

Introduction to High-Performance GPU Programming, Part-2

Dr. Hari Sadasivan,
AI Group, AMD

About Me

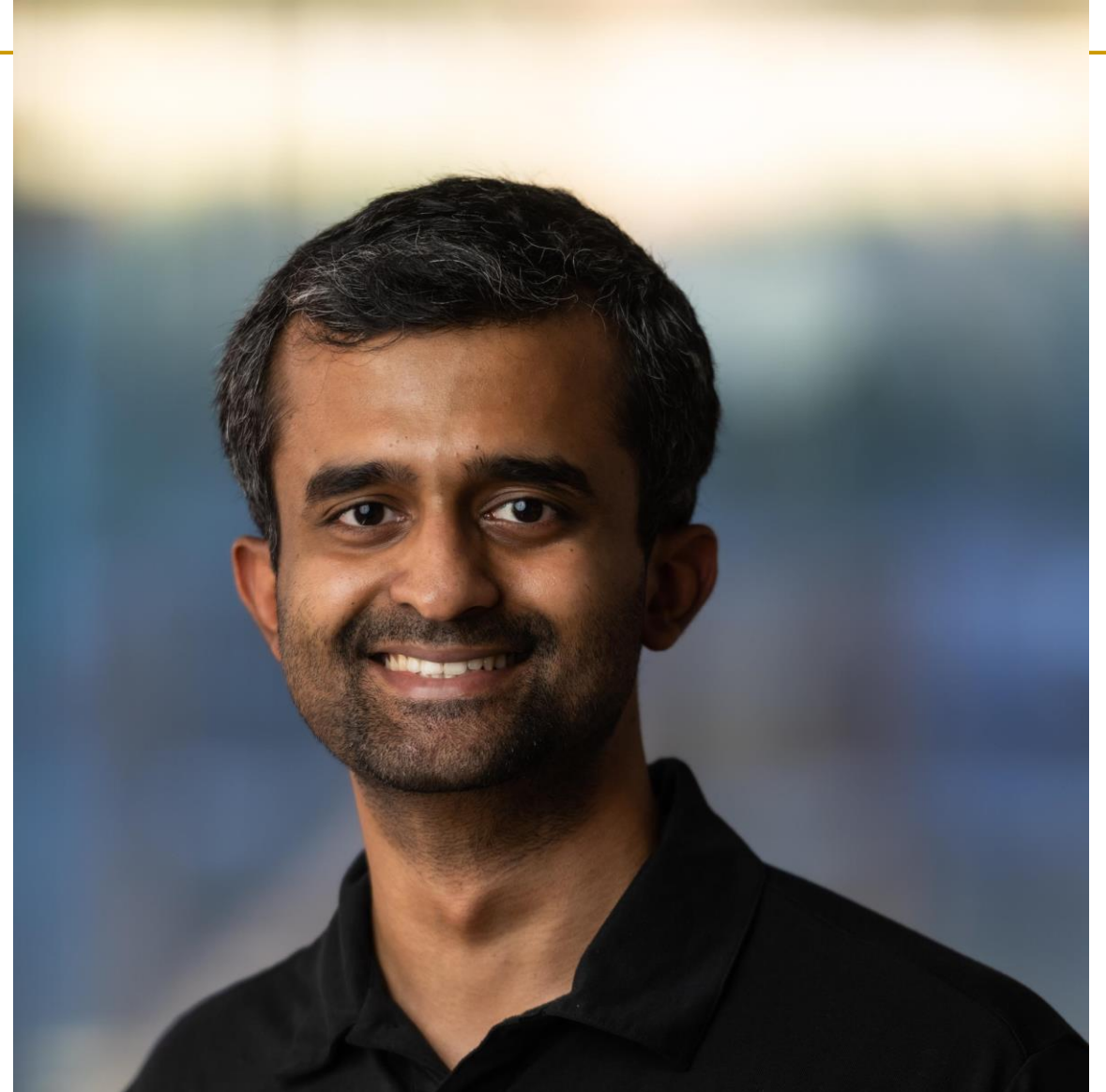
Dr. Hari Sadasivan

MTS SDE in AI Group, AMD |

Faculty (part-time), UW Seattle |

Sr Member, IEEE

- Co-founder of AMD –omics group
- Tech-lead, AMD CoE in AI at UW Seattle
 - [CSE/ECE PMP 590](#): Applied Parallel Programming on GPU



Agenda

GPUs

- Evolution
- CPU vs GPU
- SIMT Programming Model
- Programming GPUs
 - Put together a complete HIP code
- Architecture
- Profiling
- The world of GPU Optimizations for AI
 - AMD Hiring

Vector Addition

```
void vecAdd(float* A, float* B, float* C, int n) {
    int size = n * sizeof(float);
    float* A_d, B_d, C_d;

    // Allocate device memory
    hipMalloc((void **) &A_d, size);
    hipMalloc((void **) &B_d, size);
    hipMalloc((void **) &C_d, size);

    // Transfer A and B to device memory
    hipMemcpy(A_d, A, size, hipMemcpyHostToDevice);
    hipMemcpy(B_d, B, size, hipMemcpyHostToDevice);

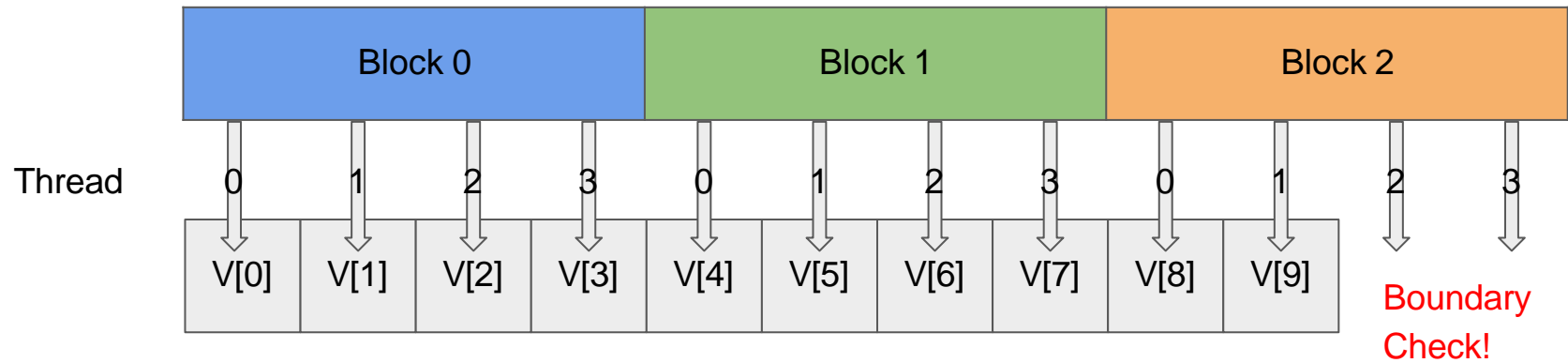
    // Kernel invocation code
    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
    hipDeviceSynchronize();

    // Transfer C from device to host
    hipMemcpy(C, C_d, size, hipMemcpyDeviceToHost);

    // Free device memory for A, B, C
    hipFree(A_d); hipFree(B_d); hipFree(C_d);
}
```

Mapping to the data

Suppose we use 1d thread blocks of size 4



$$i = \text{blockIdx}.x * \text{blockDim}.x + \text{threadIdx}.x$$

Questions

1. How many floating operations are being performed in the vector add kernel? Give your answer in terms of N and explain.
2. How many global memory reads and writes are being performed by the vector add kernel? Give your answer in terms of N .

Questions

1. How many floating operations are being performed in the vector add kernel? Give your answer in terms of N and explain.

N , one for each pair of input vector elements

2. How many global memory reads and writes are being performed by the vector add kernel? Give your answer in terms of N .

Reads: $2N$, one for each of the two input vectors elements.

Writes: N , one for each output vector element.

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.

Streams

- A stream in HIP is a queue of tasks (e.g. kernels, memcpyys, events).
 - Tasks enqueued in a stream **complete in order on that stream**.
 - Tasks being executed in different streams are allowed to overlap and share device resources.
- Streams are created via:
`hipStream_t stream;`
`hipStreamCreate(&stream);`
- And destroyed via:
`hipStreamDestroy(stream);`
- Passing `0` or `NULL` as the `hipStream_t` argument to a function instructs the function to execute on a stream called the 'NULL Stream':
 - No task on the NULL stream will begin until **all previously enqueued tasks in all other streams have completed**.
 - Blocking calls like `hipMemcpy` run on the NULL stream.

Streams

- Suppose we have 4 small kernels to execute:

```
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);
```

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);
```



Streams

Changing to asynchronous memcpyys 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);
```

NULL Stream					
Stream1	HToD1	myKernel1	DToH1		
Stream2		HToD2	myKernel2	DToH2	
Stream3			HToD3	myKernel3	DToH3

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.

Questions

1. If we want to copy 5000 bytes of data from host array `h_A` (`h_A` is a pointer to element 0 of the source array) to device array `d_A` (`d_A` is a pointer to element 0 of the destination array), what would be an appropriate API call for the data copy in HIP?

Questions

1. If we want to copy 5000 bytes of data from host array h_A (h_A is a pointer to element 0 of the source array) to device array d_A (d_A is a pointer to element 0 of the destination array), what would be an appropriate API call for the data copy in HIP?

Ans: `hipMemcpy(d_A, h_A, 5000, hipMemcpyHostToDevice);`

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->hip-clang->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:

hipBLAS

The diagram consists of three colored rectangular boxes. At the top is a purple box containing the text 'hipBLAS'. Below it are two smaller boxes: a red box on the left containing 'rocBLAS' and a green box on the right containing 'cuBLAS'. This visualizes hipBLAS as a higher-level library that interfaces with the lower-level rocBLAS and cuBLAS backends.

rocBLAS

cuBLAS

Why libraries?

- Code reuse
- High Performance
 - Maximize compute
 - Maximize memory bandwidth
- No need to deal with low level GPU code

Math library equivalents

CUBLAS

ROCBLAS

Basic Linear Algebra Subroutines

CUFFT

ROCFFT

Fast Fourier Transforms

THRUST

ROCTHRUST

STL Library for parallel algos

CUB

ROCPRIM

Optimized Parallel Primitives

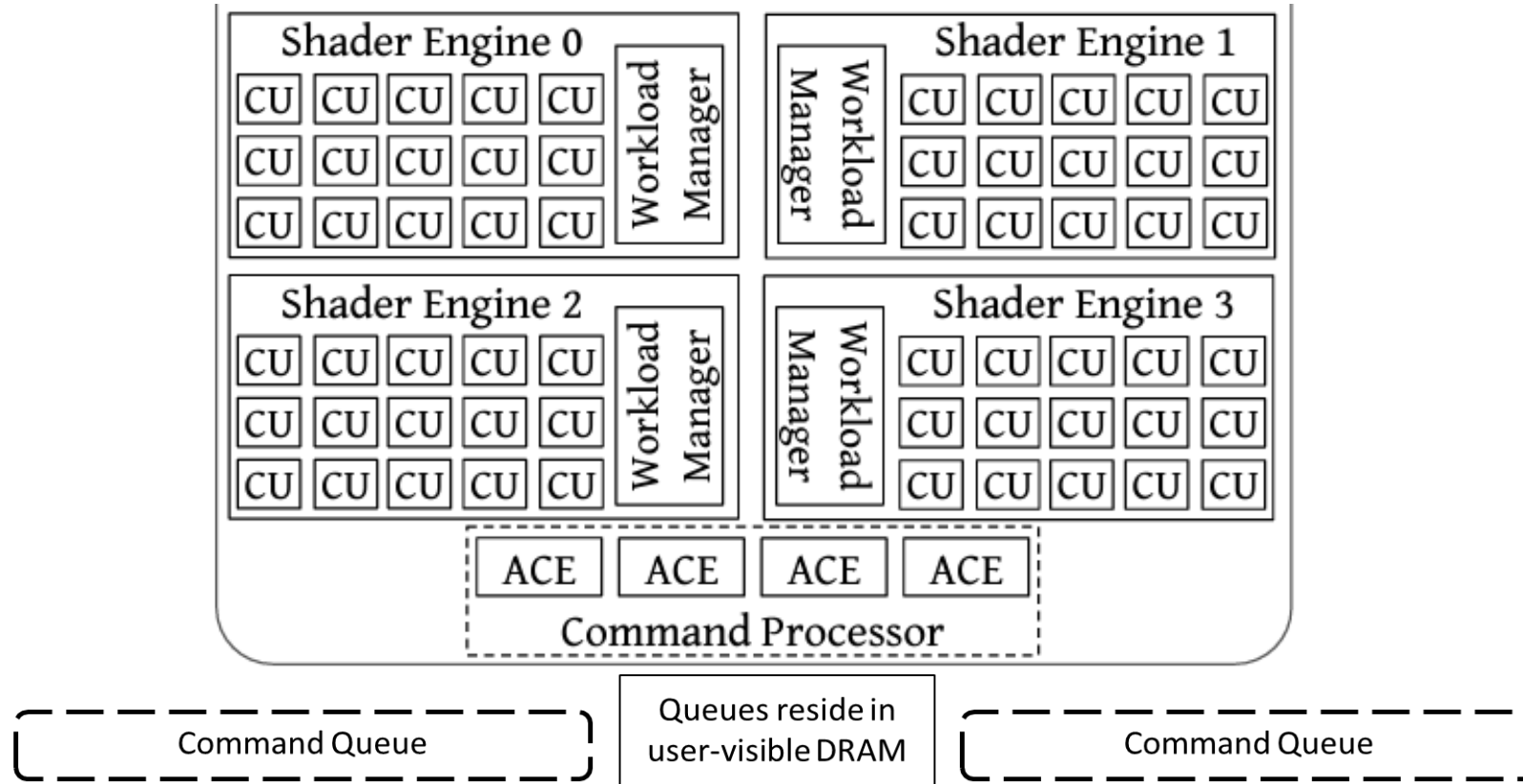
EIGEN

EIGEN

C++ Template Library for Linear Algebra

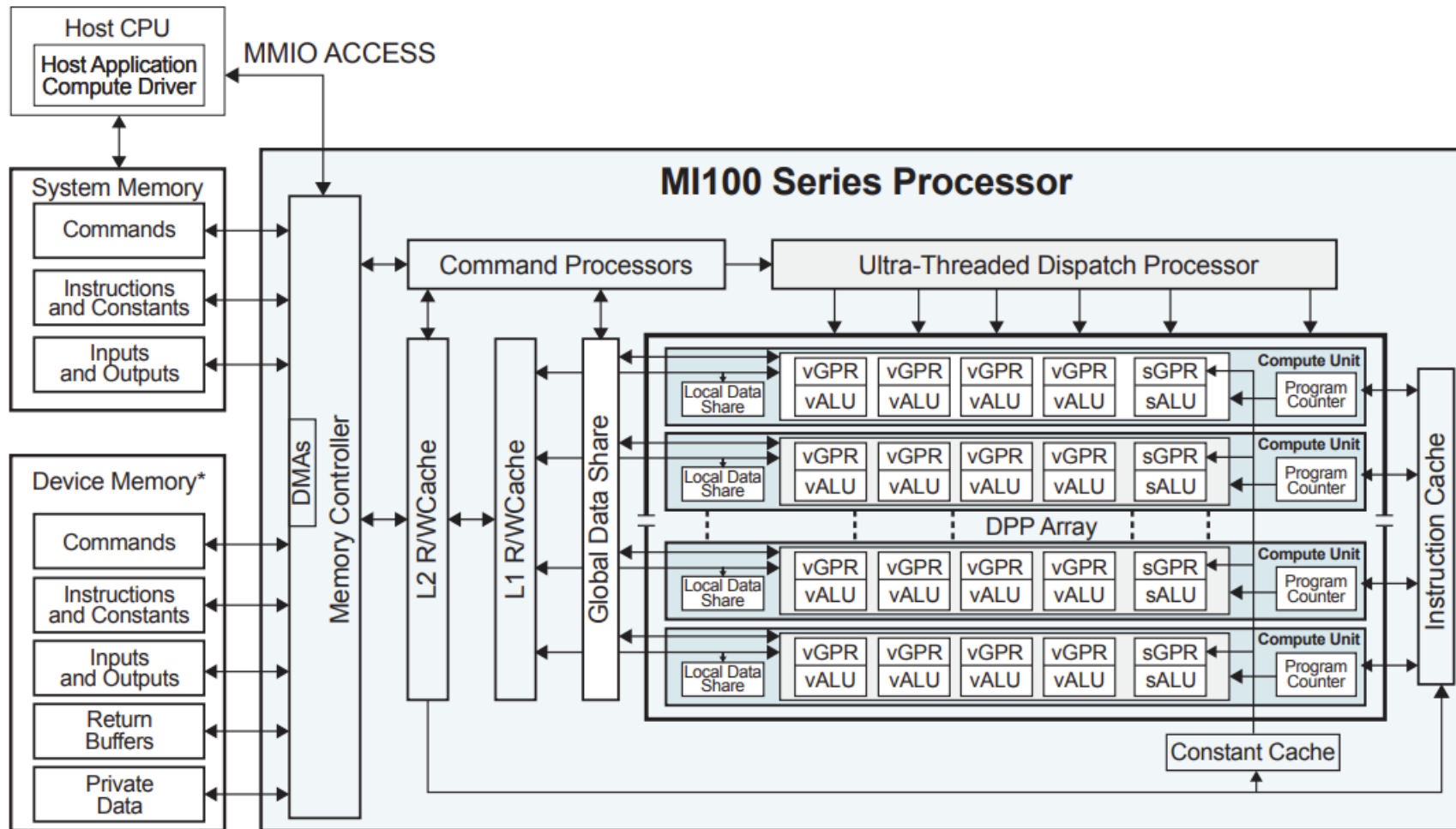
MORE INFO AT: [GITHUB.COM/ROCM-DEVELOPER-TOOLS/HIP](https://github.com/ROCm-developer-tools/hip) □ [HIP_PORTING_GUIDE.MD](#)

AMD GCN HW Layout



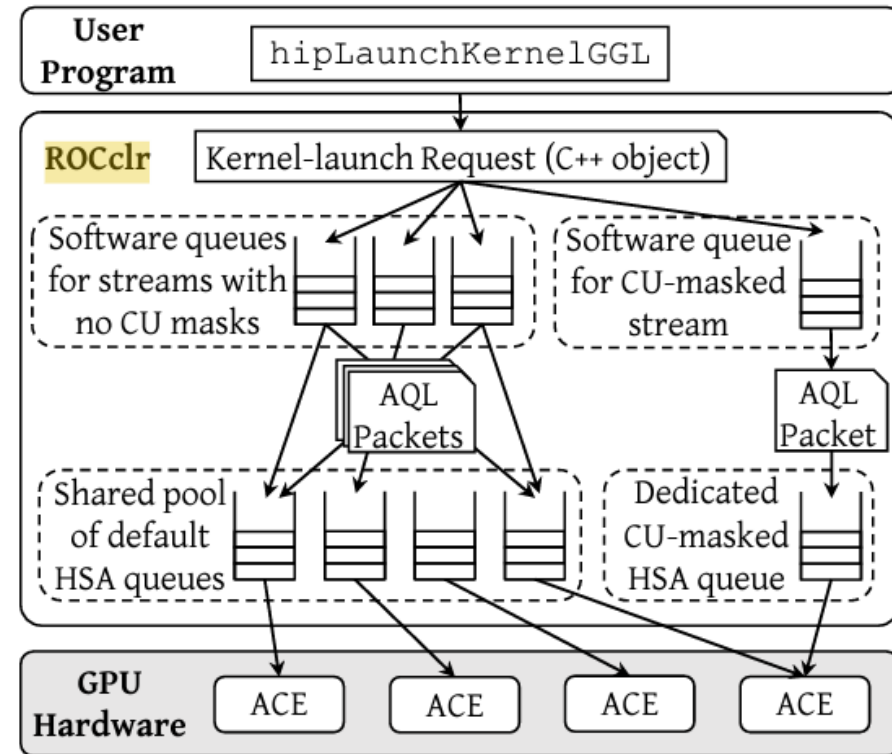
Nathan Otterness and James H. Anderson. 2021. Exploring AMD GPU Scheduling Details by Experimenting With “Worst Practices”.

MI100 micro-architecture



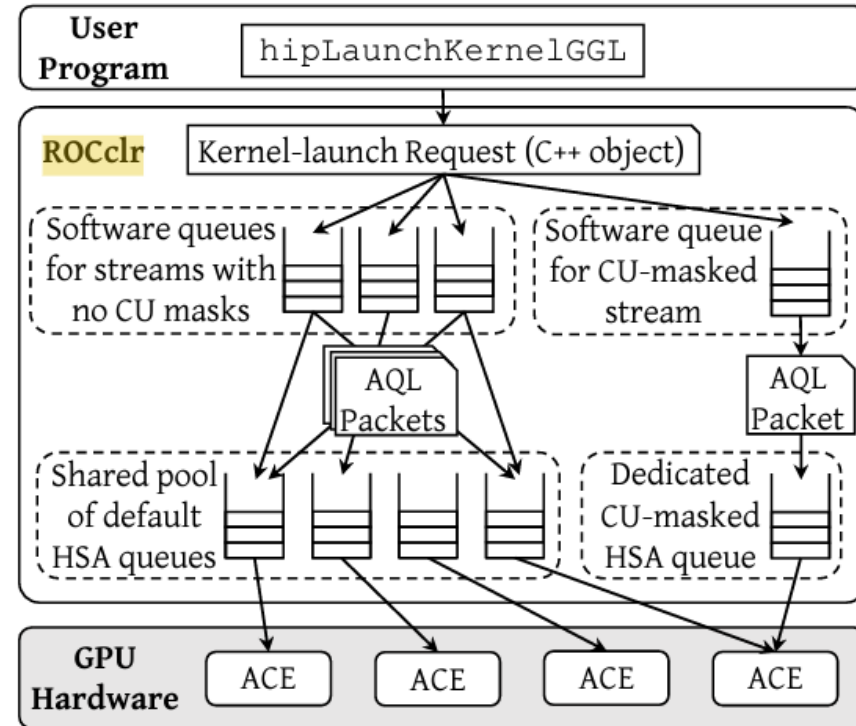
ROCm queue management

1. A user program calls the hipLaunchKernelGGL API function to launch a kernel.



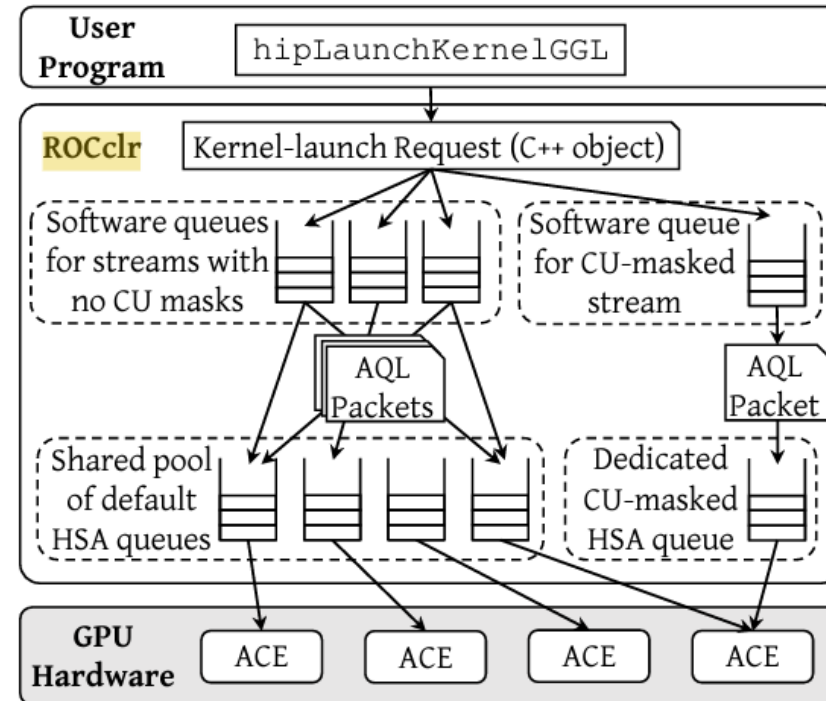
ROCm queue management

1. A user program calls the `hipLaunchKernelGGL` API function to launch a kernel.
2. The HIP runtime inserts a kernel-launch command into a software queue managed by the ROCclr runtime library.



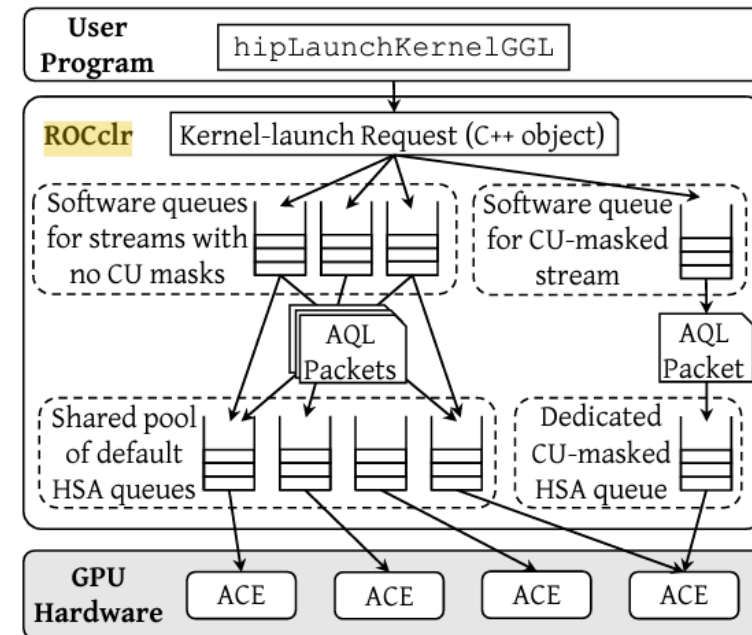
ROCm queue management

1. A user program calls the `hipLaunchKernelGGL` API function to launch a kernel.
2. The HIP runtime inserts a kernel-launch command into a software queue managed by the ROCclr runtime library.
3. ROCclr converts the kernel-launch command into an AQL (architected queuing language) packet.



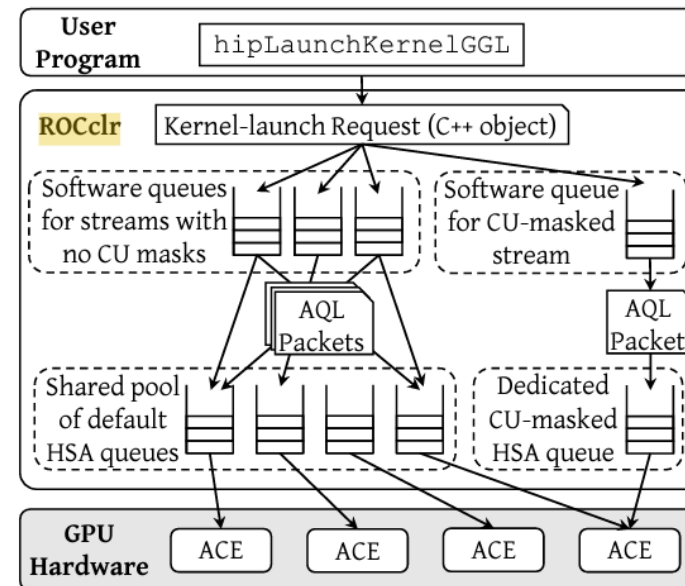
ROCm queue management

1. A user program calls the `hipLaunchKernelGGL` API function to launch a kernel.
2. The HIP runtime inserts a kernel-launch command into a software queue managed by the ROCclr runtime library.
3. ROCclr converts the kernel-launch command into an AQL (architected queuing language) packet.
4. ROCclr inserts the AQL packet into an HSA (heterogeneous system architecture) queue.



ROCm queue management

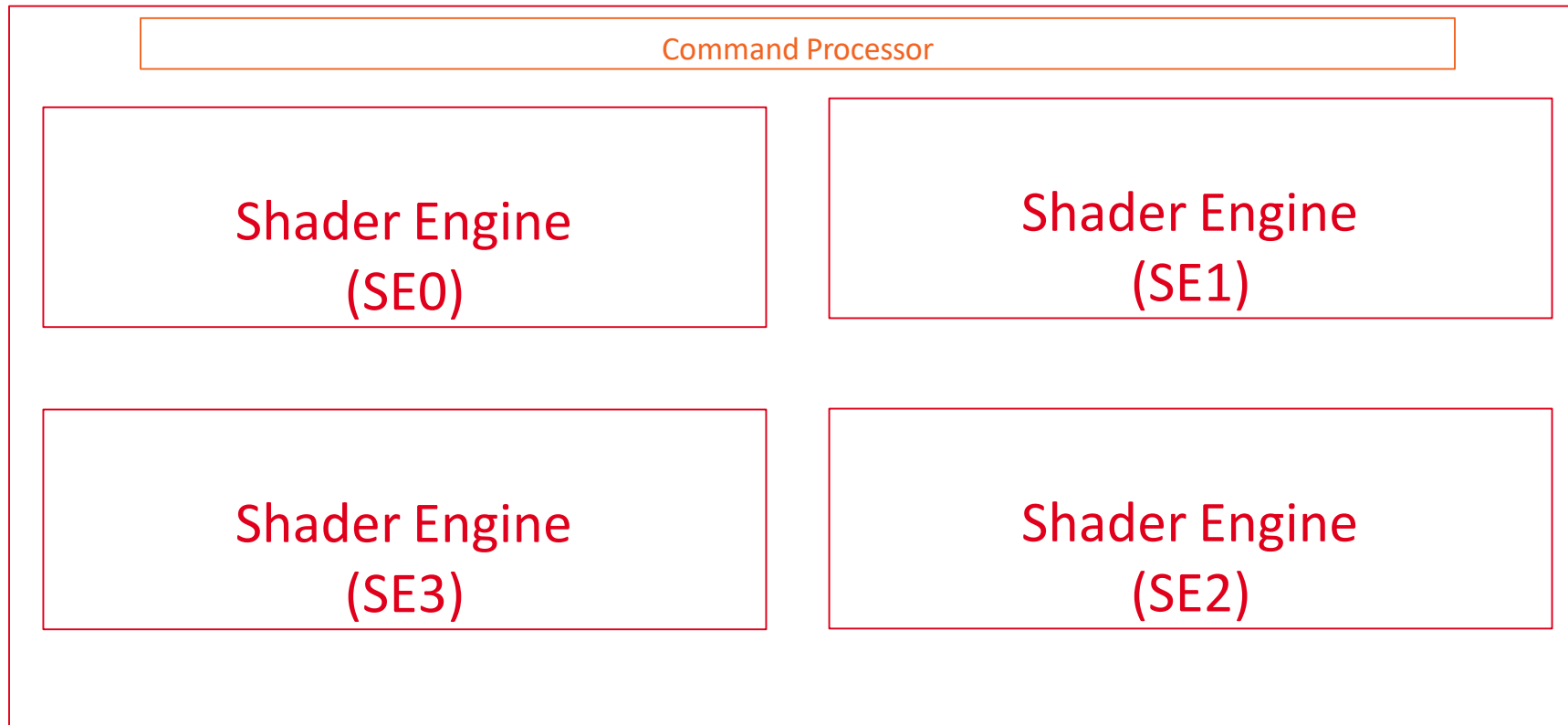
1. A user program calls the `hipLaunchKernelGGL` API function to launch a kernel.
2. The HIP runtime inserts a kernel-launch command into a software queue managed by the ROCclr runtime library.
3. ROCclr converts the kernel-launch command into an AQL (architected queuing language) packet.
4. ROCclr inserts the AQL packet into an HSA (heterogeneous system architecture) queue.
5. In hardware, an asynchronous compute engine (ACE) processes HSA queues, assigning kernels to compute hardware.



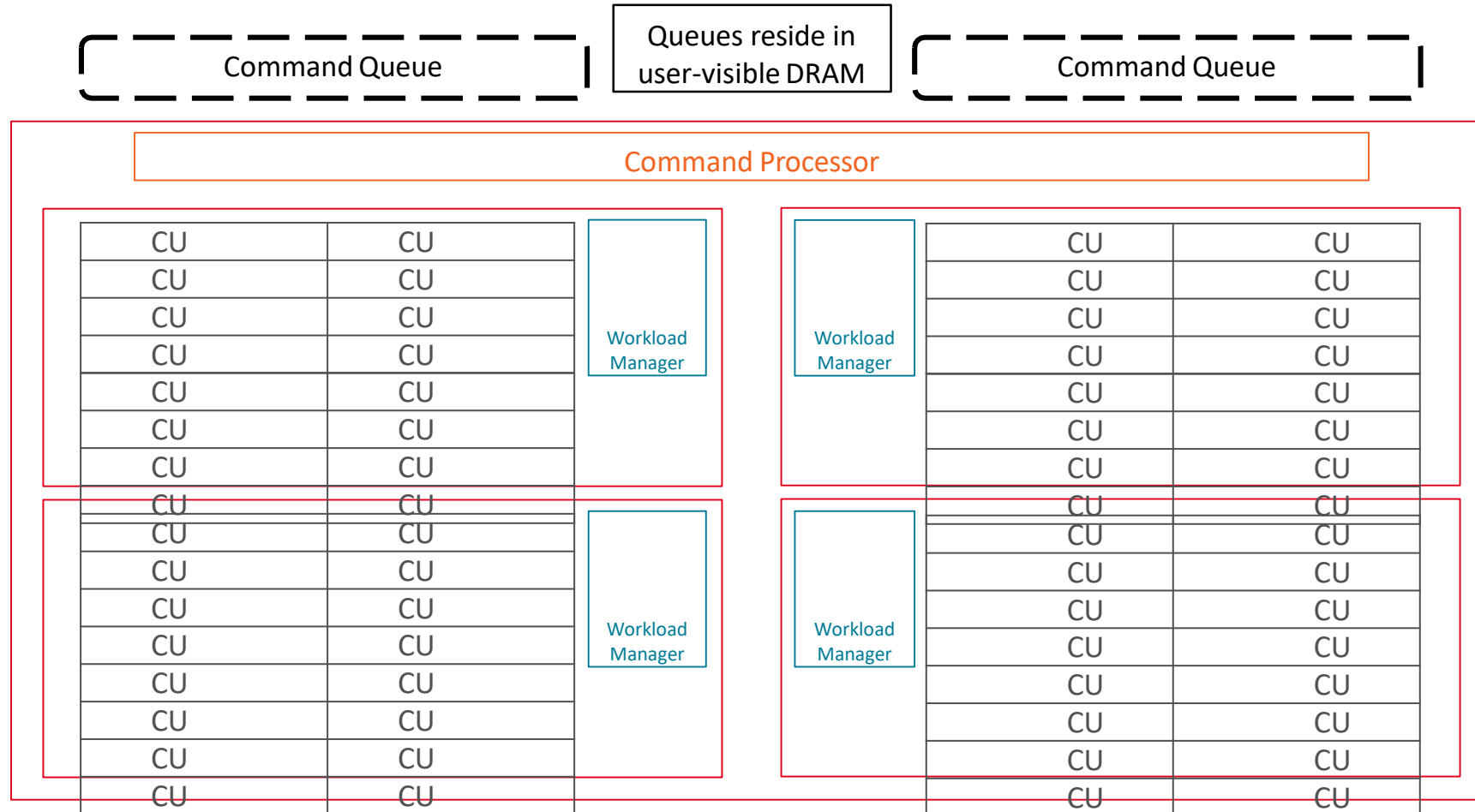
HSA Queue Management

- HSA API creates & manages the memory-mapped queues & commands that interface with driver & software.
- HSA queues (4 in total) are ring buffers of AQL packets & shared directly between GPU & userspace memory
- On creation of a new HSA queue, amdgpu driver sends GPU an updated runlist
 - list of HSA queues & memory locations
- For in-stream ordering, ROCm uses both HW (barrier AQL packets) & SW (ROCclr's SW queues) mechanisms.

AMD GCN GPU Hardware Layout



AMD GCN GPU Hardware Layout

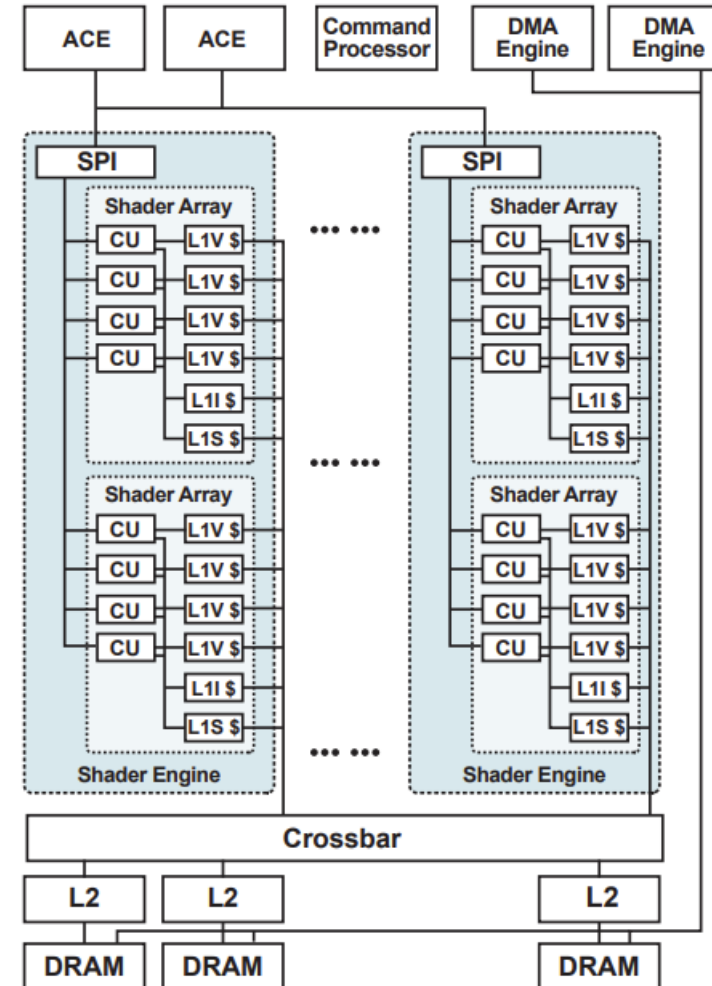


Hardware Configuration Parameters on Modern AMD GPUs

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

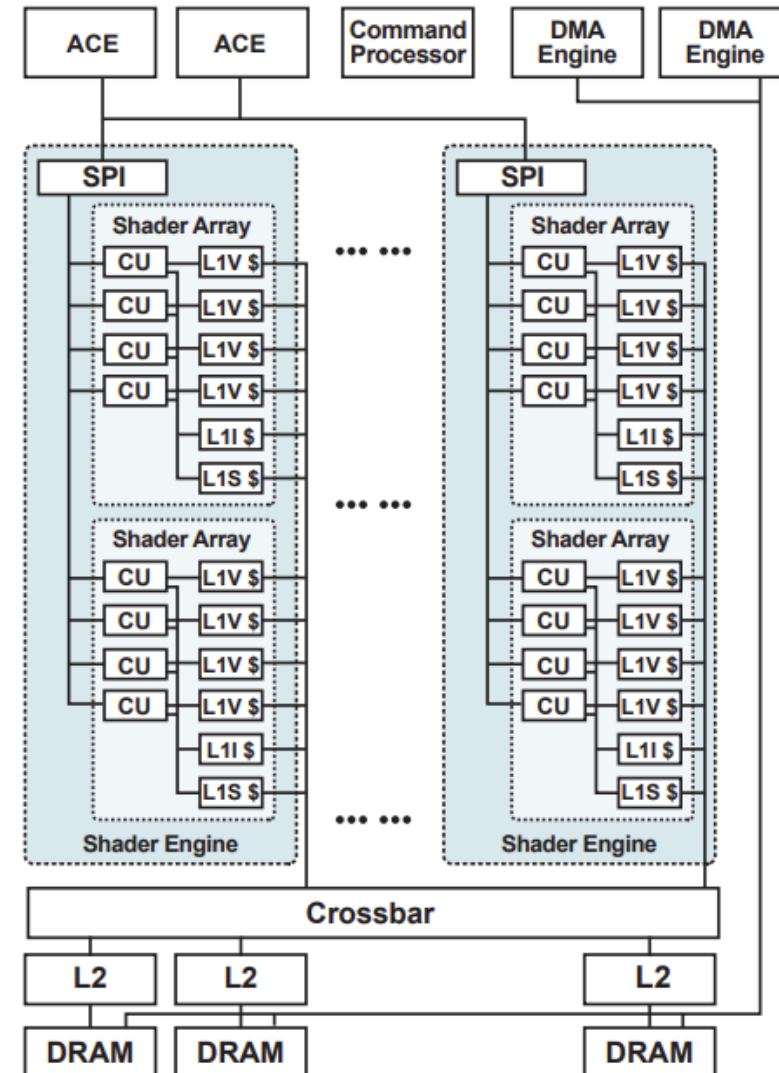
GPU Organization

1. Controlling
 - Command Processor (CP)
 - Asynchronous Compute Engine (ACE)
 - Direct Memory Access (DMA)
2. User-programmable shader
 - Shader Engine
 - Shader Arrays
 - Compute Units (CU)
3. Memory
 - L2 cache, Memory controller



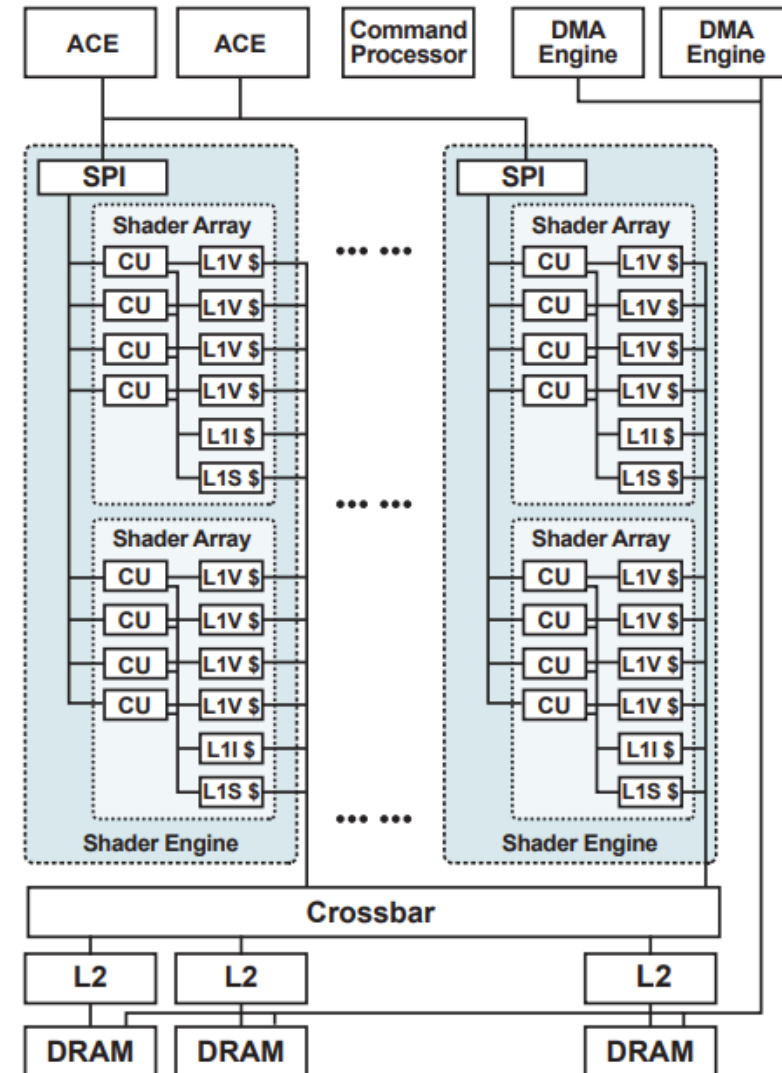
Controlling

- Command Processor (CP) receives mem-copy/kernel launch commands from CPU.
- Kernel launching command is forwarded to Asynchronous Compute Engine (ACE).
- DMA engines oversees mem-copy



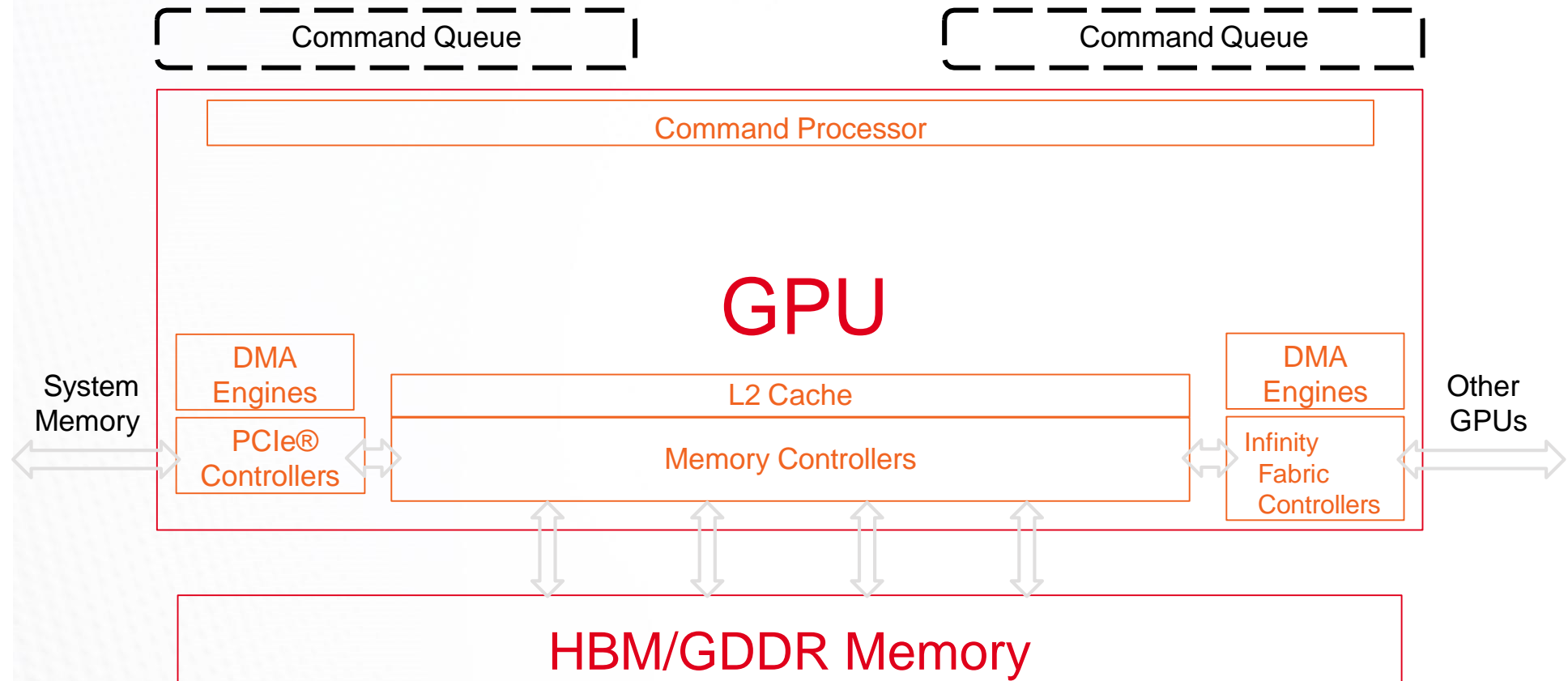
Workgroup dispatching for kernels

- Asynchronous Compute Engines (ACEs) handle kernel launch commands.
- ACEs break down kernels into workgroups & distribute to Shader Pipe Input (SPI) blocks.
- SPI breaks down workgroups into wavefronts, dispatches wavefronts to CUs & initializes registers.
- SPI guarantees all wavefronts in a workgroup are dispatched to same CU.



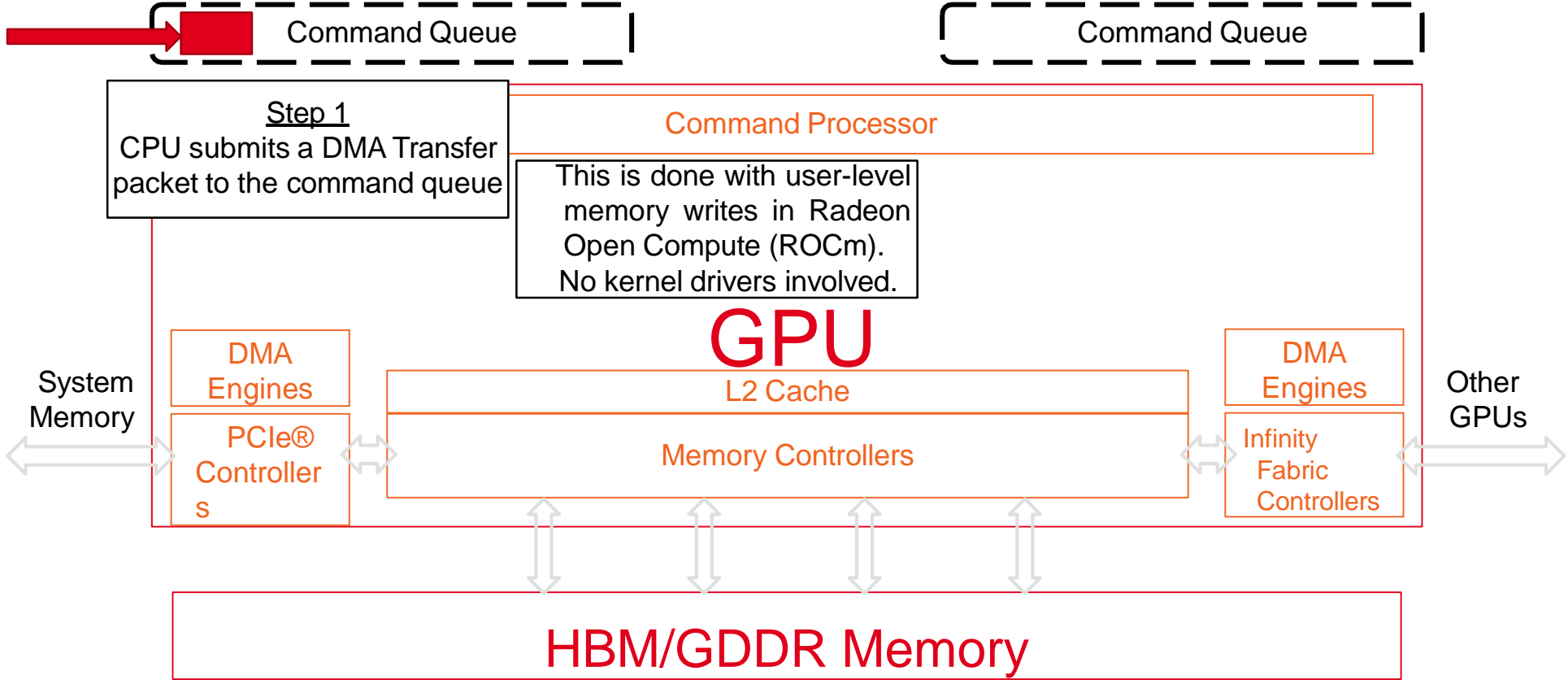
Scheduling memcopy to a GPU

GPU Memory, I/O, and Connectivity



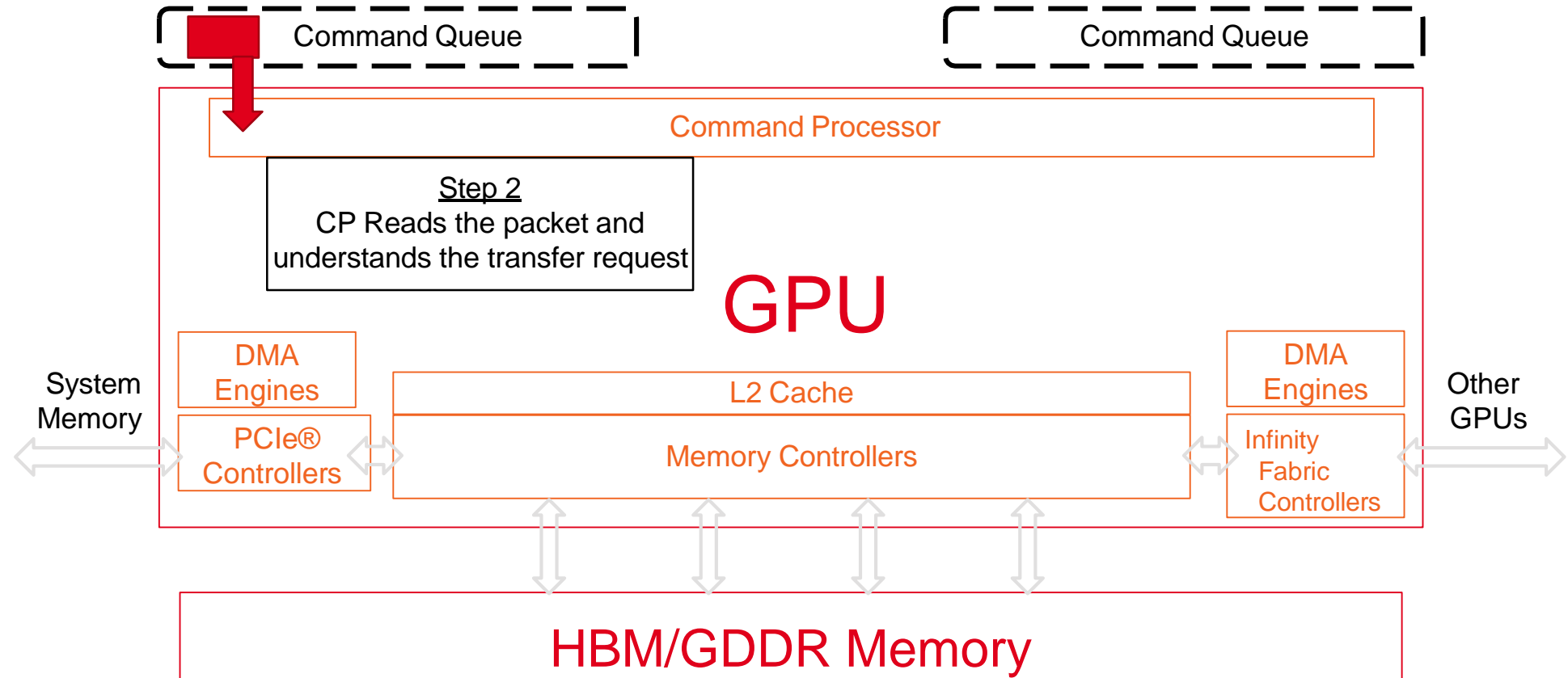
Scheduling memcopy to a GPU

DMA Engines Accept Work from the Same Queues



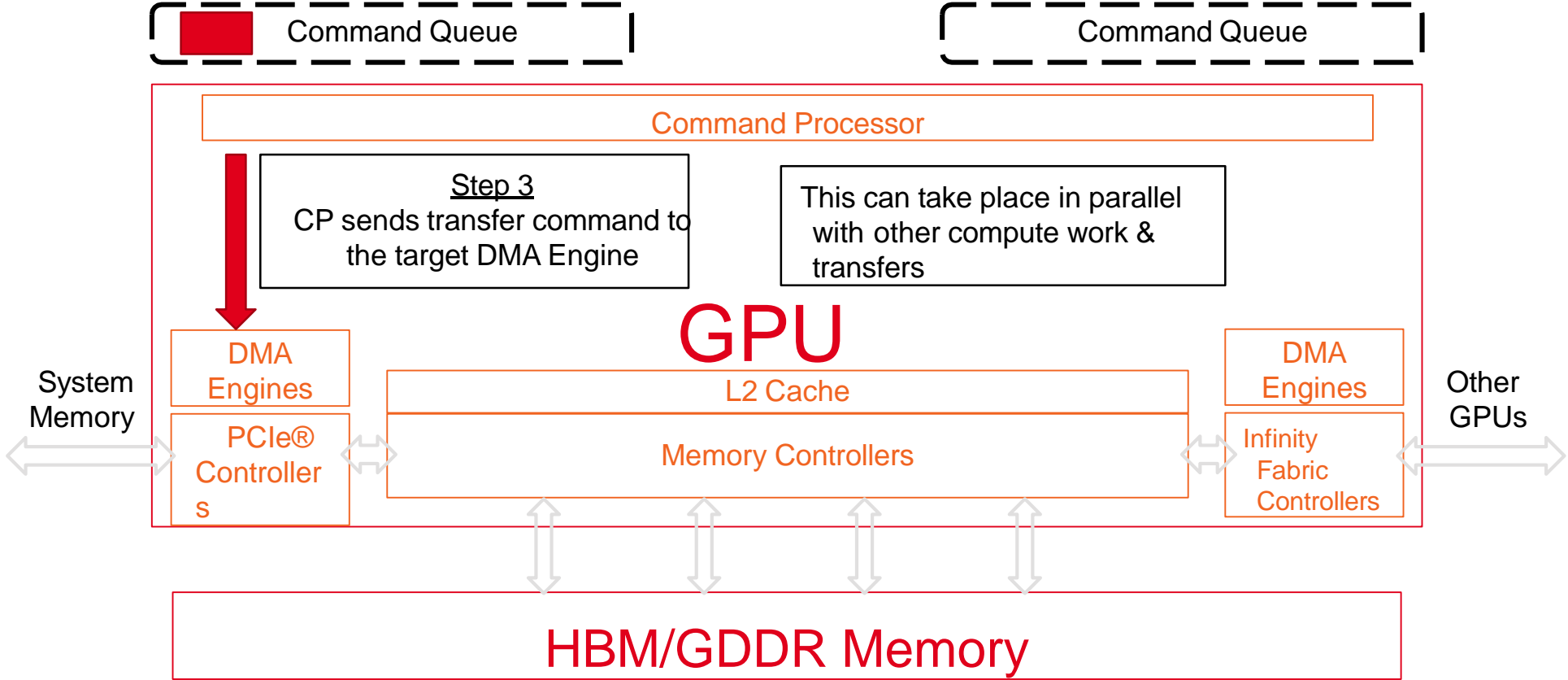
Scheduling memcopy to a GPU

DMA Engines Accept Work from the Same Queues



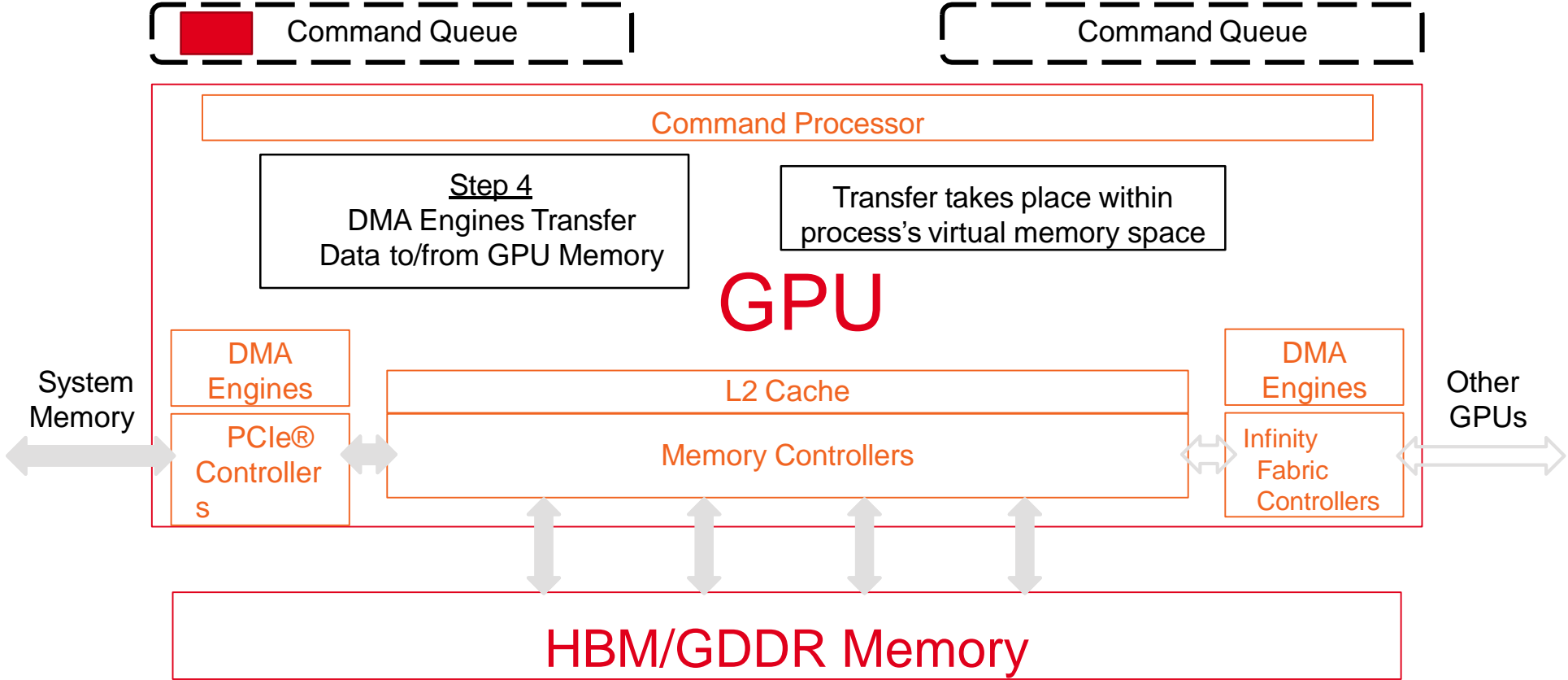
Scheduling memcopy to a GPU

DMA Engines Accept Work from the Same Queues

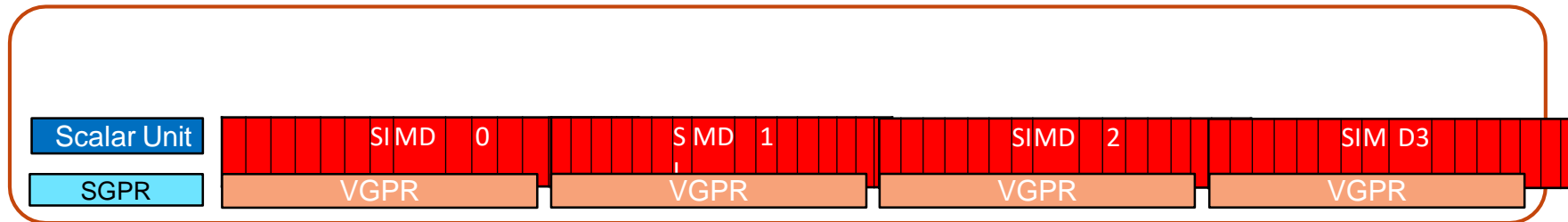


Scheduling memcopy to a GPU

DMA Engines Accept Work from the Same Queues

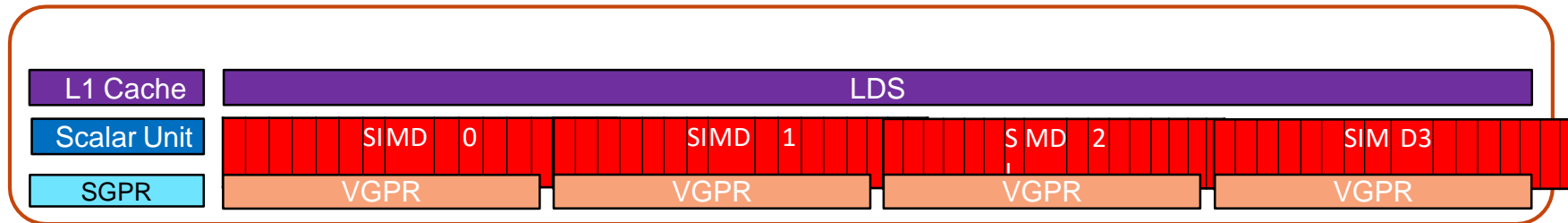


The GCN Compute Unit (CU)



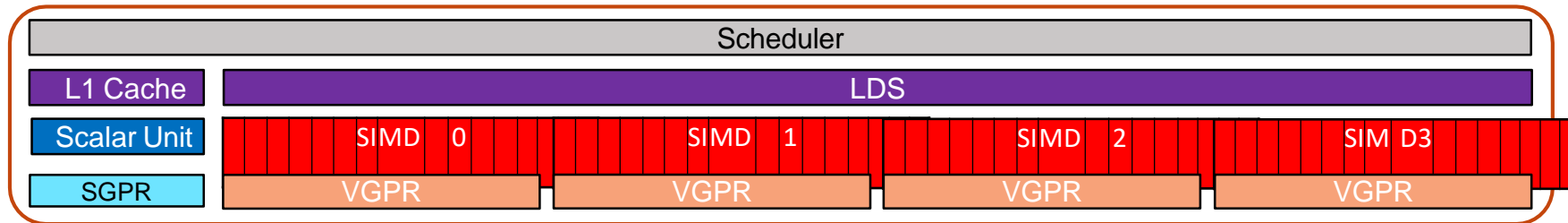
- SIMD Units
 - 4x SIMD vector units (each 16 lanes wide)
 - 4x 64KiB (256KiB total) Vector General-Purpose Register (VGPR) file
 - A maximum of 256 total registers per SIMD lane – each register is 64x 4-byte entries
 - Instruction buffer for 10 wavefronts on each SIMD unit
 - Each wavefront is local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

The GCN Compute Unit (CU)



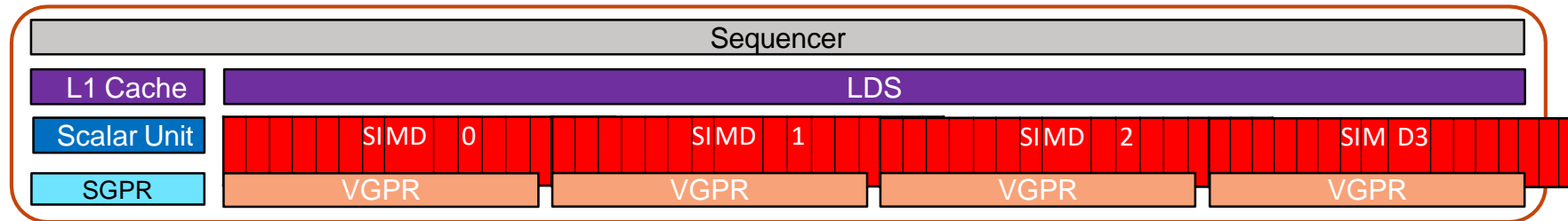
- 64KiB Local Data Share (LDS, or shared memory)
 - 32 banks with conflict resolution
 - Can share data between all threads in a block
- 16 KiB Read/Write L1 vector data cache
 - Write-through; L2 cache is the coherence point – shared by all CUs

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)

The GCN Compute Unit (CU)

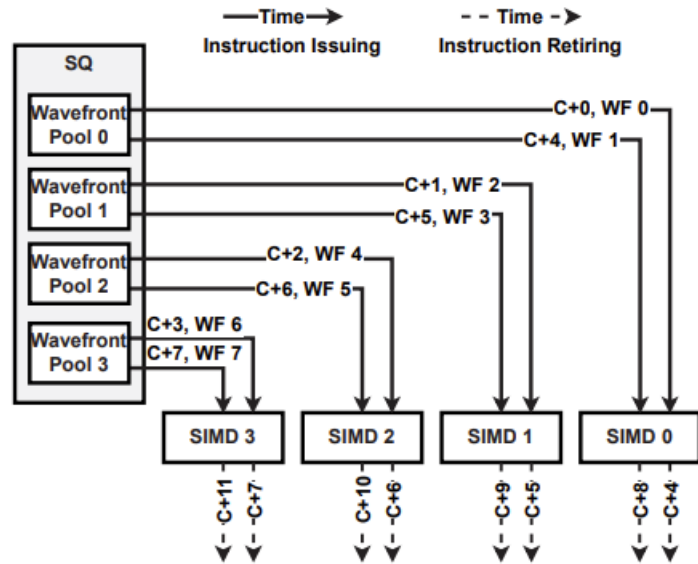


- Sequencer
 - At each clock, waves on **1 SIMD unit** are considered for execution (Round Robin scheduling among SIMDs)
 - At most **1 instruction per wavefront** may be issued
 - At most **1 instruction from each category** may be issued (S/V ALU, S/V GPR, LDS, global, branch, etc)
 - **Maximum of 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

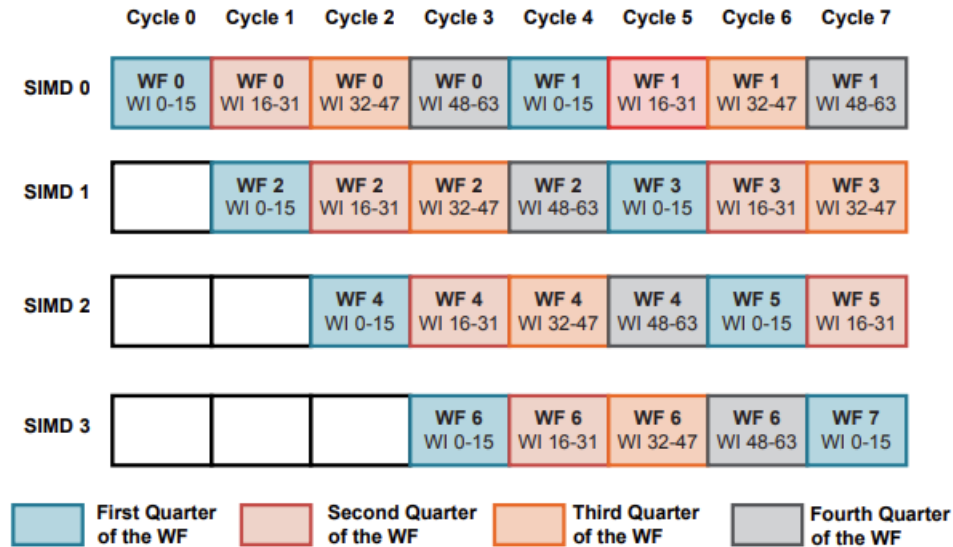
Registers contd.

- SPI dynamically maps HW registers to logical ones
- However, the following special registers are physically located in SQ's wavefront slots.
 - Program Counter (PC) stores 32b address
 - EXEC
 - 64b execution mask for predicated execution
 - Comparison registers:
 - VCC-64b
 - SCC-1b

Calculating GPU throughput



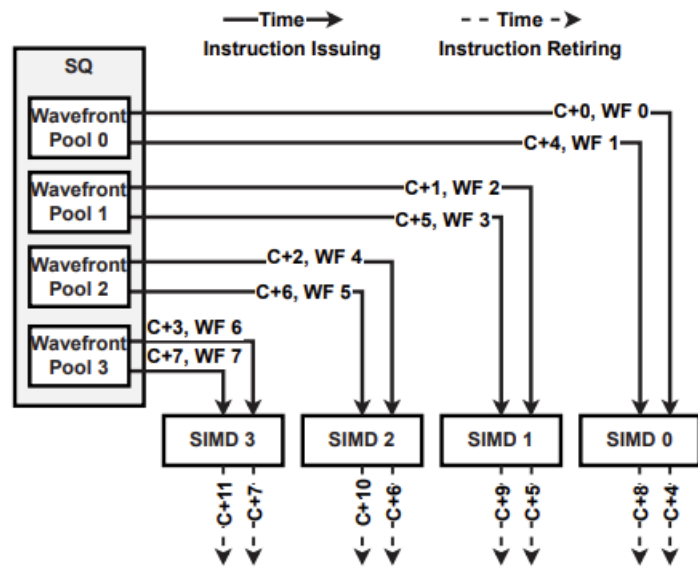
The timeline of instruction issuing



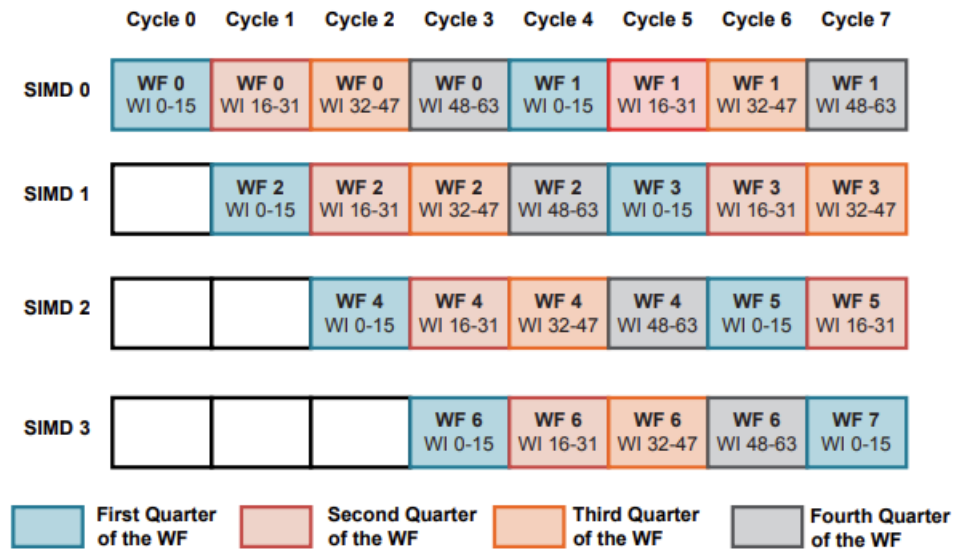
The time table of wavefronts executing in the SIMD units

- For MI100 with 120CUs,
- Instruction throughput= 120CUs x 4 SIMDs x 16 ALUs = 7680 instructions/cycle (IPC)
- Fused Mul-Add is 2ops/cycle, clock is 1.5GHz.
- Theoretical max throughput (TFLOPS)= ?

Calculating GPU throughput



The timeline of instruction issuing

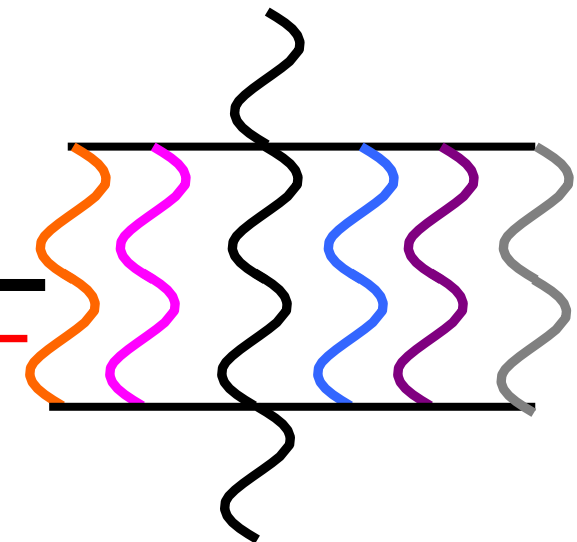


The time table of wavefronts executing in the SIMD units

- For MI100 with 120CUs,
- Instruction throughput= 120CUs x 4 SIMDs x 16 ALUs = 7680 instructions/cycle (IPC)
- Fused Mul-Add is 2ops/cycle, clock is 1.5GHz.
- Theoretical max throughput = $7680 \times 1.502 \times 10^9 \times 2 = 23 \text{ TFLOPS}$

Amdahl's Law calculates upper bound on SpeedUp

- Amdahl's Law
 - f : Parallelizable fraction of a program
 - S : Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{S}}$$


The diagram shows a horizontal pipeline of processors. A thick black line represents the serial portion of the program, labeled '1' above it. This line is divided into two segments: a black segment on the left labeled '1 - f' and a red segment on the right labeled 'f'. Below the red segment is a red line labeled 'S', representing the number of processors. To the right of the red segment, the pipeline branches into S parallel paths, each represented by a colored wavy line (orange, magenta, black, blue, purple, grey). These paths converge back into a single black line on the right, representing the serial portion of the program.

- Maximum speedup limited by serial portion: Serial bottleneck

Speed of Light Instruction Throughput can be derived

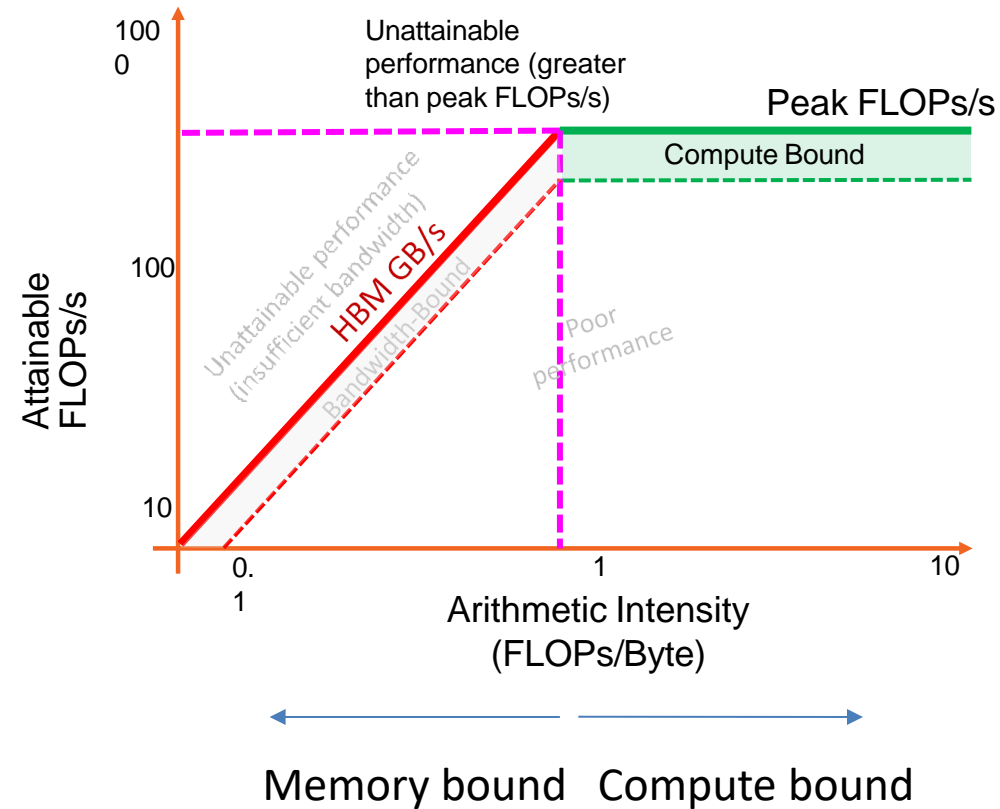
- Estimation by hand
 - #CUs x CLK x #Ops/cycle/CU
 - MI250 peak FP32 vector performance:
 - $104 * 1.7 * (2 * 64) = 22.63 \text{TFLOPS}$
- Additionally, profiler displays SOL (more later)

Arithmetic Intensity checks if kernel is memory/compute bound

- Arithmetic Intensity (AI)
 - FLOPs/Bytes
 - For a GEMM problem of size A (MxK), B (KxN) & C(MxN):
 - $AI = (\#FLOPs) / (\#Bytes)$
 $= 2MNK / 2(MK + NK + MN) = MNK / (MK + NK + MN)$
- Estimating bounds:
 - FP16 SOL = 181TFLOPS/1.6TB/s = 113 FLOPs/B
 - If M or N=1, $AI < 1 \Rightarrow$ memory bound
 - For GEMM MxKxN: 8192x128x8192,
AI= 124.1FLOPs/B \Rightarrow compute bound

Roofline model guides optimization

- Attainable FLOPs/s =
 - $\text{Min}(\text{peak GB/s}, \text{peak FLOPs/s})$
- Machine Balance:
 - Where $AI = \frac{\text{Peak FLOPs/s}}{\text{Peak GB/s}}$



Optimization Strategies

- Know your architecture
- Optimize memory layouts, execution config
- Profile & find the bottleneck
- Balance compute & memory operations:
 - Datatypes
 - Arithmetic costs
 - Latencies
- Better hardware utilization:
 - Control flow
 - Reduce instruction count
 - High throughput instructions
 - Parallel compute resources
- Maximize data-independence
 - Minimize communication or synchronization required

rocprof Profiler

- Command line profiler
- GPU HW counters vary by architecture:
 - rocprof -list-basic
 - rocprof -list-derived
- rocprof -stats <your_app>

results.stats.csv

```
-----  
"Name", "Calls", "TotalDurationNs", "AverageNs", "Percentage"  
"vecAdd(double*, double*, double*, int) [clone.kd]", 1, 5920, 5920, 100.0  
-----
```

- <https://rocm.docs.amd.com/projects/rocprofiler/en/latest/rocprof.html>

rocprof flags

- To get help:
rocprof -h
- Useful housekeeping flags:
 - --timestamp <on|off> - turn on/off gpu kernel timestamps
 - --basenames <on|off> - turn on/off truncating gpu kernel names (i.e., removing template parameters and argument types)
 - -o <output csv file> - Direct counter information to a particular file name
 - -d <data directory> - Send profiling data to a particular directory
 - -t <temporary directory> - Change the directory where data files typically created in /tmp are placed. This allows you to save these temporary files.
- Flags directing rocprofiler activity:
 - -i input<.txt|.xml> - specify an input file (note the output files will now be named input.*)
 - --hsa-trace - to trace GPU Kernels, host HSA events (more later) and HIP memory copies.
 - --hip-trace - to trace HIP API calls
 - --roctx-trace - to trace roctx markers
 - --kfd-trace - to trace GPU driver calls
- Advanced usage
 - -m <metric file> - Allows the user to define and collect custom metrics. See rocprofiler/test/tool/*.xml on GitHub for examples.

rocprof contd.

- rocprof -i input.txt -o vadd_profile.csv ./vadd
- Measure RD/WR between cache & memory
 - TCC_EA_RDREQ
 - TCC_EA_WRREQ

input.txt

```
pmc: TCC_EA_RDREQ_sum, TCC_EA_WRREQ_sum  
range: 0:1  
gpu: 0  
kernel: vecAdd
```

results.stats.csv

```
Index,KernelName,gpu-id,queue-id,queue-index,pid,tid,grd,wgr,lds,scr,vgpr  
,sgpr,fbar,sig,obj,TCC_EA_RDREQ_sum,TCC_EA_WRREQ_sum  
0,"vecAdd(double*, double*, double*, int) [clone .kd]"  
,0,0,0,15239,15242,10240,256,0,0,12,24,0,0x0,0x7f9ba4c45800,2610,1280
```

RDs = 2 X WRs for vecAdd

rocpof: Commonly Used GPU Counters

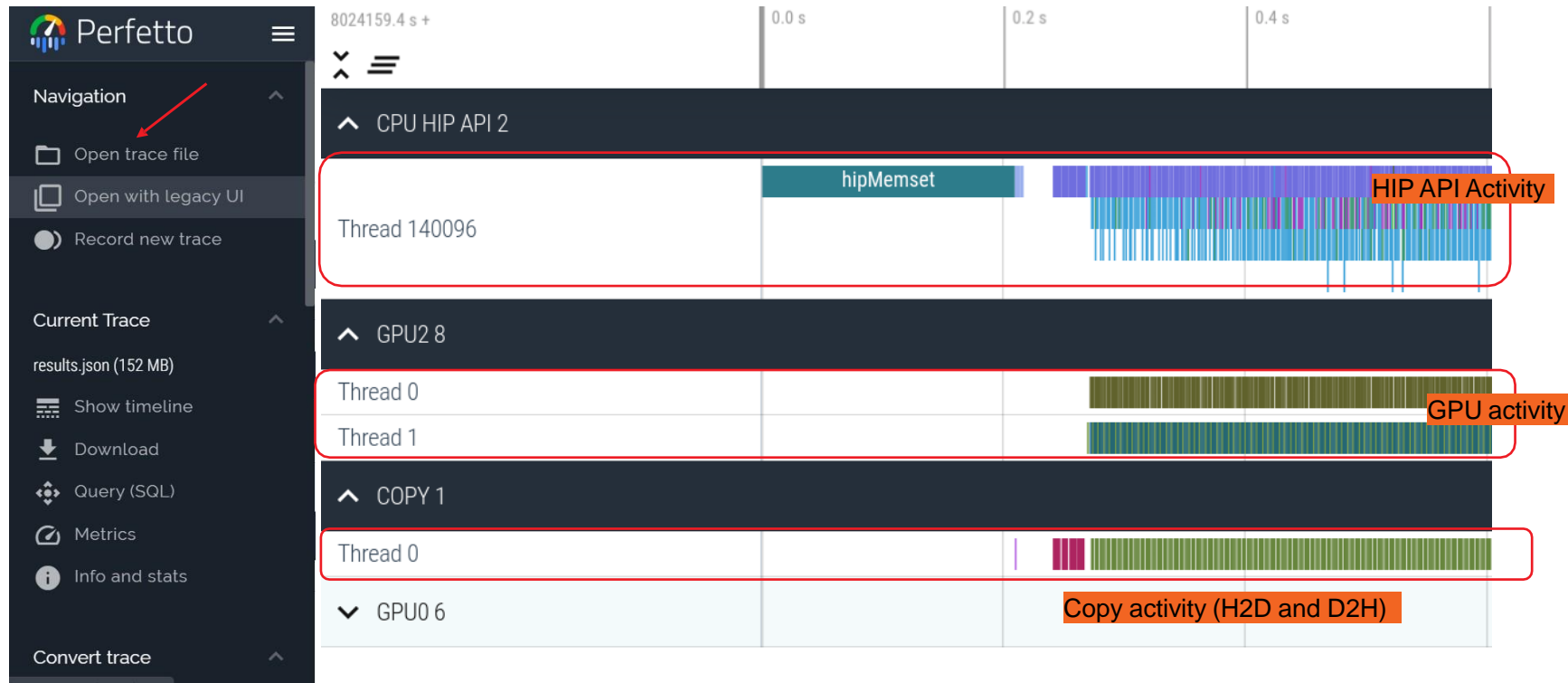
VALUUtilization	The percentage of ALUs active in a wave. Low VALUUtilization is likely due to high divergence or a poorly sized grid
VALUBusy	The percentage of GPUTime vector ALU instructions are processed. Can be thought of as something like compute utilization
FetchSize	The total kilobytes fetched from global memory
WriteSize	The total kilobytes written to global memory
L2CacheHit	The percentage of fetch, write, atomic, and other instructions that hit the data in L2 cache
MemUnitBusy	The percentage of GPUTime the memory unit is active. The result includes the stall time
MemUnitStalled	The percentage of GPUTime the memory unit is stalled
WriteUnitStalled	The percentage of GPUTime the write unit is stalled

rocpof + Perfetto: Collecting and Visualizing Application Traces

- rocprof can collect traces

\$ /opt/rocm/bin/rocprof --hip-trace <app with arguments>

This will output a .json file that can be visualized using the chrome browser and Perfetto (<https://ui.perfetto.dev/>)



Omniperf profiler: rocprof metrics + SOL

[Public]

We use the example sample/vcopy.cpp from the Omniperf installation folder:

Compile with hipcc:

```
$ hipcc -o vcopy vcopy.cpp
```

Profile with Omniperf:

```
$ omniperf profile -n vcopy_all -- ./vcopy 1048576 256
```

A new directory will be created called workloads/vcopy_all

Analyze the profiled workload:

```
$ omniperf analyze -p workloads/vcopy_all/mi200/ &> vcopy_analyze.txt
```

0. Top Stat						
	KernelName	Count	Sum(ns)	Mean(ns)	Median(ns)	Pc
0	vecCopy(double*, double*, double*, int, int) [clone .kd]	1	341123.00	341123.00	341123.00	100.0

2. System Speed-of-Light

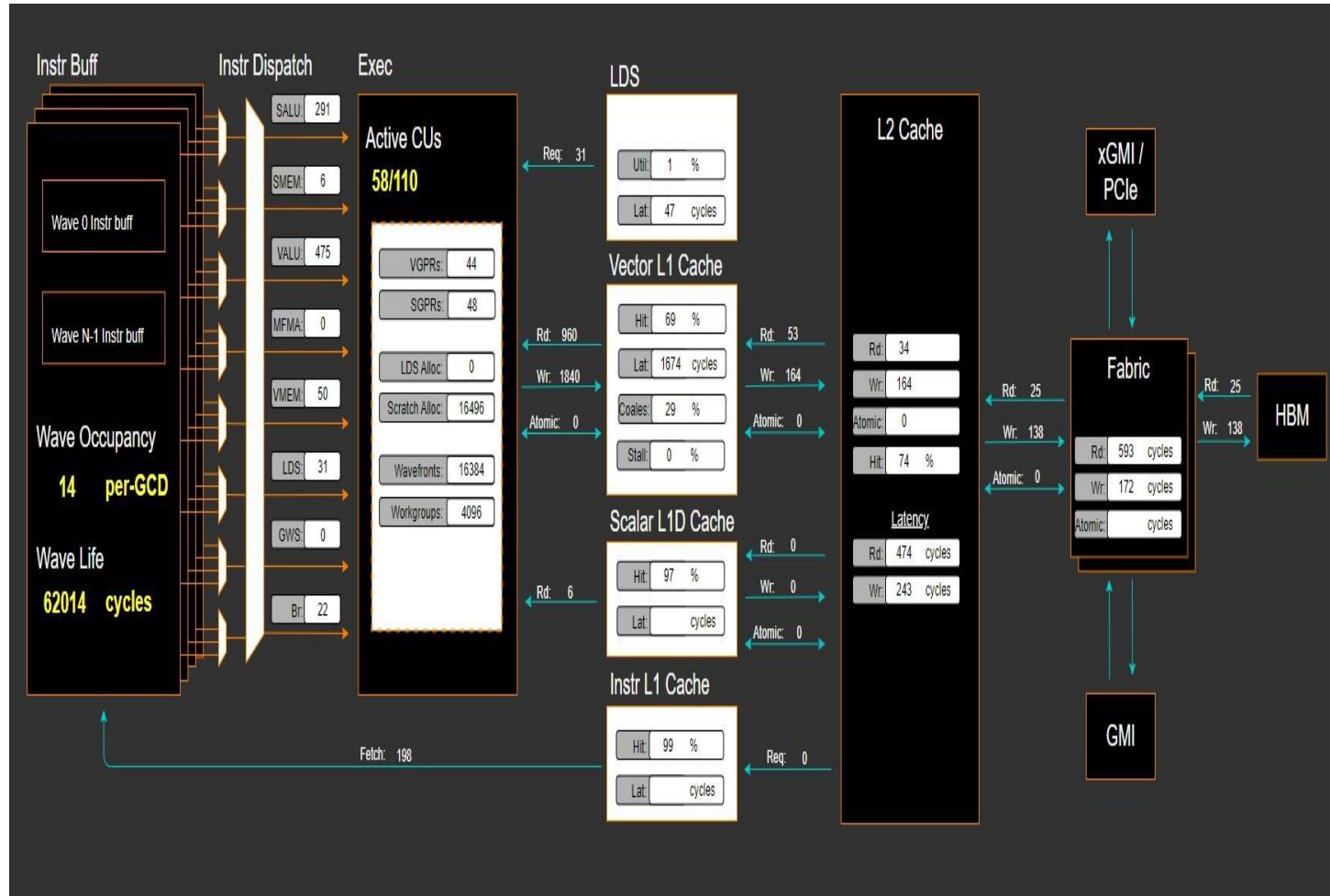
Index	Metric	Value	Unit	Peak	PoP
2.1.0	VALU FLOPs	0.00	Gflop	23936.0	0.0
2.1.1	VALU IOPs	89.14	Giop	23936.0	0.37242200388114116
2.1.2	MFMA FLOPs (BF16)	0.00	Gflop	95744.0	0.0
2.1.3	MFMA FLOPs (F16)	0.00	Gflop	191488.0	0.0
2.1.4	MFMA FLOPs (F32)	0.00	Gflop	47872.0	0.0
2.1.5	MFMA FLOPs (F64)	0.00	Gflop	47872.0	0.0
2.1.6	MFMA IOPs (Int8)	0.00	Giop	191488.0	0.0
2.1.7	Active CUs	58.00	Cus	110	52.72727272727273
2.1.8	SALU Util	3.69	Pct	100	3.6862586934167525
2.1.9	VALU Util	5.90	Pct	100	5.895531580380328
2.1.10	MFMA Util	0.00	Pct	100	0.0
2.1.11	VALU Active Threads/Wave	32.71	Threads	64	51.10526315789473
2.1.12	TBC - Tbcu	0.00	Tbcu/cycle	5	10.576640821020212

7.1 Wavefront Launch Stats

Index	Metric	Avg	Min	Max	Unit
7.1.0	Grid Size	1048576.00	1048576.00	1048576.00	Work items
7.1.1	Workgroup Size	256.00	256.00	256.00	Work items
7.1.2	Total Wavefronts	16384.00	16384.00	16384.00	Wavefronts
7.1.3	Saved Wavefronts	0.00	0.00	0.00	Wavefronts
7.1.4	Restored Wavefronts	0.00	0.00	0.00	Wavefronts
7.1.5	VGPRs	44.00	44.00	44.00	Registers
7.1.6	SGPRs	48.00	48.00	48.00	Registers
7.1.7	LDS Allocation	0.00	0.00	0.00	Bytes
7.1.8	Scratch Allocation	16496.00	16496.00	16496.00	Bytes

Omniperf profiler: rocprof metrics + SOL

[Public]



SW Optimizations for AI can be classified into 3

Reducing compute

- Model Compression
 - Pruning
 - Quantization
 - Knowledge distillation
- Efficient architectures
 - Lightweight models
 - Linear Attention
- Sparse Computation

Increasing FLOPs/Byte

- Memory access optimization
 - Layout, Hierarchy
- Compute re-use
 - Caching, Kernel fusion
- Mixed-precision

Parallelism

- Inter-device
 - Data, model, pipeline
- Intra-device
 - Concurrency

AMD Hiring in AI

1. 80% of software engineer roles will adopt AI by '27 (ACM tech news, ITPro, Oct 3, '24)
 2. Internships & Full-time
 - <https://careers.amd.com/careers-home/jobs>
 - <https://www.linkedin.com/in/hariss->
-