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

9/6/2019

Agenda

- Introduction (5 minutes)
- AMD GPU Hardware (10 minutes)
- GPU Programming Concepts (45 minutes)
- GPU Programming Software (15 minutes)
- Porting existing CUDA codes to HIP (15 mins)

AMD

What we won’t cover today (but is still important)

- Profiling:
  - rocprofiler / roctracer: libraries for collecting GPU hardware counters and application traces
  - Install: `sudo apt install rocprofiler-dev roctracer-dev`
- Debugging:
  - rocr_debug_agent: print state of wavefronts on memory violation / signals
  - HIP debugging tips
  - In kernel printf

What we won’t cover today (but is still important)

- AOMP (AMD OpenMP Compiler):
  - OpenMP 4.5+ support, “target” pragmas, device offloading
- GPU Libraries:
  - hipBLAS: BLAS functionality on GPUs
  - rocFFT: FFTs on GPUs
  - rocRAND: random number generation
  - rocPRIM / hipCUB: high performance GPU primitives
  - Tensile: GEMMs, tensor contractions
  - hipSPARSE: BLAS for sparse matrices / vectors
  - rocALUTION: iterative sparse solvers
  - MIOpen, TensorFlow, PyTorch: machine learning

Comments

- Slides will be provided
- Ask questions in the google doc; we will be monitoring it
- Focus is on single node / device:
  - Little discussion of MPI, or multi-node

Please share any feedback or ask questions in the Google Doc

AMD

AMD GCN GPU Hardware

Nick Curtis <nicholas.curtis@amd.com>

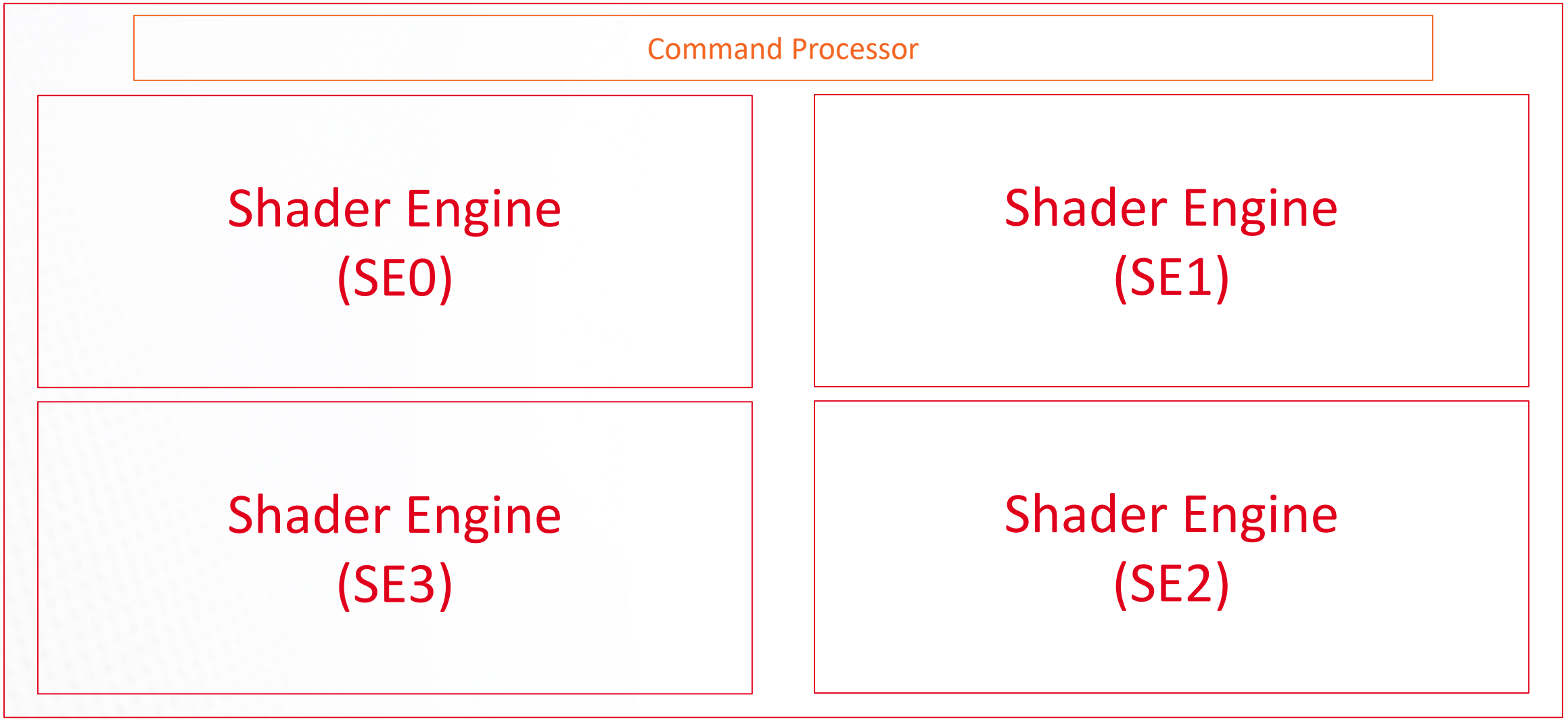
AMD GCN GPU HARDWARE

AGENDA

Introduction	GCN Hardware Overview
	AMD GPU Compute Terminology
AMD GPU Architecture	GPU Memory and I/O System
	GCN Compute Unit Internals

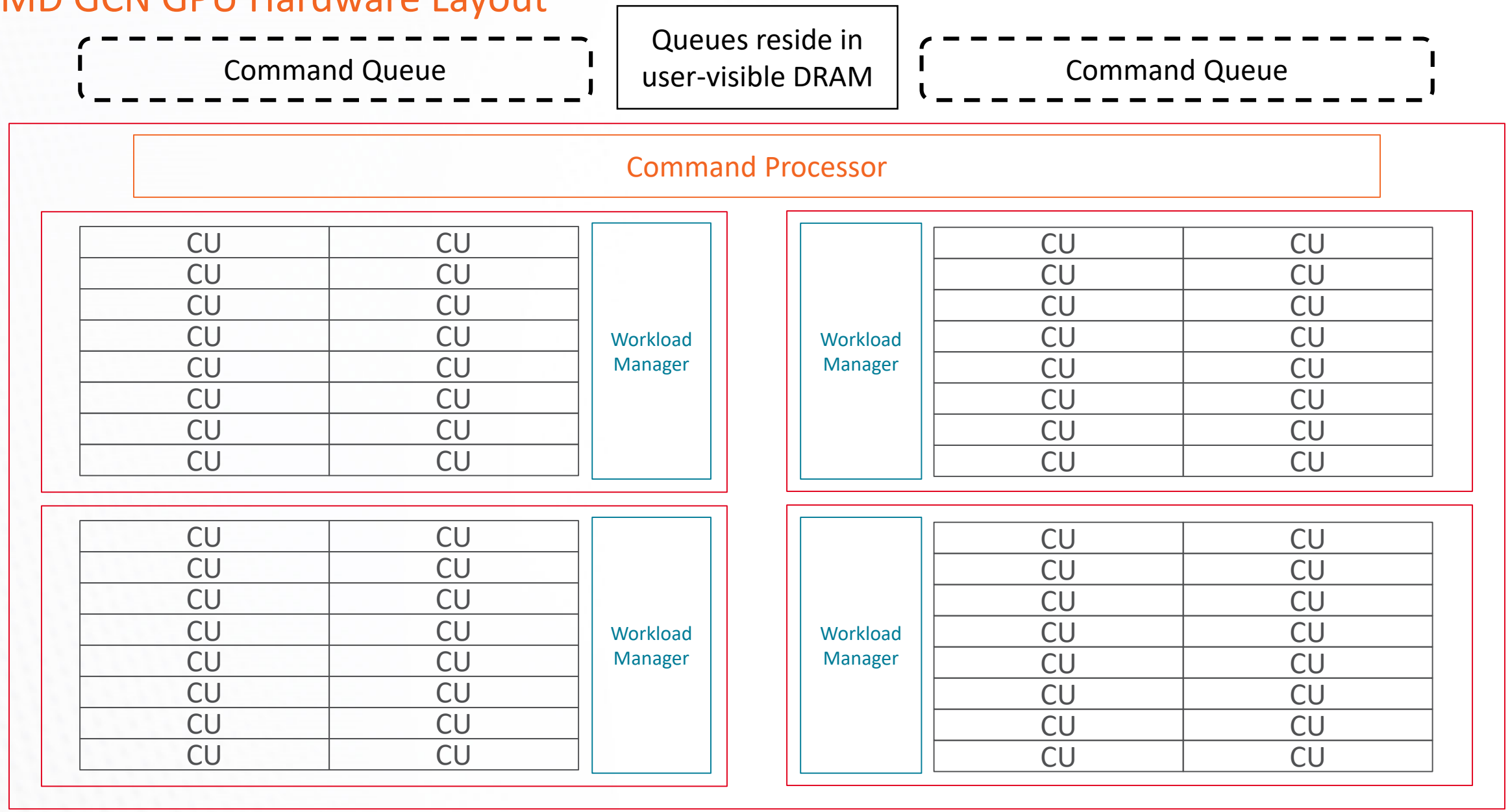
AMD

AMD GCN GPU Hardware Layout



AMD

AMD GCN GPU Hardware Layout



AMD

Hardware Configuration Parameters on Modern AMD GPUs

GPU SKU	Shader Engines	CUs / SE
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

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.

AMD

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

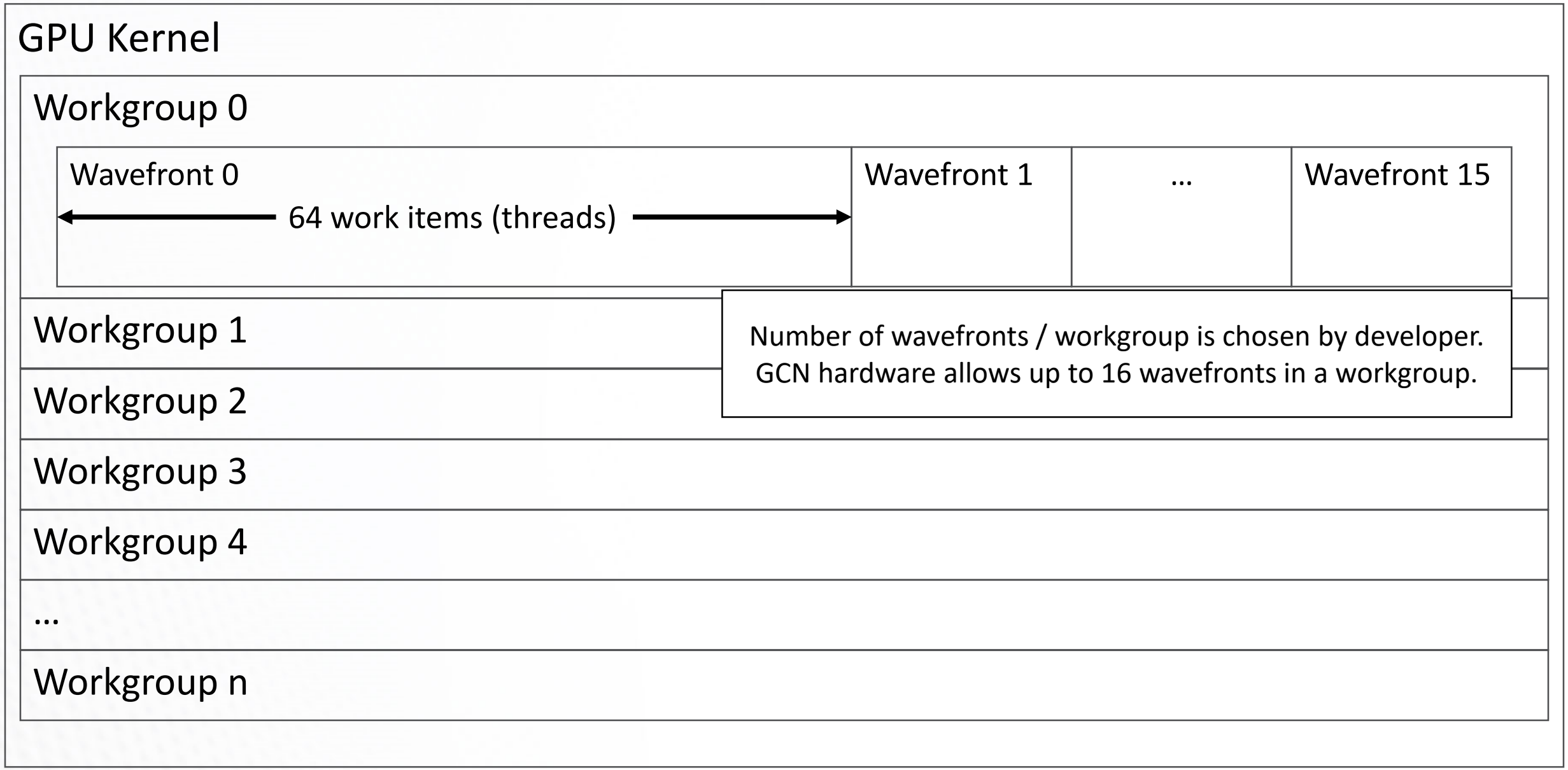
Workgroup 4

…

Workgroup n

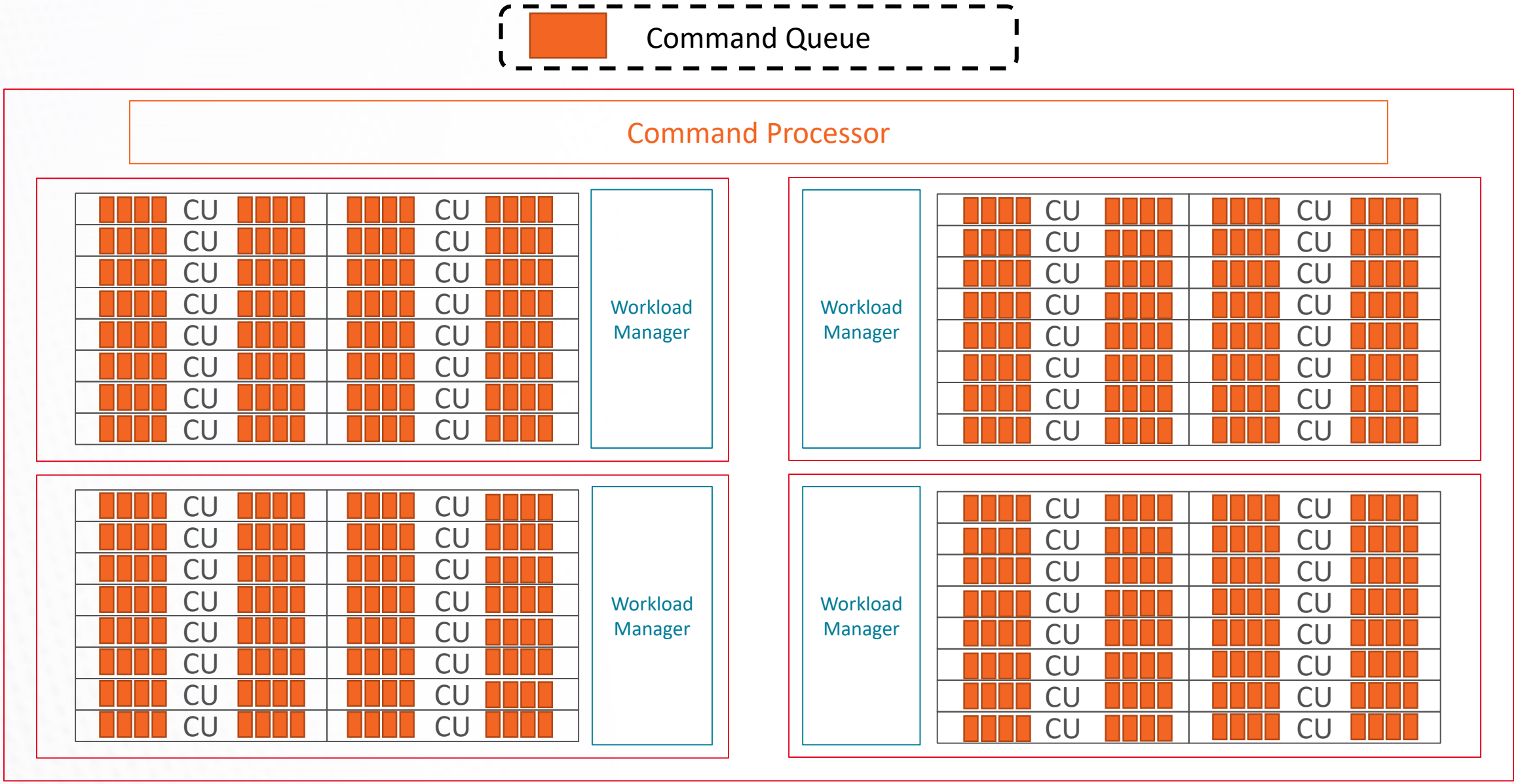
AMD

Overview of GPU Kernels



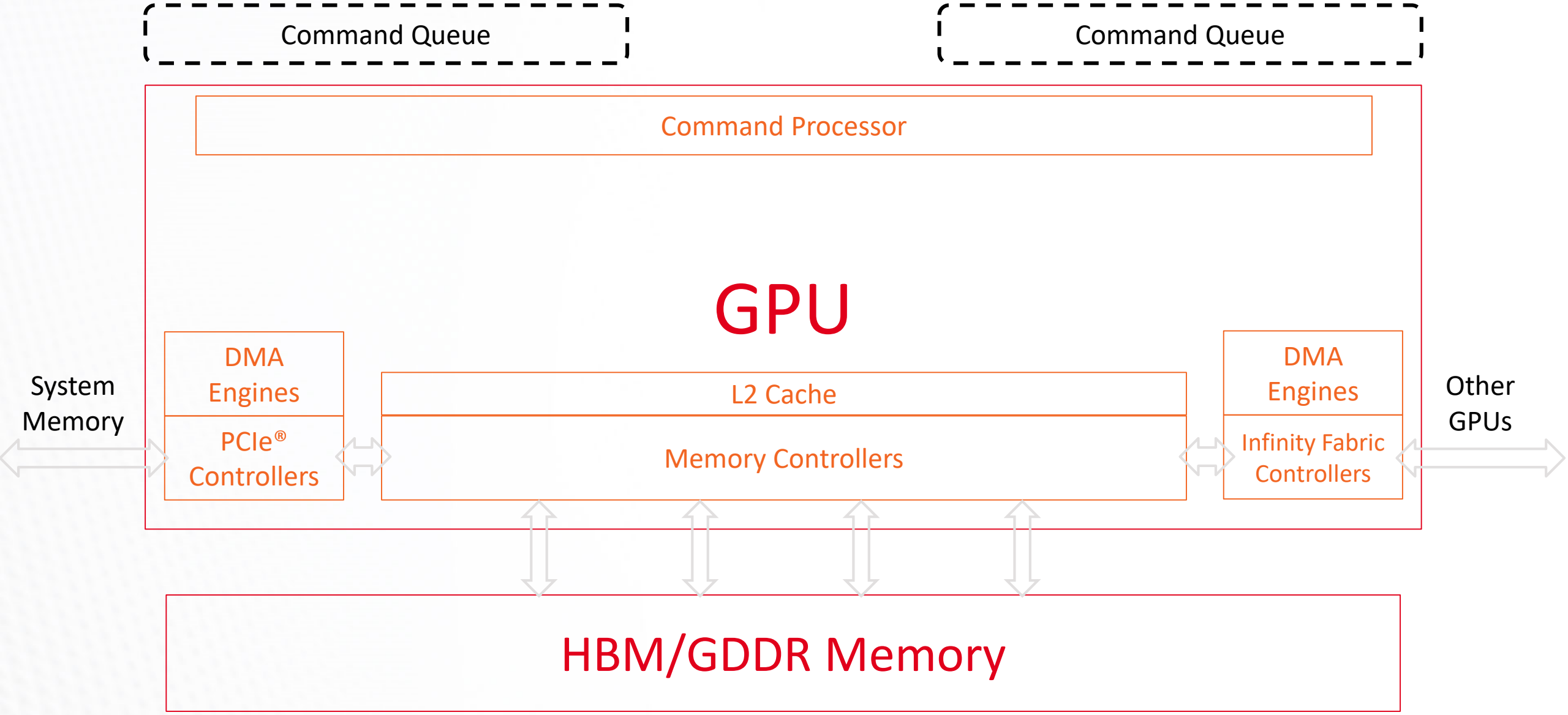
AMD

Scheduling work to a GPU



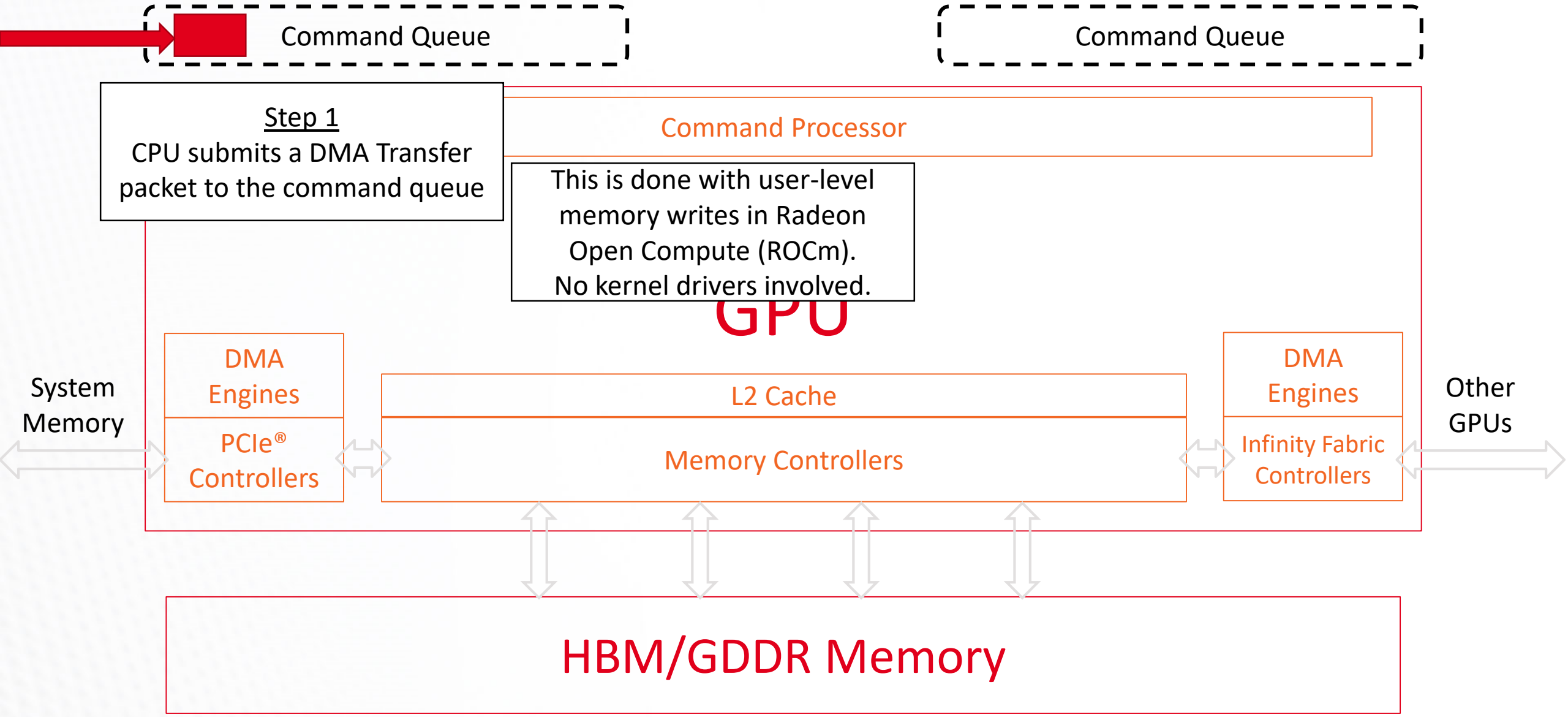
AMD

GPU Memory, I/O, and Connectivity



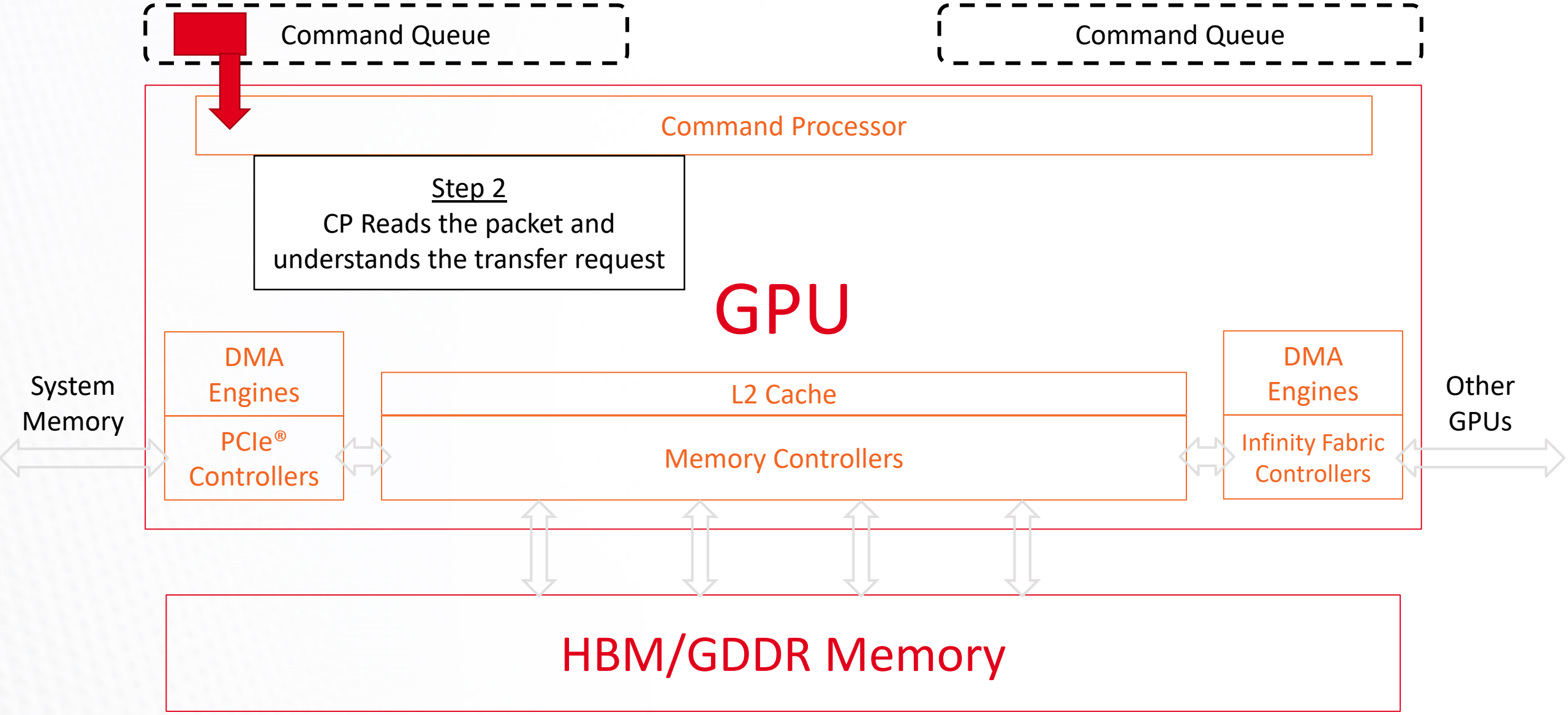
AMD

DMA Engines Accept Work from the Same Queues



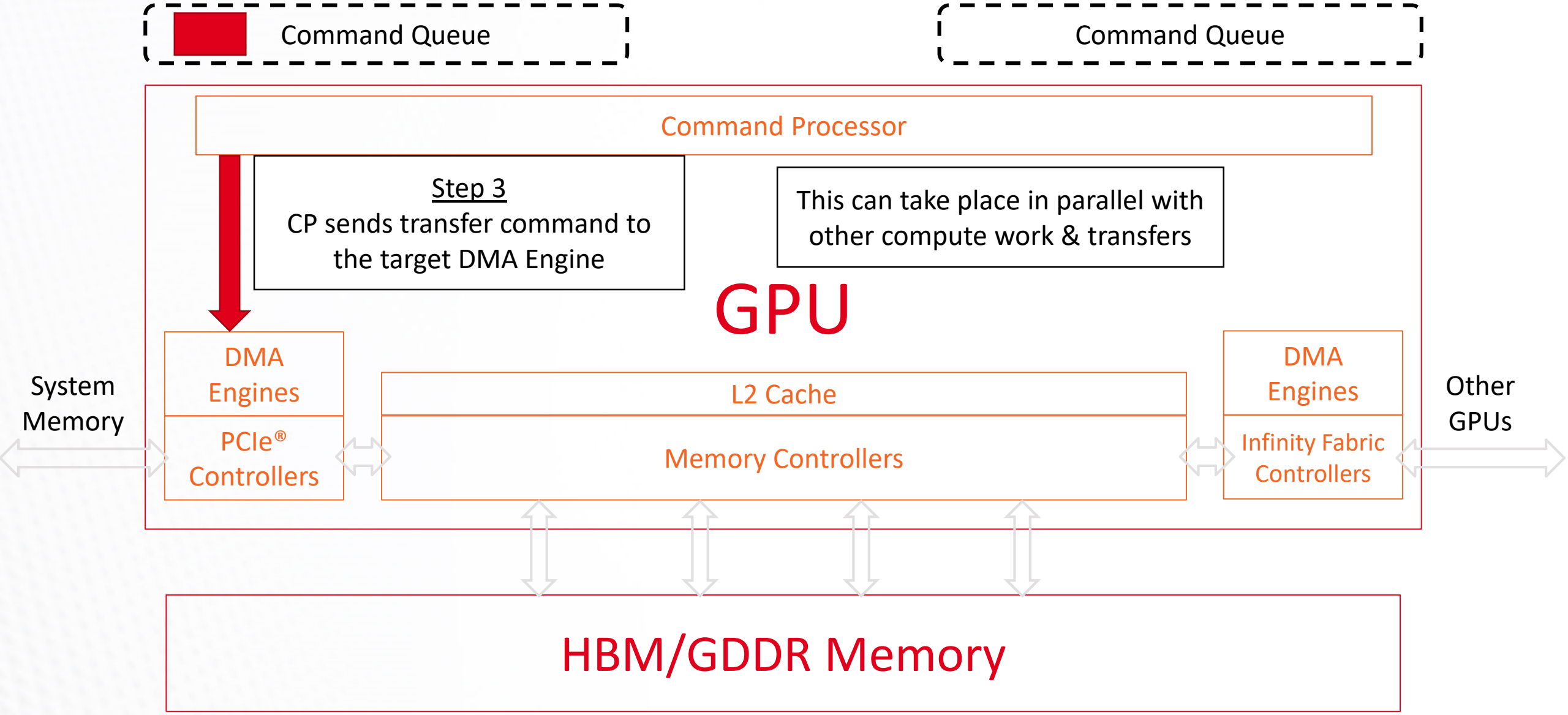
AMD

DMA Engines Accept Work from the Same Queues



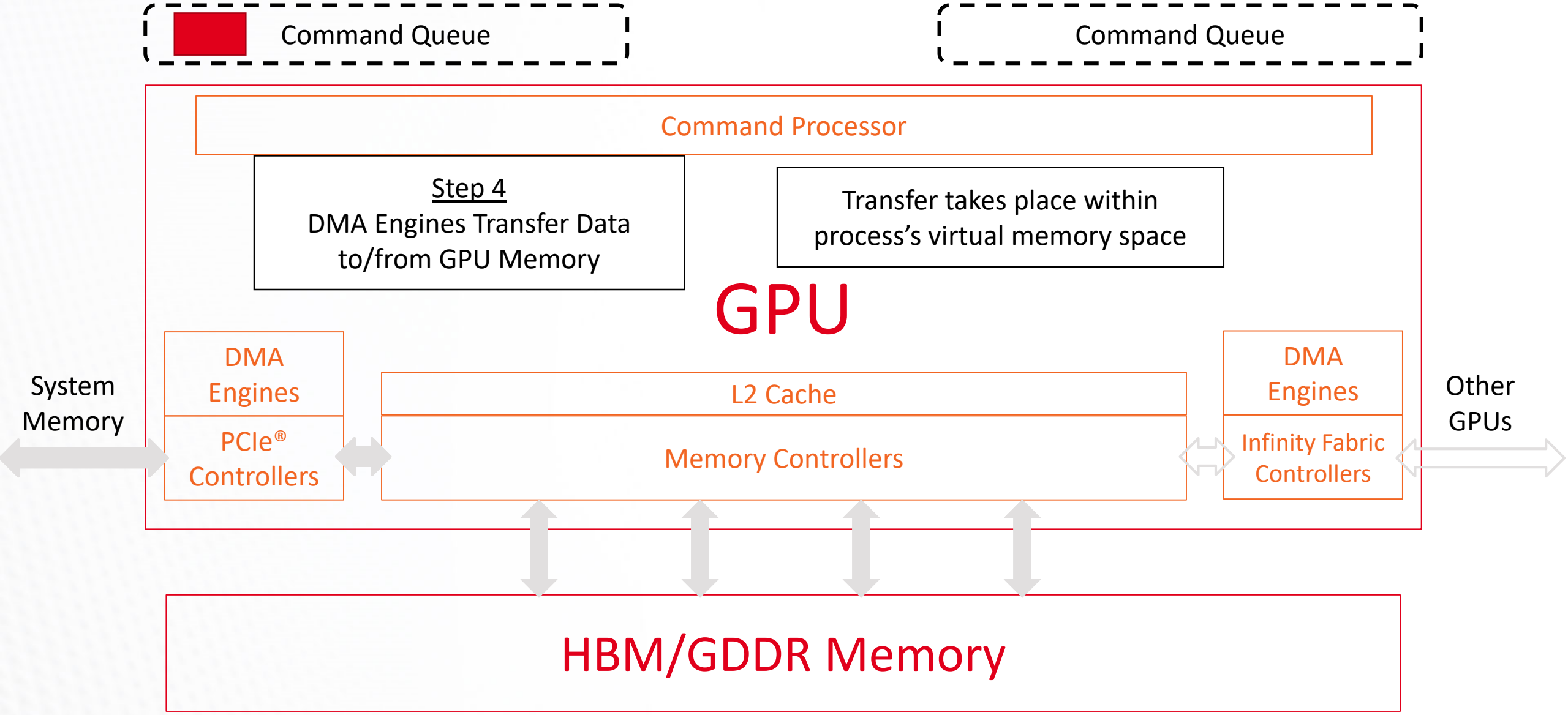
AMD

DMA Engines Accept Work from the Same Queues



AMD

DMA Engines Accept Work from the Same Queues



AMD

The GCN Compute Unit (CU)

Compute Unit (CU)

- The workload manager sends work packages (i.e. blocks of threads) to the Compute Units (CUs)
  - Blocks are executed in wavefronts (groups of 64 threads in SIMD)
  - All wavefronts in a block are guaranteed to reside in the same CU
  - The CU's scheduler can hold wavefronts from many blocks
  - At most 40 wavefronts total per CU

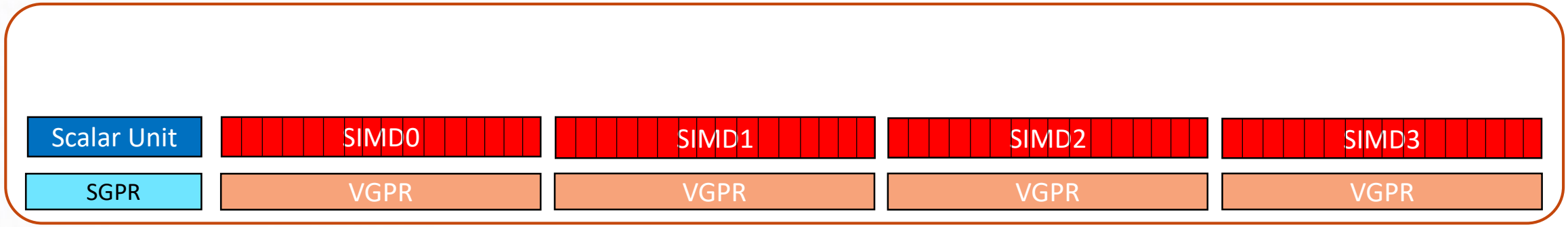
AMD

The GCN Compute Unit (CU)

- The Scalar Unit (SU)
  - Shared by all threads in each wavefront, accessed on a per-wavefront level
  - Threads in a wavefront performing the exact same operation can offload this instruction to the SU
  - Used for control flow, pointer arithmetic, loading a common value, etc.
  - Has its own pool of Scalar General-Purpose Register (SGPR) file, 12.5KiB per CU

AMD

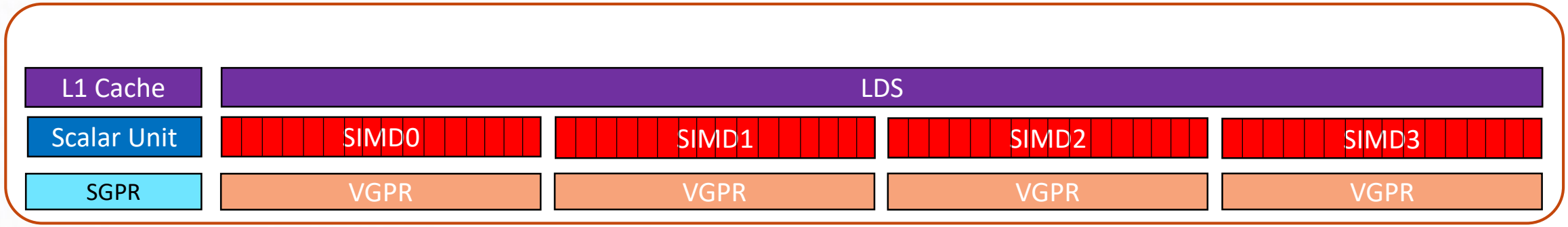
The GCN Compute Unit (CU)



- SIMD Units
  - 4x SIMD vector units (each 16 lanes wide)
  - 4x 64KiB (256KiB total) Vector General-Purpose Register (VGPR) file
    - A maximum of 256 total registers per SIMD lane – each register is 64x 4-byte entries
  - Instruction buffer for 10 wavefronts on each SIMD unit
    - Each wavefront is local to a single SIMD unit, not spread among the 4 (more on this in a moment)

AMD

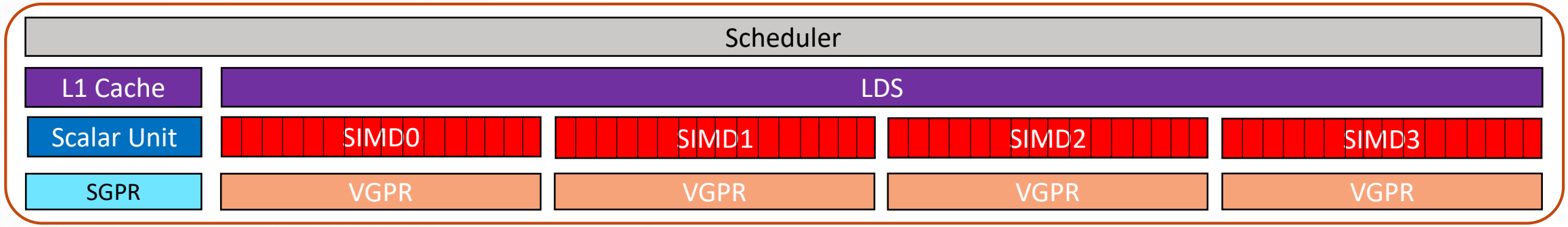
The GCN Compute Unit (CU)



- 64KiB Local Data Share (LDS, or shared memory)
  - 32 banks with conflict resolution
  - Can share data between all threads in a block
- 16 KiB Read/Write L1 vector data cache
  - Write-through; L2 cache is the coherence point – shared by all CUs

AMD

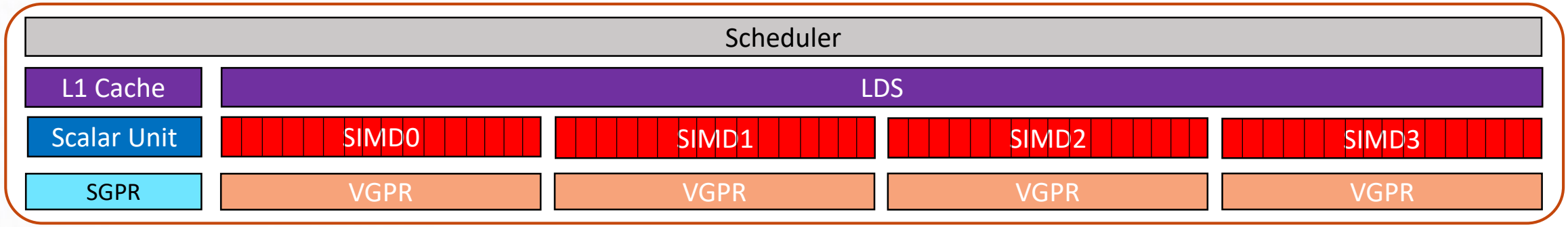
The GCN Compute Unit (CU)



- Scheduler
  - Buffer for 40 wavefronts – 2560 threads
  - Separate decode/issue for
    - VALU, VGPR load/store
    - SALU, SGPR load/store
    - LDS load/store
    - Global mem load/store
    - Special instructions (NoOps, barriers, branch instructions)

AMD

The GCN Compute Unit (CU)



- Scheduler
  - At each clock, waves on **1 SIMD unit** are considered for execution (Round Robin scheduling among SIMDs)
  - At most **1 instruction per wavefront** may be issued
  - At most **1 instruction from each category** may be issued (S/V ALU, S/V GPR, LDS, global, branch, etc)
  - **Maximum of 5** instructions issued to wavefronts on a single SIMD, per cycle per CU
  - Some instructions take 4 or more cycles to retire (e.g. FP32VALU instruction on 1 wavefront using 16-wide SIMD)
    - Round robin scheduling of SIMDs hides execution latency
    - Programmer can still ‘pretend’ CU operates in 64-wide SIMD

AMD

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	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

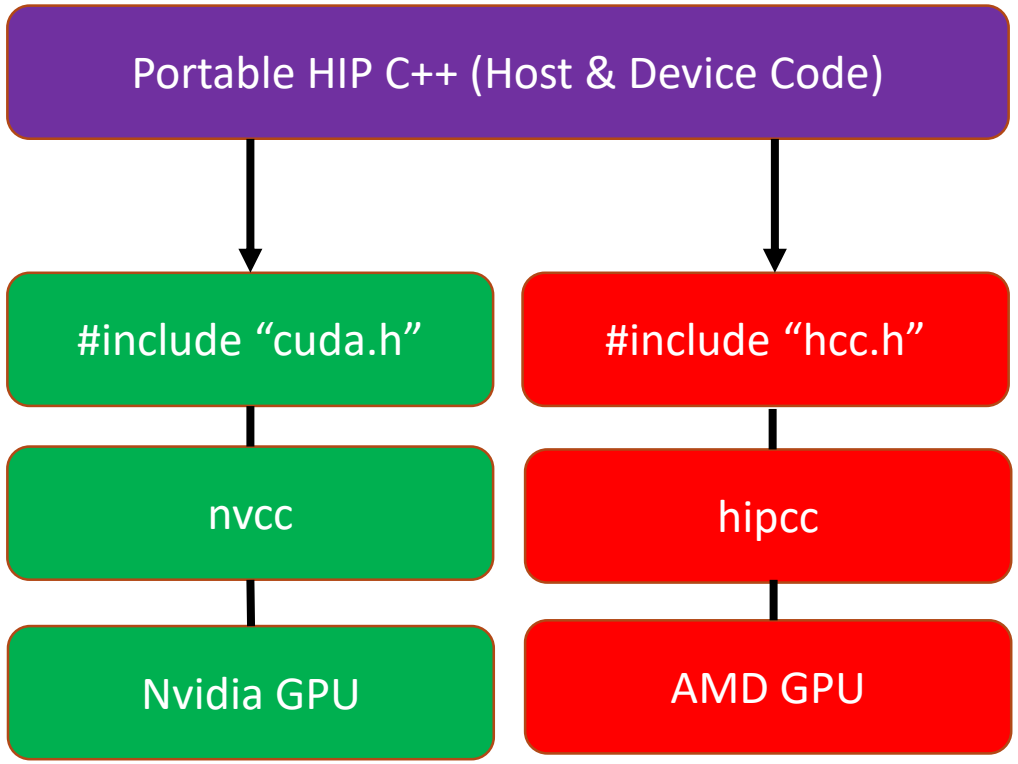
Damon McDougall <damon.mcdougall@amd.com>

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.



AMD

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”

AMD

HIP API

- Device Management:
  - `hipSetDevice()`, `hipGetDevice()`, `hipGetDeviceProperties()`
- Memory Management
  - `hipMalloc()`, `hipMemcpy()`, `hipMemcpyAsync()`, `hipFree()`
- Streams
  - `hipStreamCreate()`, `hipSynchronize()`, `hipStreamSynchronize()`, `hipStreamFree()`
- 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()`

AMD

Kernels, memory, and structure of host code

AMD

Device Kernels: The Grid

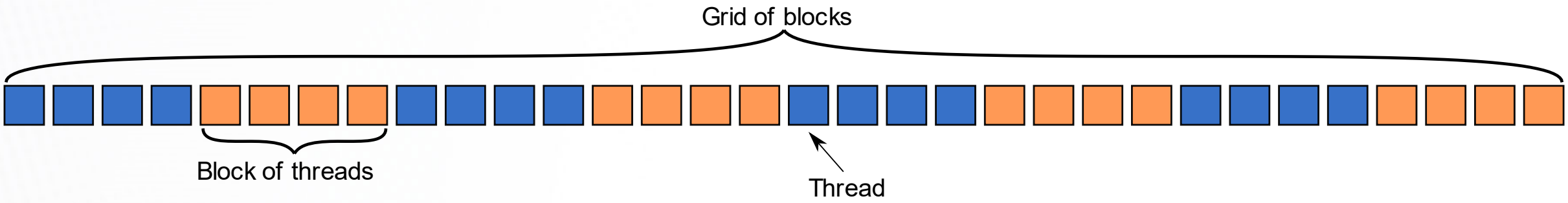
- In HIP, kernels are executed on a 3D ”grid”
  - You might feel comfortable thinking in terms of a mesh of points, but it’s not required
- The “grid” is what you can map your problem to
  - It’s not a physical thing, but it can be useful to think that way
- AMD devices (GPUs) support 1D, 2D, and 3D 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



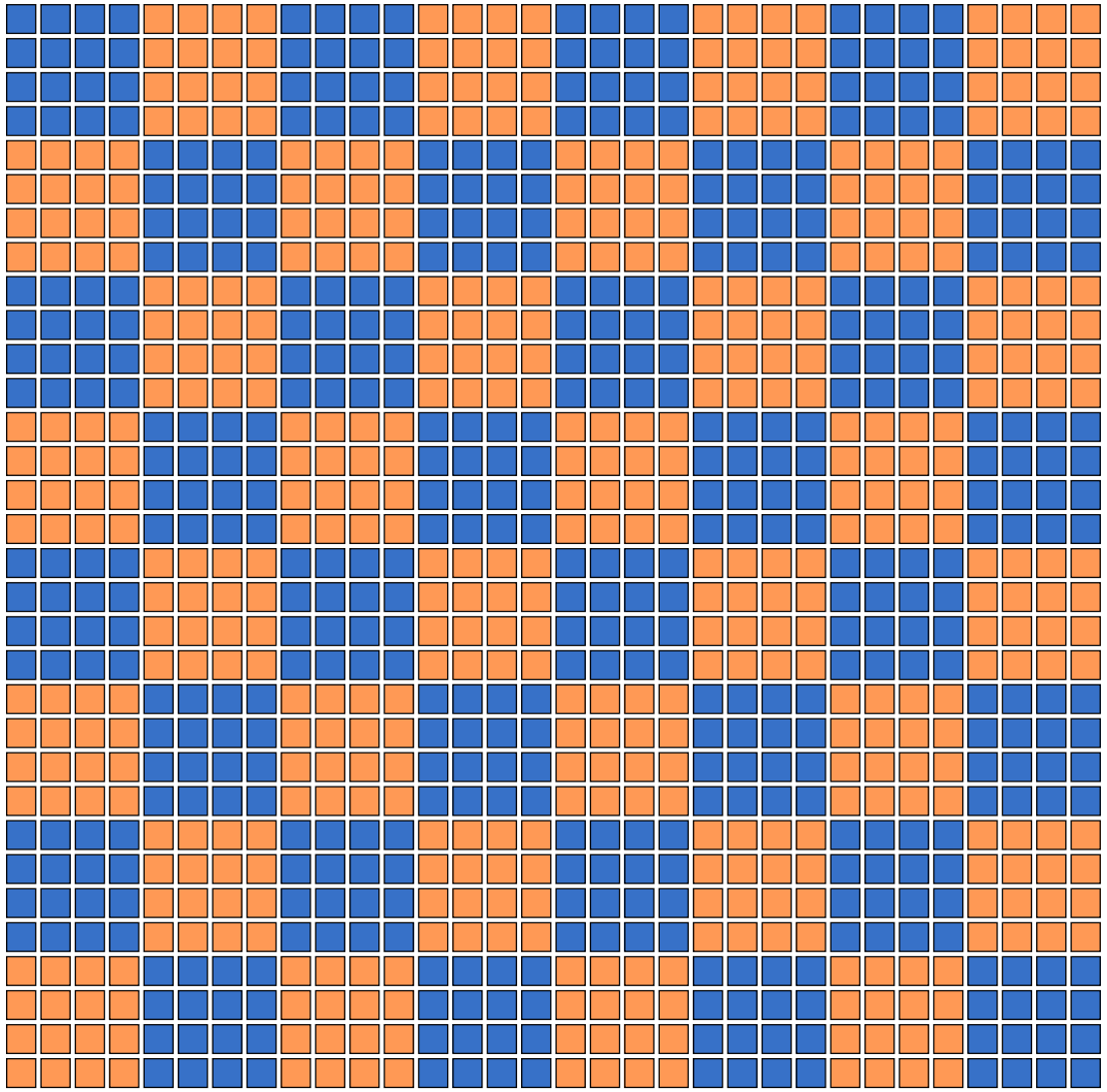
Threads in grid have access to:

- 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`

AMD

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`
- Etc.

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)

AMD

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);
```


AMD

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);          //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(); } }
```


AMD

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

  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
}
```

```
__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(); } }
```


AMD

AMD

Device management and asynchronous computing

AMD

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 swapping the currently selected device during runtime.
- MPI ranks can set different devices or over-subscribe (share) devices.

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.

Blocking vs Nonblocking API functions

- 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
  - If you know the kernel will take some time, this is a good area to do some work (i.e. MPI comms) on the host

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

- The non-blocking version is `hipMemcpyAsync`

```
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.

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

  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));   //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 waits for the kernel to finish here

AMD

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:

AMD

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);
```

NULL Stream			
Stream1		myKernel1	
Stream2		myKernel2	
Stream3		myKernel3	
Stream4		myKernel4	

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 memcpy **bandwidth increases significantly when host memory is pinned**.
  - It is good practice to allocate host memory that is frequently transferred to/from the device as pinned memory.

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);
```

AMD

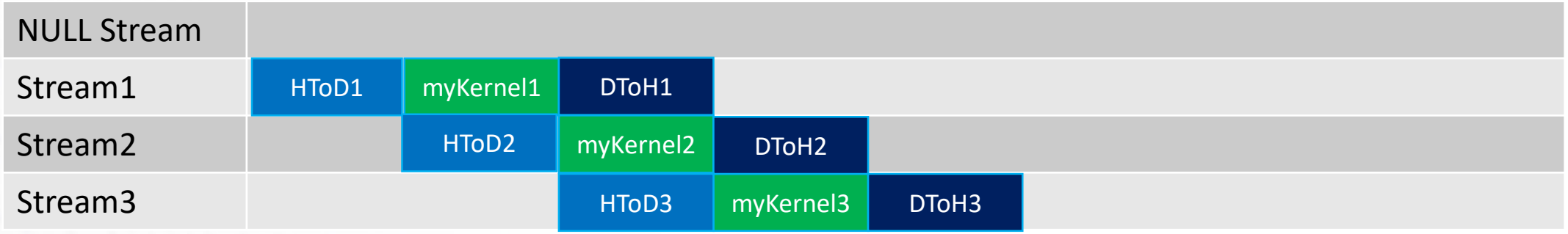
Streams

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);
```



AMD

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

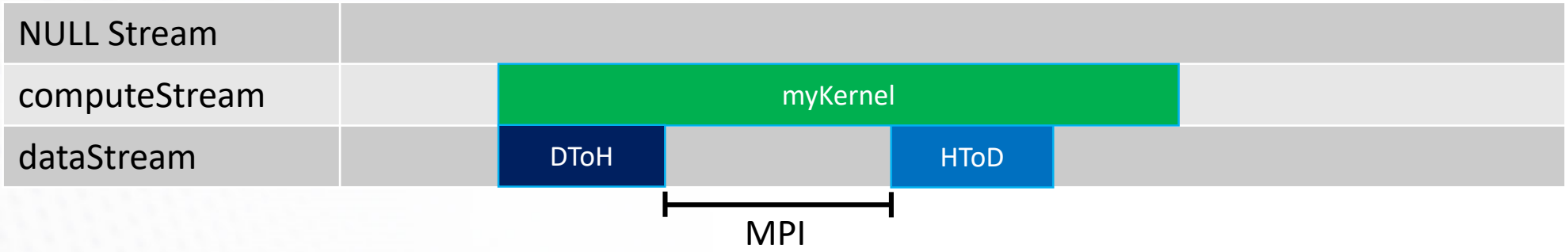
A common use-case for streams is MPI traffic:

```
//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);
```



AMD

Streams

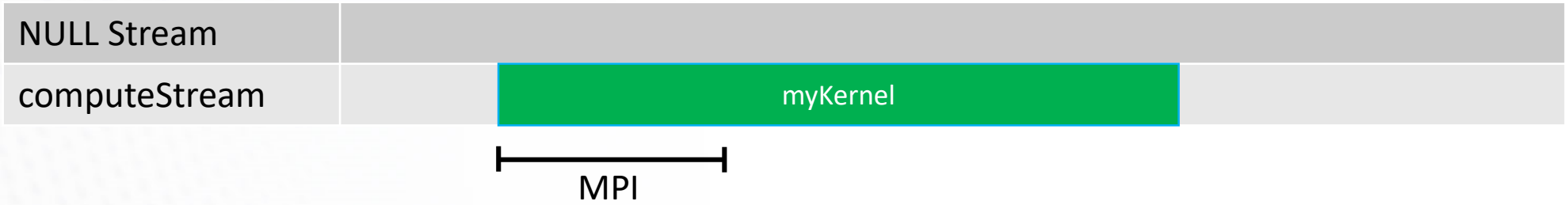
With a GPU-aware MPI stack, the Host<->Device traffic can be omitted:

```
//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);
```



AMD

Device code, shared memory, and thread synchronization

AMD

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;
//}
```


AMD

Memory declarations in Device Code

- Malloc/free not supported in device code.

- Variables/arrays can be declared on the stack.

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

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

- Stack variables declared as __shared__:
  - Allocated once per block in LDS memory
  - Shared and accessible by all threads in the same block
  - Access is faster than device global memory (but slower than register)
  - Must have size known at compile time

Shared Memory

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

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

  //all wavefronts must reach this point before any wavefront is allowed to continue.
                          //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
  …
}
```


AMD

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

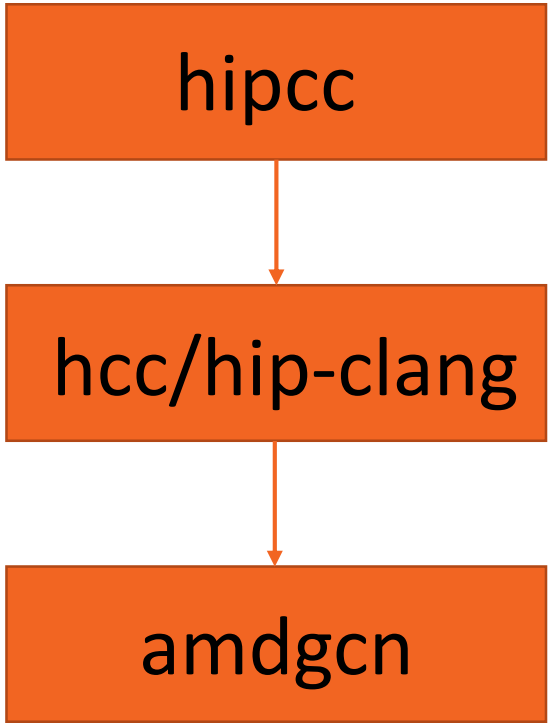
- Device Management:
  - `hipSetDevice()`, `hipGetDevice()`, `hipGetDeviceProperties()`
- Memory Management
  - `hipMalloc()`, `hipMemcpy()`, `hipMemcpyAsync()`, `hipFree()`
- Streams
  - `hipStreamCreate()`, `hipSynchronize()`, `hipStreamSynchronize()`, `hipStreamFree()`
- Events
  - `hipEventCreate()`, `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()`

AMD GPU Software

René van Oostrum <rene.vanoostrum@amd.com>

AMD GPU Compilers

- AMD supports several compilers that emit AMDGCN assembly
  - hcc
    - A fork of clang, will eventually be deprecated.
  - hip-clang
    - Based on upstream version of clang, currently in development
  - AOMP
- hipcc
  - A perl script that will invoke nvcc or hcc (or hip-clang).
  - Will automatically set the command line options needed by each compiler.
- hcc
  - Compiles HIP applications
  - Fork of clang
  - Generates and links AMDGCN from HIP device code
  - All the x86 host code is handled as usual by clang
- AOMP (AMD OpenMP Compiler)
  - Compiles C/C++ code with OpenMP “target” pragmas
  - Links with libomptarget to produce a binary that can offload work to the GPU



AMD

AMD GPU Compilers: the FORTRAN story

- FORTRAN is a technology important to the US Department of Energy
- AMD has plans to support OpenMP 4.5+ target offload from FORTRAN with two open source options
  - F18 (based on llvm)
  - gfortran
- FORTRAN compiler work is an ongoing effort
- See the Frontier spec sheet for what is expected to be supported on Frontier
  - https://www.olcf.ornl.gov/wp-content/uploads/2019/05/frontier_specsheet.pdf

hipcc usage

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

```
hipcc dotprod.cpp -o dotprod
```

Set HIPCC_VERBOSE=7 to see a bunch of useful information

- Compile and link lines
- Various paths

```
$ HIPCC_VERBOSE=7 hipcc dotprod.cpp -o dotprod
HIP_PATH=/opt/rocm
HIP_PLATFORM=hcc
HSA_PATH=/opt/rocm/hsa
HCC_HOME=/opt/rocm/hcc
hipcc-args: dotprod.cpp -o dotprod
hipcc-cmd: /opt/rocm/hcc/bin/hcc -hc  -D__HIPCC__ -isystem /opt/rocm/hcc/include
-isystem /opt/rocm/include/hip/hcc_detail/cuda -isystem /opt/rocm/hsa/include -Wno-deprecated-register
-isystem /opt/rocm/profiler/CXLActivityLogger/include -isystem /opt/rocm/include
-DHIP_VERSION_MAJOR=1 -DHIP_VERSION_MINOR=5 -DHIP_VERSION_PATCH=19284
-D__HIP_ARCH_GFX900__=1  -D__HIP_ARCH_GFX906__=1
dotprod.cpp -o dotprod
```


Inspecting the AMD GCN ISA

- You can inspect the AMD GCN ISA that was emitted by hcc (remember this is just clang)
- The command you need is `extractkernel`
- This is roughly equivalent to `objdump` (except it only dumps the GCN assembly, not x86)
- The GCN ISA is publicly available: https://developer.amd.com/wp-content/resources/Vega_Shader_ISA_28July2017.pdf

```
$ /opt/rocm/bin/extractkernel -i vectoradd
Generated GCN ISA for gfx900 at: vectoradd-000-gfx900.isa
$ grep v_add vectoradd.000-gfx900.isa
        v_add_u32_e32 v1, s1, v1                                // 000000001138: 68020201
        v_add3_u32 v0, s0, v0, v1                               // 000000001154: D1FF0000 04060000
        v_add_co_u32_e32 v2, vcc, s2, v0                        // 00000000119C: 32040002
        v_addc_co_u32_e32 v3, vcc, v3, v1, vcc                  // 0000000011A0: 38060303
        v_add_co_u32_e32 v4, vcc, s4, v0                        // 0000000011A8: 32080004
        v_addc_co_u32_e32 v5, vcc, v5, v1, vcc                  // 0000000011AC: 380A0305
        v_add_co_u32_e32 v0, vcc, s0, v0                        // 0000000011C0: 32000000
        v_addc_co_u32_e32 v1, vcc, v6, v1, vcc                  // 0000000011C4: 38020306
        v_add_f32_e32 v2, v2, v3                                // 0000000011CC: 02040702
```


AMD

Installing ROCm

- Requirements:
  - Linux®!
    - Ubuntu: ROCm can be installed using a Debian repo.
    - CentOS/RHEL: ROCm can be installed using a yum repo.
    - Other distros: you must build from source (support planned for SUSE based distributions).
  - To run on AMD hardware you need discrete GPUs in families GFX8 (Polaris) or GFX9 (Vega)
  - APUs aren't currently officially supported
- ROCm has been on a monthly release cycle
- ROCm is now compatible with AMD drivers in some upstream linux kernels.
- ROCm can be installed with:
  - ROCK kernel driver (from ROCm repos)
    - Only supported on Ubuntu, CentOS/RHEL
  - AMD drivers in some upstream kernels
    - Should work on more distributions
- Latest install instructions can be found here: https://github.com/RadeonOpenCompute/ROCm
- Also check out: https://github.com/RadeonOpenCompute/Experimental_ROC

AMD

Querying System

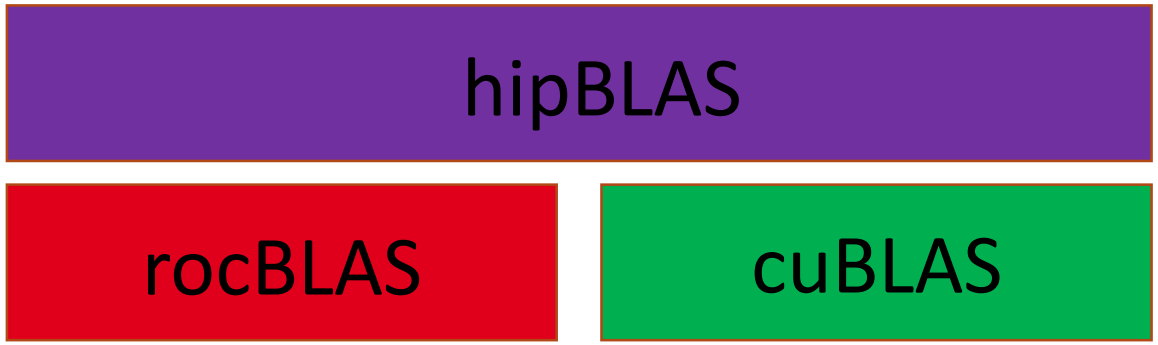
- rocminfo: Queries and displays information on the system's hardware
  - More info at: https://github.com/RadeonOpenCompute/rocminfo
- Querying ROCm version:
  - If you install ROCm in the standard location (/opt/rocm) version info is at: /opt/rocm/.info/version-dev
  - Can also run the command 'dkms status' and the ROCm version will be displayed
- 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

AMD GPU Libraries

- A note on naming conventions:
  - roc* -> AMGCN library usually written in HIP
  - cu* -> NVIDIA PTX libraries
  - hip* -> usually interface layer on top of roc*/cu* backends
- hip* libraries:
  - Can be compiled by hipcc and can generate a call for the device you have:
    - hipcc->hcc->AMD GCN ISA
    - hipcc->nvcc (inlined)->NVPTX
  - Just a thin wrapper that marshals calls off to a “backend” library:
    - corresponding roc* library backend containing optimized GCN
    - corresponding cu* library backend containing NVPTX for NVIDIA devices
  - E.g., hipBLAS is a marshalling library:



AMD

Decoder ring: Math library equivalents

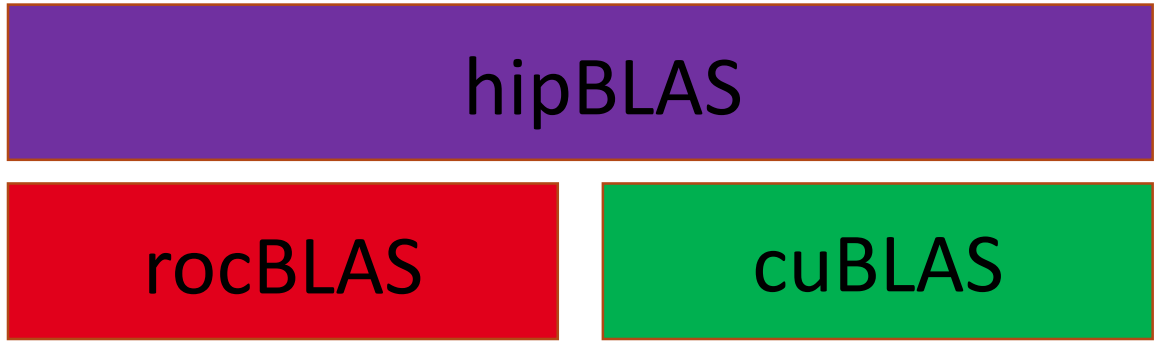
CUBLAS	**ROCBLAS**	Basic Linear Algebra Subroutines
CUFFT	**ROCFFT**	Fast Fourier Transforms
THRUST	**ROCTHRUST**	Deep Learning Library
CUB	**ROCPRIM**	Optimized Parallel Primitives
EIGEN	**EIGEN**	C++ Template Library for Linear Algebra

MORE INFO AT: GITHUB.COM/ROCM-DEVELOPER-TOOLS/HIP → HIP_PORTING_GUIDE.MD

AMD

AMD GPU Libraries: BLAS

- rocBLAS – `sudo apt install rocblas`
  - Source code: https://github.com/ROCmSoftwarePlatform/rocBLAS
  - Documentation: https://rocblas.readthedocs.io/en/latest/
  - Basic linear algebra functionality
    - axpy, gemv, trsm, etc
  - Use hipBLAS if you need portability between AMD and NVIDIA devices
- hipBLAS - `sudo apt install hipblas`
  - Documentation: https://github.com/ROCmSoftwarePlatform/hipBLAS/wiki/Exported-functions
  - Use this if you need portability between AMD and NVIDIA
  - It is just a thin wrapper:
    - It can dispatch calls to rocBLAS for AMD devices
    - It can dispatch calls to cuBLAS for NVIDIA devices



AMD

AMD GPU Libraries: rocBLAS example

- rocBLAS
  - Documentation: https://rocblas.readthedocs.io/en/latest/
  - Level 1, 2, and 3 functionality
    - axpy, gemv, trsm, etc
  - Note: rocBLAS syntax matches BLAS closer than hipBLAS or cuBLAS
    - Use hipBLAS only if you need portability between AMD and NVIDIA devices
  - Link with: `-lrocblas`

```
#include <rocblas.h>

int main(int argc, char ** argv) {
  rocblas_int N = 500000;

  // Allocate device memory
  double * dx, * dy;
  hipMalloc(&dx, sizeof(double) * N);
  hipMalloc(&dy, sizeof(double) * N);

  // Allocate host memory (and fill up the arrays) here
  std::vector<double> hx(N), hy(N);

  // Copy host arrays to device
  hipMemcpy(dx, hx.data(), sizeof(double) * N, hipMemcpyHostToDevice);
  hipMemcpy(dy, hy.data(), sizeof(double) * N, hipMemcpyHostToDevice);

  const double alpha = 1.0;
  rocblas_handle handle;
  rocblas_create_handle(&handle);
  rocblas_status status;
  status = rocblas_daxpy(handle, N, &alpha, dx, 1, dy, 1);
  rocblas_destroy_handle(handle);

  // Copy result back to host
  hipMemcpy(hy.data(), dy, sizeof(double) * N, hipMemcpyDeviceToHost);
  hipFree(dx);
  hipFree(dy);
  return 0;
}
```


AMD

Some Links to Key Libraries

- BLAS
  - rocBLAS (https://github.com/ROCmSoftwarePlatform/rocBLAS)
  - hipBLAS (https://github.com/ROCmSoftwarePlatform/hipBLAS)
- FFTs
  - rocFFT (https://github.com/ROCmSoftwarePlatform/rocFFT)
- Random number generation
  - rocRAND (https://github.com/ROCmSoftwarePlatform/rocRAND)
  - hipRAND (https://github.com/ROCmSoftwarePlatform/hipRAND)
- Sparse linear algebra
  - rocSPARSE (https://github.com/ROCmSoftwarePlatform/rocSPARSE)
  - hipSPARSE (https://github.com/ROCmSoftwarePlatform/hipSPARSE)
- Iterative solvers
  - rocALUTION (https://github.com/ROCmSoftwarePlatform/rocALUTION)
- Parallel primitives
  - rocPRIM (https://github.com/ROCmSoftwarePlatform/rocPRIM)
  - hipCUB (https://github.com/ROCmSoftwarePlatform/hipCUB)

More links to key libraries

Machine Learning libraries and Frameworks

- Tensorflow: https://github.com/ROCmSoftwarePlatform/tensorflow-upstream
- Pytorch: https://github.com/ROCmSoftwarePlatform/pytorch
- MIOpen (similar to cuDNN): https://github.com/ROCmSoftwarePlatform/MIOpen
- Tensile: https://github.com/ROCmSoftwarePlatform/Tensile
- RCCL (ROCm analogue of NCCL): https://github.com/ROCmSoftwarePlatform/rccl

Porting CUDA Applications to HIP

Paul Bauman <paul.bauman@amd.com>

Objectives

This training:

- Demonstrates how to convert CUDA codes into HIP
- Explains the meaning of the term ‘hipify’
- Provides a simple means to examine port quality
- Provides an idea of the common ‘gotchas’ of porting apps

Getting started with HIP

CUDA VECTOR ADD

```
__global__ void add(int n,
                    double *x,
                    double *y){
  int index = blockIdx.x * blockDim.x
              + threadIdx.x;
  int stride = blockDim.x * gridDim.x;

  for (int i = index; i < n; i += stride){
    y[i] = x[i] + y[i];
  }
}
```

HIP VECTOR ADD

```
__global__ void add(int n,
                    double *x,
                    double *y){
  int index = blockIdx.x * blockDim.x
              + threadIdx.x;
  int stride = blockDim.x * gridDim.x;

  for (int i = index; i < n; i += stride){
    y[i] = x[i] + y[i];
  }
}
```

KERNELS ARE SYNTACTICALLY THE SAME

AMD

CUDA APIs vs HIP API

CUDA	HIP
cudaMalloc(&d_x, N*sizeof(double));	hipMalloc(&d_x, N*sizeof(double));
cudaMemcpy(d_x, x, N*sizeof(double), cudaMemcpyHostToDevice);	hipMemcpy(d_x, x, N*sizeof(double), hipMemcpyHostToDevice);
cudaDeviceSynchronize();	hipDeviceSynchronize();

AMD

Launching a kernel

CUDA KERNEL LAUNCH SYNTAX

```
some_kernel<<<gridsize, blocksize,
              shared_mem_size, stream>>>
              (arg0, arg1, ...);
```

HIP KERNEL LAUNCH SYNTAX

```
hipLaunchKernelGGL(some_kernel,
                   gridsize, blocksize,
                   shared_mem_size, stream,
                   arg0, arg1, ...);
```


AMD

Difference between HIP and CUDA

Some things to be aware of writing HIP, or porting from CUDA:

- AMD GCN hardware ‘warp’ size = 64 (warps are referred to as ‘wavefronts’ in AMD documentation)
- Device and host pointers allocated by HIP API use flat addressing
  - Unified virtual addressing is enabled by default
  - Unified memory is available, but does not perform optimally currently
- Dynamic parallelism not currently supported
- CUDA 9+ thread independent scheduling not supported (e.g., no __syncwarp)
- Some CUDA library functions do not have AMD equivalents
- Shared memory and registers per thread can differ between AMD and Nvidia hardware
- Inline PTX or AMD GCN assembly is not portable

Despite differences, majority of CUDA code in applications can be simply translated.

Enter Hipify

- AMD provides ‘Hipify’ tools to automatically convert most CUDA code
  - Hipify-perl
  - Hipify-clang

- Good resource to help with porting: https://github.com/ROCm-Developer-Tools/HIP/blob/master/docs/markdown/hip_porting_guide.md

- In practice, large portions of many HPC codes have been automatically Hipified:
  - ~90% of CUDA code in CORAL-2 HACC
  - ~80% of CUDA code in CORAL-2 PENNANT
  - ~80% of CUDA code in CORAL-2 QMCPack
  - ~95% of CUDA code in CORAL-2 Laghos

  The remaining code requires programmer intervention

AMD

Hipify tools

- Hipify-perl:
  - Easy to use –point at a directory and it will attempt to hipify CUDA code
  - Very simple string replacement technique: may make incorrect translations
    - **sed -e 's/cuda/hip/g', (e.g., cudaMemcpy becomes hipMemcpy)**
  - Recommended for quick scans of projects

- Hipify-clang:
  - Requires clang compiler
  - More robust translation of the code. Uses clang to parse files and perform semantic translation
  - Can generate warnings and assistance for code for additional user analysis
  - High quality translation, particularly for cases where the user is familiar with the make system

Hipify-perl

- Sits in $HIP/bin/ (**export PATH=$PATH:[MYHIP]/bin**)
- Command line tool: **hipify-perl foo.cu > new_foo.cpp**
- Compile: **hipcc new_foo.cpp**
- How does this this work in practice?
  - Hipify source code
  - Check it in to your favorite version control
  - Try to build
  - Manually work on the rest

Hipify-clang

- Build from source
- hipify-clang has unit tests using LLVM lit/FileCheck (44 tests)

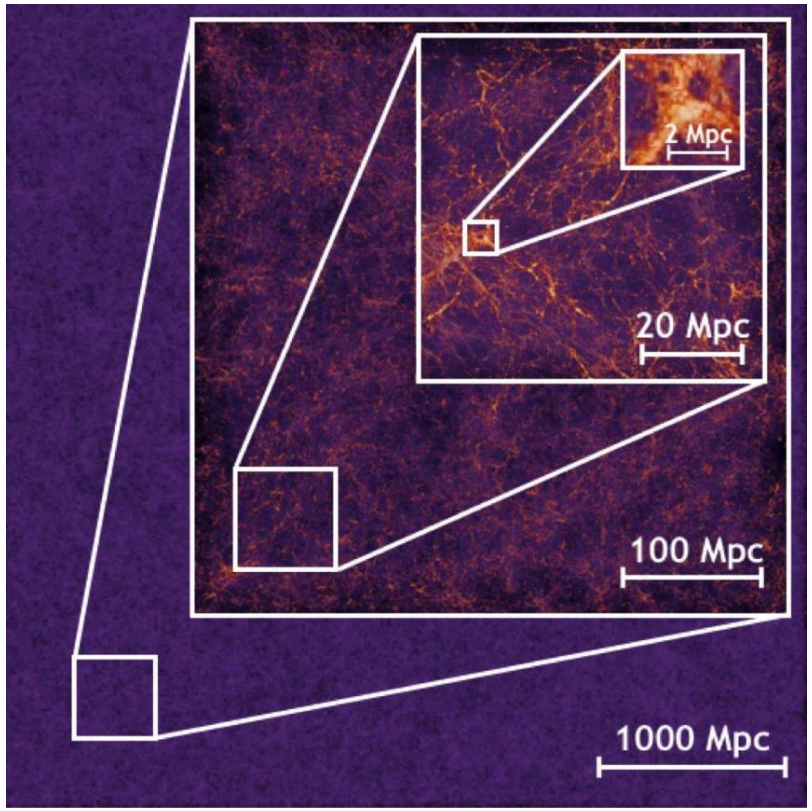
- Hipification requires same headers that would be needed to compile it with clang:
- ./hipify-clang foo.cu -I /usr/local/cuda-8.0/samples/common/inc

- https://github.com/ROCm-Developer-Tools/HIP/tree/master/hipify-clang

AMD

Example: HACC

- Hardware Accelerated Cosmology Code
- Simulates time-evolution of universe
  - Mpc = Megaparsec = 3.09 x 10^{22} meters
- **Our HIP success story:**
  - **Ported in an afternoon**
- Profiling:
  - 10% of time is spent in the tree walk
  - >80% in the short force kernels
    - (**GPU kernel**)
  - 5% in the 3d Transposes / FFTs



$$f_{SR} = (s+\epsilon)^{-3/2} - f_{grid}(s)$$

where,

$$s = \mathbf{r} \cdot \mathbf{r}$$

and,

$$f_{grid}(s) = POLY_5(s)$$

AMD

HACC: What made it a success

- **What was easy?**
  - Simple GPU kernel
  - Few library dependencies (FFTW, not in kernel)
  - No advanced CUDA features
- **What was difficult?**
  - Inline PTX: required translation to AMD GCN
  - Hand-written wave-32 code (for a reduction)

Porting HACC

CUDA

```
cudaMemcpyAsync(d_npos,h_npos,Nposbytes,
                cudaMemcpyHostToDevice,stream);
cudaMemcpyAsync(d_mask,h_mask,NmaskBytes,
                cudaMemcpyHostToDevice,stream);

calcHHCullenDehnen<<<blocksPerGrid,
               threadsPerBlock, 0, stream>>>
               (cnt, SIZE, d_npos, d_mask, rsm);

cudaMemcpyAsync(h_pos,
                d_npos+(SIZE-cnt),cntBytes,
                cudaMemcpyDeviceToHost,stream);
cudaMemcpyAsync(h_mask,d_mask,NmaskBytes,
                cudaMemcpyDeviceToHost,stream);
```

HIP

```
hipMemcpyAsync(d_npos,h_npos,Nposbytes,
               hipMemcpyHostToDevice,stream);
hipMemcpyAsync(d_mask,h_mask,NmaskBytes,
               hipMemcpyHostToDevice,stream);

hipLaunchKernelGGL((calcHHCullenDehnen),
      blocksPerGrid, threadsPerBlock, 0, stream,
      cnt, SIZE, d_npos, d_mask, rsm);

hipMemcpyAsync(h_pos,
               d_npos+(SIZE-cnt),cntBytes,
               hipMemcpyDeviceToHost,stream);
hipMemcpyAsync(h_mask,d_mask,NmaskBytes,
               hipMemcpyDeviceToHost,stream);
```


AMD

HACC: Comparison to the CUDA version

MEASUREMENT	TITAN-V W/ CUDA 9.0	MI-25 W/ HIP (ROCM 1.9)
REGISTER USE (VGPR)	32	36
REGISTER SPILLING	0	0
DIVERGENCE	92.5%	84%

AMD

Performance portability

MEASUREMENT	TITAN-V W/ CUDA 9.0	MI-25 W/ HIP (ROCM 1.9)
REGISTER USE (VGPR)	30, 74, 44, 58, 53	12, 73, 25, 38, 24
REGISTER SPILLING	0, 0, 0, 0, 0	0, 0, 0, 0, 0
DIVERGENCE	97, 95, 70, 99, 78	97, 90, 71, 98, 82

Pennant has 5 gpu kernels: Multiple entries denote each GPU kernel (GPUMain1, etc,)

Similar numbers observed in Quicksilver, HACC, etc

AMD

HIPified (ported) codes

QUICKSILVER

HACC

SW4LITE

HPL

PENNANT

LAGHOS

AND (ALREADY) MANY MORE…

AMD

Portability layers using HIP

Several portability layers are already supporting, or implementing, HIP

- RAJA
  - HIP kernel execution policies syntactically identical to CUDA
  - Official PRs under review
- Kokkos
  - HIP kernel execution policies syntactically identical to CUDA
  - Support is in Alpha and under development by Kokkos and AMD developers
- OCCA
  - OKL kernels can compile for HIP devices
  - Available in OCCA's master branch
- OpenMP 5.0
  - gcc and AMD's aomp compilers support target offload regions, interop with HIP

Tips and tricks for performance

What to look for when porting:

- Inline PTX assembly
- CUDA intrinsics
- Hard-coded dependencies on warp size, shared memory size
  - “grep 32”
  - Do not use hard coded dependencies on warp size!
- Code geared toward limited size of register file on NVIDIA hardware
- Functions implicitly inlined
- Unified Memory

QUESTIONS?

Dynamic Shared Memory

- Can actually use __shared__ arrays when sizes aren't known at compile time
  - Called dynamic shared memory
  - Declare one array using HIP_DYNAMIC_SHARED macro, use for all dynamic LDS space
  - Use the hipLaunchKernelGGL argument we haven't discussed yet

Dynamic Shared Memory

```
__global__ void reverse(double *d_a, int N) {
  HIP_DYNAMIC_SHARED(double, s_a); //dynamic array of doubles, shared in this block

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

  //all wavefronts should reach this point before any wavefront is allowed to continue.
  __syncthreads();

  d_a[tid] = s_a[N-1-tid]; //write out array in reverse order
}

int main() {
  …
  size_t NsharedBytes = N*sizeof(double);
  hipLaunchKernelGGL(reverse, dim3(1), dim3(N), NsharedBytes, 0, d_a, N); //Launch kernel
  …
}
```


AMD

Atomic Operations

Atomic functions:

- Perform a read+write of a single 32 or 64-bit word in device global or LDS memory
- Can be called by multiple threads in device code
- Performed in a conflict-free manner

- AMD GPUs support atomic operations on 32-bit integers in hardware
  - Float /double atomics implemented as atomicCAS (Compare And Swap) loops, may have poor performance

- Can check at compile time if 32 or 64-bit atomic instructions are supported on target device
  - `#ifdef __HIP_ARCH_HAS_GLOBAL_INT32_ATOMICS__`
  - `#ifdef __HIP_ARCH_HAS_GLOBAL_INT64_ATOMICS__`

Atomic Operations

Supported atomic operations in HIP:

Operation	Type, T	Notes
`T atomicAdd(T* address, T val)`	int, long long int, float, double	Adds val to *address
`T atomicExch(T* address, T val)`	int, long long int, float	Replace *address with val and return old value
`T atomicMin(T* address, T val)`	int, long long int	Replaces *address if val is smaller
`T atomicMax(T* address, T val)`	int, long long int	Replaces *address if val is larger
`T atomicAnd(T* address, T val)`	int, long long int	Bitwise AND between *address and val
`T atomicOr(T* address, T val)`	int, long long int	Bitwise OR between *address and val
`T atomicXor(T* address, T val)`	int, long long int	Bitwise XOR between *address and val

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
- 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/

CUDA features not supported by HIP

- CUDA 5.0 :
  - Dynamic Parallelism (not supported)
  - culpc functions (under development).

- CUDA 5.5 :
  - CUPTI (not directly supported, AMD GPUPerfAPI an alternative in some cases)

- CUDA 6.0
- Managed memory (under development)

CUDA features not in HIP, cont.

- CUDA 7.0
  - Per-thread-streams (under development)

- CUDA 8.0
  - Page Migration including cudaMemAdvise, cudaMemPrefetch, other cudaMem* APIs (not supported)

- **https://github.com/ROCm-Developer-Tools/HIP/blob/master/docs/markdown/hip_faq.md#what-specific-version-of-cuda-does-hip-support**

AMD

DISCLAIMER

DISCLAIMER
The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. All open source software listed in this presentation is governed by the associated open source license. Terms and limitations applicable to the purchase or use of AMD’s products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18

PCIe is a trademark (or registered trademark) of PCI-SIG Corporation.

OpenCL is a trademark of Apple Inc. used by permission by Khronos Group, Inc.

Linux is a trademark (or registered trademark) of Linus Torvalds.