CS302

GPU: Control divergence, Synchronization and Memory part I

Spring 2025

Babak Falsafi, Arkaprava Basu

parsa.epfl.ch/course-info/cs302

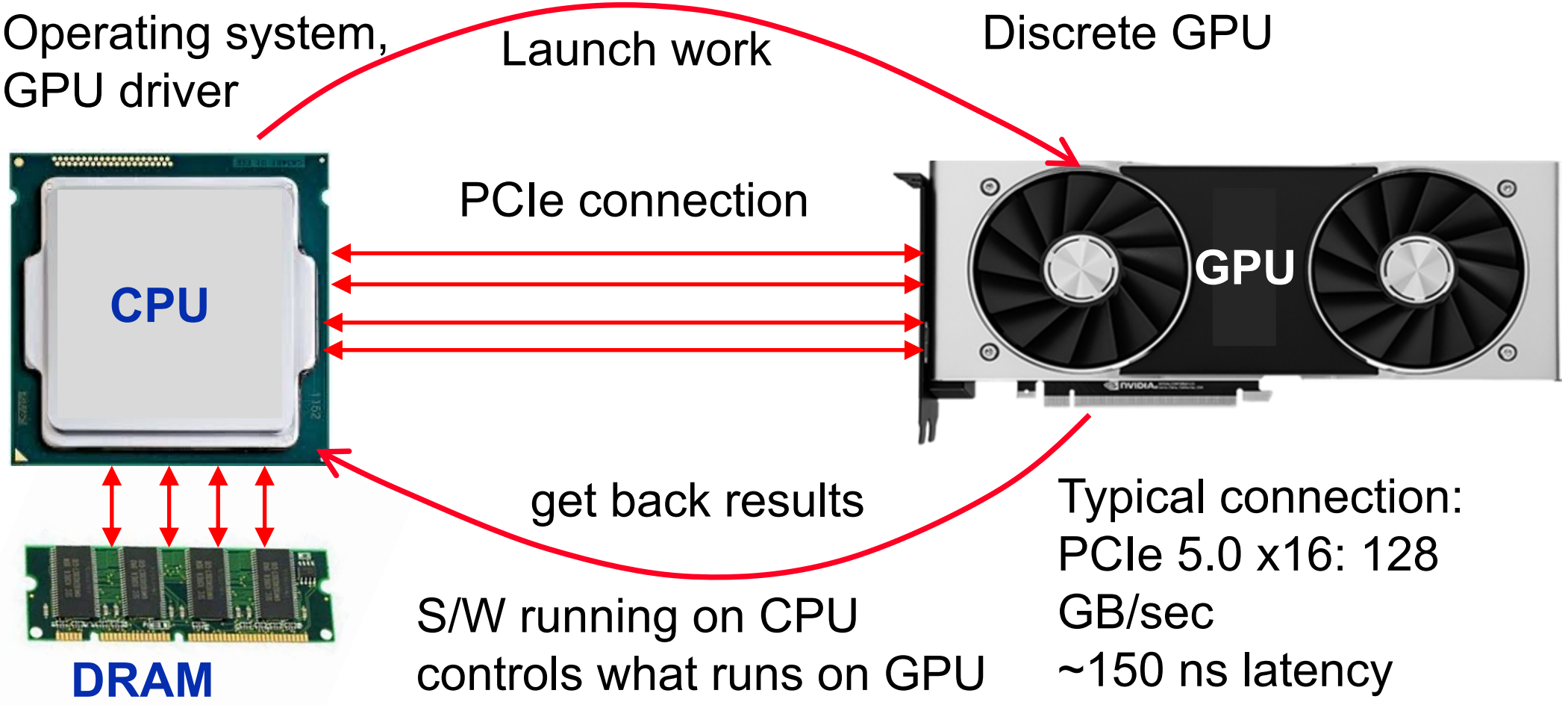
Some of the slides are from Derek R Hower, Adwait Jog, Wen-Mei Hwu, Steve Lumetta, Babak Falsafi, Andreas Moshovos, and from the companion material of the book “Programming Massively Parallel Processors”

Where are We?

M	T	W	T	F
17-Feb	18-Feb	19-Feb	20-Feb	21-Feb
24-Feb	25-Feb	26-Feb	27-Feb	28-Feb
3-Mar	4-Mar	5-Mar	6-Mar	7-Mar
10-Mar	11-Mar	12-Mar	13-Mar	14-Mar
17-Mar	18-Mar	19-Mar	20-Mar	21-Mar
24-Mar	25-Mar	26-Mar	27-Mar	28-Mar
31-Mar	1-Apr	2-Apr	3-Apr	4-Apr
7-Apr	8-Apr	9-Apr	10-Apr	11-Apr
14-Apr	15-Apr	16-Apr	17-Apr	18-Apr
21-Apr	22-Apr	23-Apr	24-Apr	25-Apr
28-Apr	29-Apr	30-Apr	1-May	2-May
5-May	6-May		8-May	9-May
12-May	13-May	14-May	15-May	16-May
19-May	20-May	21-May	22-May	23-May
26-May	27-May	28-May	29-May	30-May

- This lecture
  - Control flow divergence
  - Basic synchronization
  - GPU memory part I
- Exercise session:
  - GPU demo programming
  - Using the Izar cluster
- Next week:
  - Memory optimizations on GPUs

Review: GPU is a Co-Processor



Typical connection:
PCIe 5.0 x16: 128 GB/sec
~150 ns latency

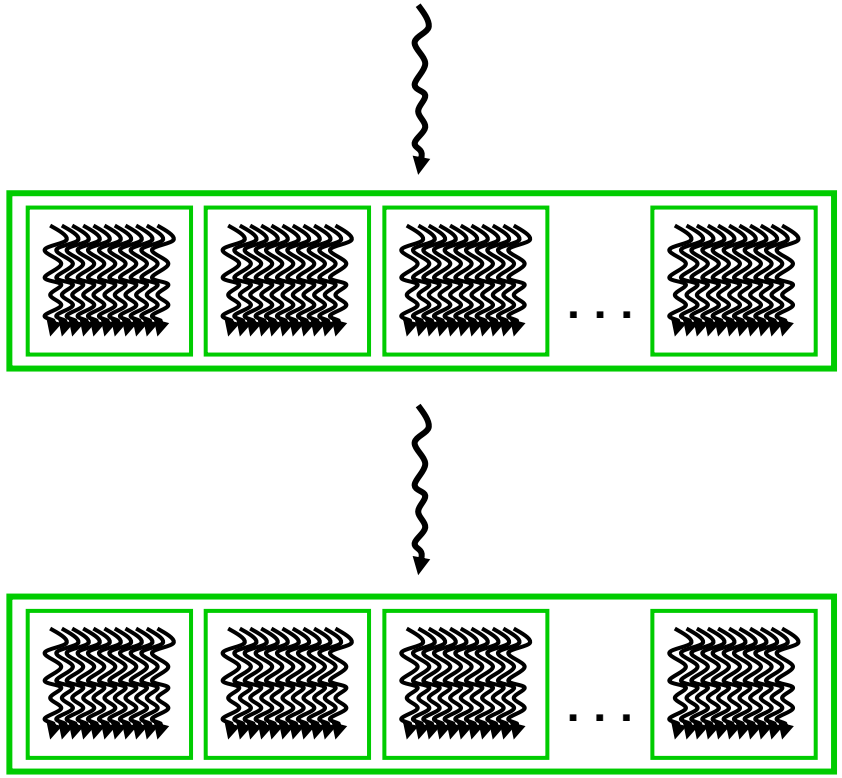
S/W running on CPU controls what runs on GPU

Review: Launching Kernel and Thread Grids

Serial Code (host/CPU)

Data parallel function (device/GPU)

KernelA<<< nBlk, nTid >>>(args);



Serial Code (host/CPU)

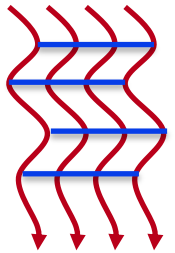
Data parallel function (device/GPU)

KernelB<<< nBlk, nTid >>>(args);

Kernels written as SPMD programs

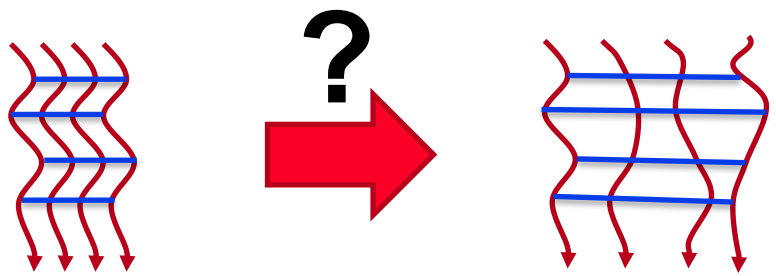
Beware of Pitfalls of GPU's SIMT Execution

- Recall: SIMT → One PC multiple thread operating on multiple data
  - Benefits → Fetch and decode one instruction, multiple concurrent execution



- But what if the code (kernel/device) code has a conditional branch?

```
if (x <= 0)
      y = 0;
else
      y = x;
```

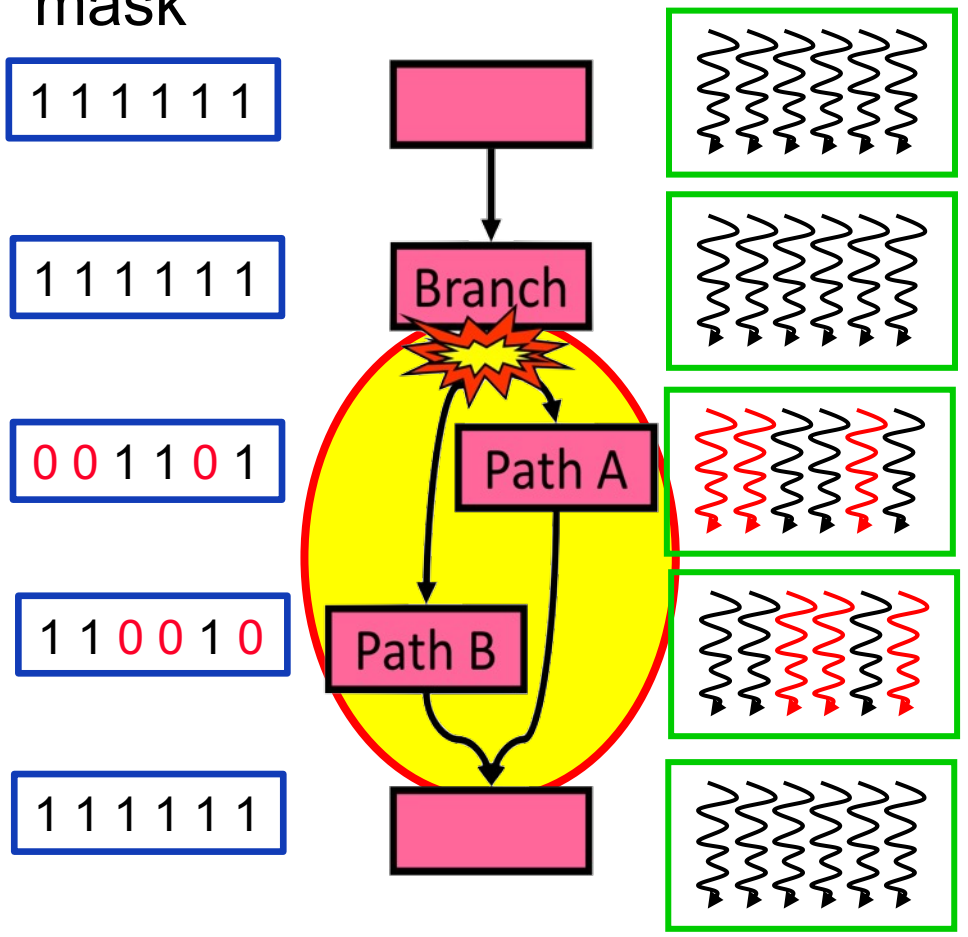


A Pitfall of GPU's SIMT Execution: Control Flow Divergence

- Threads in warp run in lockstep
  - Observation*: **Not** all have to commit*
- Support conditional branch through **predication**
  - All threads execute, but some *discard* their result (wasted compute)
  - A predication mask determines whose results to keep (discard)

Performance Loss due to Control-Flow Divergence

Predication mask



50% Performance Loss

- Execute one path at a time
  - Diverging threads will be disabled
  - Limits parallelism → lower performance

Control flow divergence

Programs with many conditional branches are not well-suited for GPU's SIMT execution

Limiting Control-Flow Divergence

- Observation: SIMT (lock-step) execution **only** within a warp
  - It's okay if **different** warps take different paths

```
if (threadIdx.x > 2) {. . .}
else {.   .   .}
```

- Two different control paths for threads in a warp
- Avoiding control flow divergence:

```
if (threadIdx.x / WARP_SIZE > 2) {. . .}
else {.   .   . }
```

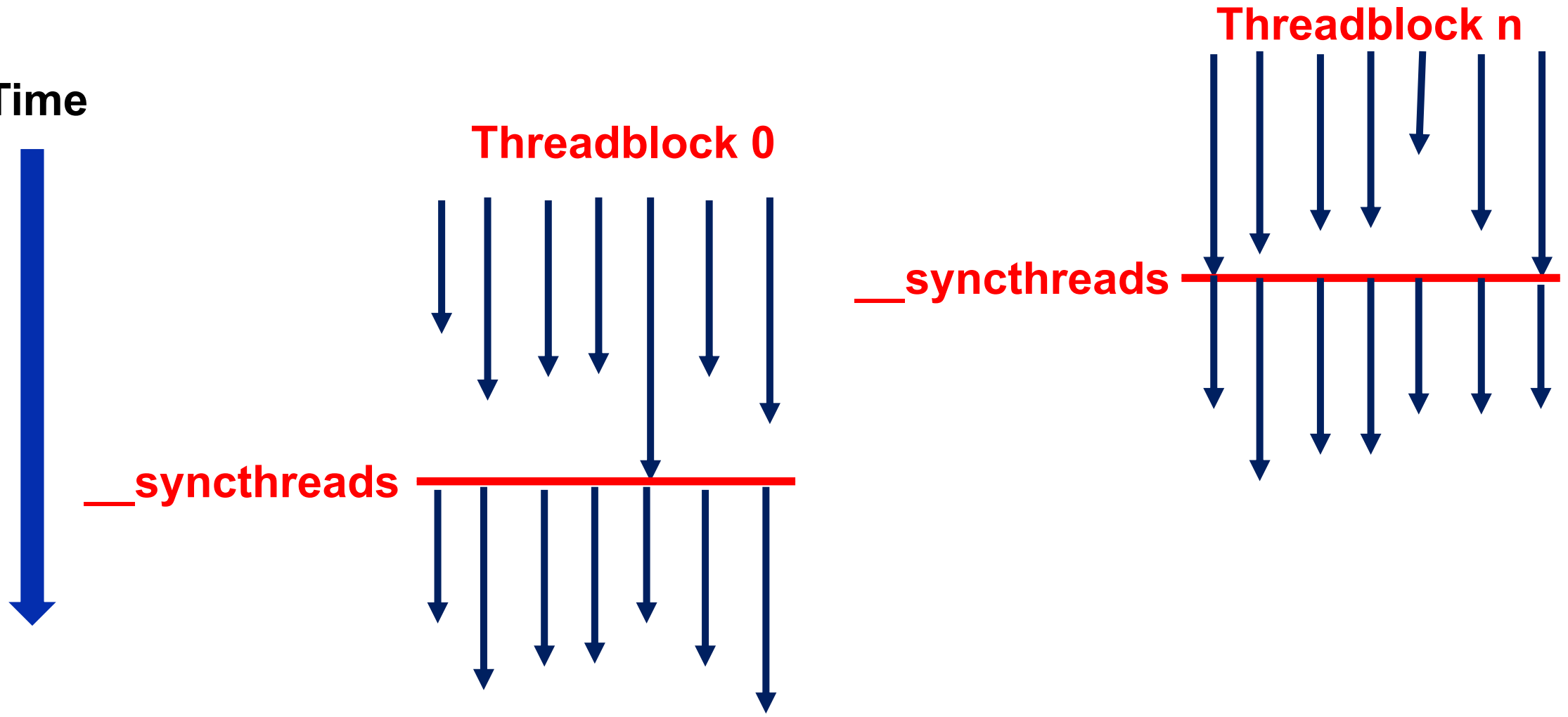
  - Two different control paths, **but** different warps
    - All threads in a given warp follow the same path
    - Divergence across warps is OK!

BASIC SYNCHRONIZATION

Barrier Synchronization in CUDA

- Commonly used synchronization: Threadblock barrier (_ _*syncthreads*)
- All threads of a given threadblock must reach the barrier before any of them can proceed past the barrier
- Threads from different threadblocks are not synchronized
  - **NOT** a global synchronization across all threads of a threadblock

Barrier Synchronization in CUDA



Barrier Synchronization in CUDA

- Commonly used synchronization: Threadblock barrier (_ _*syncthreads*)
- All threads of a given threadblock must reach the barrier before any of them can proceed past the barrier
- Threads from different threadblocks are not synchronized
  - **NOT** a global synchronization across all threads of a threadblock
- Suitable for bulk synchronous programs
  - Threads execute concurrently most of the time
  - Infrequently synchronize many threads together at certain points in the program

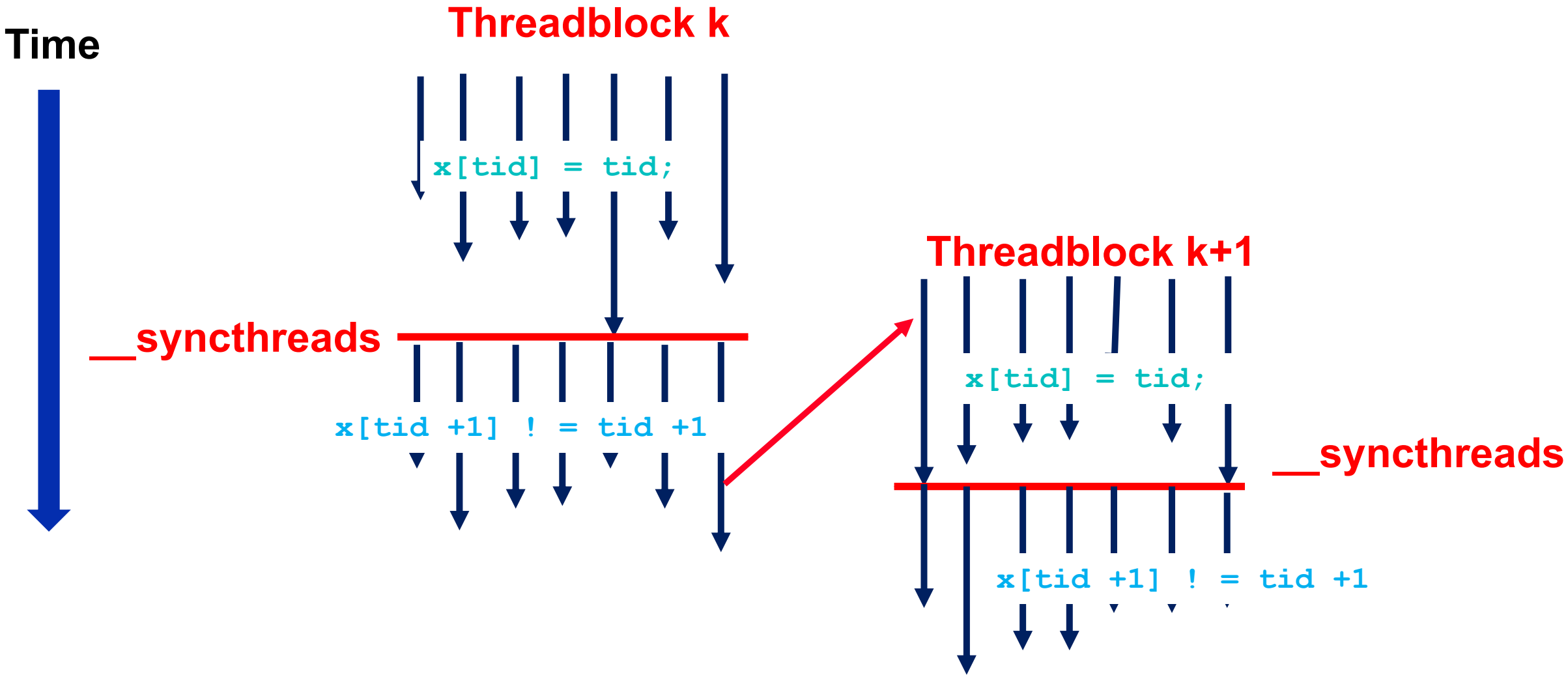
Barrier Synchronization in CUDA

- A threadblock barrier = execution barrier + a memory barrier
- Execution barrier: All threads of a threadblock reach the barrier before any of them can proceed forward
- Memory barrier: A write by any thread before the barrier is guaranteed to be visible to all threads in the threadblock after the barrier

When do you expect the statement be printed?

```
__global void test_syncthread(unsigned* x, int N) {
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    x[tid] = tid;
    __syncthreads();
    if (tid < N-1 && x[tid +1] ! = tid +1 ) {
      printf("syncthreads not enough\n");
    }
}
```

Barrier Synchronization in CUDA



Barrier Synchronization in CUDA

- A threadblock barrier = execution barrier + a memory barrier
- Execution barrier: All threads of a threadblock reach the barrier before any of them can proceed forward
- Memory barrier: A write by any thread before the barrier is guaranteed to be visible to all threads in the threadblock after the barrier

When do you expect the statement be printed?

Ans: When threads tid and tid+1 belongs to different threadblocks

```
__global void test_syncthread(unsigned* x, int N) {
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    x[tid] = tid;
    __syncthreads();
    if (tid < N-1 && x[tid +1] ! = tid +1 ) {
      printf("syncthreads not enough\n");
    }
}
```

Interactions Between Barriers and Conditionals

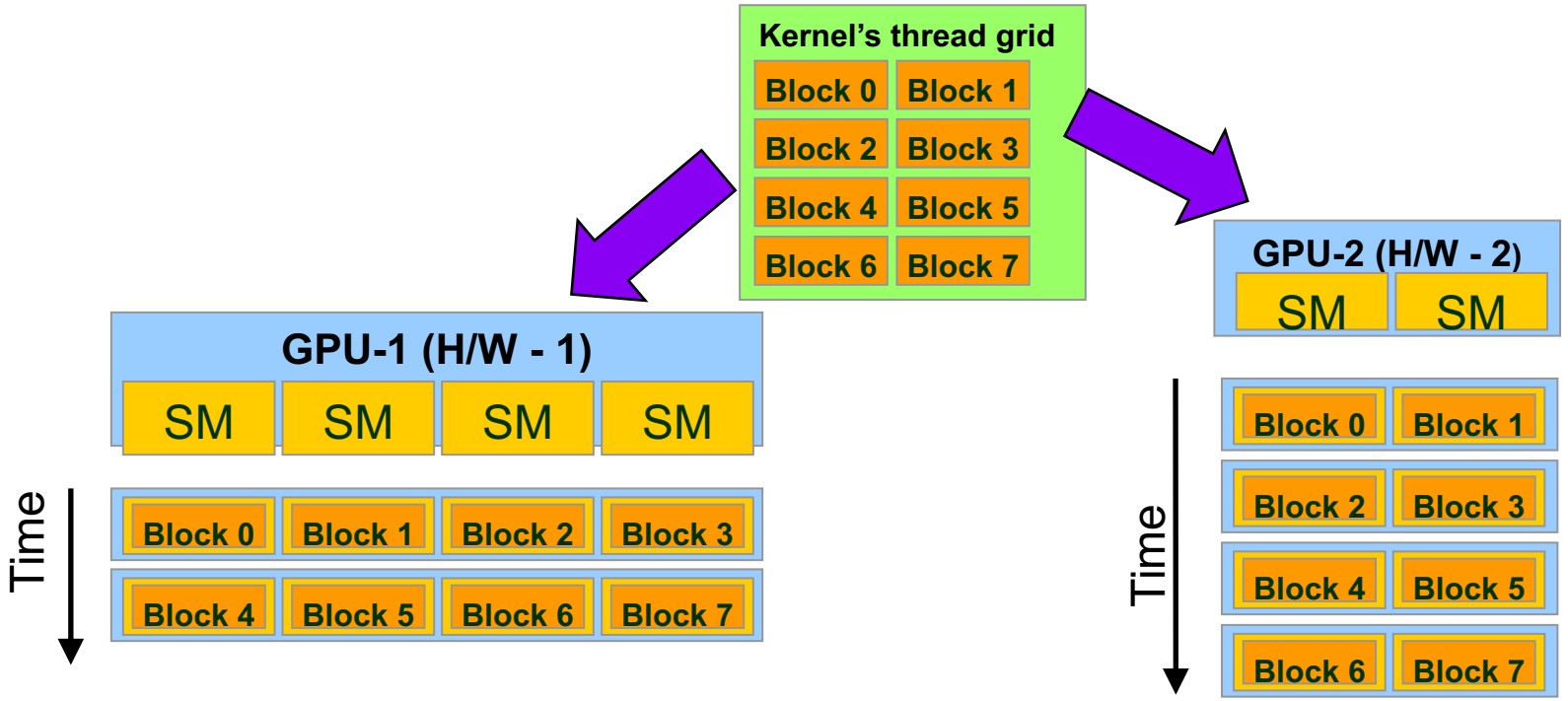
```
__global void test_syncthread(unsigned* x, int N) {
    ……
    if(threadIdx % 2 == 0) {
        ……
        __syncthreads();
    }
    else {
        ……
        __syncthreads();
    }
}
```

Distinct barriers – not all threads of a threadblock will execute the same barrier → Possible deadlock

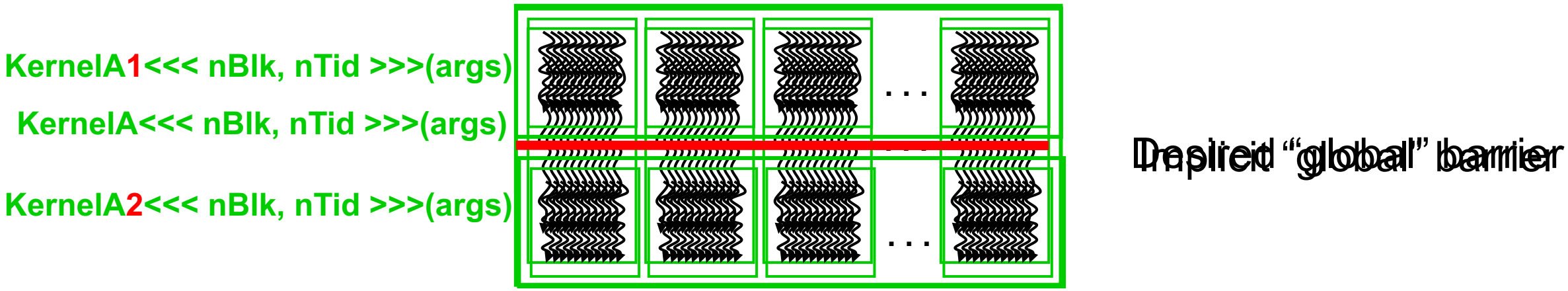
- What do you think can happen to this code?
- Be careful when using barriers within conditional statements
  - Works only if all threads of a threadblock are guaranteed to either hit the barrier or none would

Synchronizing across All Threads of a Kernel?

- No explicit *global* barrier across all threads of a kernel
- **Not** all thread blocks of a kernel may execute simultaneously
  - No specific order of execution amongst the thread blocks
  - Enforcing a barrier can deadlock if only *some* thread blocks are running



Kernel Decomposition for Global Synchronization

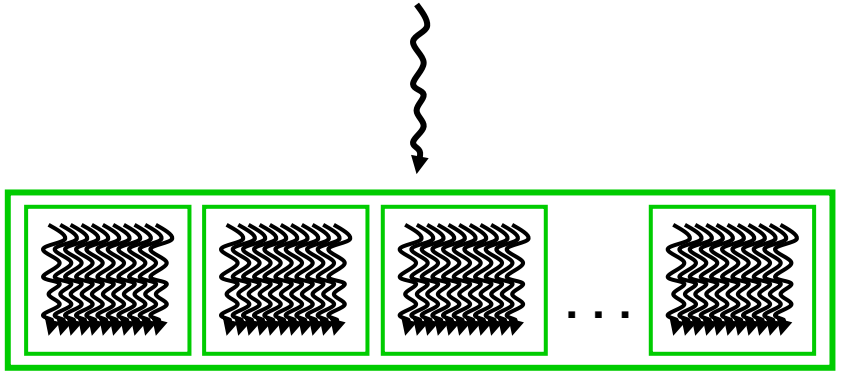


- Implicit "global" barrier at each kernel boundary
  - When a kernel completes, all threads have completed their operations

Kernel Launch is Asynchronous

Serial Code (host/CPU)

KernelA<<< nBlk, nTid >>>(args);

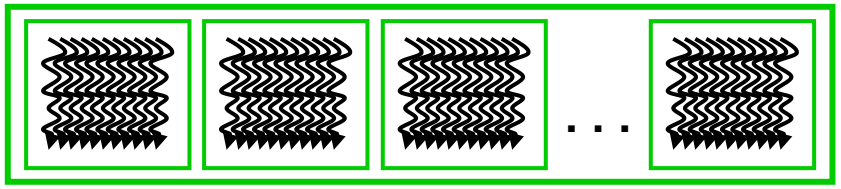


Would not wait for the kernel to finish

Serial Code (host/CPU)

cudaDeviceSynchronize to wait for a kernel to finish before the host code proceeds

KernelB<<< nBlk, nTid >>>(args);



```
cudaMemcpy((void*)host_array, (void*)gpu_array,
           NBYTES, cudaMemcpyDeviceToHost);
```

cudaMemcpy is blocking --- no need for cudaDeviceSynchronize

Note: KernelB will only start executing after KernelA finishes execution even without cudaDeviceSynchronize

GPU MEMORY SYSTEM

Overview: CPU vs. GPU memory system

CPU

- Limited number of registers
  - OS saves/restores registers
- One virtual address space (per process)
- Large caches for latency hiding

GPU

- Many registers
  - H/W saves/restores registers
- Many different address spaces
  - Global memory
    - Caches
  - Local memory
  - Constant memory
  - Shared memory (scratchpad)
- Small caches – for b/w filtering

Many Registers for Fast Context Switch

	CPU	GPU
Registers/thread	16-32	32 (up to 256)
Num physical registers/per "core" (or SM)	~150-256	**~64K**

Registers provide the fastest access

- 1 Cycle latency
- 100s of TBs of bandwidth
- Per-thread variables declared in kernels are kept in registers

Large register file key to latency hiding

- Keep register state of many warps (threads) ready
- "Zero cycle" H/W context switch when a warp hits long latency event

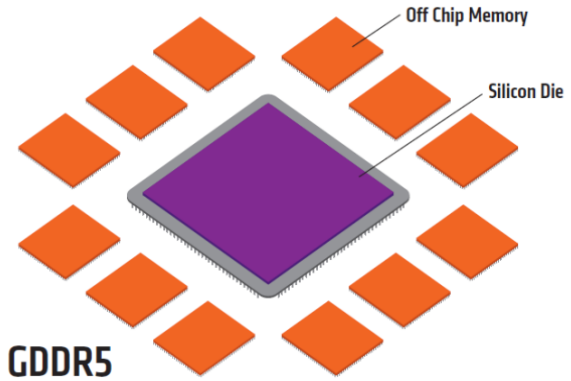
Allocated on register for each thread

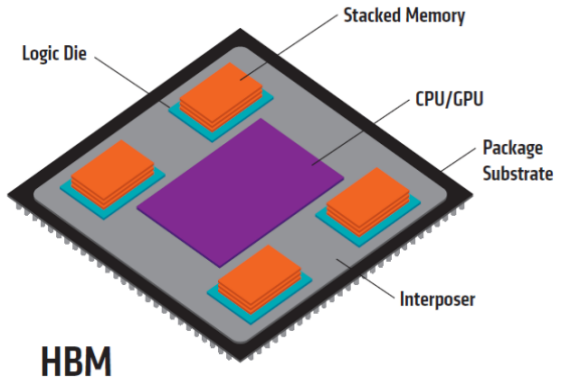
```
__global__ void vecadd_kernel(float* x, float* y, float* z, int N) {
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    z[i] = x[i] + y[i];
}
```

Global Memory of GPU

- Off chip memory on the GPU board
- Goal is to provide high bandwidth – H100 supports 3.5 TB/sec
- Latency is a lessor concern
- But, low capacity (typically, in 10s of GBs, e.g., 80 GB)

GDDR (Graphics DDR):
Lower bandwidth
Cheaper
Less power-efficient





HBM (High Bandwidth Memory):
Higher bandwidth
Costlier
Power-efficient

Global Memory of GPU

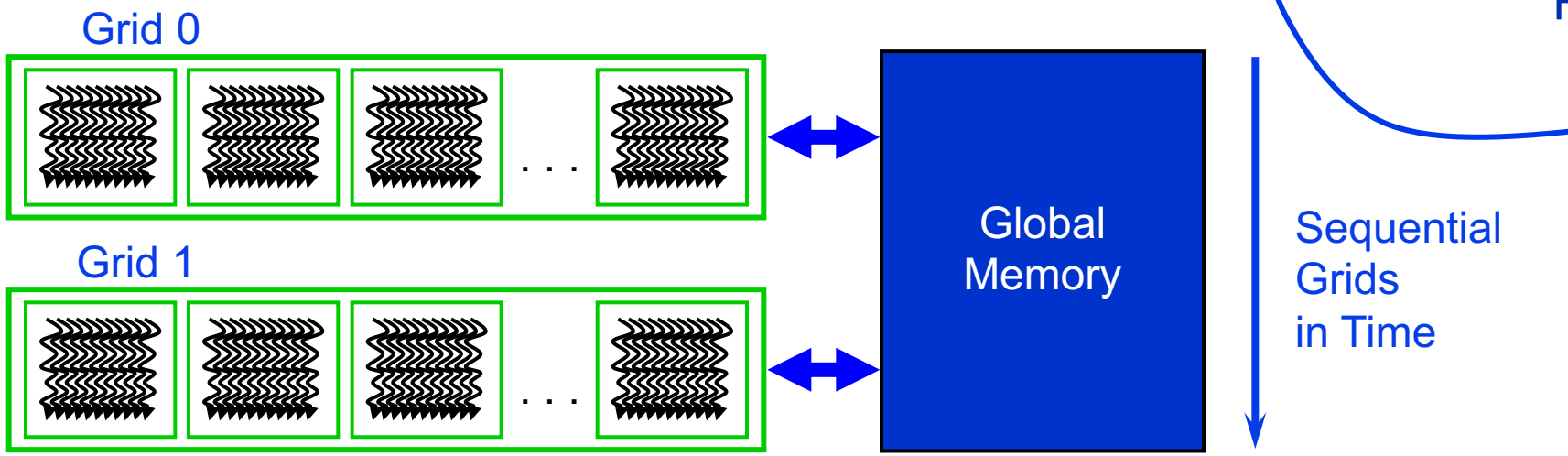
- Global memory (per application):
  - Accessible to all threads in a grid
  - Remains live (unless freed) across kernel launched in an application
  - Enables inter-grid communication

```
void vecadd(float* x, float* y, float* z,
      int N) {
   // Allocate GPU memory
   float *x_d, *y_d, *z_d;

cudaMalloc((void**) &x_d, N*sizeof(float));
N*sizeof(float));
```

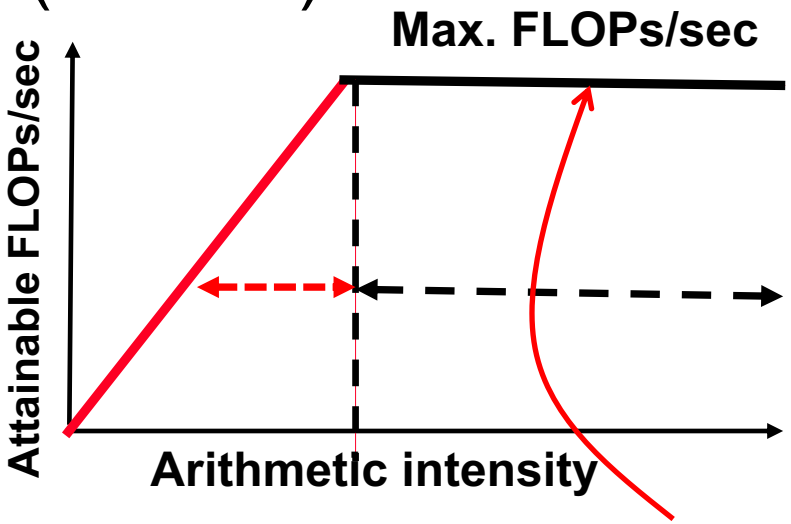
Host code

Allocates global GPU memory



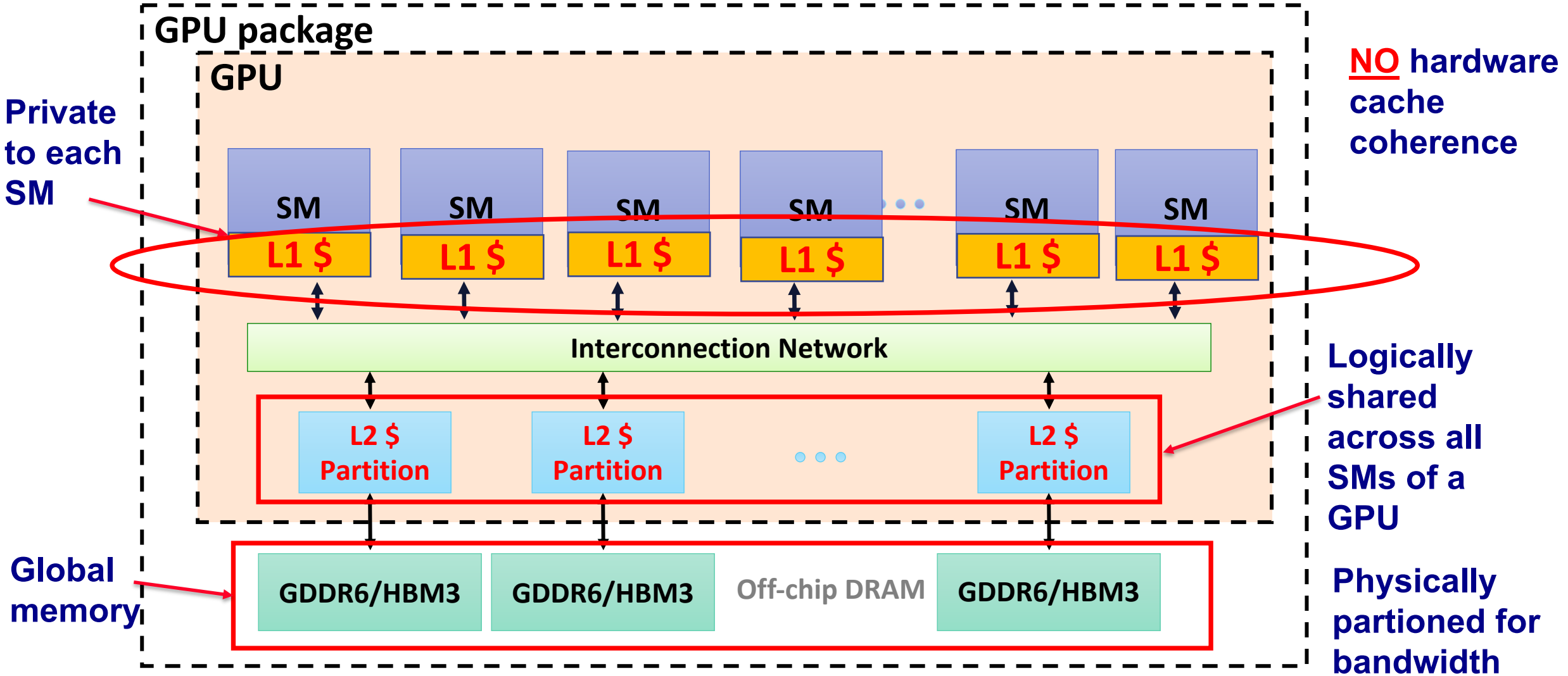
Bridging the Gap between Compute and (Global) Memory B/W

- GPUs have plenty of compute (FLOPS)
  - Max FLOPS of H100 GPU: 51 Teraflops (FP32)
- (Global) Memory bandwidth is relatively limited (GB/sec)
  - Max Mem. Bandwidth of H100 GPU: 2TB/sec
- Remember the roofline model?
  - Needs very high arithmetic intensity to achieve the FLOPs
  - How much airth. Intensity needed to leverage full TFLOPS?
    - ▲Ans: 25.5
- GPU memory hierarchy built to help bridge this gap between FLOPS and GB/sec
  - Caching as a bandwidth filter
  - Memory access coalescing
  - Specialized memory



Want to be here. But needs high arithmetic intensity

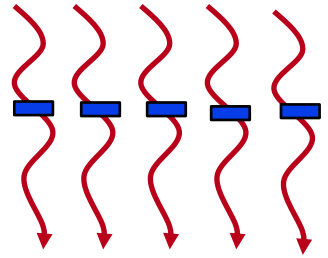
GPU Cache Hierarchy



GPU vs CPU Cache Hierarchy

Typical CPU vs GPU caches	CPU	GPU
L1 data cache capacity	32KB	16 KB
Active threads sharing L1 D Cache	2 (2-way SMT)	2048
L1 D Cache capacity / thread	**16KB**	**8 bytes**
Last level cache (LLC) capacity	8MB	1MB
Active threads sharing LLC	8	163,840
LLC capacity / thread (4 cores)	**1MB**	**6.4 bytes**
Cache line size	64B	128B

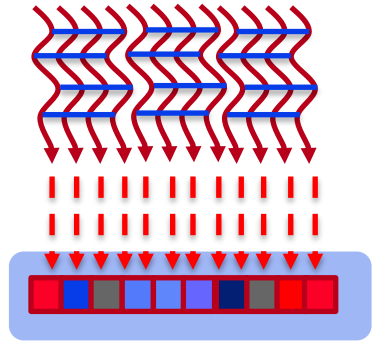
The primary purpose of the GPU cache hierarchy is to act as a bandwidth filter.



In comparison, the purpose of CPU caches is latency hiding.

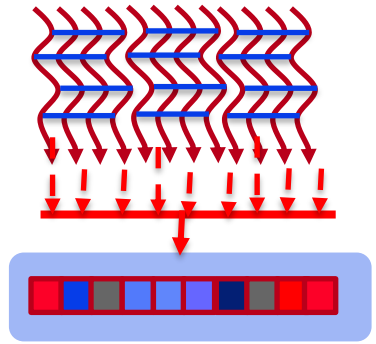
Memory Access Coalescing

- Threads in a warp executing in lock-step often *simultaneously* access different words within the same cache line
- GPU cache line size 128 bytes
- If all 32 threads of warp access contiguous 4 bytes then they map to the same cache line



Memory Access Coalescing

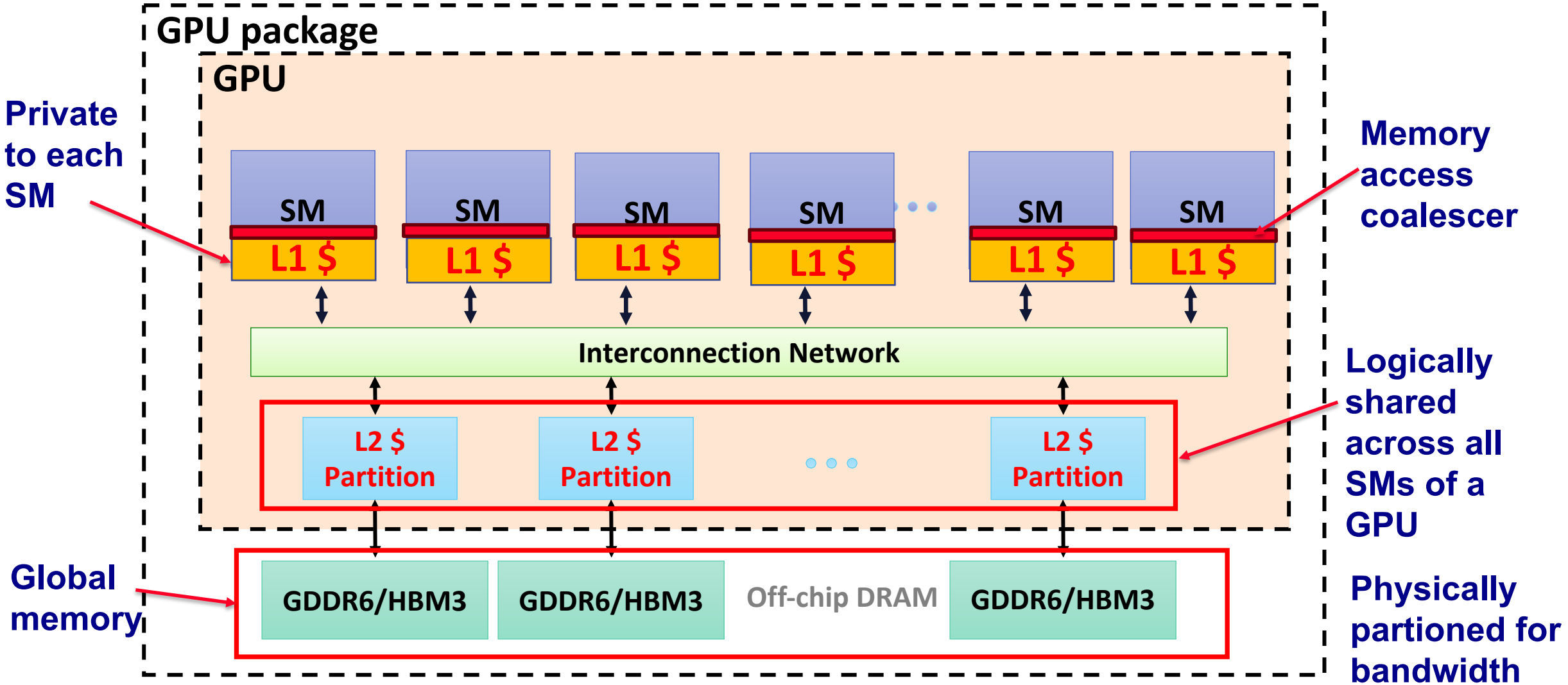
- Threads in a warp executing in lock-step often *simultaneously* access different words within the same cache line



- GPU cache line size 128 bytes
- If all 32 threads of warp access contiguous 4 bytes, they map to the same cache line

- **Hardware coalescer:** Zap accesses from threads of a warp to a single cache access
  - Up to 32X reduction in the number of cache accesses/memory accesses!

GPU Memory Access Coalescer



Opportunities for Memory Access Coalescing

- Whether memory accesses can be coalesced *depends on software*
  - e.g., how the array is indexed

- Example of coalescing: Vector addition

```
__global__ void vecadd_kernel(float* x, float* y, float* z, int N)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    z[i] = x[i] + y[i];
}
```

A warp contains consecutive threads in the x dimension followed by the y dimension

  - Accesses to **x**, **y**, and **z** are coalesced
  - E.g., threads 0 to 31 access elements 0 to 31, resulting in one memory transaction to service warp 0

Coalesced Memory Accesses - Example

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within one cache line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus transfer utilization: 100%

Full coalescing – 1 memory access per warp

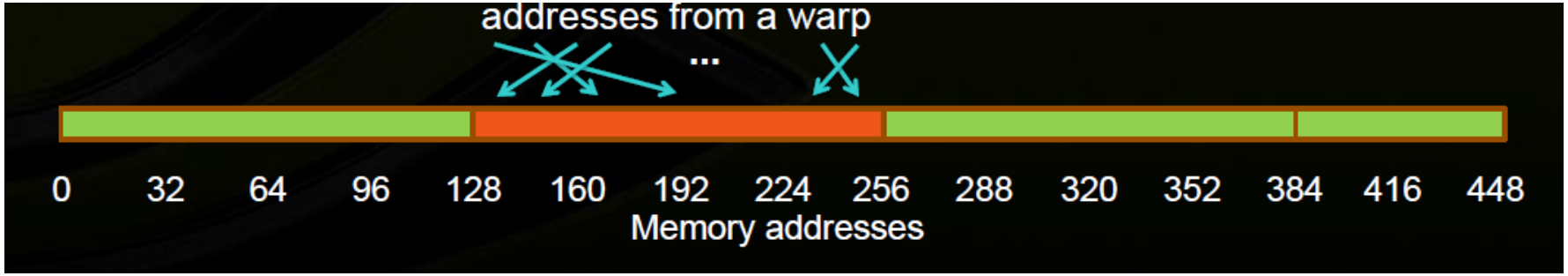
```
i = (blockIdx.x * blockDim.x) + threadIdx.x ;
A[i]
```

Coalesced Memory Accesses - Example

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within one cache line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus transfer utilization: 100%

Full coalescing – 1 memory access per warp

```
t32 = (threadIdx.x / 32) * 32;
i = (blockIdx.x * blockDim.x) + t32 + (31 - threadIdx.x %32);
A[i]
```

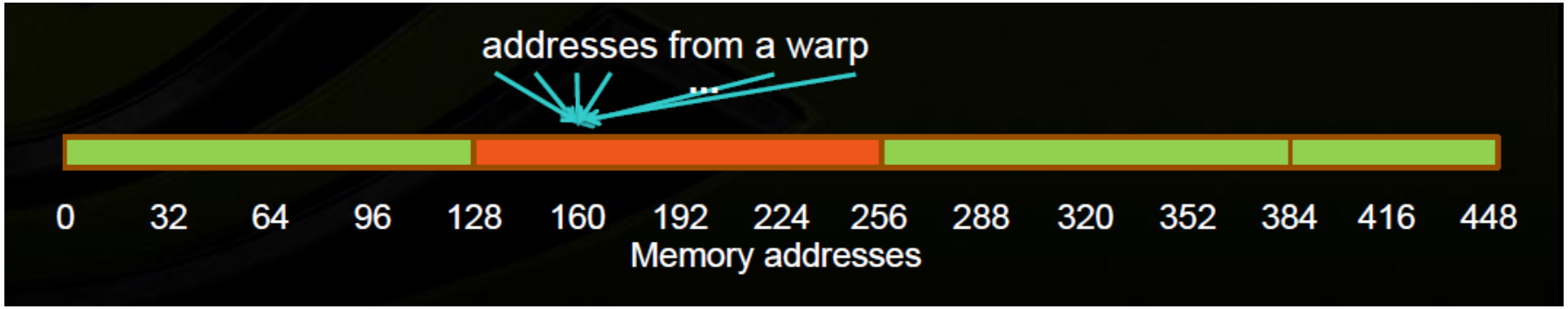


Coalesced Memory Accesses - Example

- All threads in a warp request the same 4-byte word
- Addresses fall within a single cache line
  - Warp needs four bytes
  - 128 bytes move across the bus on a miss
  - Bus transfer utilization: 3.125%

Full coalescing – 1 memory access per warp

```
i = blockidx.x * BlockDim.x;
A[i]
```



Partially Coalesced Memory Accesses - Example

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within two cache lines
  - Warp needs 128 bytes
  - 256 bytes move across the bus on a miss
  - Bus transfer utilization: 50%

Partial coalescing – 2 memory accesses per warp

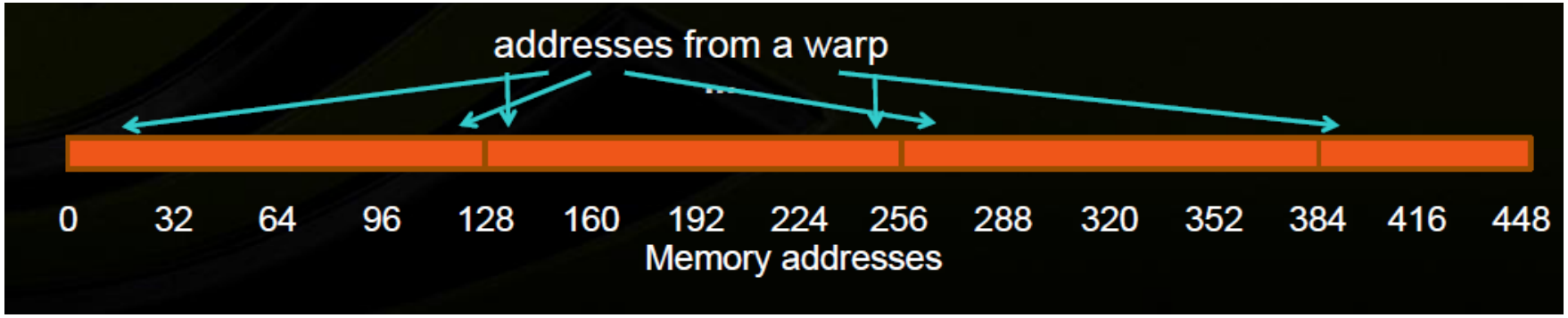
```
// OFFSET = 64
i = blockidx.x * BlockDim.x + threadIdx.x + OFFSET;
A[i]
```

Uncoalesced Memory Accesses - Example

- Warp requests 32 scattered 4-byte words
- Addresses fall within N cache lines
  - Warp needs 128 bytes
  - N*128 bytes move across the bus on a miss
  - Bus utilization: 128 / (N*128), for N=32 → 3.125%

Worst case scenario – 32 threads in a warp generates 32 accesses

```
i = blockidx.x * BlockDim.x + threadidx.x * 32;
A[i]
```



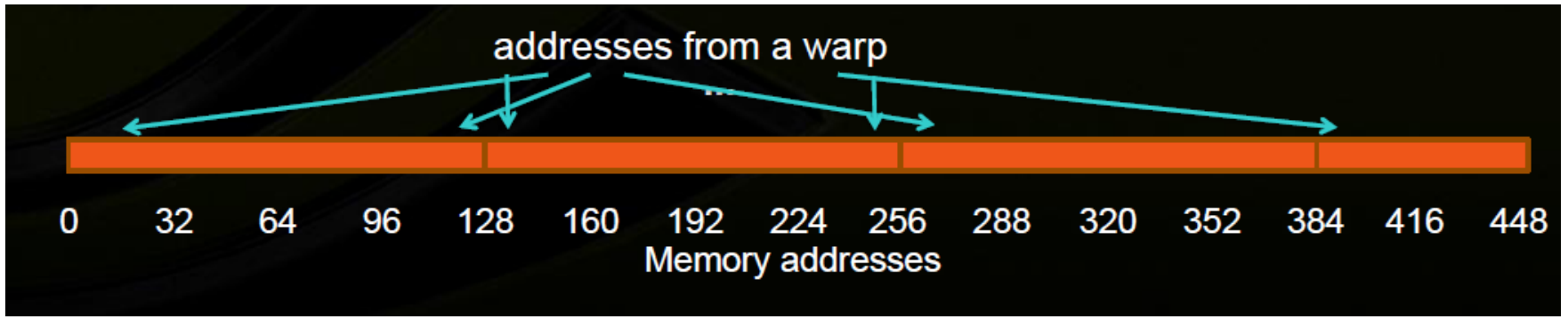
Indirect Memory Accesses – Unpredictable Coalescing

- Memory dependent indexing causes uncoalesced access
  - Also called irregular memory access
- Addresses may ***not*** fall within a single cache line
  - In the worst case each thread in a warp will access different cache line
  - Bus transfer utilization: Unpredictable

Unpredictable scenario -- but memory dependent addressing leads to uncoalesced access

```
i = Index[j];
A[i]
```

Depends upon content of the Index array



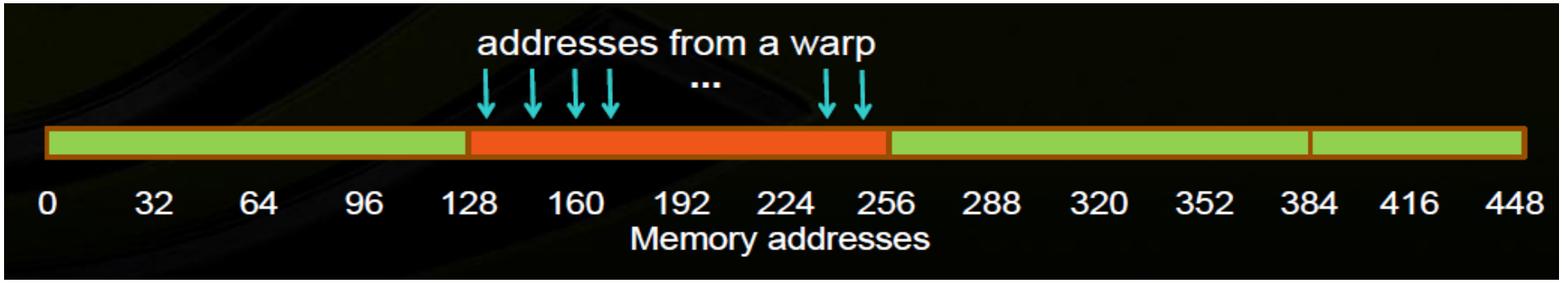
Indirect Memory Accesses – Unpredictable Coalescing

- Memory dependent indexing causes uncoalesced access
  - Also called irregular memory access
- Addresses may ***not*** fall within a single cache line
  - In the worst case each thread in a warp will access different cache line
  - Bus transfer utilization: Unpredictable

Unpredictable scenario -- but memory dependent addressing leads to uncoalesced access

```
i = Index[j];
A[i]
```

Depends upon content of the Index array



Coalesced or Uncoalesced Memory Accesses?

- Would the following indexing coalesce?

```
int row = blockDim.y*blockIdx.y + threadIdx.y;
int col = blockDim.x*blockIdx.x + threadIdx.x;
for(unsigned int i = 0; i < N; ++i) {
    sum += A[row*N + i]*B[i*N + col];
}
```

- Accesses to **A** and **B** are coalesced
  - E.g., threads 0 to 31 access element 0 of **A** on the first iteration, resulting in one access to service warp 0
  - E.g., threads 0 to 31 access elements 0 to 31 of **B** on the first iteration, resulting in one access to service warp 0

Divergence Kills GPU Program Performance

- Uncoalesced Memory Accesses → Memory Access Divergence

- Beware of Divergences in the GPU program

- Control flow divergence
- Memory access divergence