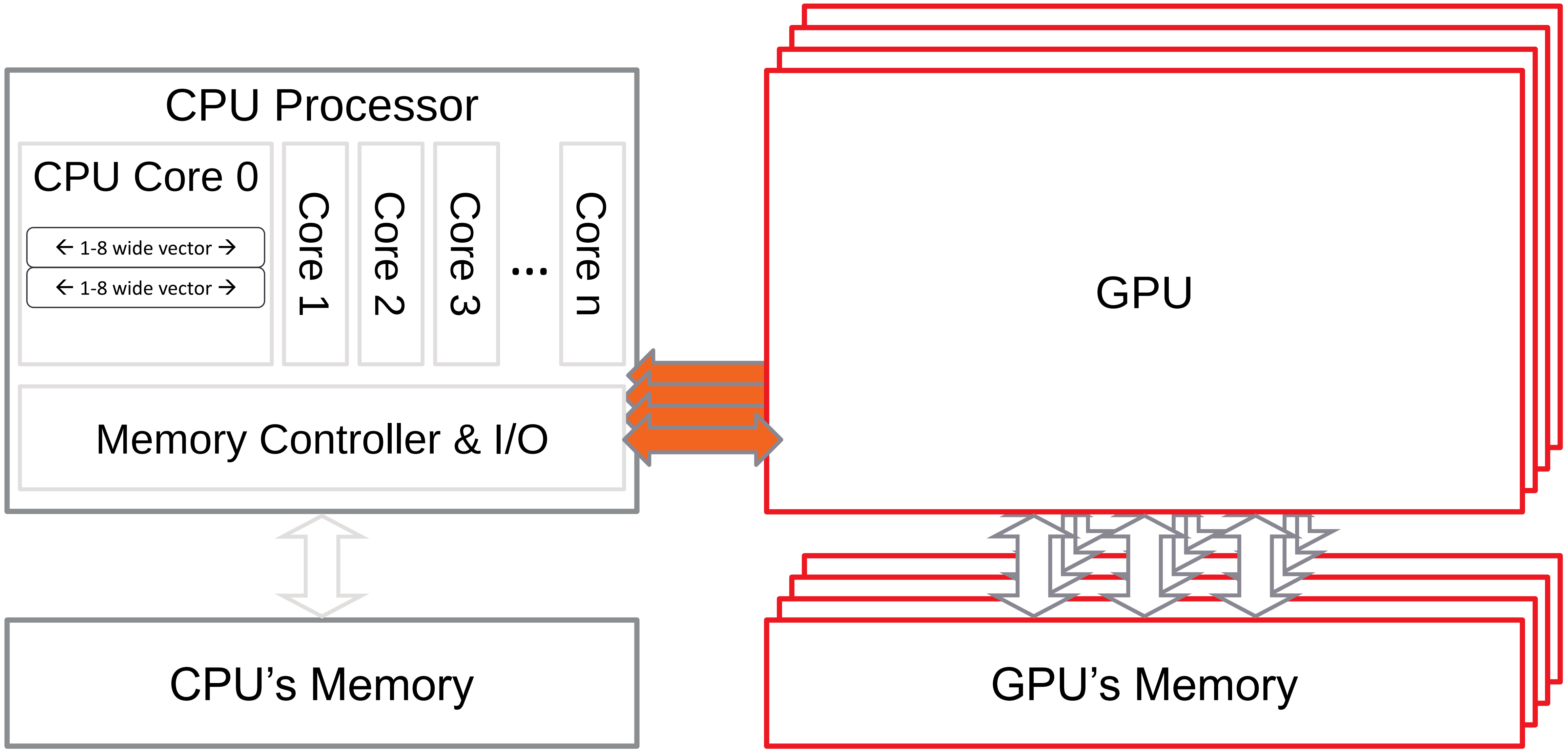
Comments

Please feel free to ask questions during the presentation in the chat and the moderator will help flag the speaker

(and maybe some AMD people will answer in the chat)

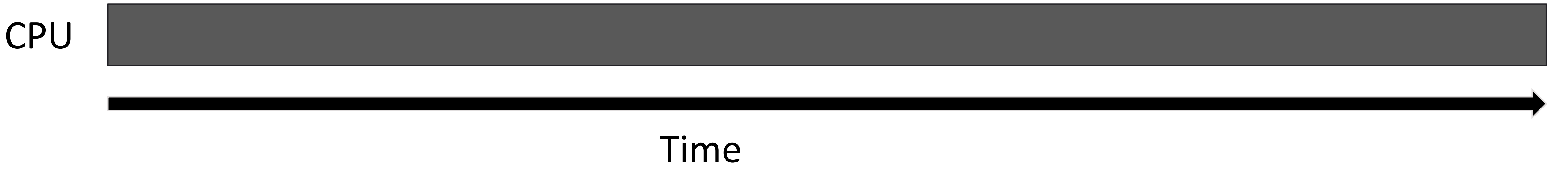
A Mile-High Overview of GPU Acceleration

Illustration of a CPU+GPU Heterogeneous System

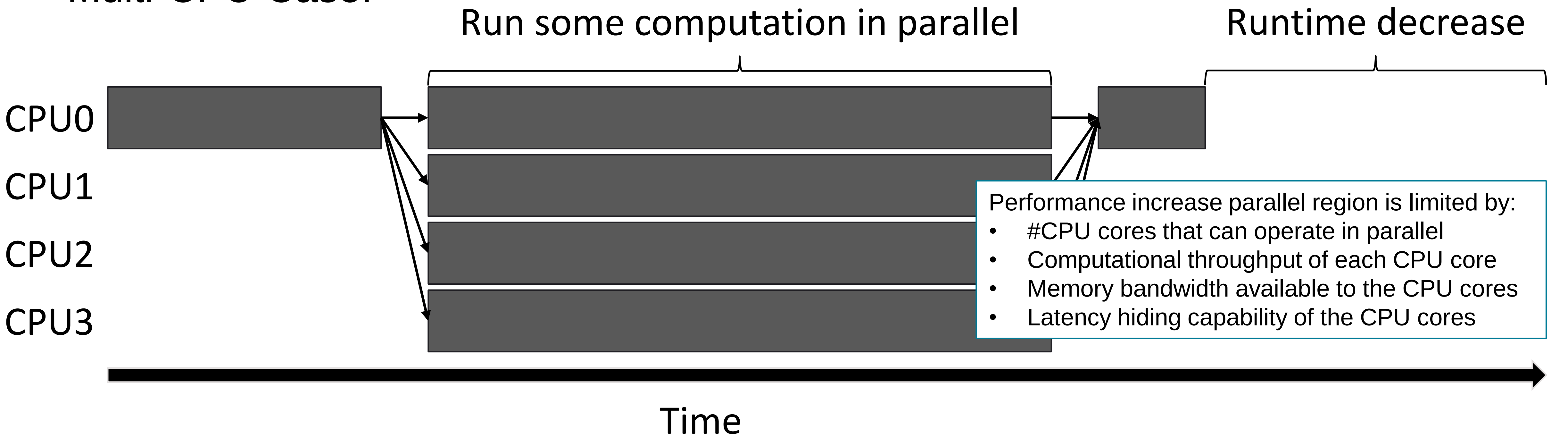


CPU Parallelism Can Accelerate Applications

Single CPU Case:

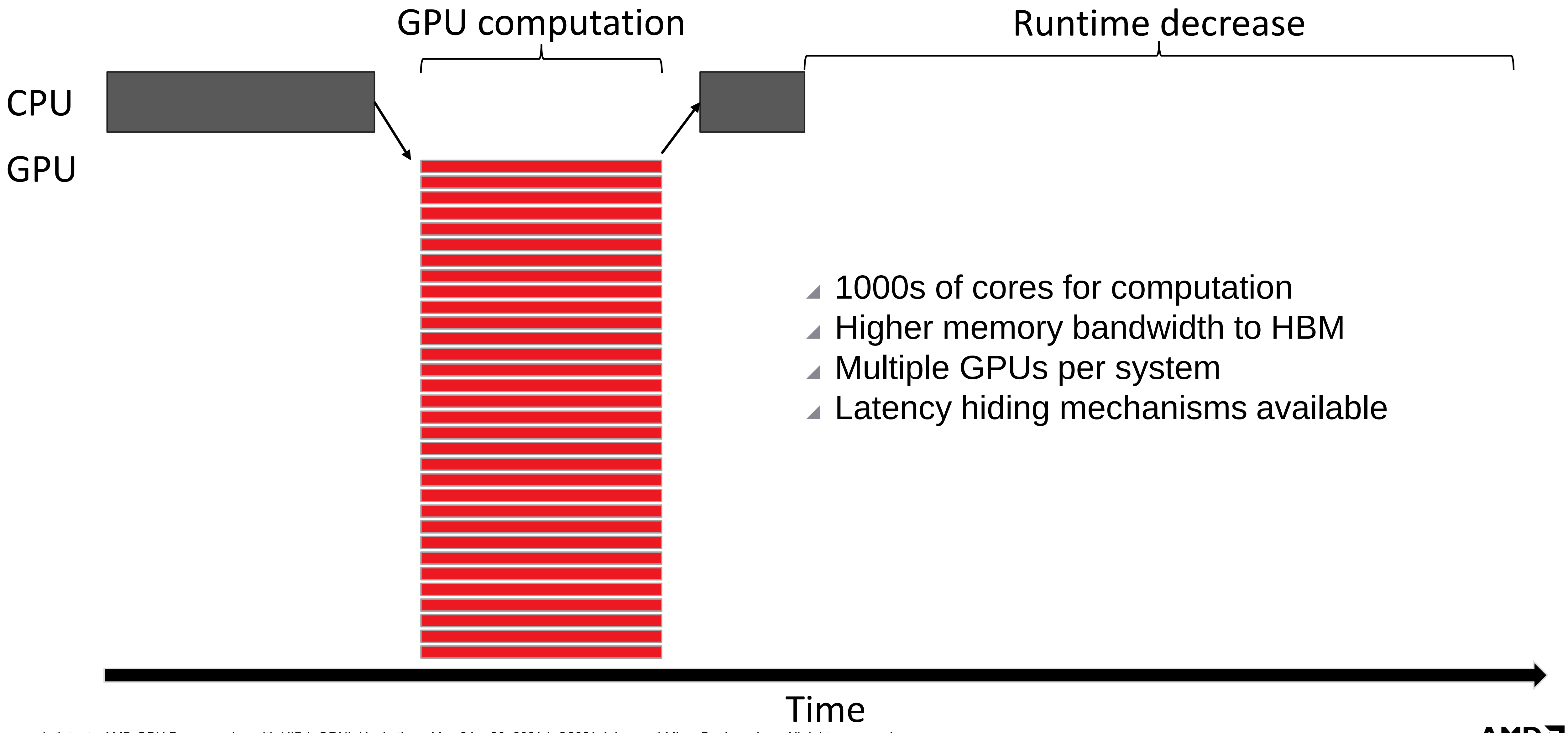


Multi-CPU Case:



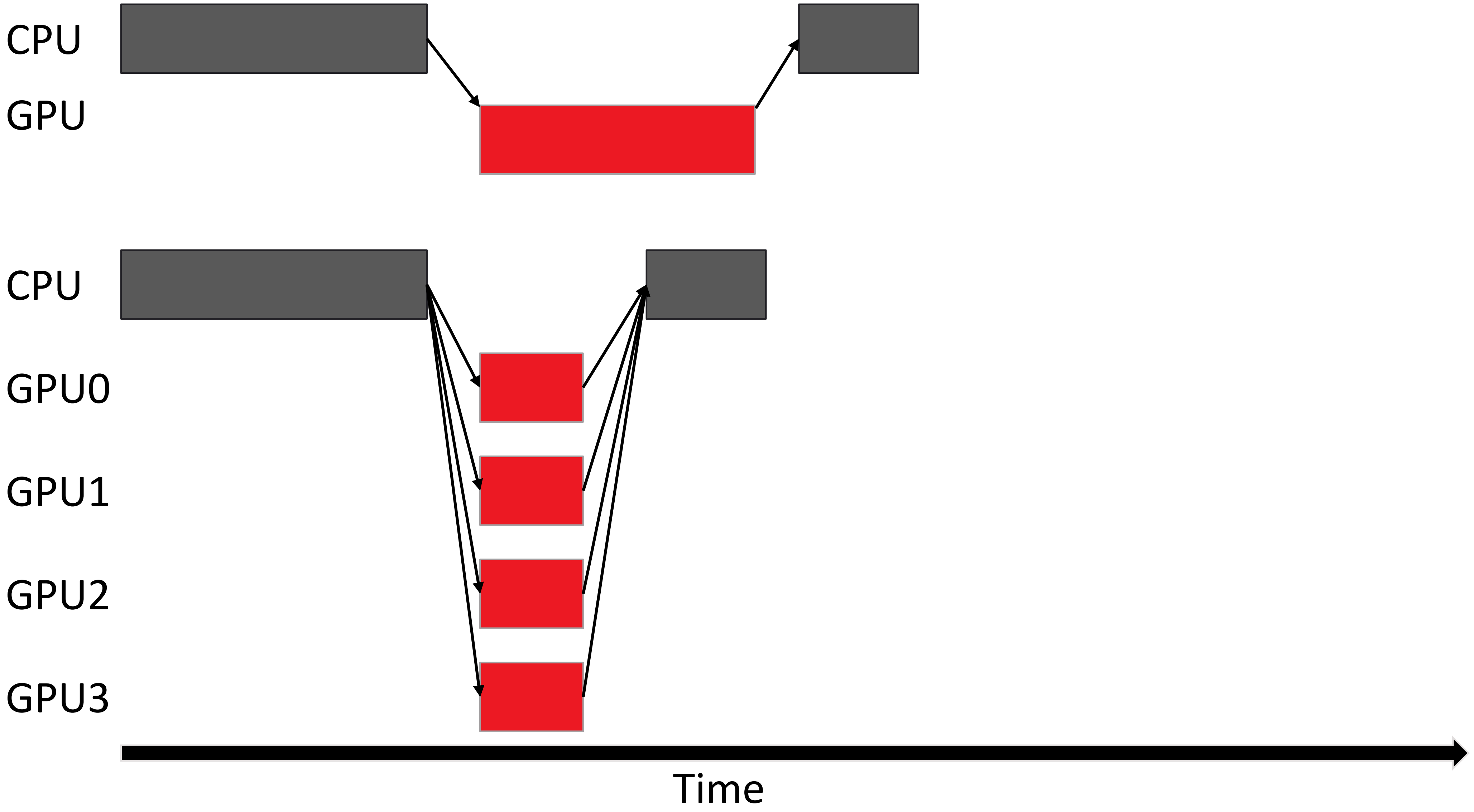
Performance increase parallel region is limited by:
- #CPU cores that can operate in parallel
- Computational throughput of each CPU core
- Memory bandwidth available to the CPU cores
- Latency hiding capability of the CPU cores

GPUs can Accelerate Parallel Regions

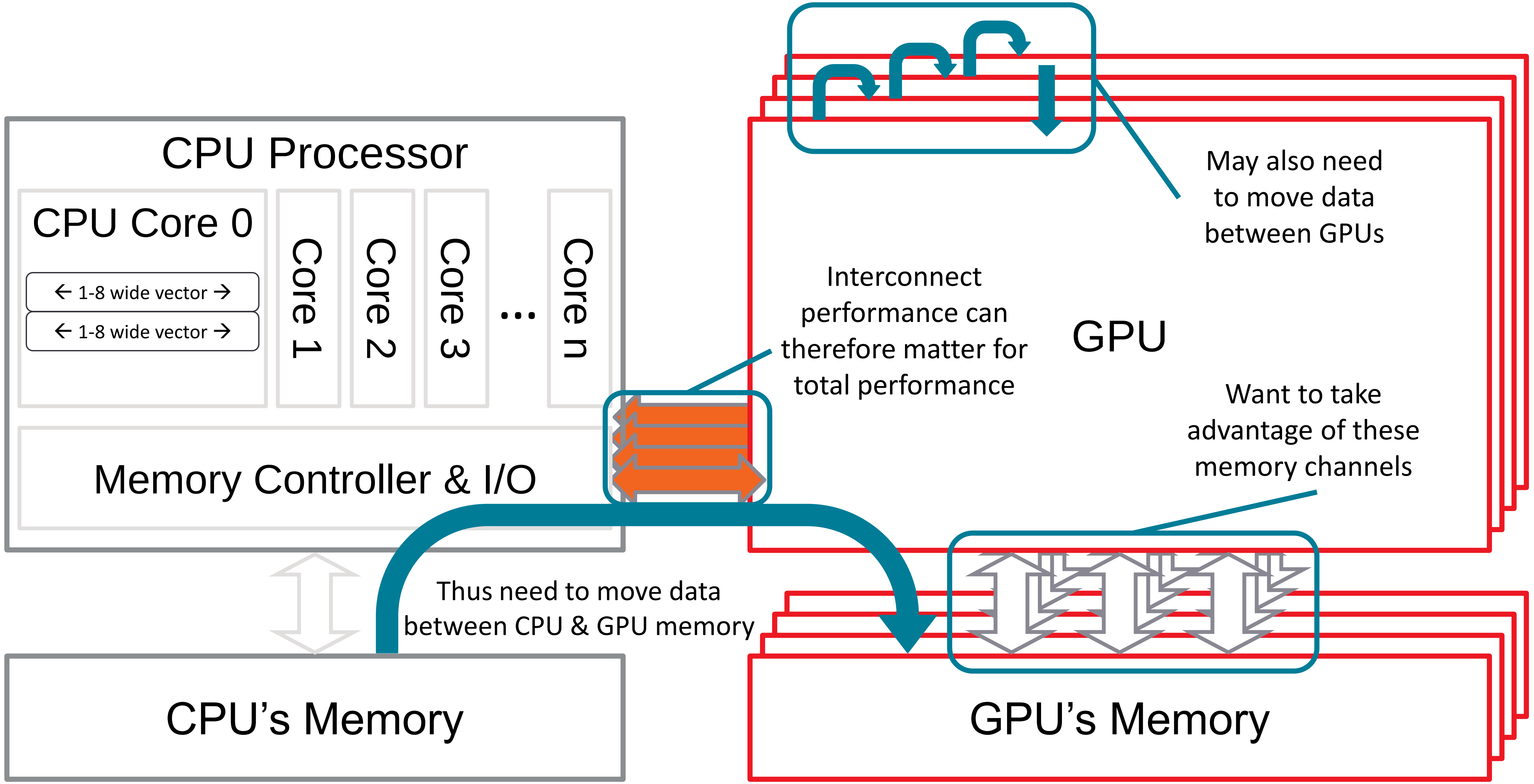


- 1000s of cores for computation
- Higher memory bandwidth to HBM
- Multiple GPUs per system
- Latency hiding mechanisms available

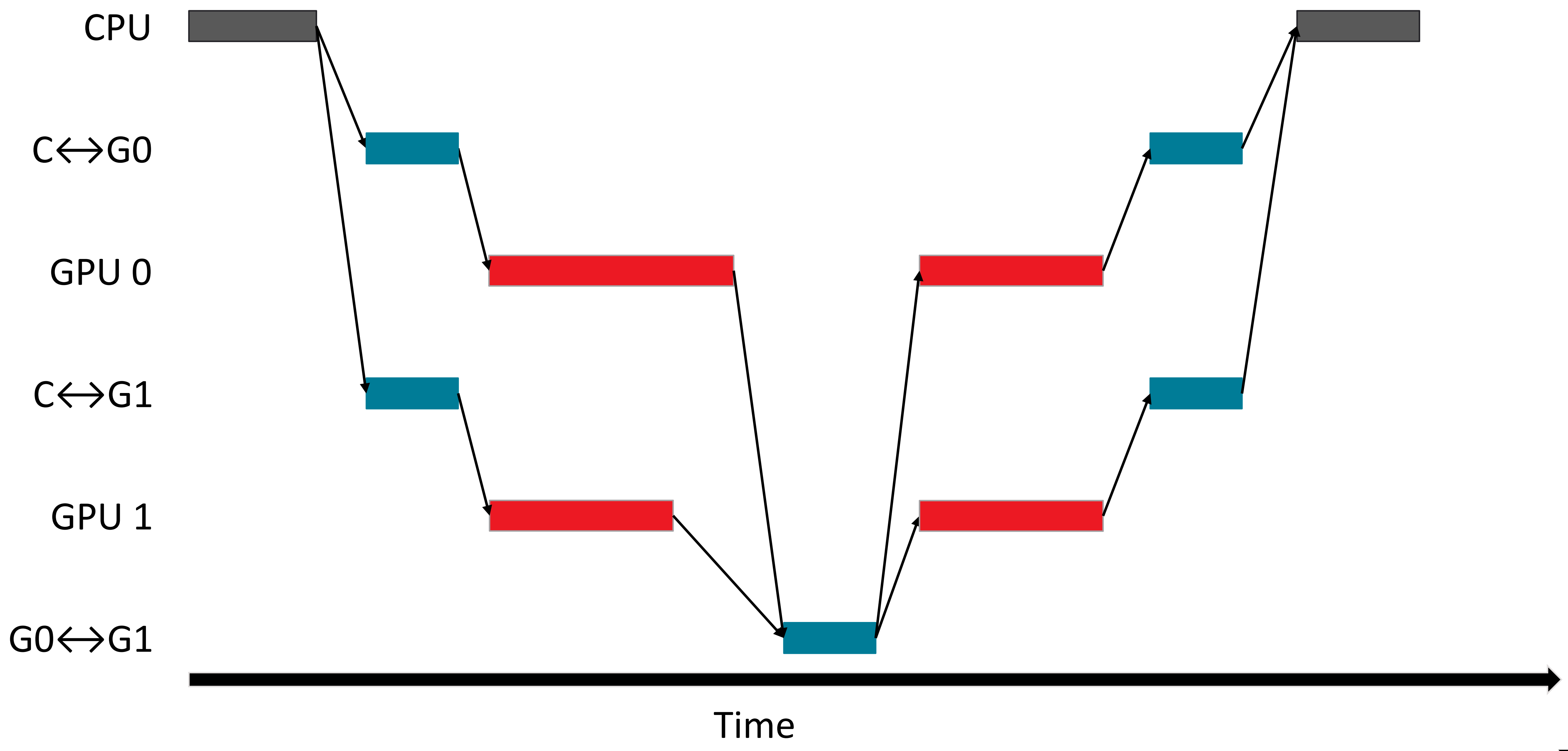
Multiple GPUs can Further Increase Performance



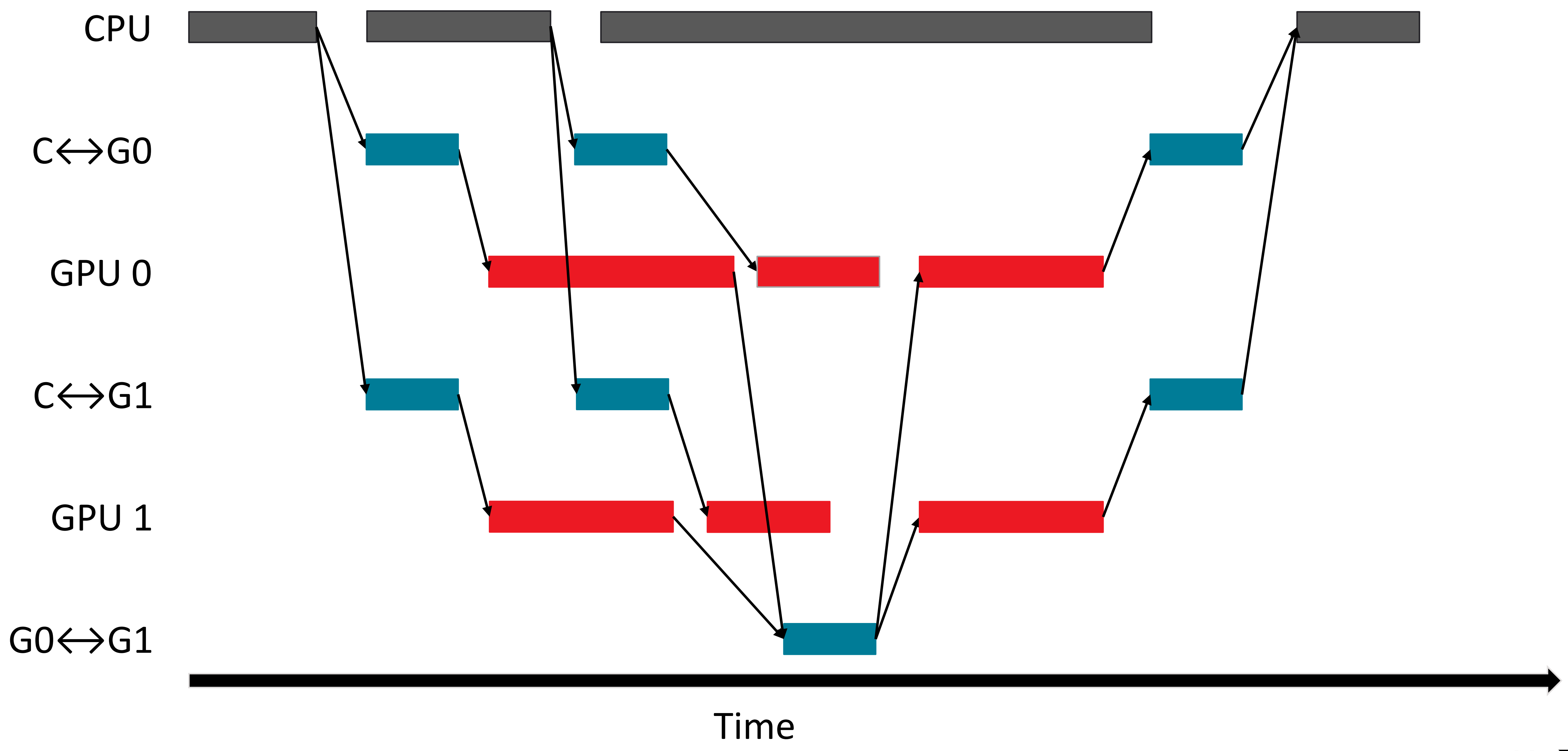
Taking Advantage of GPU Memory BW Requires Moving Data to GPU


CPU Processor
CPU Core 0
← 1-8 wide vector →
← 1-8 wide vector →
Core 1
Core 2
Core 3
…
Core n
Memory Controller & I/O
CPU's Memory
GPU
GPU's Memory
May also need to move data between GPUs
Interconnect performance can therefore matter for total performance
Want to take advantage of these memory channels
Thus need to move data between CPU & GPU memory

Computation and Communication Must Both Happen

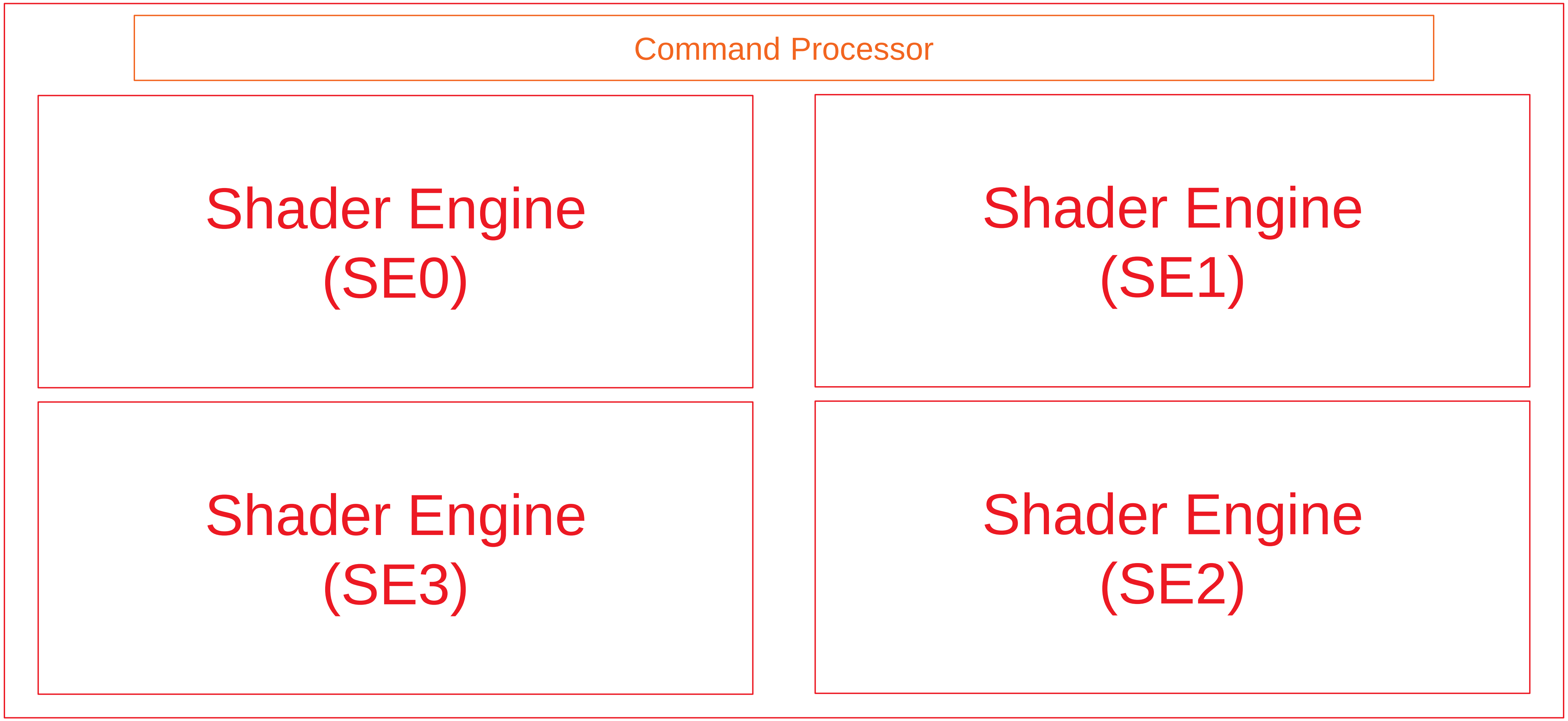


All of the Things in This Task Graph can be Overlapped

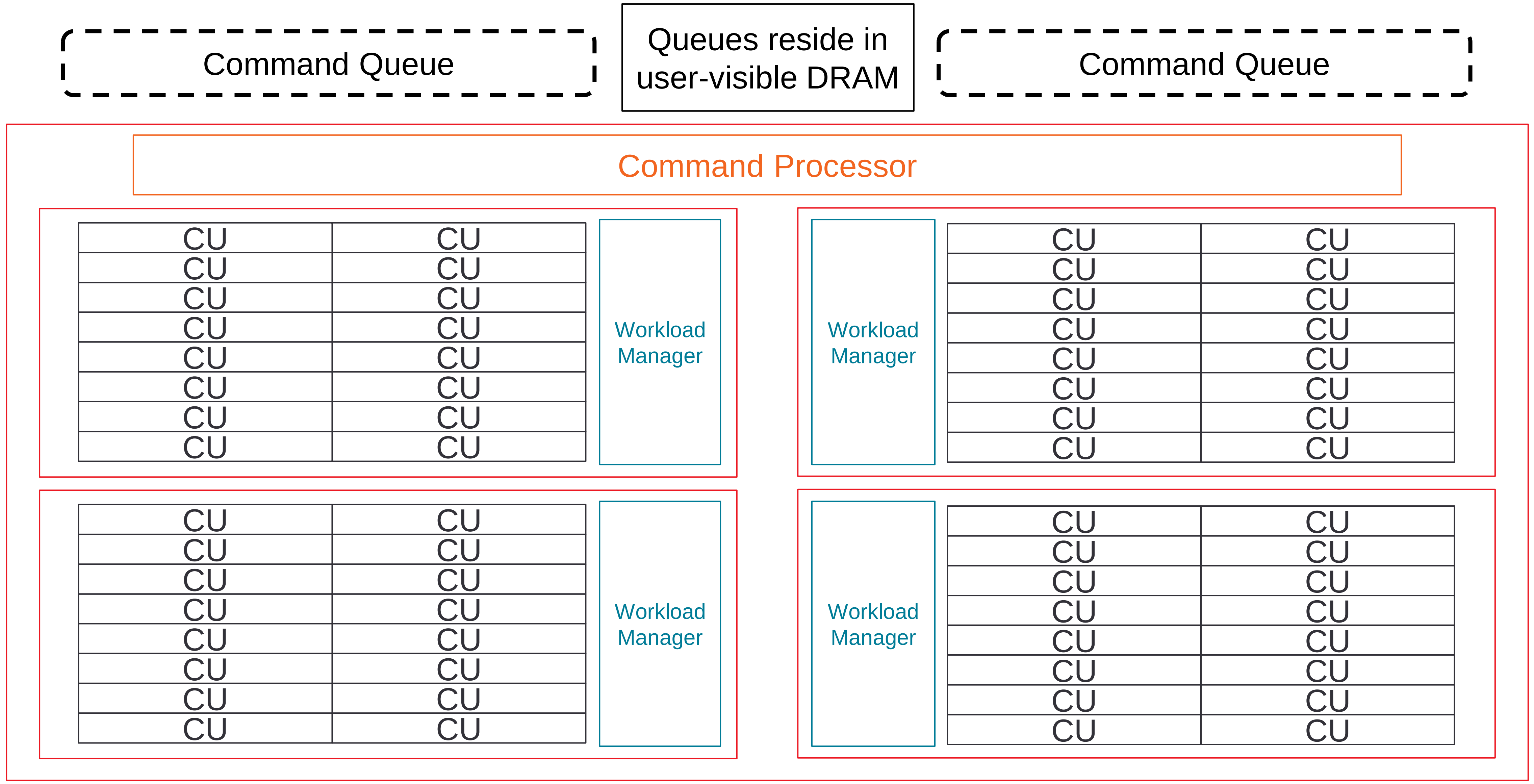


AMD GCN GPU Hardware

AMD GCN GPU Hardware Layout



AMD GCN GPU Hardware Layout



Hardware Configuration Parameters on Modern AMD GPUs

GPU SKU	Shader Engines	CUs / SE
AMD Instinct™ MI100	8	15
AMD Radeon Instinct™ MI60	4	16
AMD Radeon Instinct™ MI50	4	15
AMD Radeon™ VII	4	15
AMD Radeon Instinct™ MI25 AMD Radeon™ Vega 64	4	16
AMD Radeon™ Vega 56	4	14
AMD Radeon Instinct™ MI6	4	9
AMD Ryzen™ 5 2400G	1	11

AMD GPU Compute Terminology

Overview of GPU Kernels

GPU Kernel

Functions launched to the GPU that are executed by multiple parallel workers

Examples: GEMM, triangular solve, vector copy, scan, convolution

Overview of GPU Kernels

GPU Kernel

Workgroup 0

Group of threads that are on the GPU at the same time
Also on the same compute unit
Can synchronize together and communicate through memory in the CU

CUDA Terminology
Thread Block

Workgroup 1

Workgroup 2

Workgroup 3

Workgroup 4

…

Workgroup n

Programmer controls the number of workgroups – it's usually a function of problem size

Overview of GPU Kernels

GPU Kernel

Workgroup 0

Wavefront

Collection of resources that execute in lockstep, run the same instructions, and follow the same control-flow path. Individual lanes can be masked off
Can think of this as a vectorized thread

CUDA Terminology
Warp

Workgroup 1

Workgroup 2

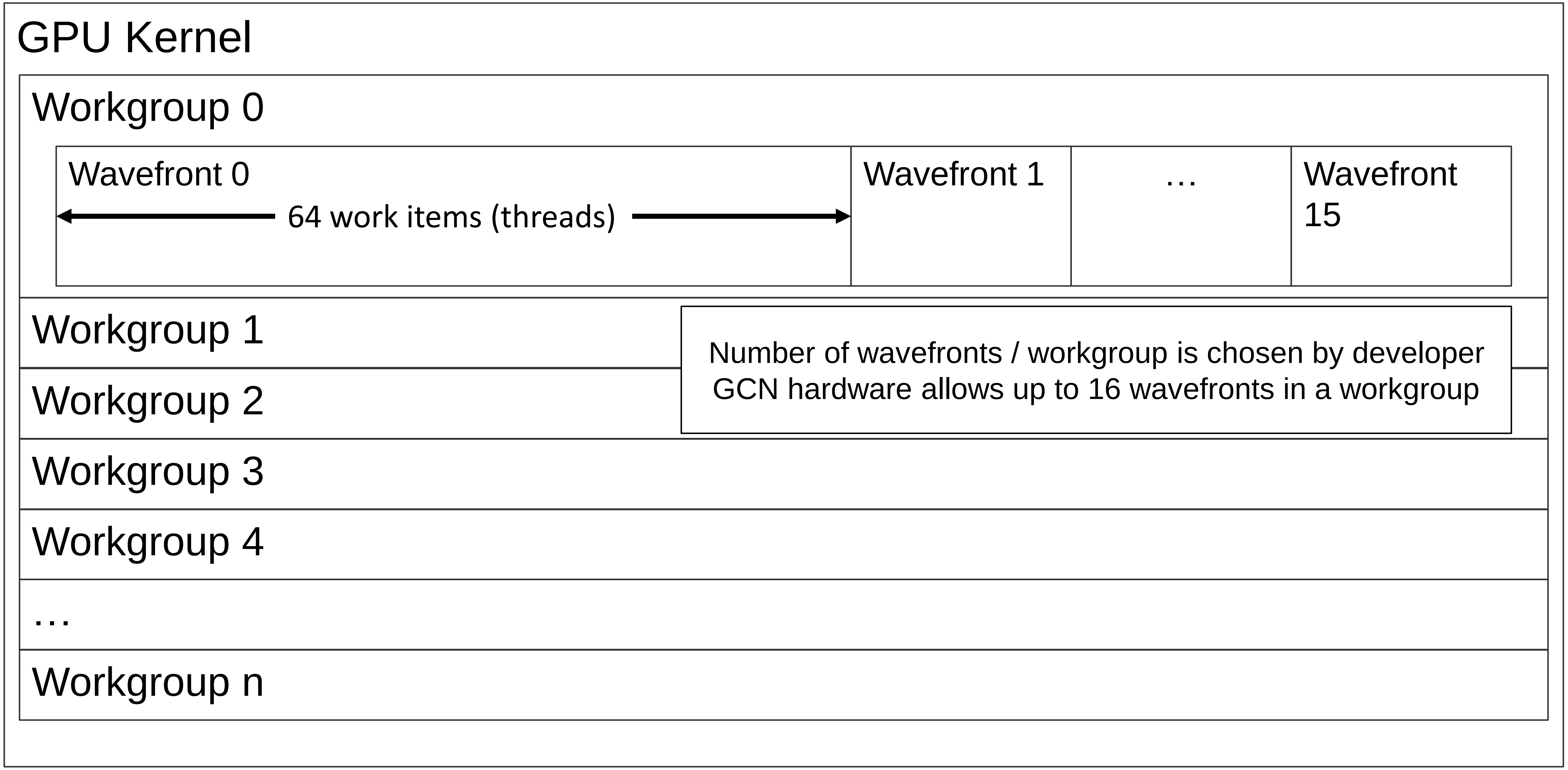
Workgroup 3

Workgroup 4

…

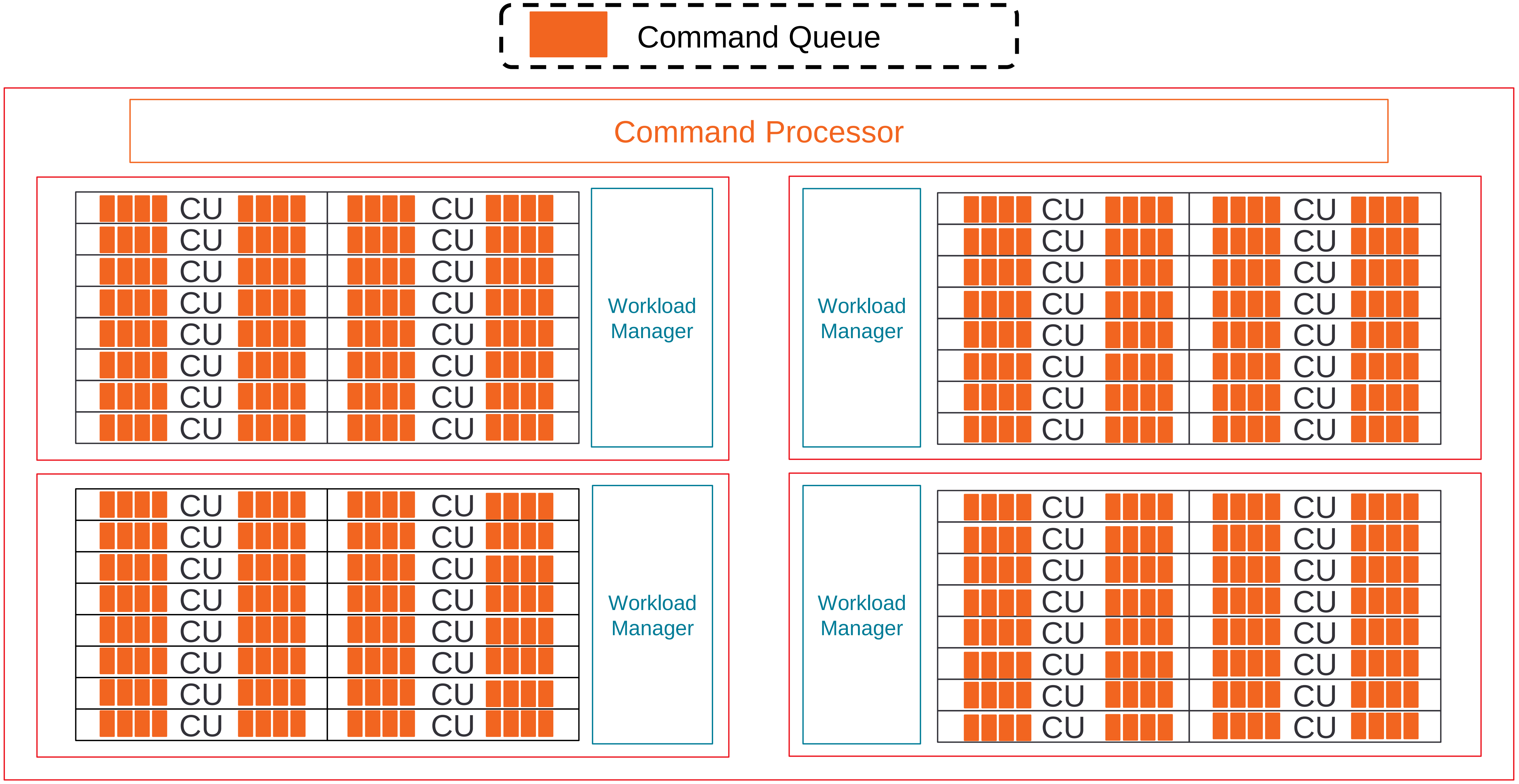
Workgroup n

Overview of GPU Kernels



Number of wavefronts / workgroup is chosen by developer
GCN hardware allows up to 16 wavefronts in a workgroup

Scheduling work to a GPU



Software Terminology

Nvidia/CUDA Terminology	AMD Terminology	Description
Streaming Multiprocessor	Compute Unit (CU)	One of many parallel vector processors in a GPU that contain parallel ALUs. All waves in a workgroups are assigned to the same CU.
Kernel	Kernel	Functions launched to the GPU that are executed by multiple parallel workers on the GPU. Kernels can work in parallel with CPU.
Warp	Wavefront	Collection of operations that execute in lockstep, run the same instructions, and follow the same control-flow path. Individual lanes can be masked off. Think of this as a vector thread. A 64-wide wavefront is a 64-wide vector op.
Thread Block	Workgroup	Group of wavefronts that are on the GPU at the same time. Can synchronize together and communicate through local memory.
Thread	Work Item / Thread	Individual lane in a wavefront. On AMD GPUs, must run in lockstep with other work items in the wavefront. Lanes can be individually masked off. GPU programming models can treat this as a separate thread of execution, though you do not necessarily get forward sub-wavefront progress.

Software Terminology

Nvidia/CUDA Terminology	AMD Terminology	Description
Global Memory	Global Memory	Device DRAM memory accessible by the GPU that goes through some layers cache
Shared Memory	Local Memory	Scratchpad that allows communication between wavefronts in a workgroup
Local Memory	Private Memory	Per-thread private memory, often mapped to registers

AMD GPU Programming Concepts

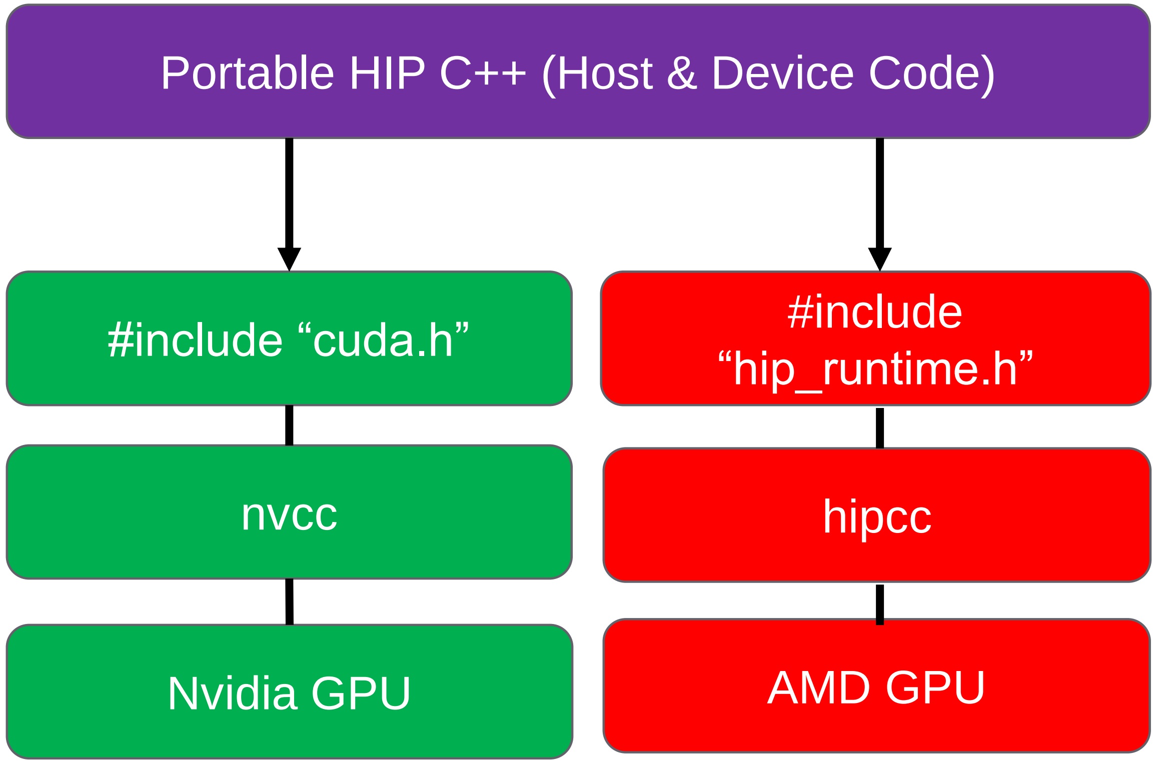
Programming with HIP: Kernels, blocks, threads, and more

What is HIP?

AMD's **H**eterogeneous-compute **I**nterface for **P**ortability, or **HIP**, is a C++ runtime API and kernel language that allows developers to create portable applications that can run on AMD's accelerators as well as CUDA devices

HIP:

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

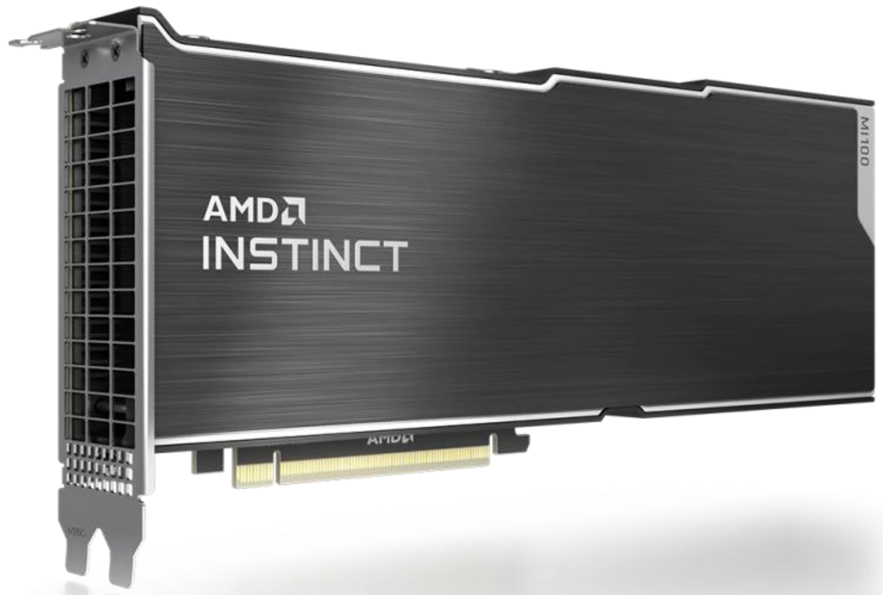


A Tale of Host and Device

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

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

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



HIP API

- Device Management:
  - `hipSetDevice(), hipGetDevice(), hipGetDeviceProperties()`
- Memory Management
  - `hipMalloc(), hipMemcpy(), hipMemcpyAsync(), hipFree(), hipHostMalloc(), hipHostFree()`
- 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()`

Kernels, Memory, and Structure of host code

Device Kernels: The Grid

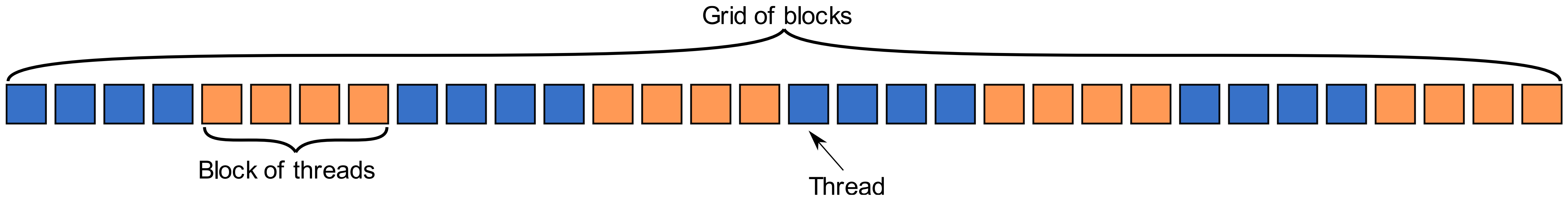
- In HIP, kernels are executed on a 3D “grid”
  - You might feel comfortable thinking in terms of a mesh of points, but it’s not required
- The “grid” is what you can map your problem to
  - It’s not a physical thing, but it can be useful to think that way
- AMD devices (GPUs) support 1D, 2D, and 3D grids, but most work maps well to 1D
- Each dimension of the grid partitioned into equal sized “blocks”
- Each block is made up of multiple “threads”
- The grid and its associated blocks are just organizational constructs
  - The threads are the things that do the work
- If you’re familiar with CUDA already, the grid + block structure is very similar in HIP

Device Kernels: The Grid

Some Terminology:

CUDA	HIP	OpenCL™
grid	grid	NDRange
block	block	work group
thread	work item / thread	work item
warp	wavefront	sub-group

The Grid: blocks of threads in 1D



Threads in grid have access to:

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

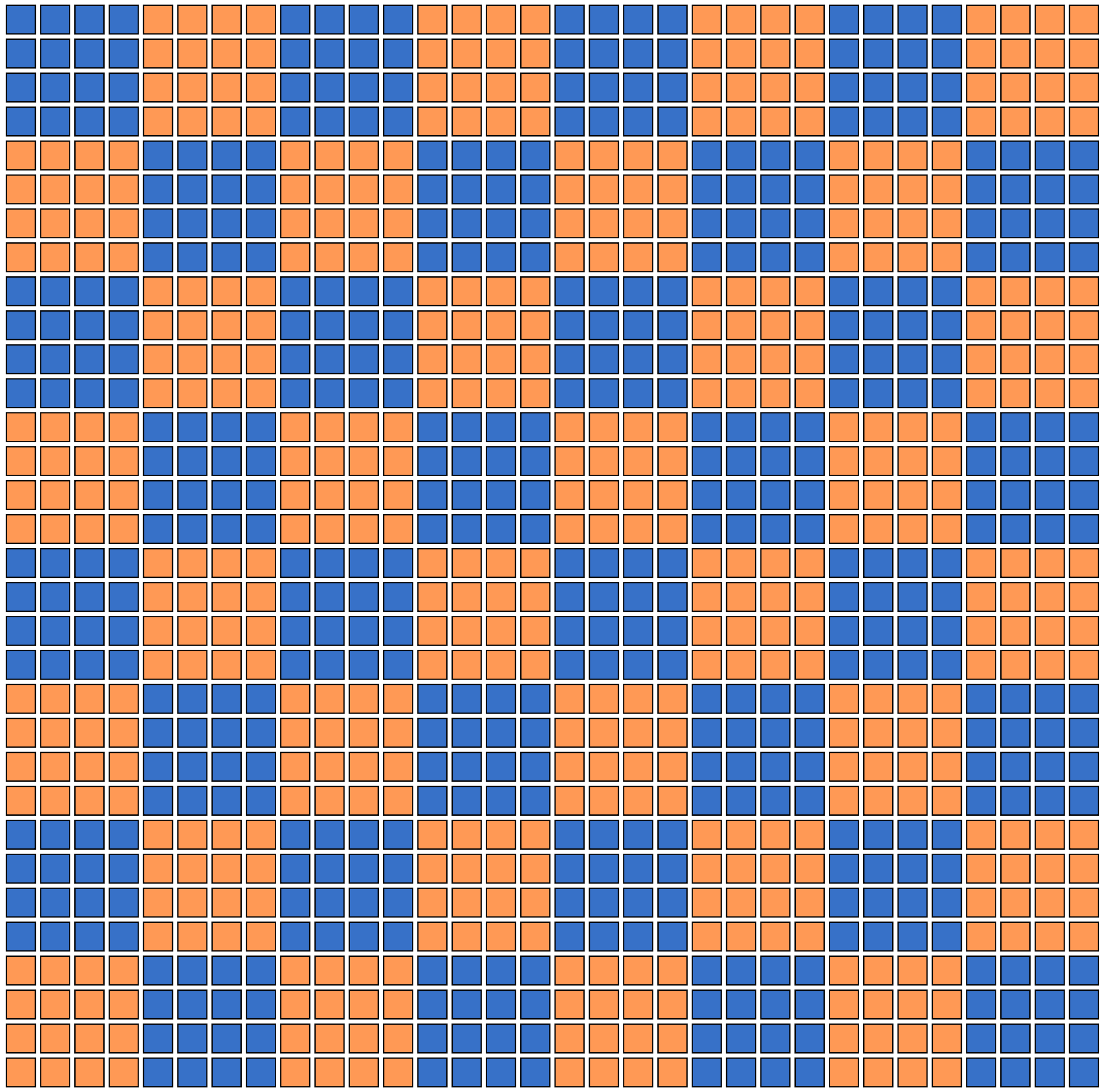
The Grid: blocks of threads in 2D

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

Threads in grid have access to:

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

- 3D (`threadIdx.z, blockIdx.z, ..`)



More info about device: rocminfo

```
Name:                    gfx908
Cache Info:
  L1:                      16(0x10) KB
Cacheline Size:          64(0x40)
Compute Unit:            120
SIMDs per CU:            4
Shader Engines:          8
Wavefront Size:          64(0x40)
Workgroup Max Size:      1024(0x400)
Workgroup Max Size per Dimension:
  x                        1024(0x400)
  y                        1024(0x400)
  z                        1024(0x400)
Max Waves Per CU:        40(0x28)
Max Work-item Per CU:    2560(0xa00)
Grid Max Size:           4294967295(0xffffffff)
Grid Max Size per Dimension:
  x                        4294967295(0xffffffff)
  y                        4294967295(0xffffffff)
  z                        4294967295(0xffffffff)
```

Kernels

A simple embarrassingly parallel loop

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

Can be translated into a GPU kernel:

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

- A device function that will be launched from the host program is called a kernel and is declared with the __global__ attribute
- Kernels should be declared void
- All pointers passed to kernels must point to memory on the device (more later)
- 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 (we'll see why in a minute)

Kernels

Kernels are launched from the host:

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

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

Analogous to CUDA kernel launch syntax:

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

Working with Templated Kernels

```
template<typename Tdata, int index>
__global__ void myKernel(int N, Tdata* d_a) {
  /* kernel code */
}



/* Host code */
int ix = 1000;
hipLaunchKernelGGL(HIP_KERNEL_NAME(MyKernel<double, ix>), blocks, threads, 0, 0, N, d_a)
```

SIMD operations

Why blocks and threads?

Natural mapping of kernels to hardware:

- Blocks are dynamically scheduled onto CUs
- All threads in a block execute on the same CU
- Threads in a block share LDS memory and L1 cache
- Threads in a block are executed in **64-wide** chunks called “wavefronts”
- Wavefronts execute on SIMD units (Single Instruction Multiple Data)
- If a wavefront stalls (e.g., data dependency) CUs can quickly context switch to another wavefront

A good practice is to make the block size a multiple of 64 and have several wavefronts (e.g., 256 threads)

Device Memory

The host instructs the device to allocate memory in VRAM and records a pointer to device memory:

```
int main() {
  …
  int N = 1000;
  size_t Nbytes = N*sizeof(double);
  double *h_a = (double*) malloc(Nbytes);                  // Allocate host memory

  double *d_a = NULL;
  hipMalloc(&d_a, Nbytes);                                 // Allocate Nbytes on device
  …
  free(h_a);                                               // Free host memory
  hipFree(d_a);                                            // Free device memory
}
```

Device Memory

The host queues memory transfers:

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

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

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

Device Memory

Can copy strided sections of arrays:

```
hipMemcpy2D(d_a,          //pointer to destination
            DLDAbytes,    //pitch of destination array
            h_a,          //pointer to source
            LDAbytes,     //pitch of source array
            Nbytes,       //number of bytes in each row
            Nrows,        //number of rows to copy
            hipMemcpyHostToDevice);
```

Error Checking

- Most HIP API functions return error codes of type `hipError_t`

```
hipError_t status1 = hipMalloc(…);
hipError_t status2 = hipMemcpy(…);
```

- If API function was error-free, returns `hipSuccess`, otherwise returns an error code

- Can also peek/get at last error returned with

```
hipError_t status3 = hipGetLastError();
hipError_t status4 = hipPeekLastError();
```

- Can get a corresponding error string using `hipGetErrorString(status)`. Helpful for debugging, e.g.,

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

Device Management

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

- Host can query number of devices visible to system:

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

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

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

- Host can query what device is currently selected:

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

- The host can manage several devices by changing the default device at runtime
- Different processes can use different devices or/and over-subscribe (share) the same device

Device Properties

The host can also query a device's properties:

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

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

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(hipSetDevice (0));
  HIP_CHECK(hipMalloc(&d_a, Nbytes));
  …
  HIP_CHECK(hipMemcpy(d_a, h_a, Nbytes, hipMemcpyHostToDevice));   //copy data to device
  hipLaunchKernelGGL(myKernel, dim3((N+256-1)/256,1,1), dim3(256,1,1), 0, 0, N, d_a); //Launch kernel
  HIP_CHECK(hipGetLastError());
  HIP_CHECK(hipMemcpy(h_a, d_a, Nbytes, hipMemcpyDeviceToHost));
  …
  free(h_a);                  //free host memory
  HIP_CHECK(hipFree(d_a));    //free device memory
}
```

The host waits for the kernel to finish here

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

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

Asynchronous Computing

Blocking vs Nonblocking API functions

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

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

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

- Like `hipLaunchKernelGGL`, this function takes an argument of type hipStream_t

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

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);
```

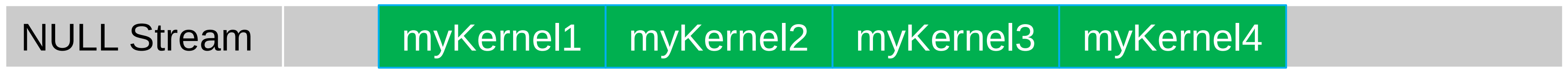
- Passing `0 or NULL` as the `hipStream_t` argument to a function instructs the function to execute on a stream called the 'NULL Stream':
  - No task on the NULL stream will begin until **all previously enqueued tasks in all other streams have completed**
  - Blocking calls like `hipMemcpy` run on the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

Streams

- Suppose we have 4 small kernels to execute:

```
hipLaunchKernelGGL(myKernel1, dim3(1), dim3(256), 0, 0, 256, d_a1);
hipLaunchKernelGGL(myKernel2, dim3(1), dim3(256), 0, 0, 256, d_a2);
hipLaunchKernelGGL(myKernel3, dim3(1), dim3(256), 0, 0, 256, d_a3);
hipLaunchKernelGGL(myKernel4, dim3(1), dim3(256), 0, 0, 256, d_a4);
```

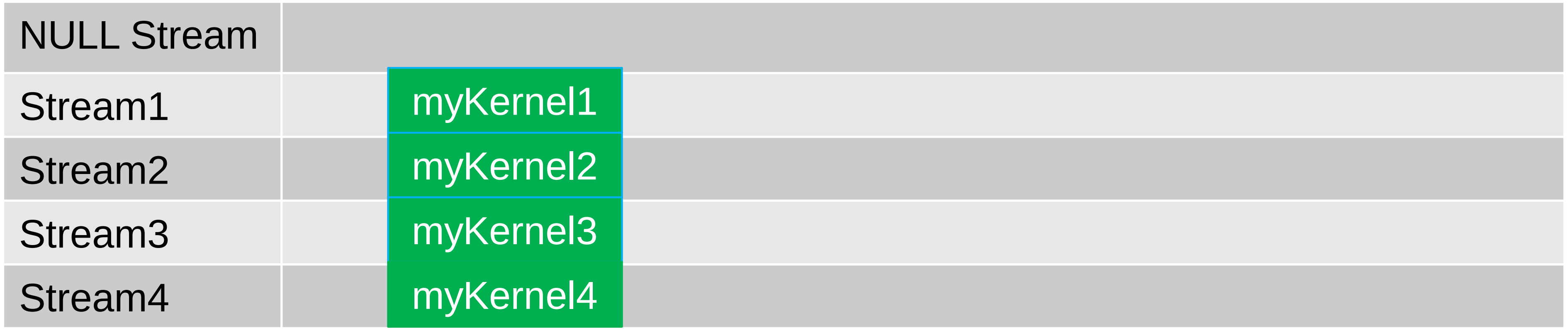
- Even though these kernels use only one block each, they'll execute in serial on the NULL stream:



Streams

- With streams we can effectively share the GPU's compute resources:

```
hipLaunchKernelGGL(myKernel1, dim3(1), dim3(256), 0, stream1, 256, d_a1);
hipLaunchKernelGGL(myKernel2, dim3(1), dim3(256), 0, stream2, 256, d_a2);
hipLaunchKernelGGL(myKernel3, dim3(1), dim3(256), 0, stream3, 256, d_a3);
hipLaunchKernelGGL(myKernel4, dim3(1), dim3(256), 0, stream4, 256, d_a4);
```



Note 1: Check that the kernels modify different parts of memory to avoid data races

Note 2: With large kernels, overlapping computations may not help performance

Streams

- There is another use for streams besides concurrent kernels:
  - Overlapping kernels with data movement

- AMD GPUs have separate engines for:
  - Host->Device memcpys
  - Device->Host memcpys
  - Compute kernels

- These three different operations can overlap without dividing the GPU’s resources
  - The overlapping operations should be in separate, non-NULL, streams
  - The host memory should be **pinned**

Pinned Memory

Host data allocations are pageable by default. The GPU can directly access host data if it is pinned instead.

- Allocating pinned host memory:

```
double *h_a = NULL;
hipHostMalloc(&h_a, Nbytes);
```

- Free pinned host memory:

```
hipHostFree(h_a);
```

- Host<->Device effective data transfer rate **increases significantly when host memory is pinned**
  - It is good practice to allocate pinned memory for data that is frequently transferred to/from the device

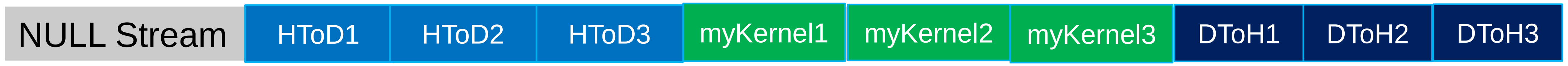
Streams

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));

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



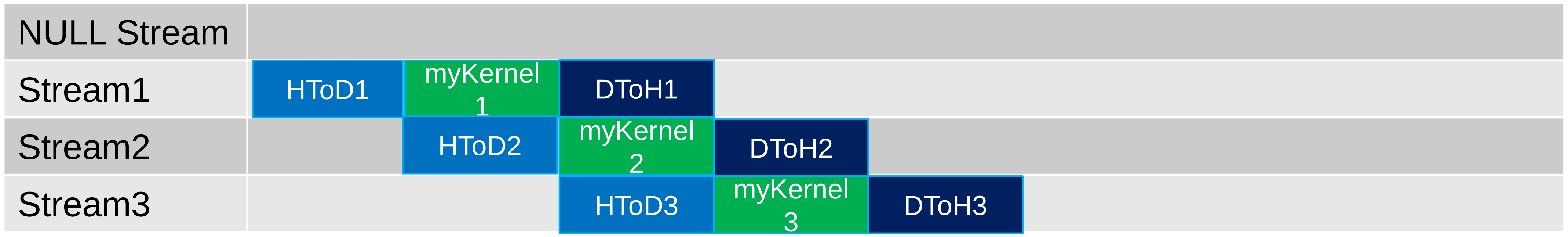
Streams

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);

hipLaunchKernelGGL(myKernel1, blocks, threads, 0, stream1, N, d_a1);
hipLaunchKernelGGL(myKernel2, blocks, threads, 0, stream2, N, d_a2);
hipLaunchKernelGGL(myKernel3, blocks, threads, 0, stream3, N, d_a3);

hipMemcpyAsync(h_a1, d_a1, Nbytes, hipMemcpyDeviceToHost, stream1);
hipMemcpyAsync(h_a2, d_a2, Nbytes, hipMemcpyDeviceToHost, stream2);
hipMemcpyAsync(h_a3, d_a3, Nbytes, hipMemcpyDeviceToHost, stream3);
```



Synchronization

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

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

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

Events

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

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

We queue an event into a stream:

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

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

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

```
hipEventDestroy(event);
```

Events

What can we do with queued events?

- `hipEventSynchronize(event);`
  - Block host until event reports complete
  - Only a synchronization point with respect to the stream where event was enqueued

- `hipEventElapsedTime(&time, startEvent, endEvent);`
  - Returns the time in ms between when two events, startEvent and endEvent, completed
  - Can be very useful for timing kernels/memcpys

- `hipStreamWaitEvent(stream, event);`
  - Non-blocking for host
  - Instructs all future work submitted to stream to wait until event reports complete
  - Primary way we enforce an ‘ordering’ between tasks in separate streams

Streams

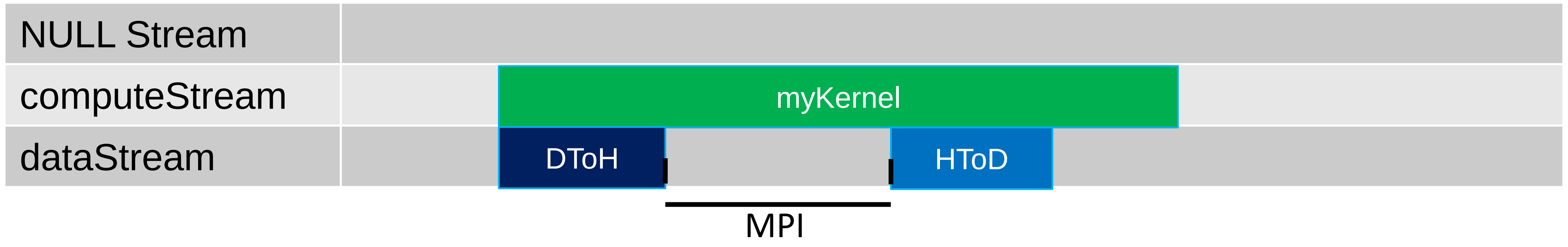
A common use-case for streams is MPI traffic:

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

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

//Exchange data with MPI
MPI_Data_Exchange(h_commBuffer);

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



Streams

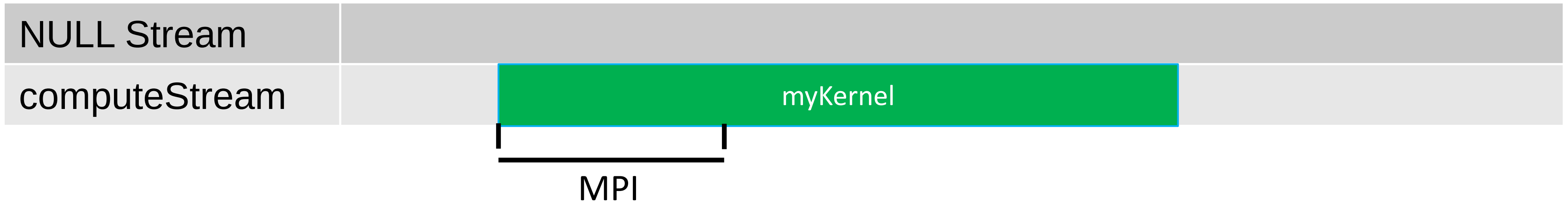
With a GPU-aware MPI stack, the Host<->Device traffic can be omitted:

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

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

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

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



Device code, Shared Memory and Thread Synchronization

Function Qualifiers

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

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

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

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

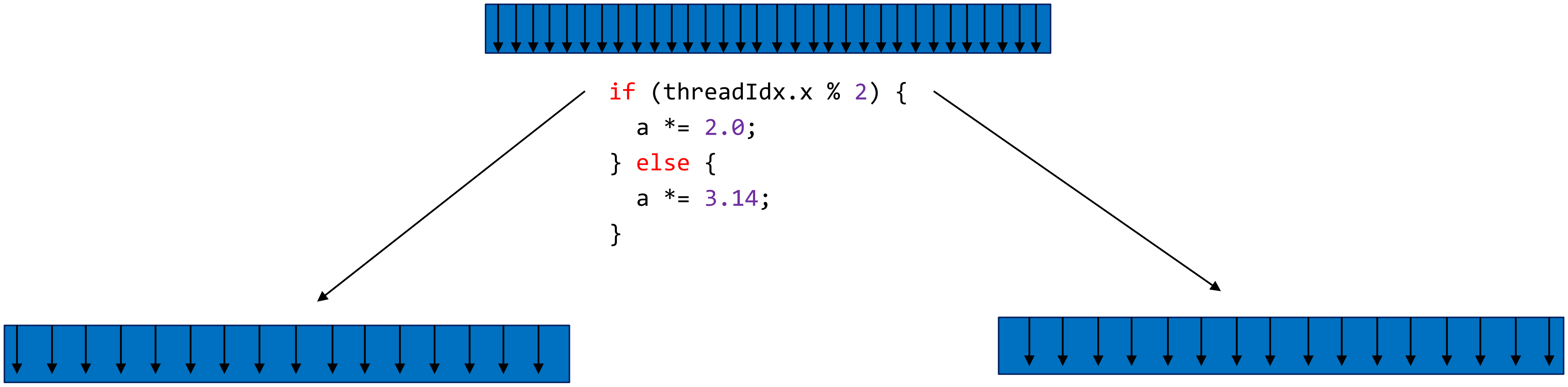
SIMD Execution

On SIMD units, be aware of divergence

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

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

SIMD Execution



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

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

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

Memory declarations in Device Code

- malloc/free not supported in device code

- Variables/arrays can be declared on the stack

- Stack variables declared in device code are allocated in registers and are private to each thread

- Threads can all access common memory via device pointers, but otherwise do not share memory
  - Important exception: __shared__ memory

- Stack variables declared as __shared__:
  - Allocated once per block in LDS memory
  - Shared and accessible by all threads in the same block
  - Access is faster than device global memory (but slower than register)
  - May be statically and/or dynamically allocated

Shared Memory

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

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

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

__syncthreads();

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

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

Thread Synchronization

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

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

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

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

HIP API

- Device Management:
  - `hipSetDevice(), hipGetDevice(), hipGetDeviceProperties()`
- Memory Management
  - `hipMalloc(), hipMemcpy(), hipMemcpyAsync(), hipFree(), hipHostMalloc(), hipHostFree()`
- 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()`

hipcc usage

Usage is straightforward. Accepts all/any flags that vanilla clang accepts, e.g.,

```
hipcc vectoradd_hip.cpp -o vectorAdd
```

Set HIPCC_VERBOSE=7 to see a bunch of useful information

- Compile and link lines, various paths

```
$ HIPCC_VERBOSE=7 hipcc -O3 vectoradd_hip.cpp
HIP_PATH=/opt/rocm-4.2.0/hip
HIP_PLATFORM=amd
HIP_COMPILER=clang
HIP_RUNTIME=rocclr
ROCM_PATH=/opt/rocm-4.2.0
HIP_ROCCLR_HOME=/opt/rocm-4.2.0/hip
HIP_CLANG_PATH=/opt/rocm-4.2.0/llvm/bin
HIP_CLANG_INCLUDE_PATH=/opt/rocm-4.2.0/llvm/lib/clang/12.0.0/include
HIP_INCLUDE_PATH=/opt/rocm-4.2.0/hip/include
HIP_LIB_PATH=/opt/rocm-4.2.0/hip/lib
DEVICE_LIB_PATH=/opt/rocm-4.2.0/amdgcn/bitcode
hipcc-args: -O3 vectoradd_hip.cpp
hipcc-cmd: "/opt/rocm-4.2.0/llvm/bin/clang"  -std=c++11 -isystem "/opt/rocm-4.2.0/llvm/lib/clang/12.0.0/include/.." -isystem
/opt/rocm-4.2.0/hsa/include -isystem "/opt/rocm-4.2.0/hip/include" --offload-arch=gfx908 --offload-arch=gfx908 --offload-
arch=gfx908 --offload-arch=gfx908 --offload-arch=gfx908 --offload-arch=gfx908 --offload-arch=gfx908 --offload-arch=gfx908 -mllvm -
amdgpu-early-inline-all=true -mllvm -amdgpu-function-calls=false -fhip-new-launch-api --driver-mode=g++ -L"/opt/rocm-
4.2.0/hip/lib" -lgcc_s -lgcc -lpthread -lm -lrt  -O3 -x hip vectoradd_hip.cpp -Wl,--enable-new-dtags -Wl,--rpath=/opt/rocm-
4.2.0/hip/lib:/opt/rocm-4.2.0/lib -lamdhip64  -L/opt/rocm-4.2.0/llvm/bin/../lib/clang/12.0.0/lib/linux -lclang_rt.builtins-x86_64
```

Querying System

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

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

AMD GPU programming resources

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

Disclaimer