Introduction to Parallel & Distributed Computing

GPU Programming with HIP (2)

GPU Memory Model & Architecture

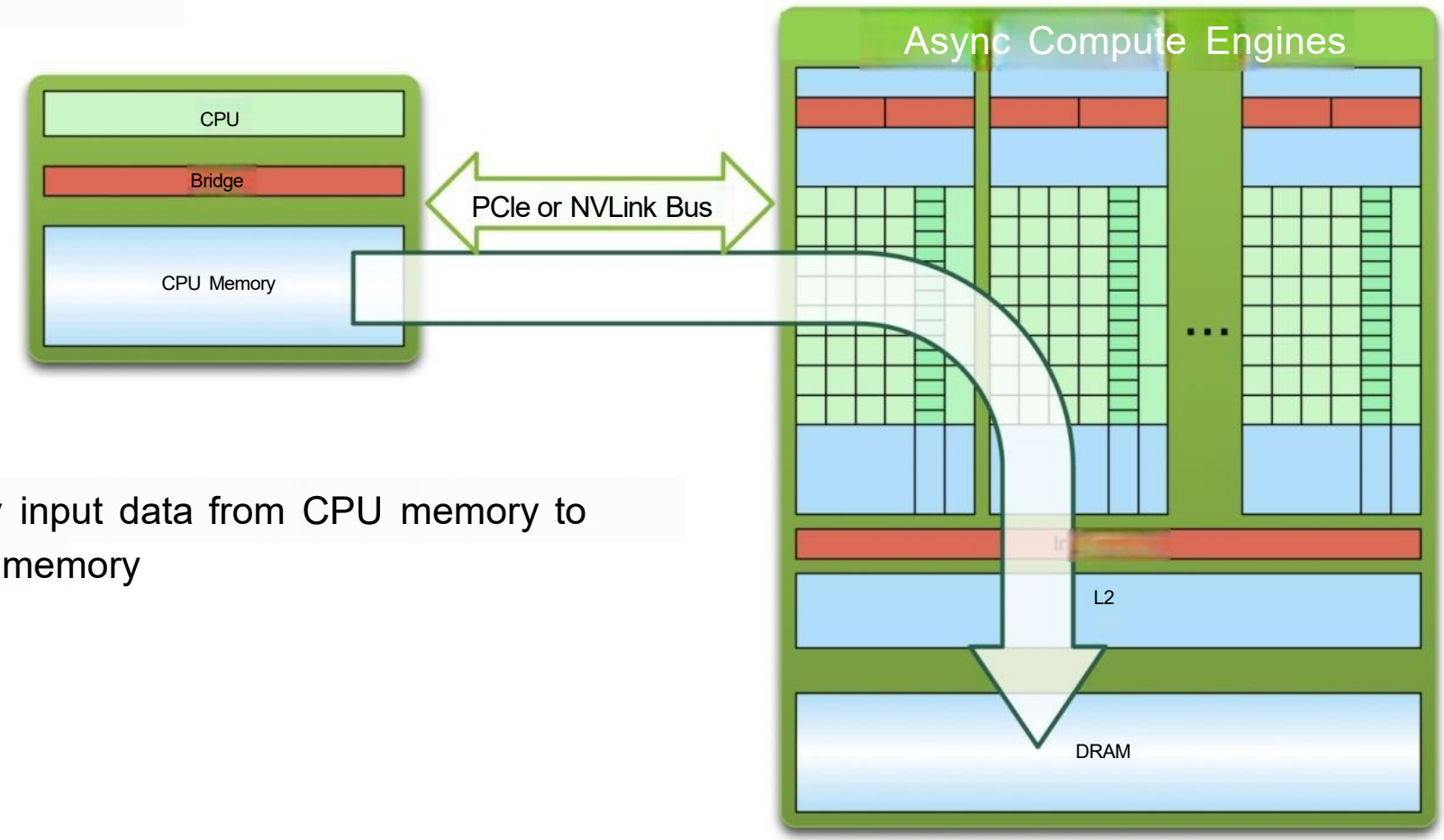
Lecture 8, Spring 2024

Instructor: 罗国杰

gluo@pku.edu.cn

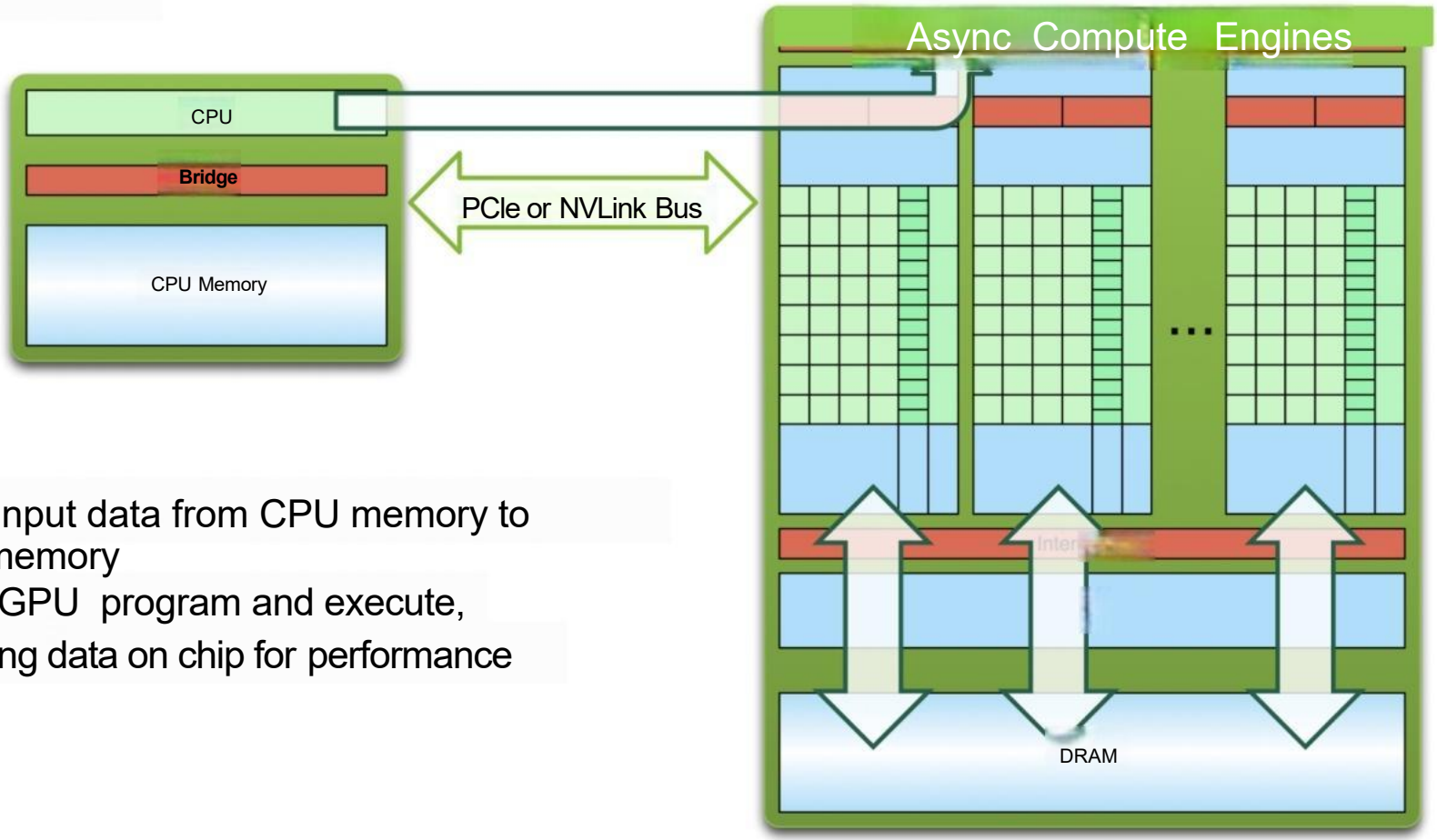
Adapted from UW CSEP 590B, Dr. Hari Sadasivan
https://courses.cs.washington.edu/courses/csep590b/

Simple GPU Processing Workflow Pattern



1. Copy input data from CPU memory to GPU memory

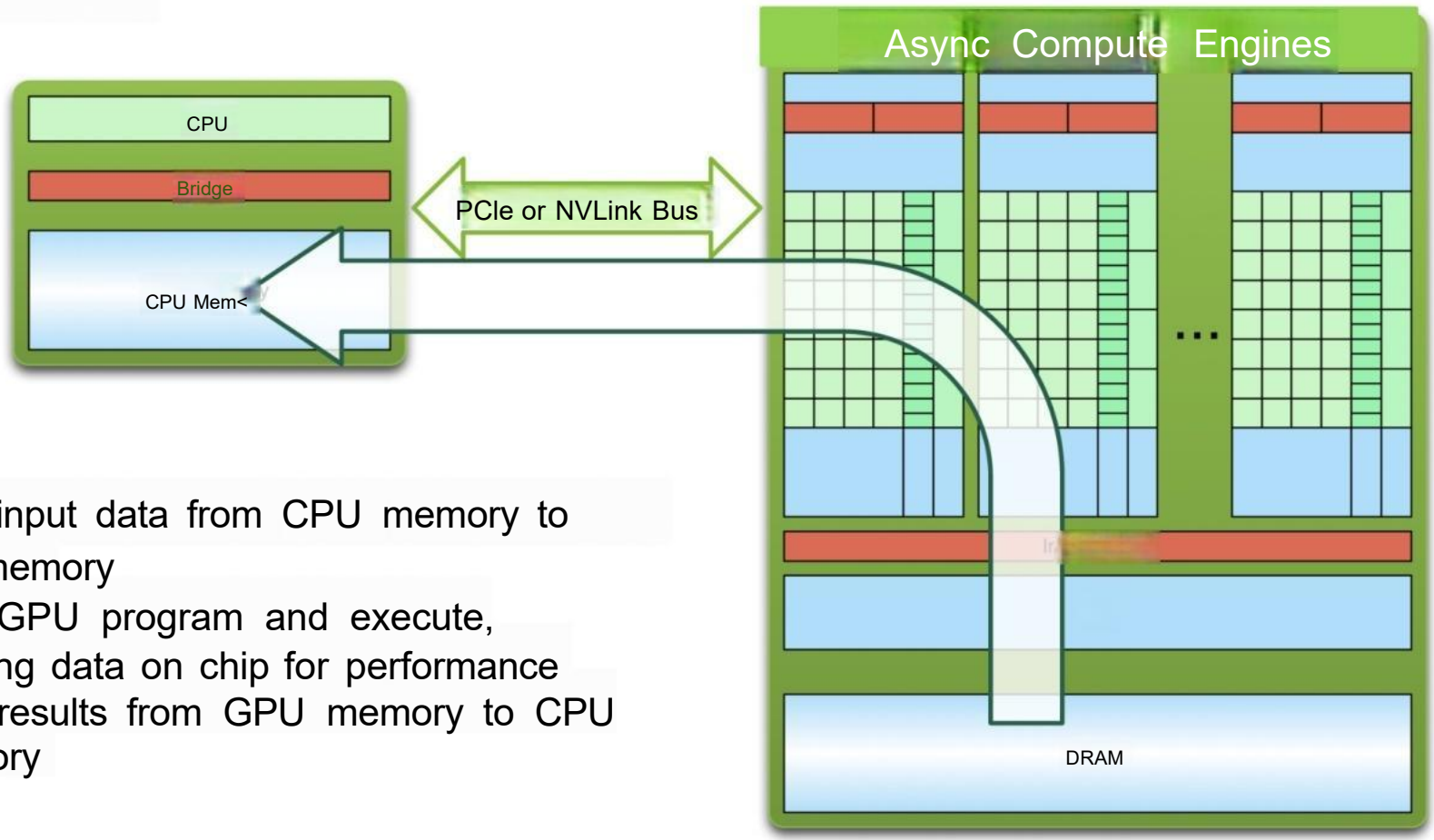
Simple GPU Processing Workflow Pattern



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

adapted from UW CSEP 590, Dr. Hari Sadasivan

Simple GPU Processing Workflow Pattern



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

adapted from UW CSEP 590, Dr. Hari Sadasivan

Recap: Keywords

- Kernel
- Grid
- Block / Workgroup
- Warp / Wavefront
- Thread / Work-item

HIP Hello world

```
1  #include <hip/hip_runtime.h>
2  __global__ void gpuHello() {
3      int tid = blockIdx.x*blockDim.x+threadIdx.x;
4      printf("Hello World from thread %d\n",tid);
5  }
6  int main() {
7      gpuHello<<<1,64>>>();
8      hipDeviceSynchronize();
9  }
```

```
hipcc helloWorld.cpp -o helloWorld
./helloWorld
```

```
Hello World from thread 0
Hello World from thread 1
Hello World from thread 2
Hello World from thread 3
Hello World from thread 4
Hello World from thread 5
Hello World from thread 6
Hello World from thread 7
Hello World from thread 8
Hello World from thread 9
Hello World from thread 10
Hello World from thread 11
Hello World from thread 12
Hello World from thread 13
Hello World from thread 14
Hello World from thread 15
Hello World from thread 16
Hello World from thread 17
Hello World from thread 18
Hello World from thread 19
```

Simple GPU Processing Workflow

1. Prepare the data
2. Copy the data to the device
3. Execute the device code
4. Copy data from the device

```
__global__ void vectorAdd(float*a, float scalar) {
    int tid = blockIdx.x * blockDim.x + threadIdx.X;
    if  (tid<N) a[tid] += scalar;
}

int  main() {
   float *h_a, scalar=5.0f, *d_a;
   hipHostMalloc((void**)&h_a,N*sizeof(float));
   for (int i=0; i<N; i++) h_a[i]=i;

   hipMalloc((void**)&d_a,N*sizeof(float));
   hipMemcpy(d_a,h_a,N*sizeof(float),
      hipMemcpyHostToDevice);

  dim3 grid((N+255)/256,1,1),block(256,1,1);
  hipLaunchKernelGGL(grid,block,0,0,d_a,scalar);

  hipMemcpy(h_a,d_a,N*sizeof(float),
     hipMemcpyDeviceToHost);

   hipHostFree(h_a);
   hipFree(d_a);

   return  0;
```

adapted from UW CSEP 590, Dr. Hari Sadasivan

Exercise: total thread number

For a vector addition,assume that the vector length is 8000, each thread calculates one output element,and the thread block size is 1024 threads. The programmer configures the kernel launch to have a minimal number of thread blocks to cover all output elements.How many threads will be in the grid?

- A. 8000
- B. 8196
- C. 8192
- D. 8200

Answer: (C)

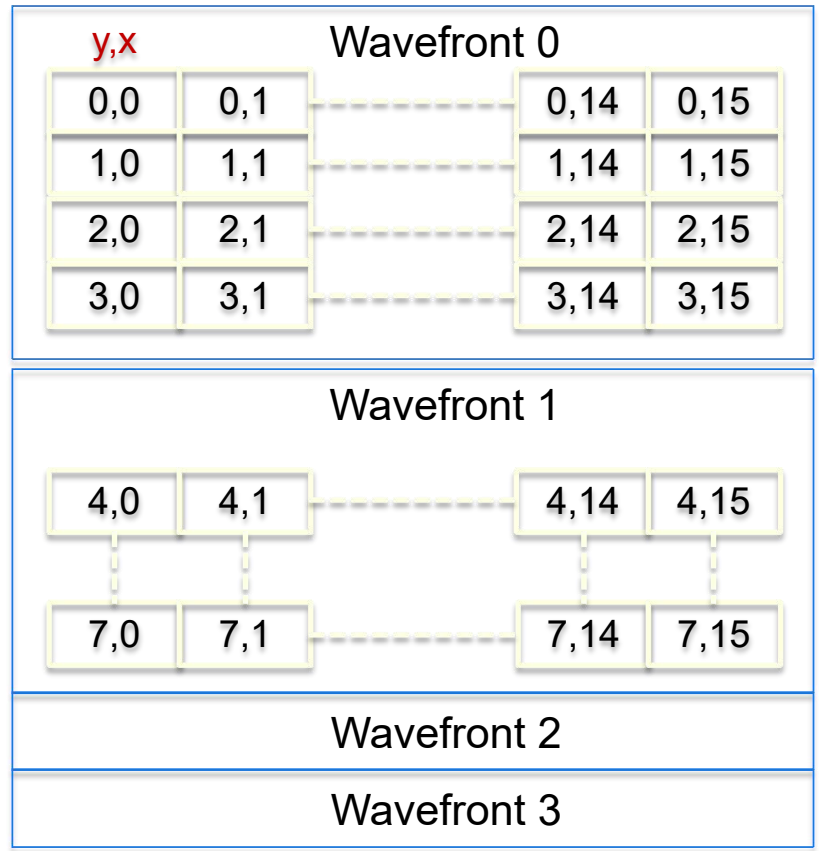
Exercise: best block size

If a Compute Unit (CU) on GPU can take up to 1536 threads and up to 8 thread blocks, which of the following block configurations would result in the most number of threads in each CU?

A. 1024 threads/block
B. 512 threads/block
C. 128 threads/block
D. None

Review: Warp/Wavefront Scheduling

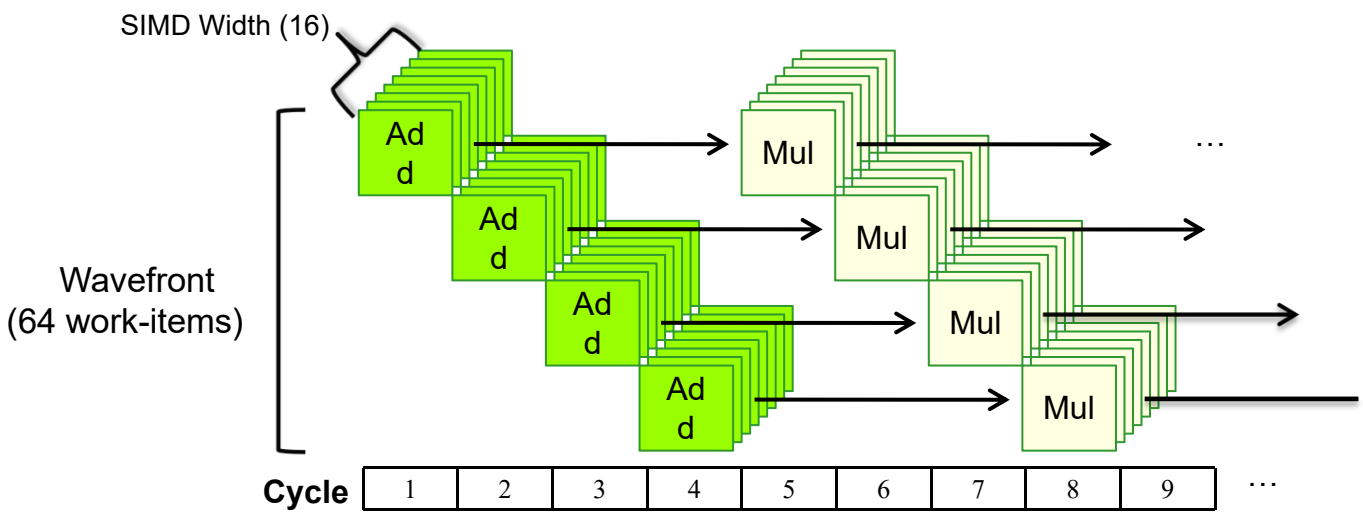
- Hardware creates wavefronts by grouping threads of a work group
  - Along the X dimension first
- All threads in a wavefront execute the same instruction
  - Threads within a wavefront move in lockstep
- Threads have their own register state and are free to execute different control paths
  - Thread masking used by HW
  - Predication can be set by compiler



Grouping of a 16x16 block into wavefronts

Review: SIMT Execution Model

- SIMD execution can be combined with pipelining
- ALUs all execute the same instruction
- Pipelining is used to break instruction into phases
- When first instruction completes (4 cycles here), the next instruction is ready to execute



Divergent Control Flow

- Instructions are issued in lockstep in a wavefront /warp for both AMD and Nvidia
- However each work item can execute a different path from other threads in the wavefront
- If work items within a wavefront go on divergent paths of flow control, the invalid paths of a work-items are masked by hardware
- Branching should be limited to a wavefront granularity to prevent issuing of wasted instructions

Predication and Control Flow

- **How do we handle threads going down different execution paths when the same instruction is issued to all the work-items in a wavefront ?**
- **Predication is a method for mitigating the costs associated with conditional branches**
  - Beneficial in case of branches to short sections of code
  - Based on fact that executing an instruction and squashing its result may be as efficient as executing a conditional
  - Compilers may replace “switch” or “if then else” statements by using branch predication

Predication for GPUs

- **Predicate is a condition code that is set to true or false based on a conditional**
- **Both cases of conditional flow get scheduled for execution**
  - Instructions with a true predicate are committed
  - Instructions with a false predicate do not write results or read operands
- **Benefits performance only for very short conditionals**

```
__global__
void test() {
      int  tid= get_local_id(0) ;
      if( tid %2 == 0)
            Do_Some_Work() ;
      else
            Do_Other_Work() ;
}
```

Predicate = True for threads 0,2,4….

Predicate = False for threads 1,3,5….

Predicates switched for the else condition

Divergent Control Flow

- **Case 1: All odd threads will execute if conditional while all even threads execute the else conditional. The if and else block need to be issued for each wavefront**
- **Case 2: All threads of the first wavefront will execute the if case while other wavefronts will execute the else case. In this case only one out of if or else is issued for each wavefront**

Case 1

```
assert(get_local_size(0) == 128);
int tid = get_local_id(0)
if ( tid % 2 == 0) //Even Work Items
    DoSomeWork()
else
    DoSomeWork2()
```

Conditional – With divergence

Case 2

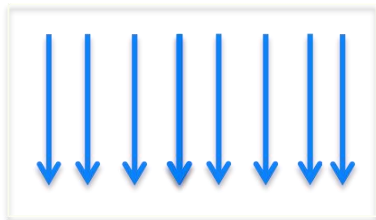
```
assert(get_local_size(0) == 128);
int tid = get_local_id(0)
if ( tid / 64 == 0)  //Full First Wavefront
    DoSomeWork()
else if (tid /64 == 1) //Full Second  Wavefront
    DoSomeWork2()
```

Conditional – No divergence

Effect of Predication on Performance

Time for Do_Some_Work = t_1 (if case)
Time for Do_Other _Work = t_2 (else case)

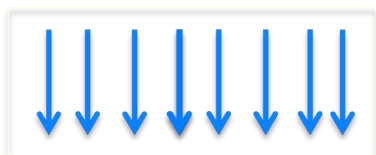
if(tid %2 == 0)



Do_Some_Work()

Green colored threads have valid results

Squash invalid results, invert mask



Do_Other _Work()

Green colored threads have valid results

Squash invalid results

T = 0

T = t_{start}

t_1

T = t_{start} + t_1

t_2

T = t_{start} + t_1 + t_2

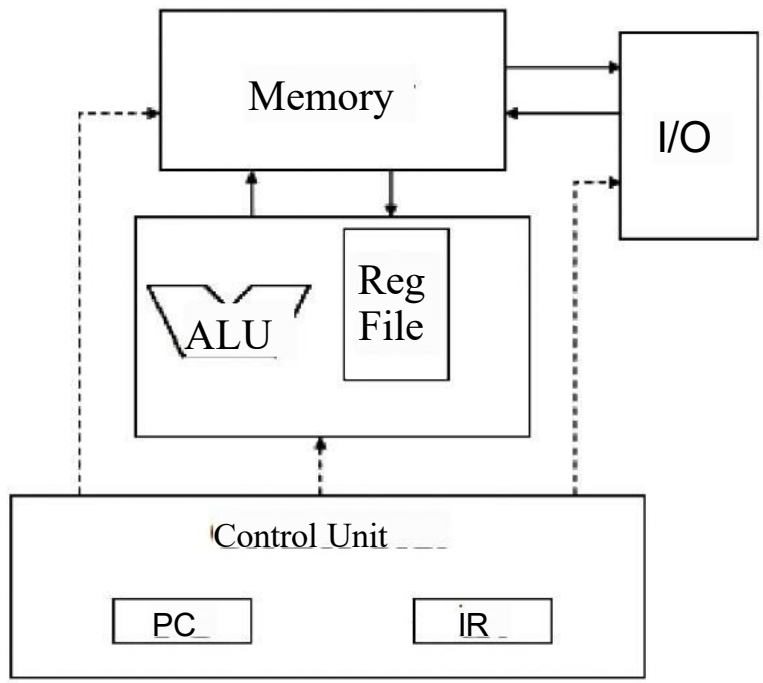
Memory Model

Registers vs Memory

Registers are fast but few

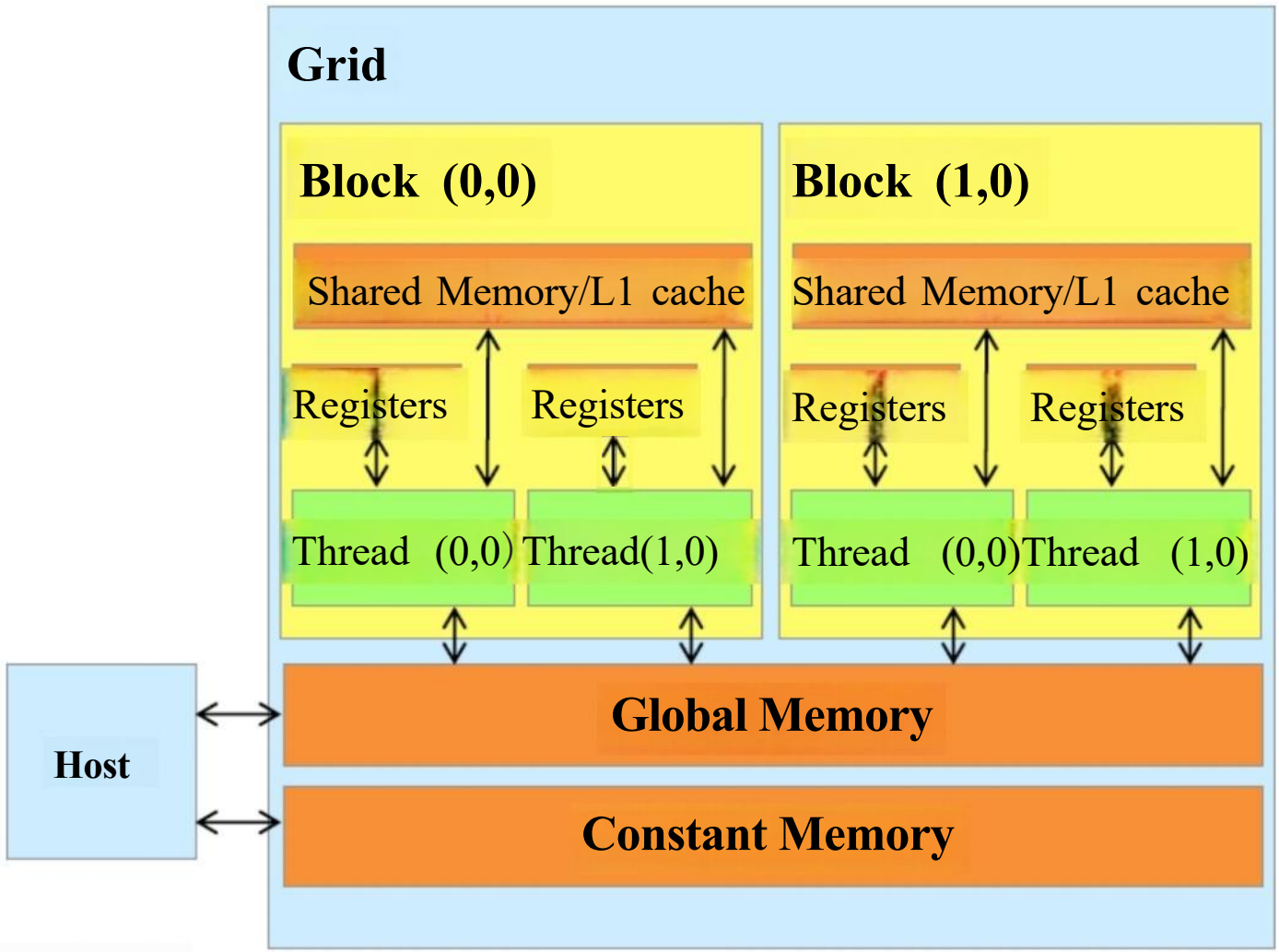
- No additional memory access instruction
- Very fast to use, however, there are very few of them

Memory is expensive (slow), but very large



adapted from UW CSEP 590, Dr. Hari Sadasivan

Simplified Memory Model



adapted from UW CSEP 590, Dr. Hari Sadasivan

Device memory hierarchy

Registers (per-thread-access)

- Used automatically
- Size on the order of kilobytes, Very fast access

Local memory (per-thread-access)

- Used automatically if all registers are reserved
- Local memory resides in global memory &very slow

Shared memory (per-block-access)

- Usage must be explicitly programmed
- Size on the order of kilobytes but fast

Global memory (per-device-access)

- Managed by the host through HIP API
- Size on the order of gigabytes but very slow

Constant Memory (per-device-access)

- Same as global memory but cached
- Uses special instructions to access

adapted from UW CSEP 590, Dr. Hari Sadasivan

HIP Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

- __device__ is optional when used with __shared__ or __constant__
- Automatic variables without any qualifier reside in a register
  - Except per-thread arrays that reside in global memory

A Common Programming Strategy

Global memory resides in device memory (DRAM)

A profitable way of performing computation on the device is to tile the input data to take advantage of fast shared memory:

- Partition data into subsets (tiles) that fit into shared memory
- Handle each data subset with one thread block by:

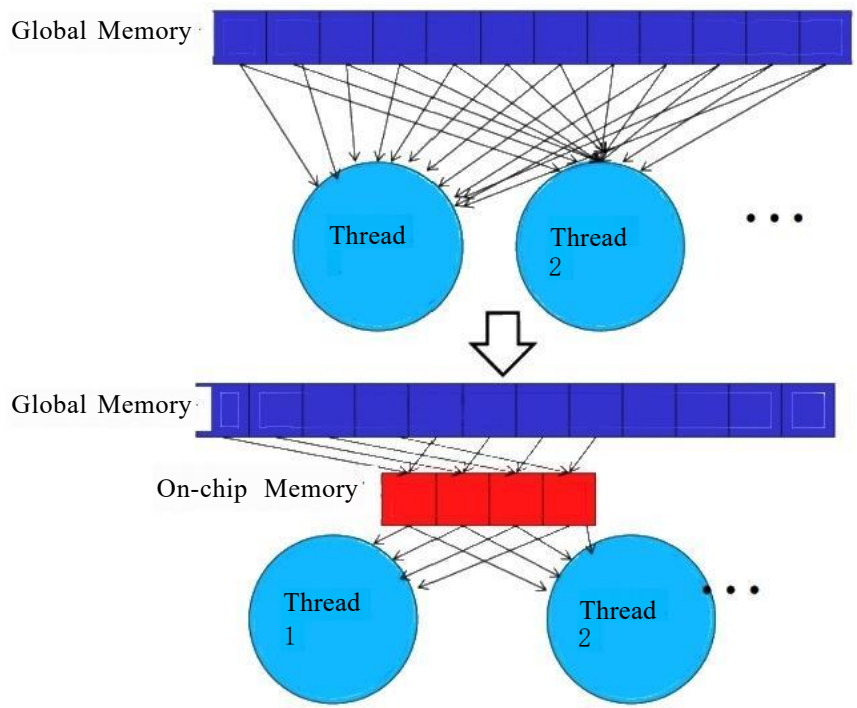
  Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism

  Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element

  Copying results from shared memory to global memory

Shared memory

```
__global__ void MatrixMulKernel(float * M, float* N, float*P, int width) {
    __shared__ float subTileM[TILE_SIZE][TILE_SIZE];
    __shared__ float subTileN[TILE_SIZE][TILE_SIZE];
    ... ...
```



adapted from UW CSEP 590, Dr. Hari Sadasivan

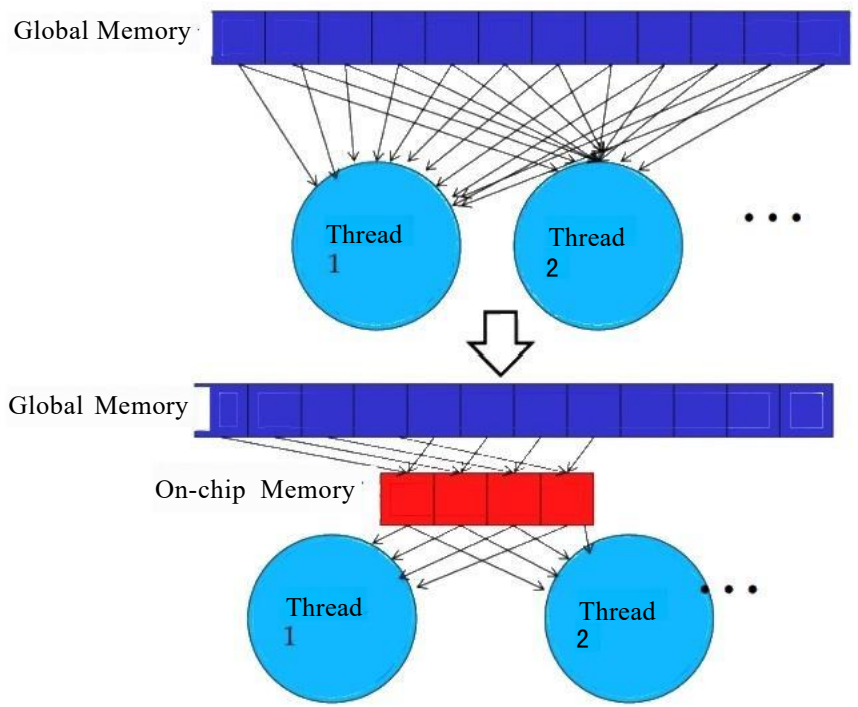
Outline of Technique

1. Identify a tile of global data that is accessed by multiple threads

2. Load the tile from global memory into on-chip memory

3. Have the multiple threads access their data from the on-chip memory

4. Move on to the next block/tile



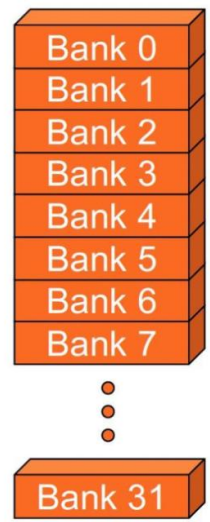
adapted from UW CSEP 590, Dr. Hari Sadasivan

Shared memory

- In a parallel machine, many threads access memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth

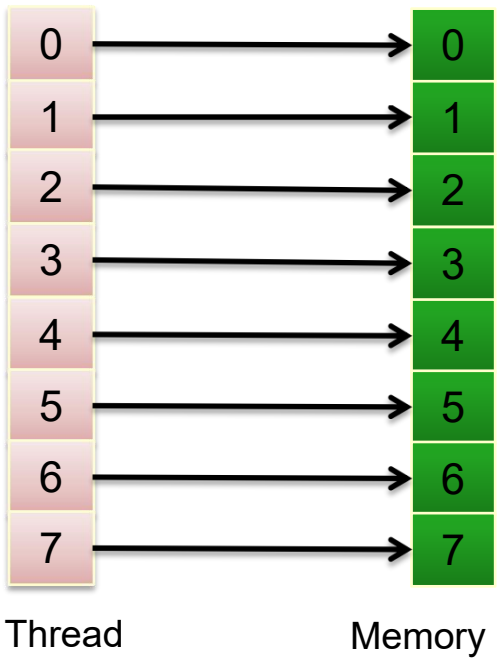
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks

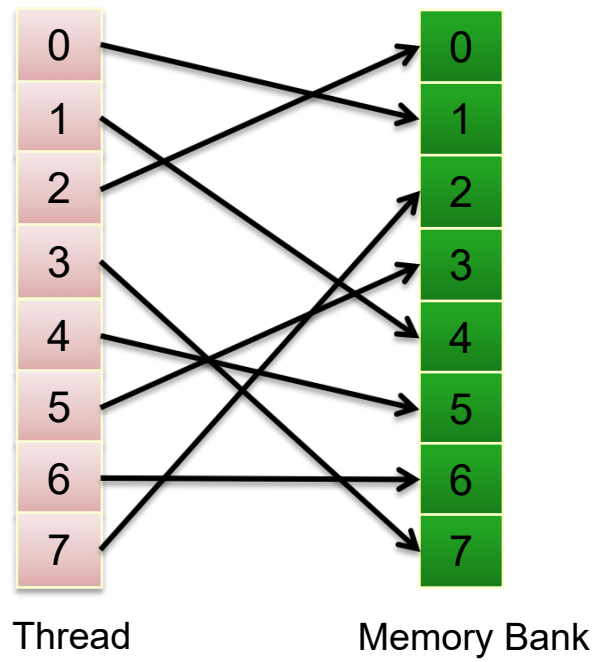
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized



Shared Memory: the Fast Case (1/2)

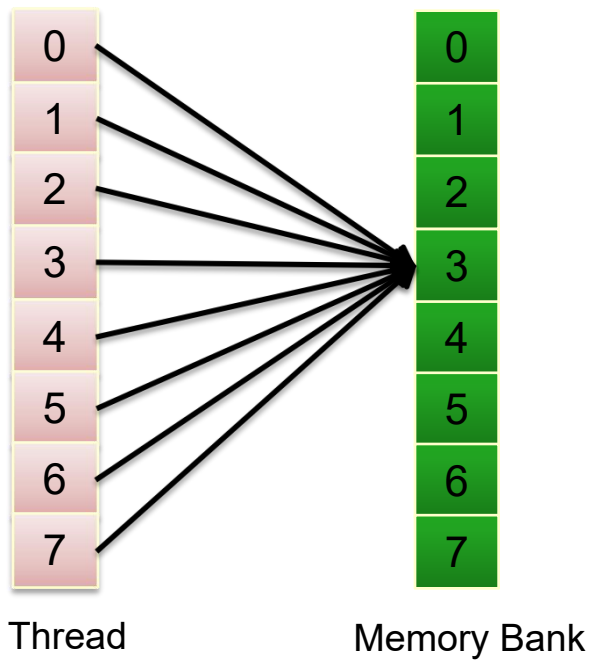
- **If there are no bank conflicts, each bank can return an element without any delays**
  - Both of the following patterns will complete without stalls on current GPU hardware





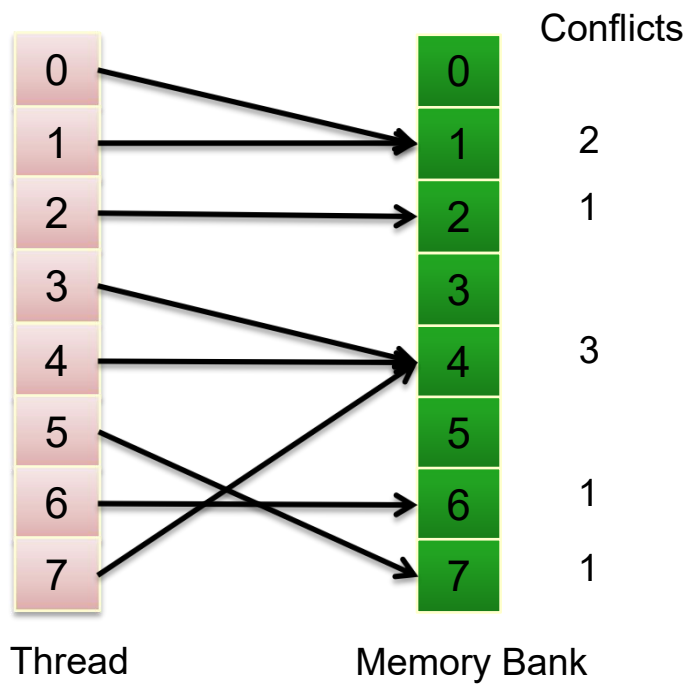
Shared Memory: the Fast Case (2/2)

- **If all accesses are to the same address, then the bank can perform a broadcast and no delay is incurred**
  - The following will only take one access to complete assuming the same data element is accessed



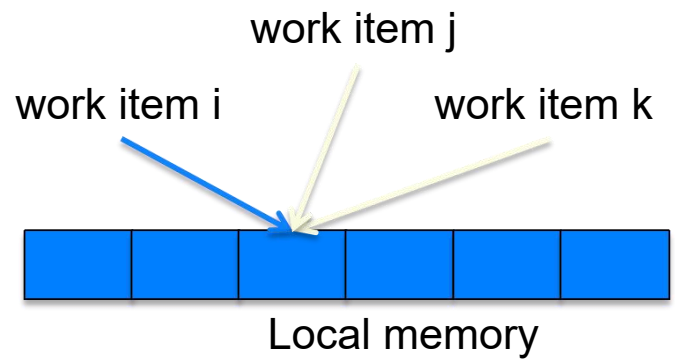
Shared Memory: the Slow Case

- **If multiple accesses occur to the same bank, then the bank with the most conflicts will determine the latency**
  - The following pattern will take 3 times the access latency to complete



Warp Voting Example

- **Implicit synchronization per instruction allows for techniques like warp voting**
  - Useful for devices without atomic shared memory operations
  - We discuss warp voting with the 256-bin Histogram example
- **For 64 bin histogram, we build a subhistogram per thread**
  - Shared memory per block for 256 bins
  - 256 bins * 4Bytes * 64 threads / block = 64KB
  - some GPUs have only 16KB of shared memory
- **Alternatively, build subhistorgram per warp**
  - shared memory required per block
  - 256 bins * 4Bytes * 2 warps / block = 2KB

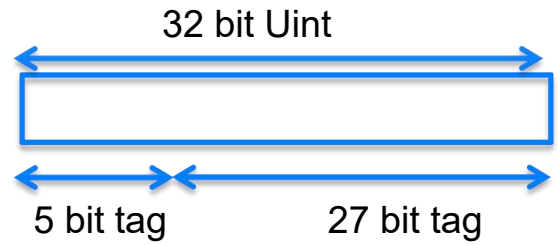


Shared memory write combining on allows **ONLY** one write from work-items i,j or k to succeed

By tagging bits in local memory and rechecking the value a work-item could know if its previously attempted write succeeded

Warp Voting for Histogram256

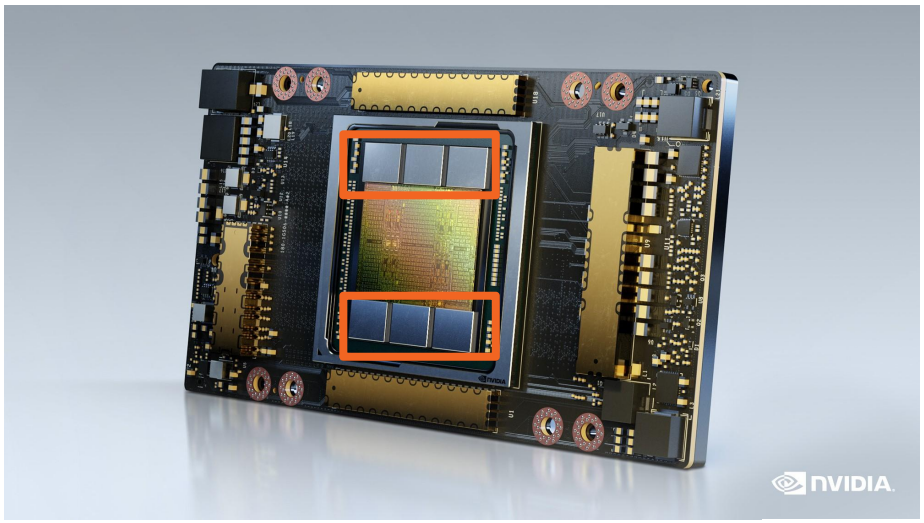
- **Build per warp subhistogram**
  - Combine to per block subhistogram
  - Share memory budget in per warp sub histogram technique allows us to have multiple blocks active
- **Handle conflicting writes by threads within a warp using warp voting**
  - Tag writes to per warp subhistogram with intra-warp thread ID
  - This allows the threads to check if their writes were successful in the next iteration of the while loop
- **Worst case : 32 iterations done when all 32 threads write to the same bin**

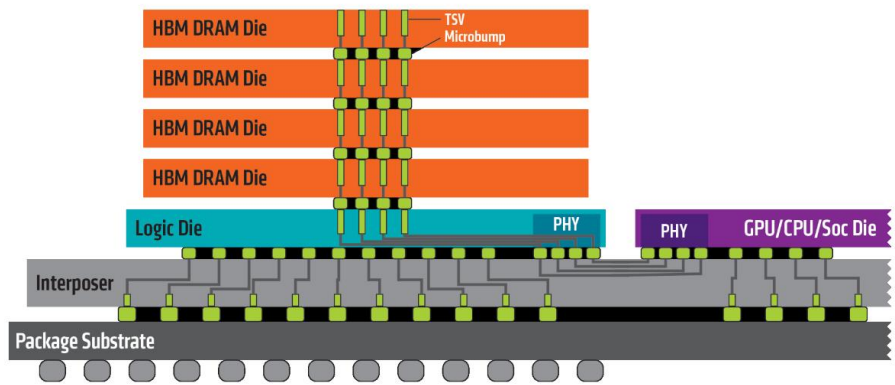



```
void addData256(
  volatile __local uint * l_WarpHist,
  uint data, uint workitemTag) {

  unsigned int count;
  do {
    // Read the current value from histogram
    count = l_WarpHist[data] & 0x07FFFFFU;
    // Add the tag and incremented data to
    // the position in the histogram
    count = workitemTag | (count + 1);
    l_WarpHist[data] = count;
  }
  // Check if the value committed to local memory
  // If not go back in the loop and try again
  while(l_WarpHist[data] != count);
}
```

Global Memory: High-B/W Memory





Global memory is slow

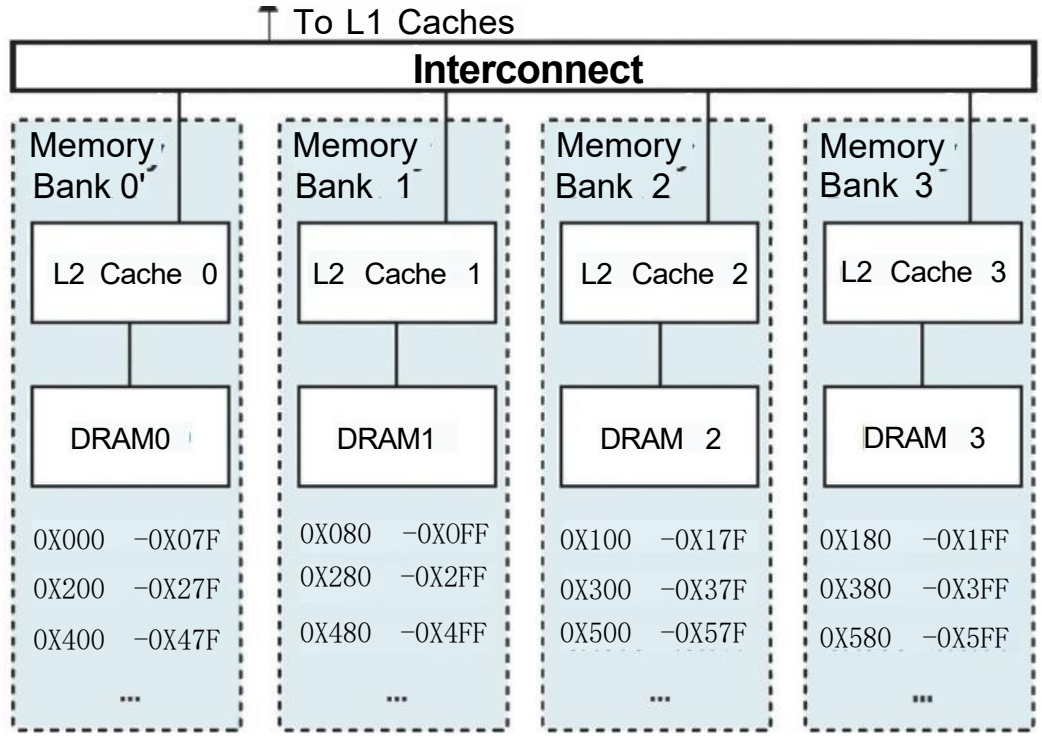
Latency:

- L1 cache - few 10s of cycles.
- L2 cache - few hundreds of cycles
- Global Memory - 1000 cycles

How do we hide global memory access latency?

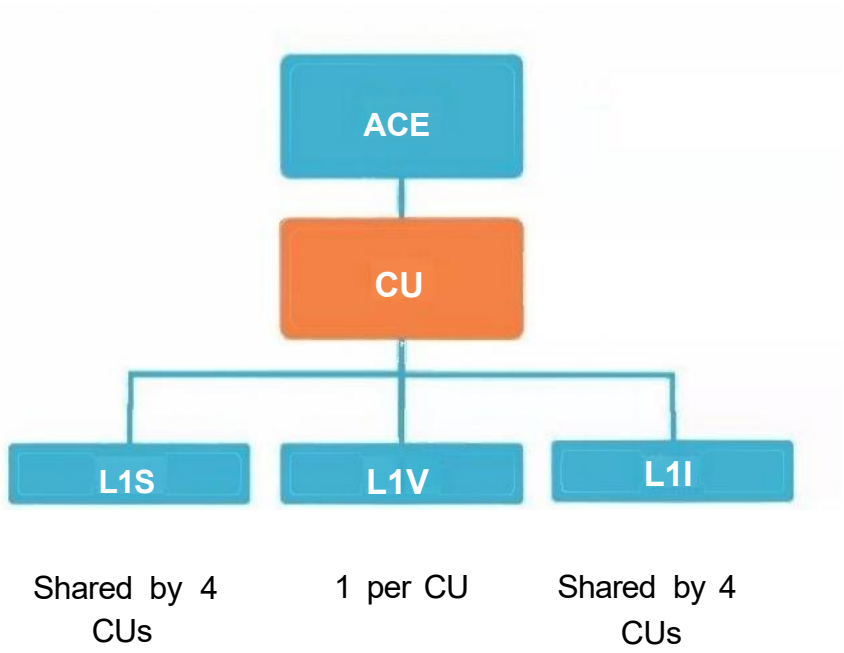
Global memory is slow

- DRAM (low-latency) is now replaced with HBM (higher throughput)
- Memory side caching
  - L2 or Texture Cache Channel (TCC)
- This is still not enough for the CUs.



L1 cache

- L2 is memory side cache while L1 is for CUs
- L1 is write-through cache



Hiding DRAM latency with Multiple Banks

With one bank, time stil wasted in between bursts

Latency can be hidden by having multiple banks

Need many threads to simultaneously access memory to keep all banks busy

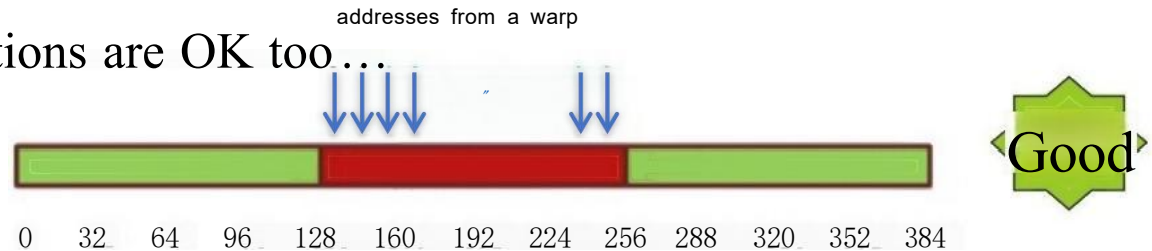
- Achieved with having high occupancy in GPU cores
- Similar idea to hiding pipeline latency in the core

Slide credit:Izzat El Hajj

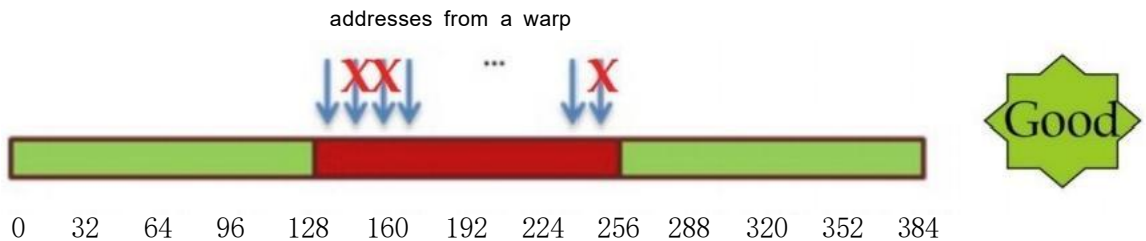
Coalesced access: reading floats

All threads participate,accesses fit within a region of 128 bytes.

Permutations are OK too…



Not all threads participate,accesses fit within a region of 128 bytes



Misaligned accesses (fit in two regions of 128 bvtes):

adapted from UW CSEP 590, Dr. Hari Sadasivan

Memory Coalescing (I)

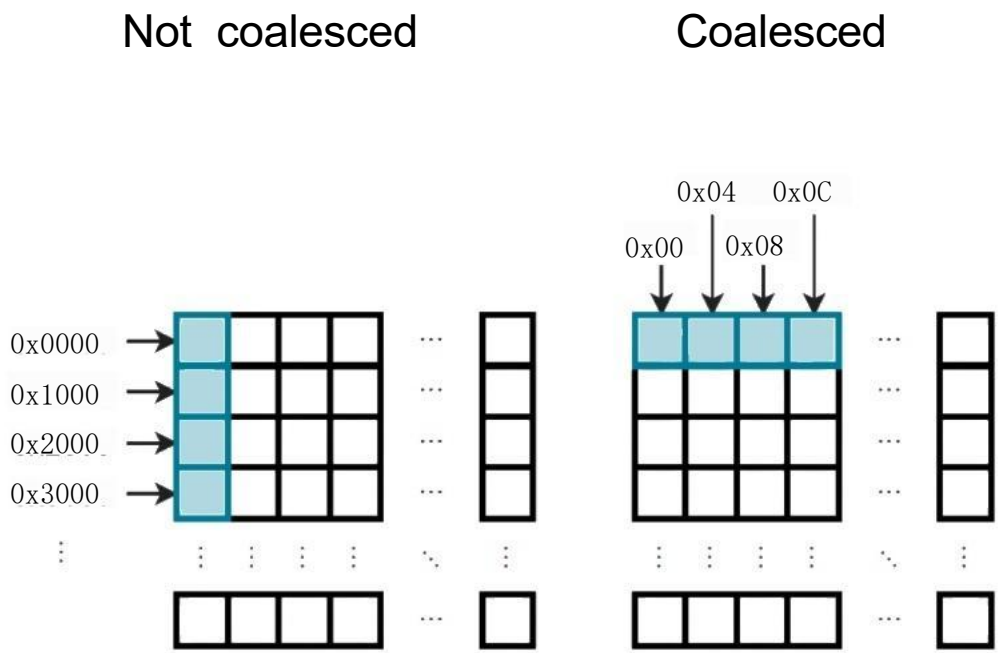
- When threads in the same warp access consecutive memory locations in the same burst,the accesses can be combined and served by one burst
  - One DRAM transaction is needed
  - Known as memory coalescing

- If threads in the same warp access locations not in the same burst, accesses cannot be combined
  - Multiple transactions are needed
  - Takes longer to service data to the warp
  - Sometimes called memory divergence

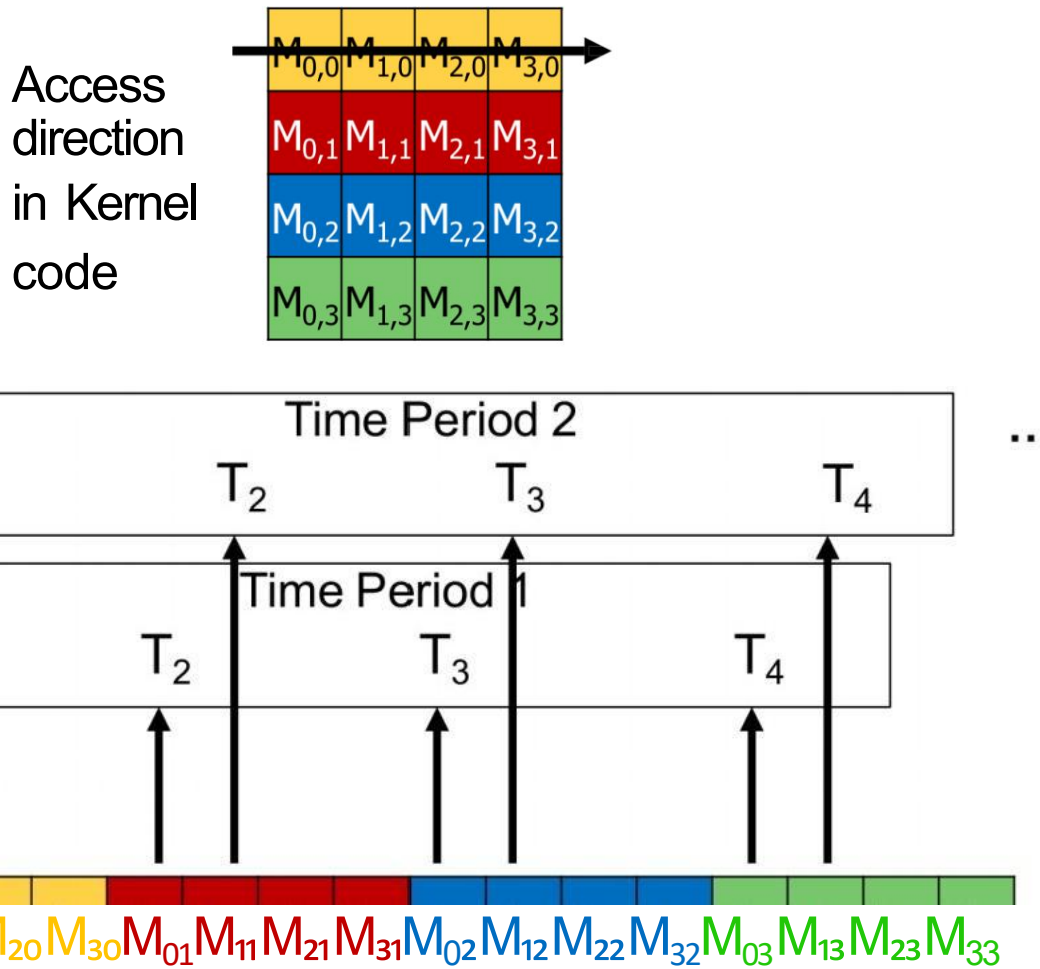
Slide credit:Izzat El Hajj

Memory Coalescing (II)

- When accessing global memory, we want to make sure that concurrent threads access nearby memory locations
- Peak bandwidth utilization occurs when all threads in a warp access one or more cache line(s).

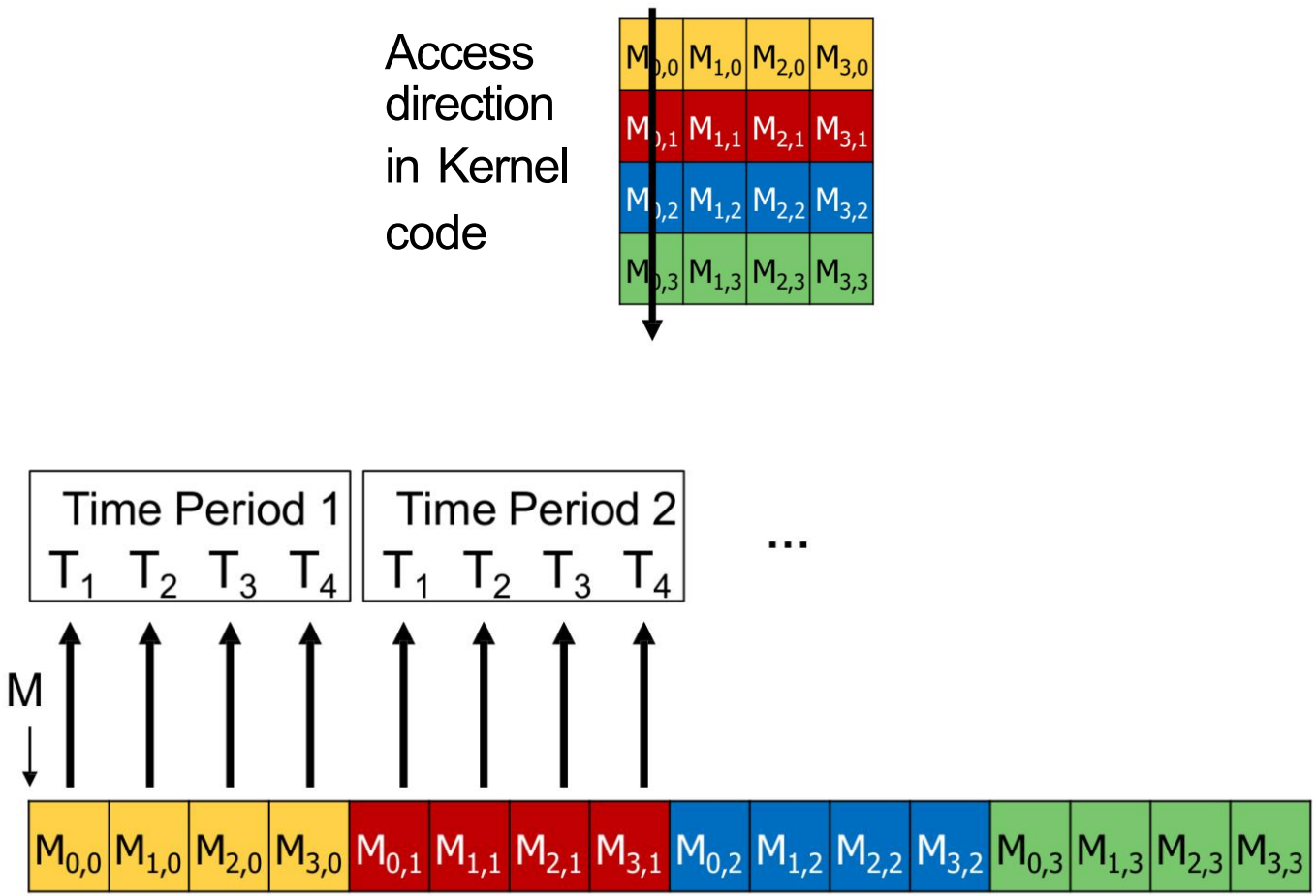


Uncoalesced Memory Accesses



adapted from UW CSEP 590, Dr. Hari Sadasivan

Coalesced Memory Accesses



adapted from UW CSEP 590, Dr. Hari Sadasivan

Thread Mapping

- **GPU thread id ⟺ data item**
  - **Proper mappings can align with hardware and provide large performance benefits**
  - **Improper mappings can be disastrous to performance**

Thread Mapping: Matrix Mult.

- Consider a serial matrix multiplication algorithm

```
for (i1=0; i1 < M; i1++)
    for (i2=0; i2< N; i2++)
        for (i3=0; i3 < P; i3++)
            C[i1][i2] += A[i1][i3]*B[i3][i2];
```

- This algorithm is suited for output data decomposition
  - We will create *NM* threads
    - Effectively removing the outer two loops
  - Each thread will perform *P* calculations
    - The inner loop will remain as part of the kernel
- Should the index space be M×N or N×M?

Thread Mapping: Matrix Mult.

- **Thread mapping 1: with an M×N index space, the kernel would be:**

```
int tx = get_global_id(0);
int ty = get_global_id(1);
for(i3=0; i3<P; i3++)
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

- **Thread mapping 2: with an N×M index space, the kernel would be:**

```
int tx = get_global_id (0);
int ty = get_global_id (1);
for(i3=0; i3<P; i3++)
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

- **Both mappings produce functionally equivalent versions of the program**

Thread Mapping: Matrix Mult.

- **The discrepancy in execution times between the mappings is due to data accesses on the global memory bus**
  - **Assuming row-major data, data in a row (i.e., elements in adjacent columns) are stored sequentially in memory**
  - **To ensure *coalesced accesses*, consecutive threads in the same wavefront should be mapped to columns (the second dimension) of the matrices**
    - **This will give coalesced accesses in Matrices B and C**
    - **For Matrix A, the iterator *i3* determines the access pattern for row-major data, so thread mapping does not affect it**

Thread Mapping: Matrix Mult.

- **In mapping 1, consecutive threads (*tx*) are mapped to different rows of Matrix C, and non-consecutive threads (*ty*) are mapped to columns of Matrix B**
  - **The mapping causes inefficient memory accesses**

```
int tx = get_global_id(0);
int ty = get_global_id(1);
for(i3=0; i3<P; i3++)
    C[tx][ty] += A[tx][i3]*B[i3][ty];
```

Thread Mapping: Matrix Mult.

- **In mapping 2, consecutive threads (*tx*) are mapped to consecutive elements in Matrices B and C**
  - **Accesses to both of these matrices will be coalesced**
    - **Degree of coalescence depends on the workgroup and data sizes**

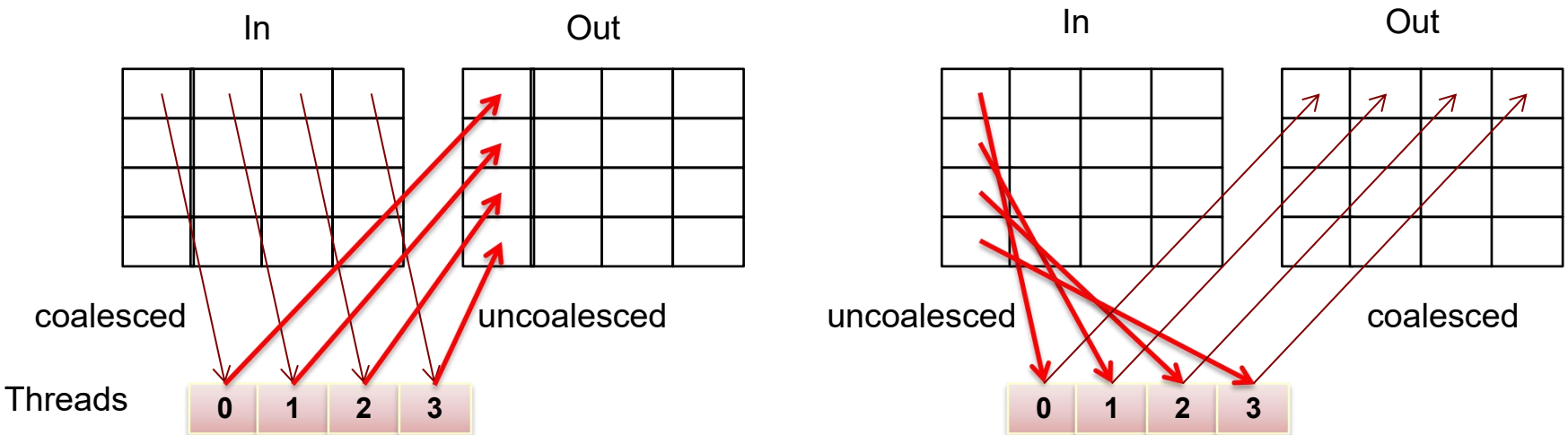
```
int tx = get_global_id (0);
int ty = get_global_id (1);
for(i3=0; i3<P; i3++)
    C[ty][tx] += A[ty][i3]*B[i3][tx];
```

Thread Mapping: Matrix Mult.

- In general, threads can be created and mapped to any data element by manipulating the values returned by the thread identifier functions
- The following matrix transpose example will show how thread IDs can be modified to achieve efficient memory accesses

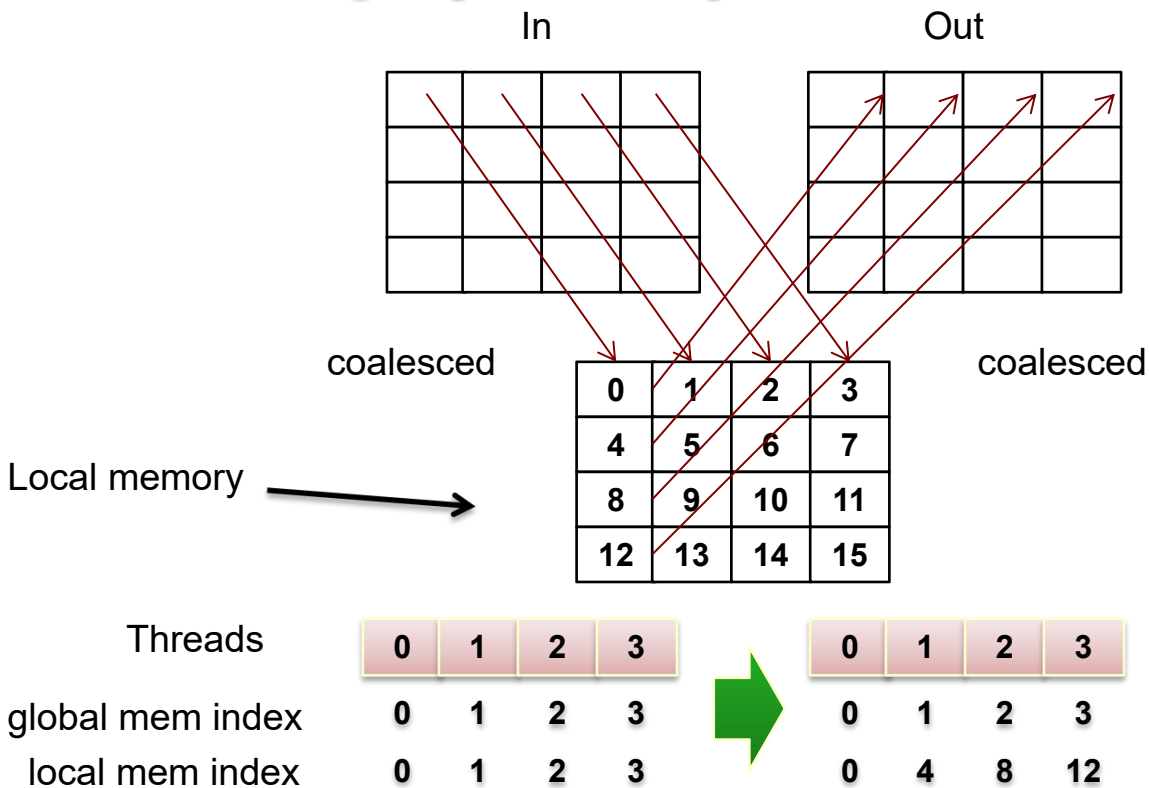
Thread Mapping: Matrix Transpose

- A matrix transpose is a straightforward technique
  - Out(x,y) = In(y,x)
- No matter which thread mapping is chosen, one operation (read/write) will produce coalesced accesses while the other (write/read) produces uncoalesced accesses
  - Note that data must be read to a temporary location (such as a register) before being written to a new location

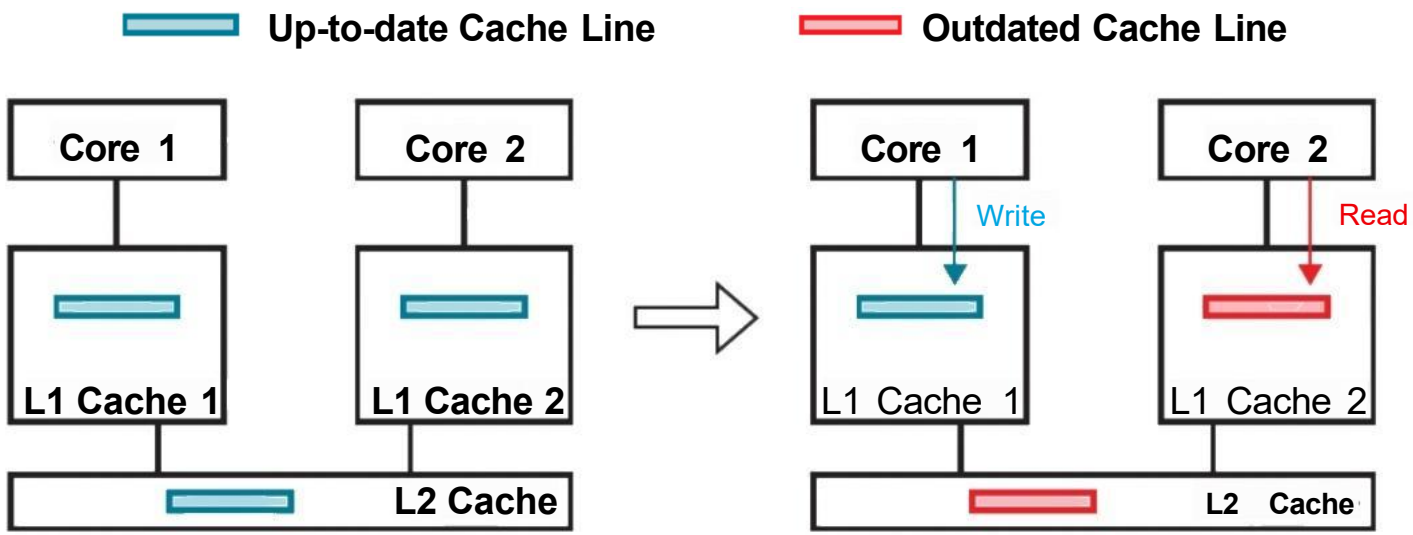


Thread Mapping: Matrix Transpose

- If local memory is used to buffer the data between reading and writing, we can rearrange the thread mapping to provide coalesced accesses in both directions
  - Note that the work group must be square

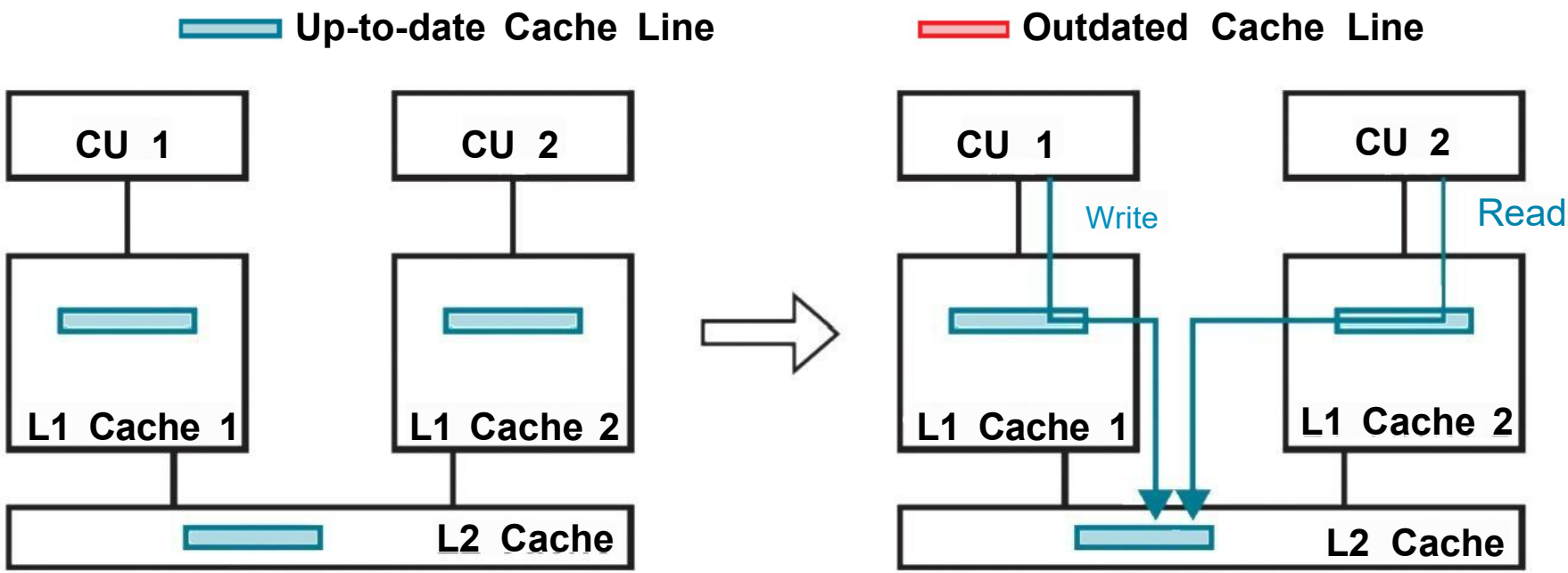


CPU needs cache coherency



GPU does not need cache coherency

- GPU does not need a coherency protocol
- What if 2 CUs write to different parts of same cache line (false sharing-to be discussed..)



adapted from UW CSEP 590, Dr. Hari Sadasivan

GPU does not need cache coherency

- Cross thread communication on global memory addresses is not general thing in GPU

- One could use

  · Global memory fences

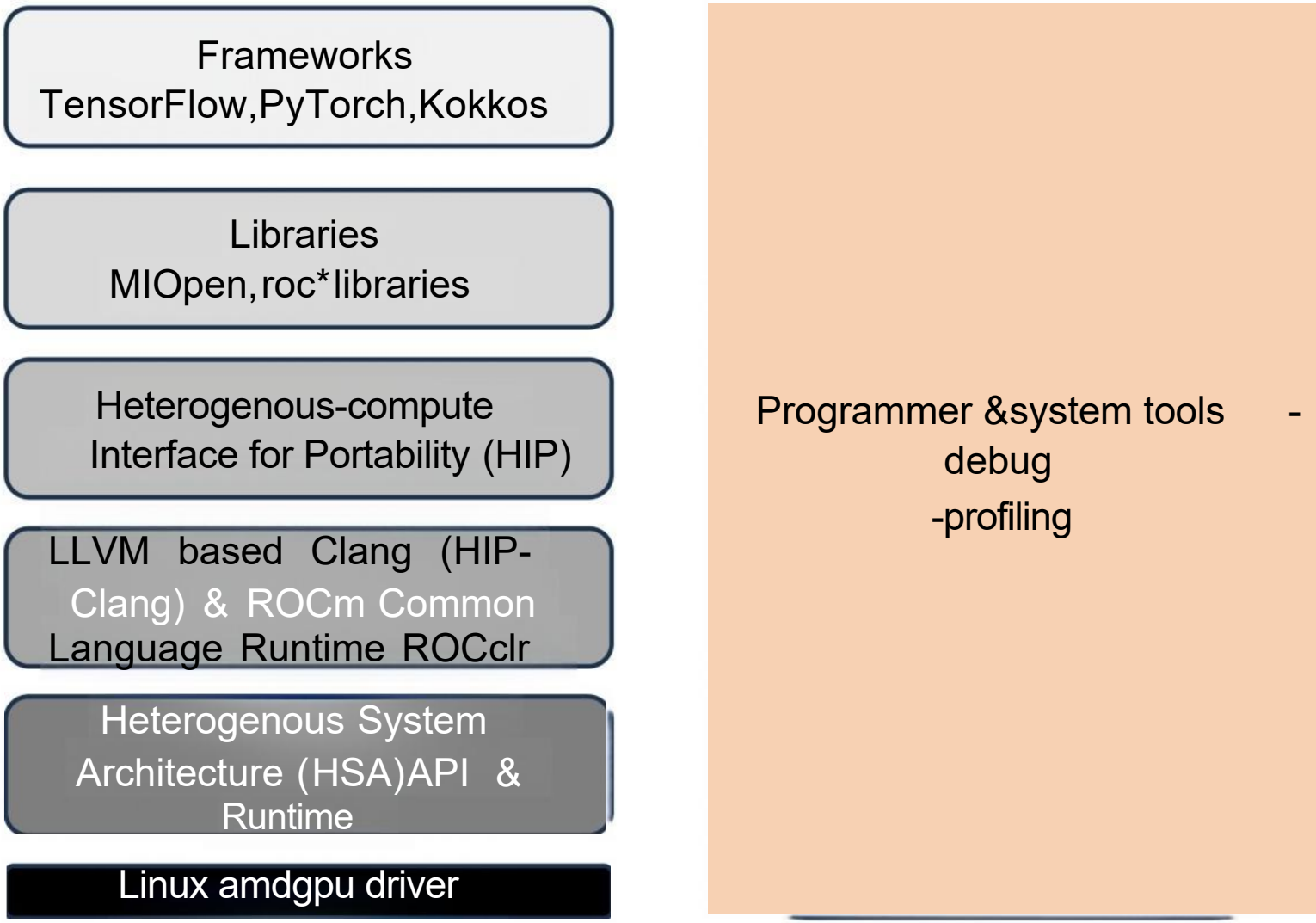
  · Atomic operations (handled at L2)

  · Separate kernels to read & write

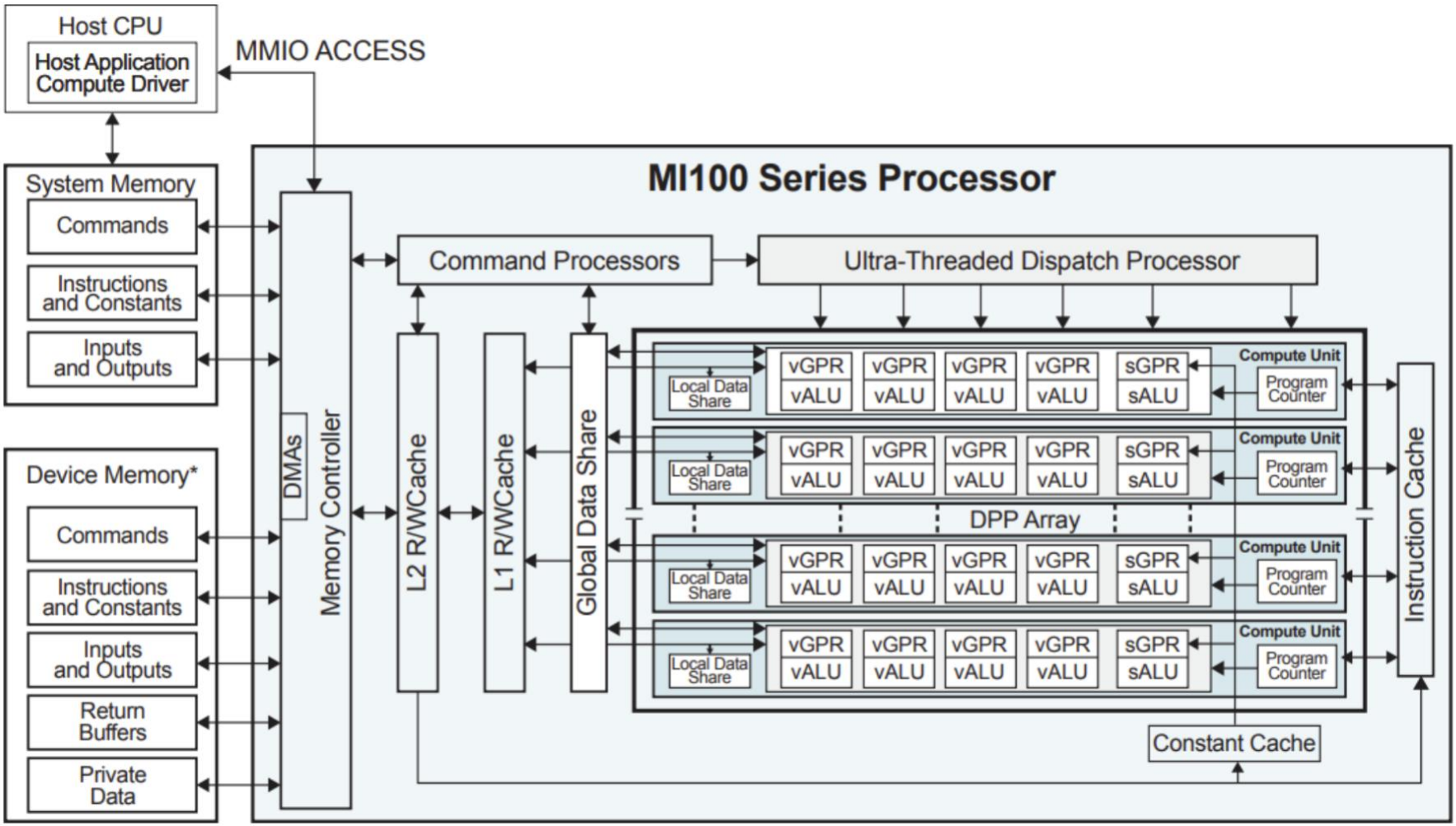
Handling false sharing

- Different CUs could write to different parts of same cache
- Without a careful design, 2nd write will overwrite first in L2
- Write-mask-based design
  - CUs & L1 send a write using bits (dirty mask-4B per cache line)
  - Allows CUs to write in parallel
  - Represents bytes to be updated in L2 cache.

Recap:ROCm software stack



MI100 micro-architecture



RQCm queue management akeme

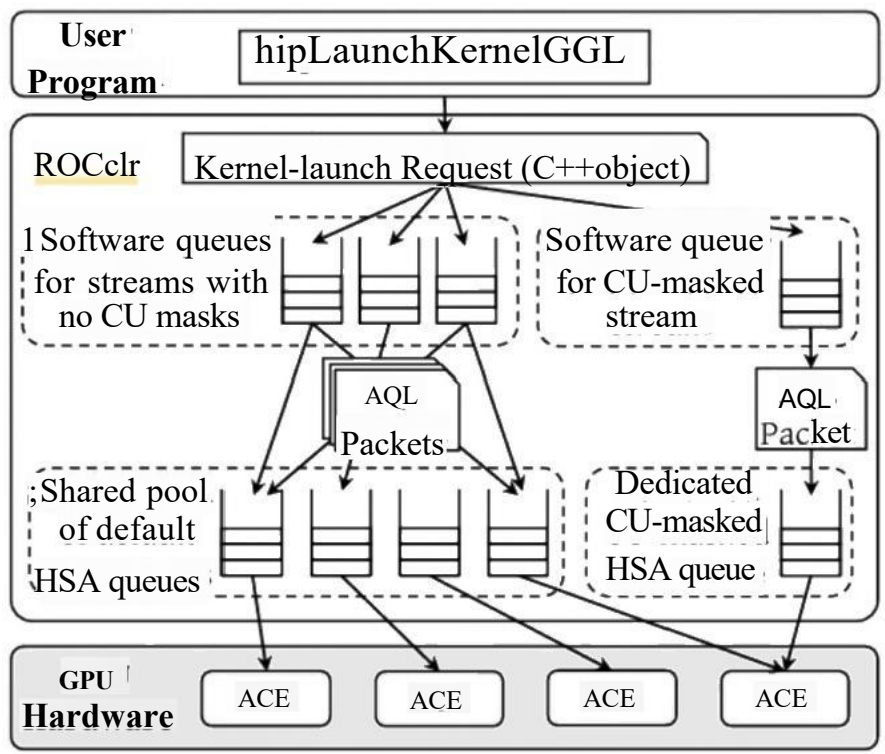
1.A user program calls the hiplaunchkernelGGL API function to launch a kernle

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 commal language)packet.

4.ROCclr inserts the AQL packet into an HSA queue.

5.In hardware,an asynchronous compute en assigning kernels to compute hardware.



HSA Queue Management

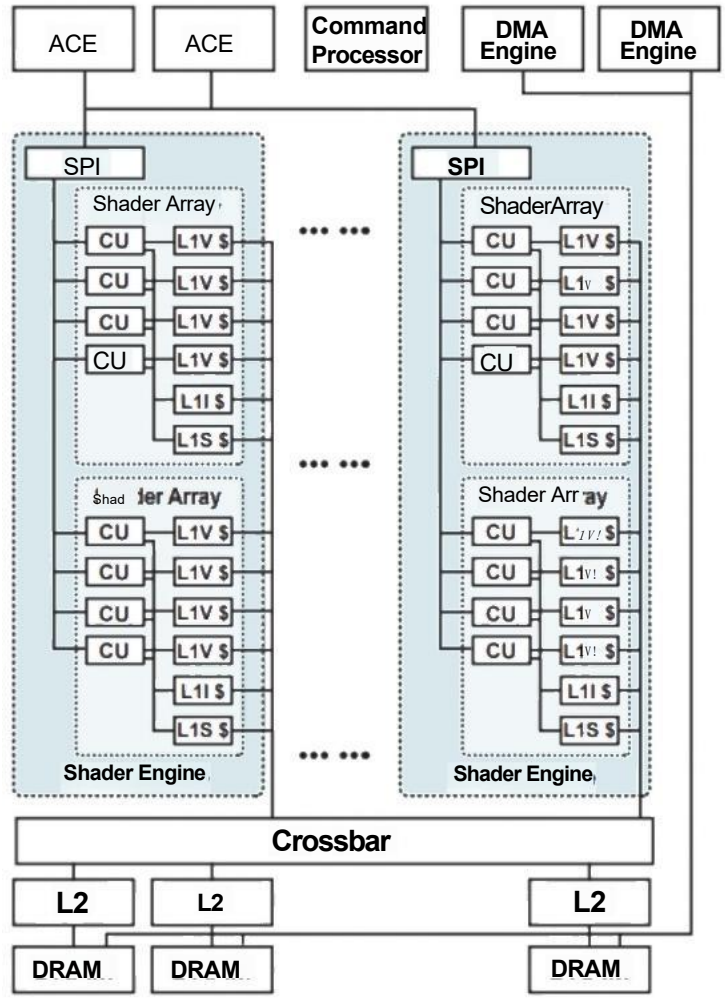
- HSA API creates & manages the memory-mapped queues & commands that interface with driver & software.
- HSA queues (4 in total)are ring buffers of AQL packets & shared directly between GPU & userspace memory
- On creation of a new HSA queue,amdgpu driversends GPU an updated runlist
  list of HSA queues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

GPU Organization

1.Controlling
- Command Processor (CP)
- Asynchronouse Compute Engine(ACE)
- Direct Memory Access (DMA)

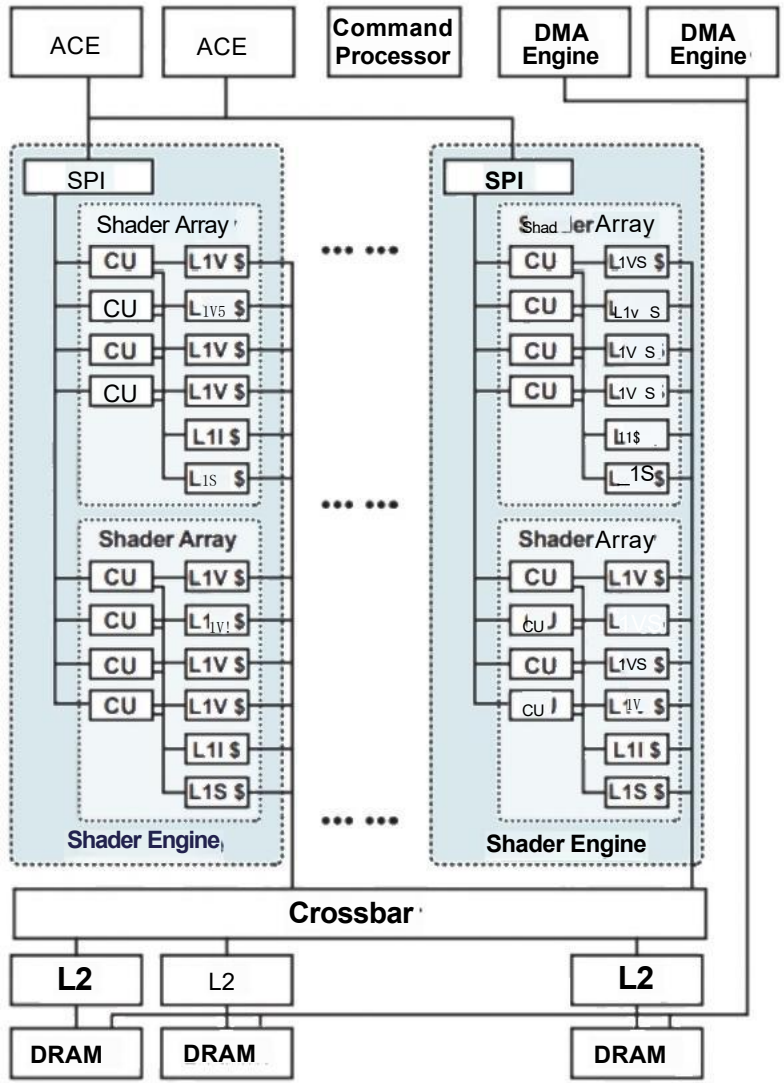
2.User-programmable shader
- Shader Engine
- Shader Arrays
- Compute Units (CU)

3.Memory
- L2 cache, Memory controller



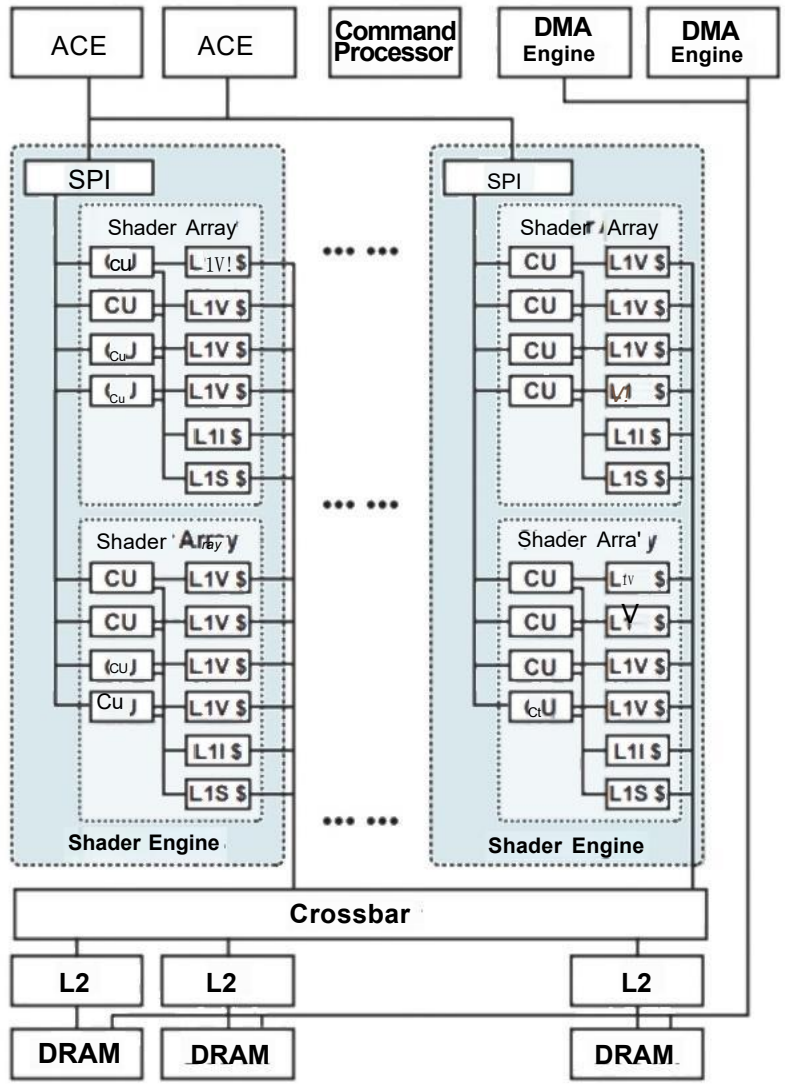
Controlling

- Command Processor (CP) recei launch commands from CPU.
- Kernel launching command is fo Asynchronous Compute Engine
- DMA engines oversees mem-co

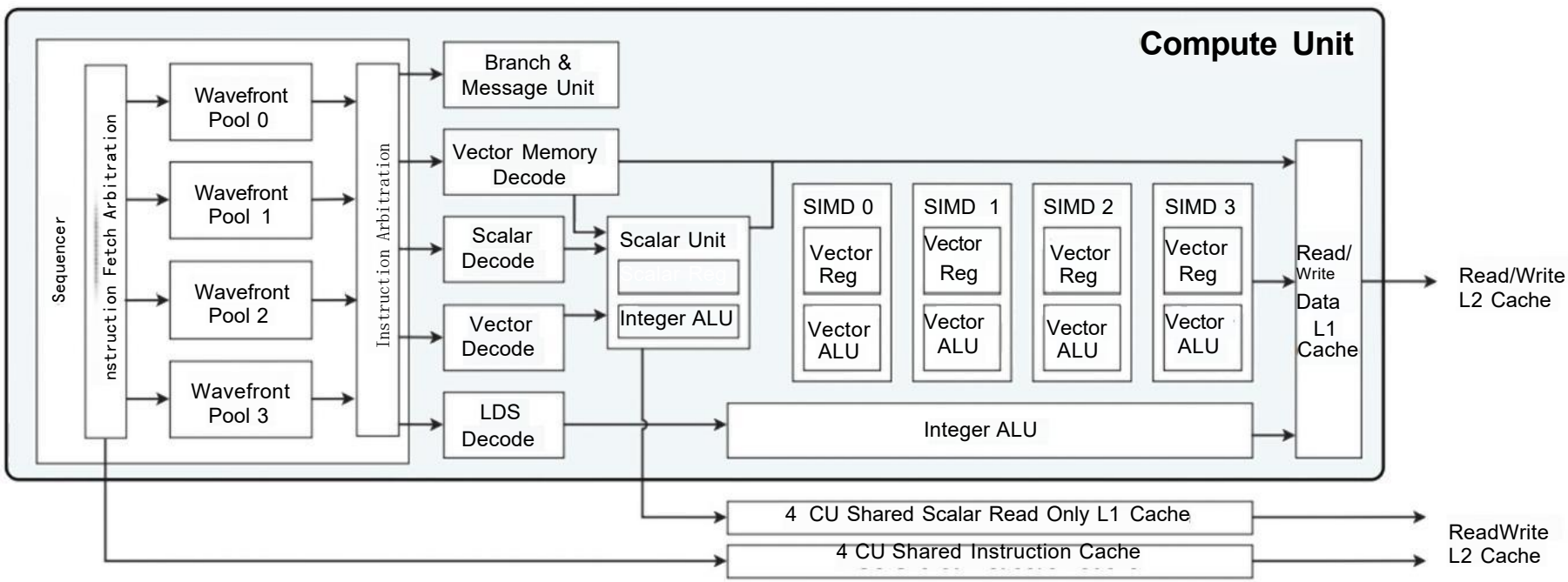


Workgroup dispatching

- Asynchronous Compute Engines launch commands.
- ACEs break down kernels into w to Shader Pipe Input (SPI)blocks
- SPI breaks down workgroups intt dispatches wavefronts to CUs &
- SPI guarantees all wavefronts in dispatched to same CU.



Sequencer

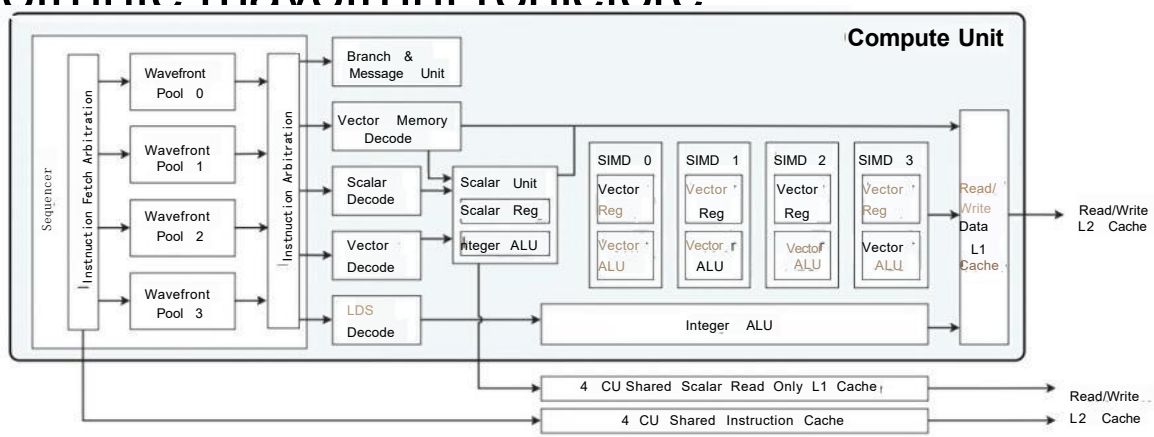

Compute Unit
Sequencer
nstruction Fetch Arbitration
Wavefront Pool 0
Wavefront Pool 1
Wavefront Pool 2
Wavefront Pool 3
Instruction Arbitration
Branch & Message Unit
Vector Memory Decode
Scalar Decode
Vector Decode
LDS Decode
Scalar Unit
Integer ALU
SIMD 0
SIMD 1
SIMD 2
SIMD 3
Vector Reg
Vector ALU
Integer ALU
Read/ Write
Data L1 Cache
Read/Write L2 Cache
4 CU Shared Scalar Read Only L1 Cache
4 CU Shared Instruction Cache
ReadWrite L2 Cache

Sequencer

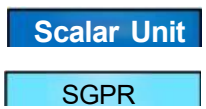
·Sequencer (SQ)issues instructions to Execution Units (EUs).

· SQ holds 4 pools of wavefront wavefront registers (including PC)&instr

·Instruction Fetching:I dispatched wavefront



The GCN Compute Unit (CU)



·The Scalar Unit (SU)

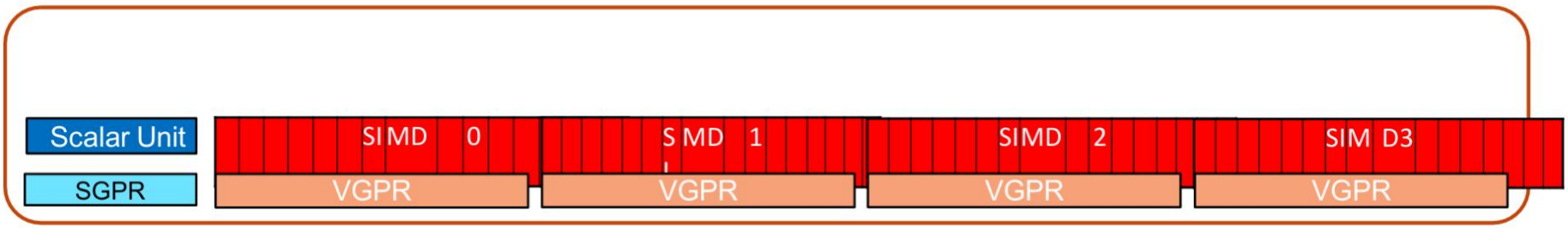
-Shared by all threads in each wavefront,accessed on a per-wavefront level

-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

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

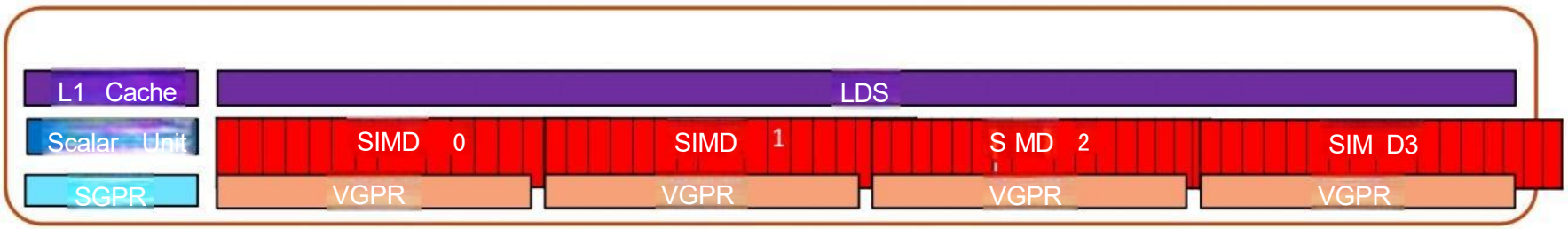
-A maximum of 256 total registers per SIMD lane -each register is 64x 4-byte entries

-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)

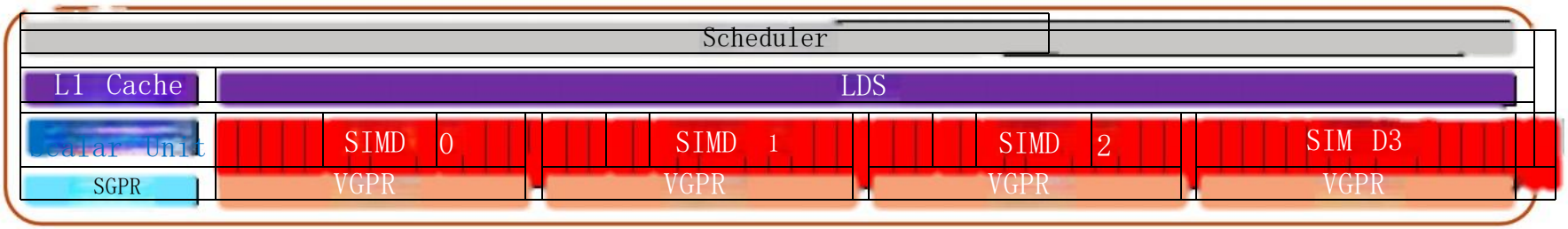
AMD ORNL training series

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

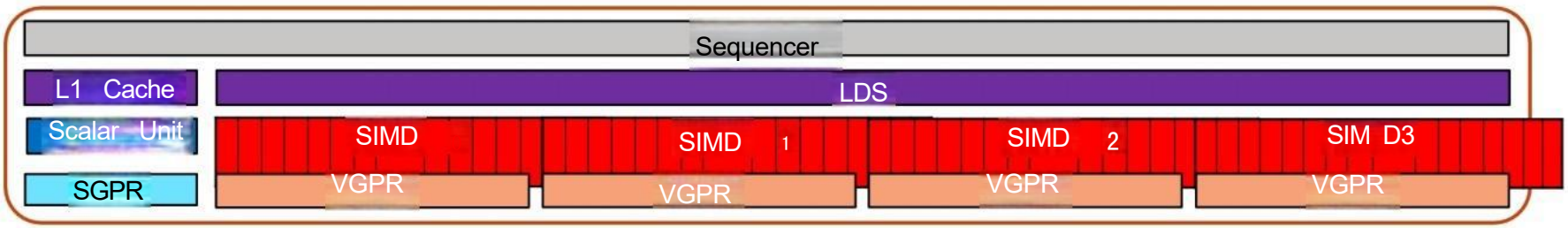
-LDS load/store

-Global mem load/store

-Special instructions (NoOps,barriers,branch instructions)

AMD ORNL training series

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 (SN ALU,SN GPR,LDS, global,branch,etc)

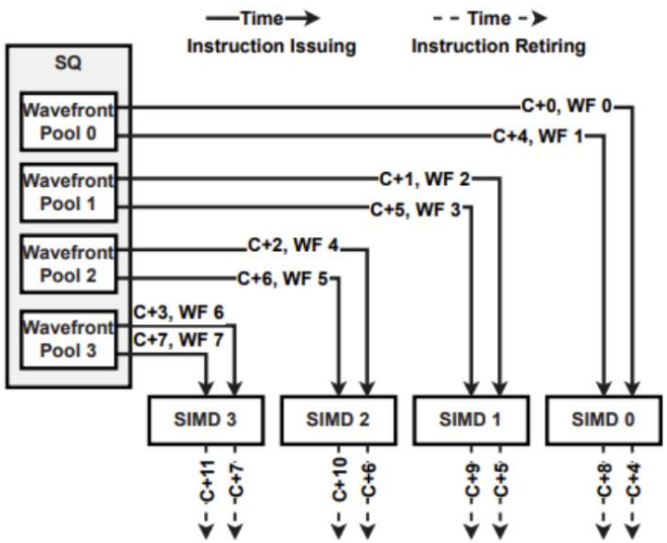
- **Maximum of 5** instructions issued to wavefronts on a single SIMD,per cycle per CU

-Some instructions take 4 or more cycles to retire (e.g.FP32VALU instruction on 1 wavefront using 16-wide SIMD)

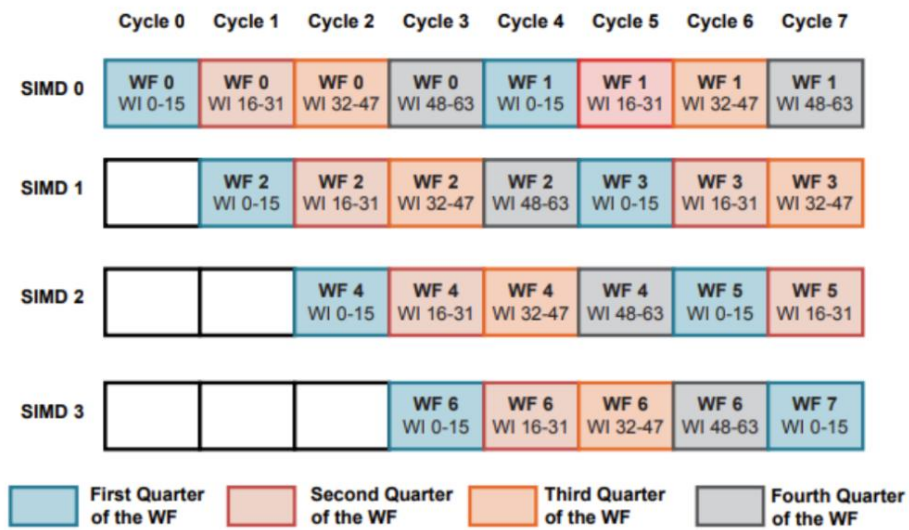
-Round robin scheduling of SIMDs hides execution latency

-Programmer can still pretend'CU operates in 64-wide SIMD

Calculating GPU throughput



The timeline of instruction issuing



The time table of wavefronts executing in the SIMD units

Few more observations on CPU vs GPU

- Common fetch & decode for multiple data elements saves chip area & energy
- In a CPU, instructions executed in a pipeline come from same thread.
  - In a GPU,instructions executed at different stages of the pipeline originate from different wavefronts.
- GPU can do zero overhead context-switching.