CSE 6230:
HPC Tools and Applications

+

Lecture 13: GPU Programming

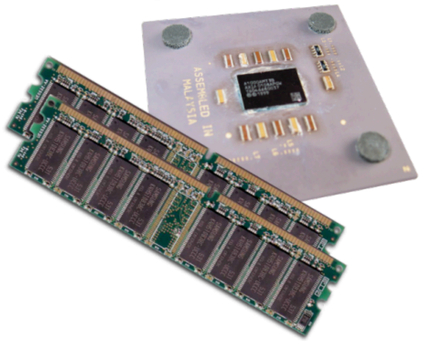
Helen Xu

hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational
Science and Engineering

Heterogeneous Computing

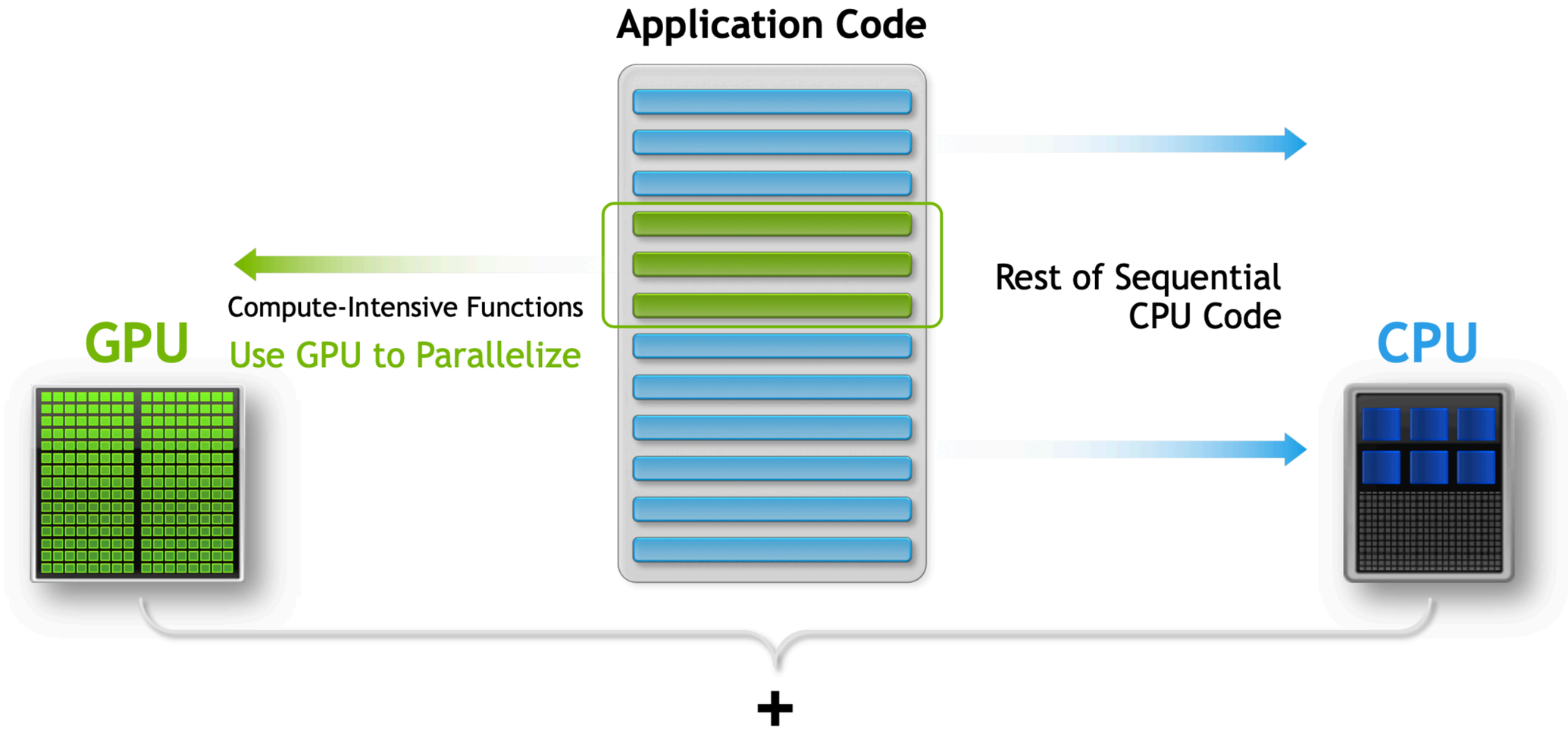
Host: CPU and its memory (host memory)



Device: GPU and its memory (device memory)

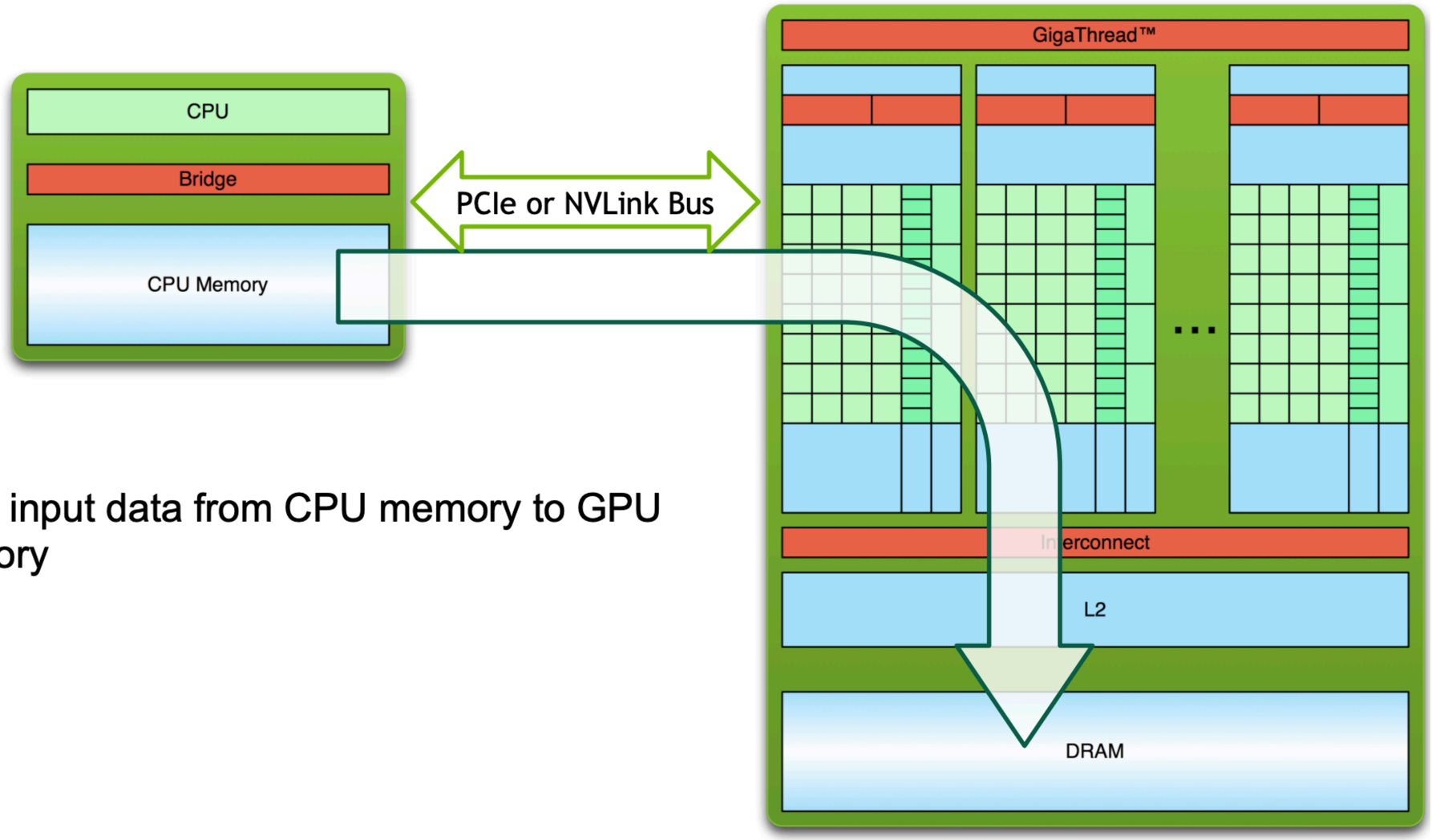
From OLCF Cuda Training Series

CPU / GPU Relationship



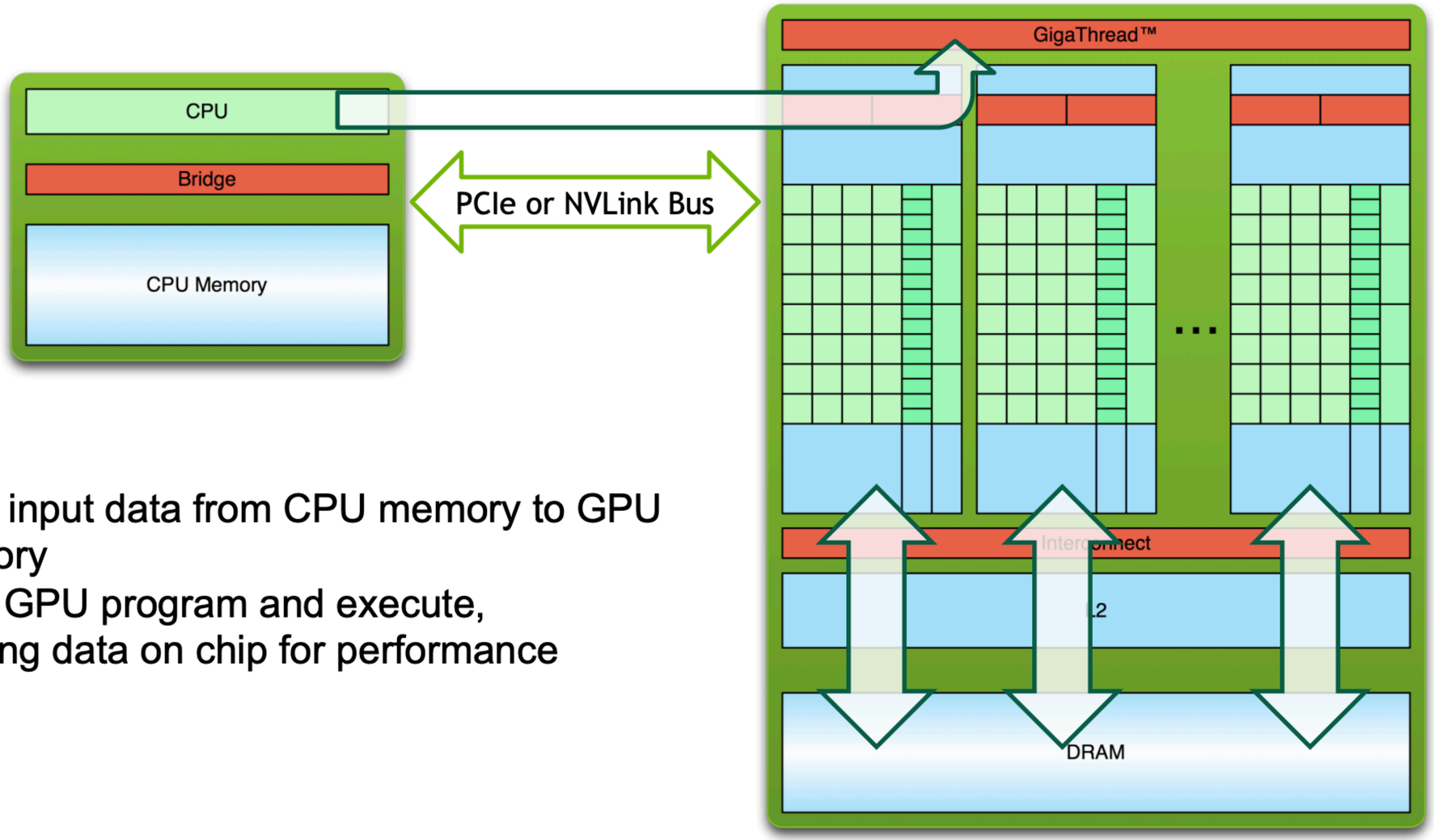
From OLCF Cuda Training Series

Simple Processing Flow



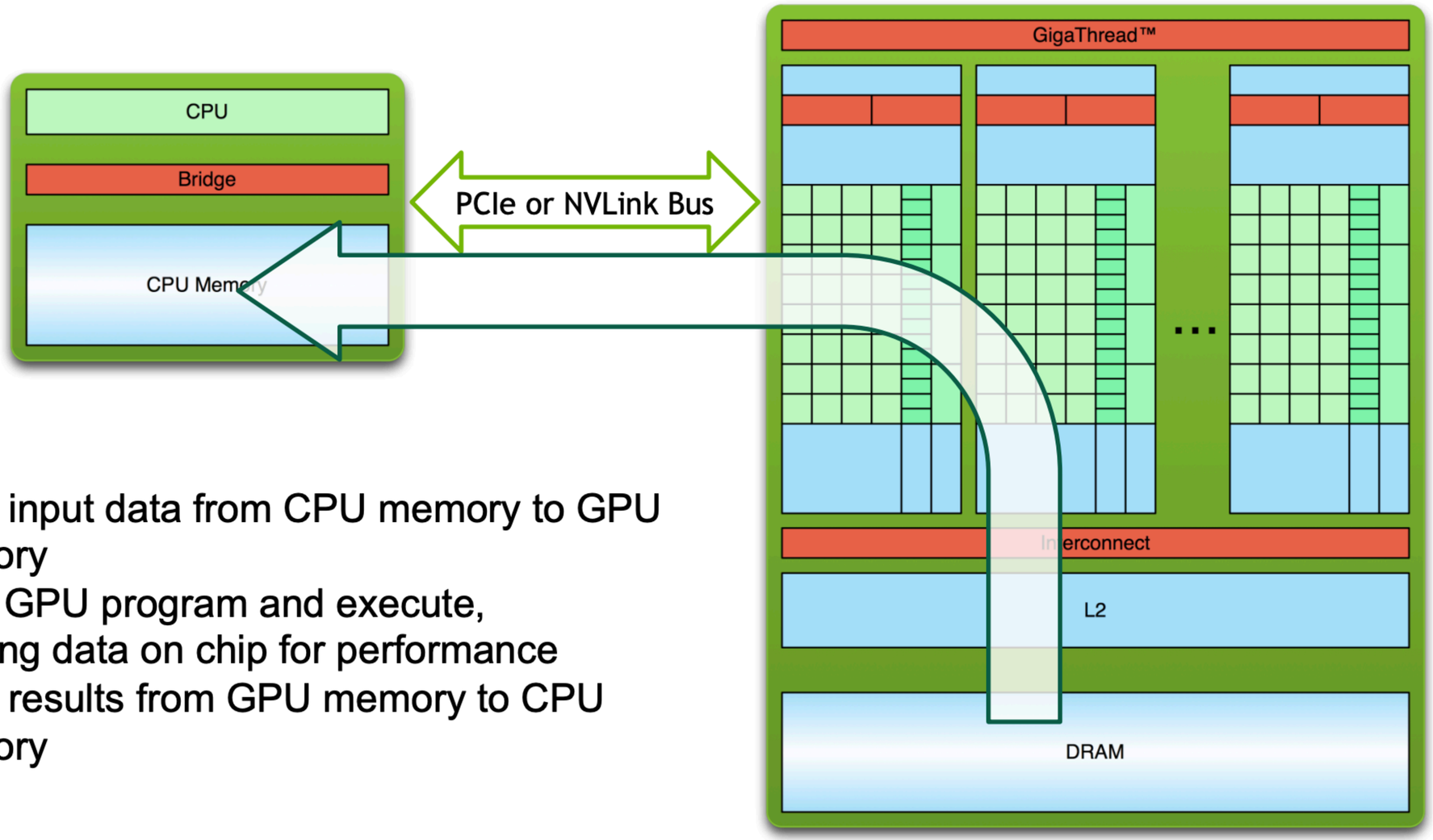
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



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

Simple Processing Flow



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

Cuda Programming 101

What is CUDA?

CUDA - Compute Unified Device Architecture

- Expose GPU parallelism for general-purpose computing
- Expose/Enable performance

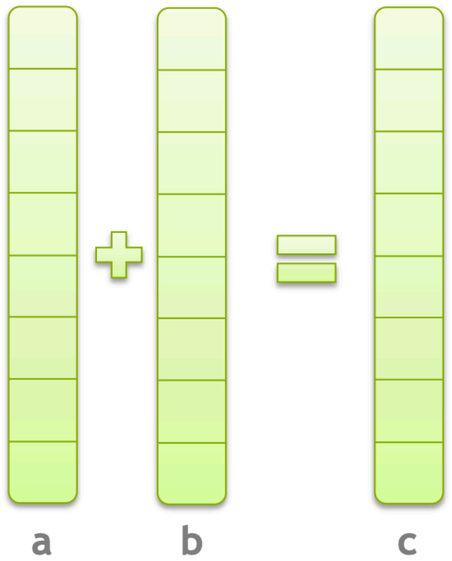
CUDA C++

- Based on industry-standard C++
- Set of extensions to enable heterogeneous programming
- Straightforward APIs to manage devices, memory etc.

Example: Vector Addition

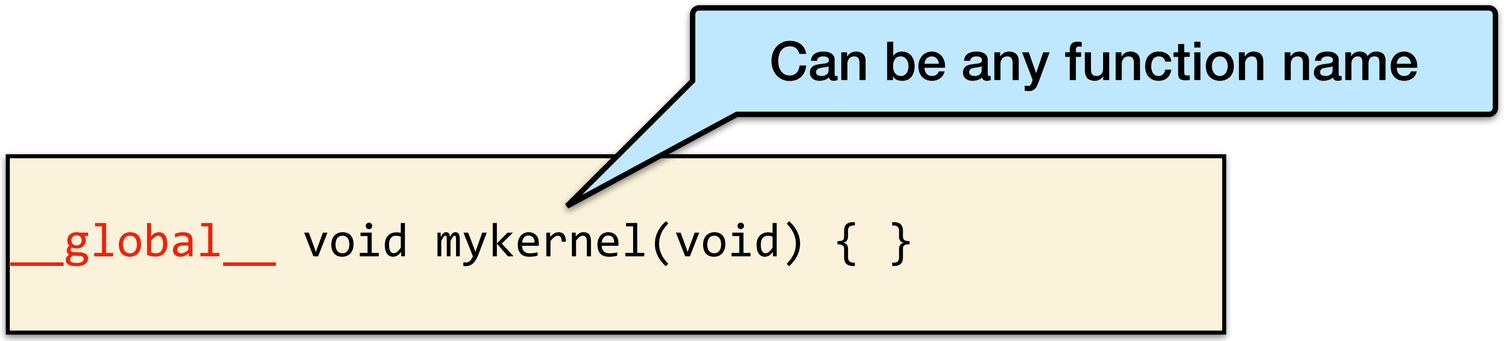
GPU computing is about **massive parallelism**.

Vector addition is embarrassingly parallel (pleasingly parallel) - all the elements are independent.



From OLCF Cuda Training Series

GPU Kernels: Device Code



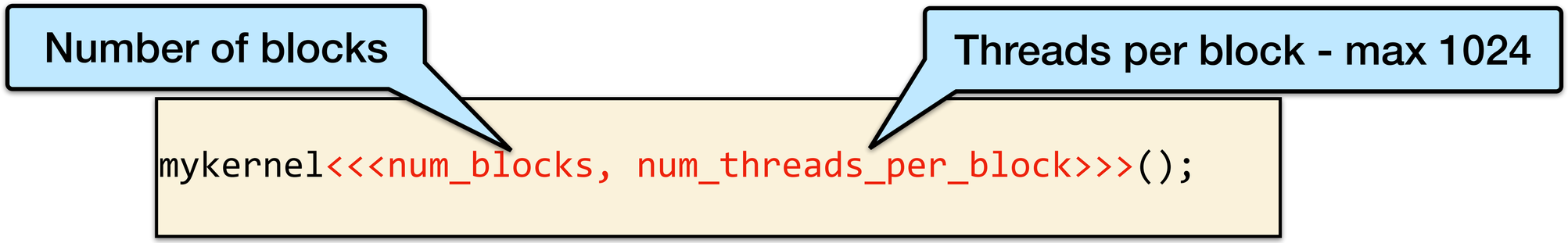
CUDA C++ keyword `__global__` indicates a function that:

- **Runs on the device**
- Is called from host code (can also be called from other device code)

`nvcc` separates source code into **host and device components**

- Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
- Host functions (e.g. `main()`) processed by standard host compiler (e.g., `gcc`)

GPU Kernels: Device Code



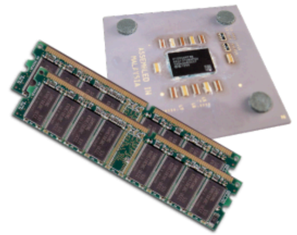
- **Triple angle brackets** mark a call to device code - also called a “kernel launch”
- The parameters inside the triple angle brackets are the **CUDA kernel execution configuration**
- That’s all that is required to execute a function on the GPU!

Memory Management

Host and device memory are **separate entities**.

Device pointers point to GPU memory
- Typically passed to device code
- Typically not dereferenced in host code

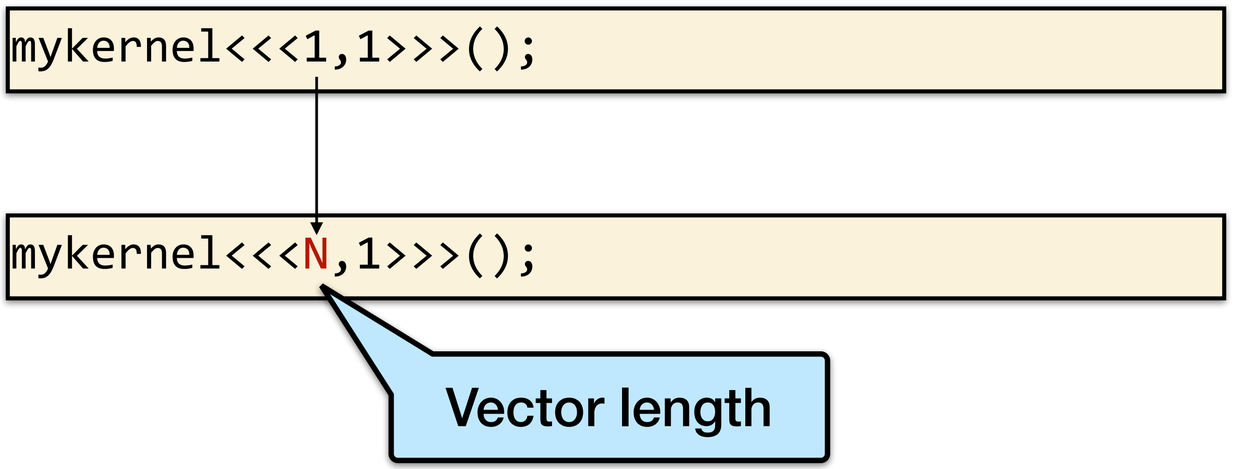
Host pointers point to CPU memory
- Typically not passed to device code
- Typically not dereferenced in device code



Simple CUDA API for handling device memory
- `cudaMalloc(), cudaFree(), cudaMemcpy()`
- Similar to the C equivalents `malloc(), free(), memcpy()`

Running Code in Parallel

GPU computing is about massive parallelism - how do we run code in parallel on the device?



Instead of executing `add()` once, execute it N times in parallel.

Vector Addition on the Device

With `add()` running in parallel we can do vector addition

Terminology: each parallel invocation of `add()` is referred to as a **block**

- The set of all blocks is referred to as a grid
- Each invocation can refer to its **block index** using `blockIdx.x`

```
__global__ void add(int *a, int *b, int *c) {
  c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

By using `blockIdx.x` to index into the array, each block handles a different index

Built-in variables like `blockIdx.x` are zero-indexed (C/C++ style), 0..N-1, where N is from the kernel execution configuration indicated at the kernel launch

Vector Addition on the Device

```
#define N 512

int main(void) {
  int *a, *b, *c; // host copies of a, b, c
  int *d_a, *d_b, *d_c; // device copies of a, b, c
  int size = N * sizeof(int);

  // Alloc space for device copies of a, b, c
  cudaMalloc((void **)&d_a, size);
  cudaMalloc((void **)&d_b, size);
  cudaMalloc((void **)&d_c, size);

  // Alloc space for host copies of a, b, c and setup input values
  a = (int *)malloc(size); random_ints(a, N);
  b = (int *)malloc(size); random_ints(b, N);
  c = (int *)malloc(size);

  …
```

Host pointers

Device pointers

Allocate space on device with cudaMalloc

Allocate space on host with malloc

Vector Addition on the Device

```
  // Copy inputs to device
  cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

  // Launch add() kernel on GPU with N blocks
  add<<N,1>>(d_a, d_b, d_c);

  // Copy result back to host
  cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

  // Cleanup
  free(a); free(b); free(c);
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
  return 0;
}
```

Pass data from host to device

Do vector addition

Copy result to host

Cleanup both host and device

CUDA Threads

Terminology: a block can be split into **parallel threads**

Let's change `add()` to use parallel threads instead of parallel blocks

```
__global__ void add(int *a, int *b, int *c) {
  c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

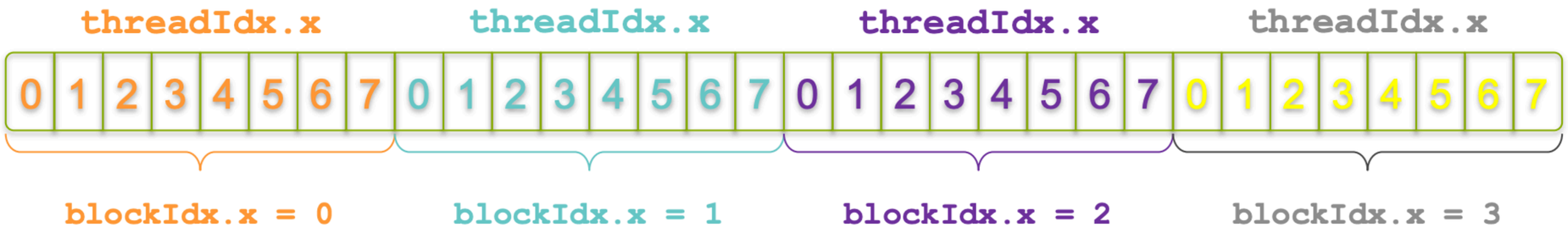
We use `threadIdx.x` instead of `blockIdx.x`

Need to make one change in `main()`: `add<<< 1, N >>>();`

Indexing Arrays With Blocks and Threads

Let's adapt vector addition to use both blocks and threads - no longer as simple as using `blockIdx.x` and `threadIdx.x`

Consider indexing an array with one element per thread (8 threads/block):

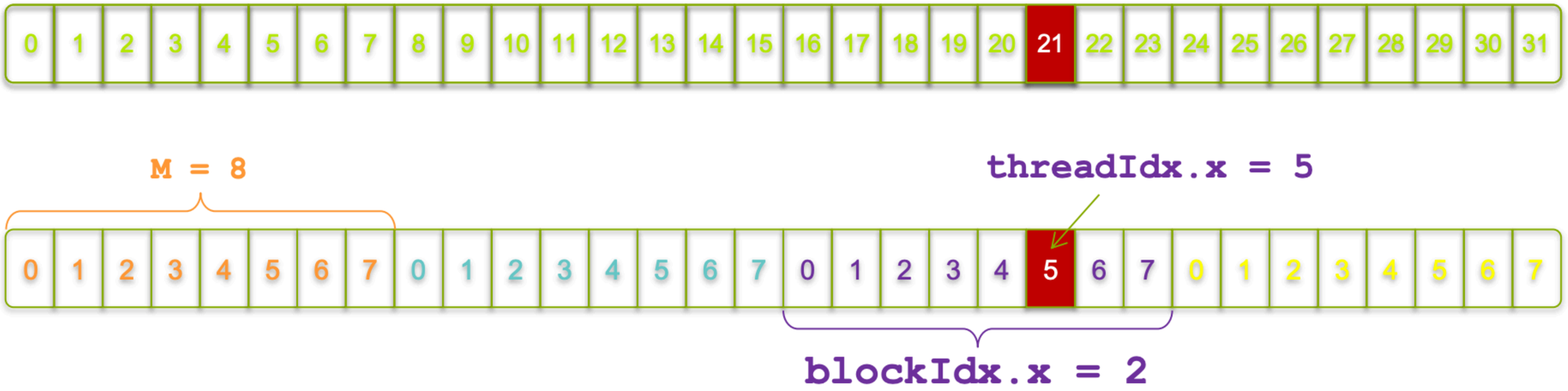


With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

Indexing Arrays With Blocks and Threads

Which thread will operate on the red element?




```
int index = threadIdx.x + blockIdx.x * M;
          =       5     +      2       * 8;
          = 21;
```

Vector Addition with Blocks and Threads

Use the built-in variable `blockDim.x` for threads per block:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

Combined version of `add()` to use parallel threads and parallel blocks:

```
__global__ void add(int *a, int *b, int *c) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  c[index] = a[index] + b[index];
}
```

Vector Addition with Blocks and Threads

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512

int main(void) {
  int *a, *b, *c; // host copies of a, b, c
  int *d_a, *d_b, *d_c; // device copies
  int size = N * sizeof(int);

  // Alloc space for device copies of a, b, c
  cudaMalloc((void **)&d_a, size);
  cudaMalloc((void **)&d_b, size);
  cudaMalloc((void **)&d_c, size);

  // Alloc space for host copies of a, b, c and setup input values
  a = (int *)malloc(size); random_ints(a, N);
  b = (int *)malloc(size); random_ints(b, N);
  c = (int *)malloc(size);

  …
```

Host pointers

Device pointers

Allocate space on device with cudaMalloc

Allocate space on host with malloc

Vector Addition with Blocks and Threads

```
  …

  // Copy inputs to device
  cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

  // Launch add() kernel on GPU
  add<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>(d_a, d_b, d_c);

  // Copy result back to host
  cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

  // Cleanup
  free(a); free(b); free(c);
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
  return 0;
}
```

Pass data from host to device

Do vector addition

Copy result to host

Cleanup both host and device

Handling Arbitrary Vector Sizes

Typical problems are not friendly multiples of `blockDim.x`

Avoid accessing beyond the end of the arrays:

Thread check

```
__global__ void add(int *a, int *b, int *c, int n) {
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  if (index < n)
    c[index] = a[index] + b[index];
}
```

Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

Why Bother With Threads?

Threads seem unnecessary

- They add a level of complexity
- What do we gain?

Unlike parallel blocks, threads have mechanisms to:

- Communicate
- Synchronize

CUDA Shared Memory

1D Stencil

Consider applying a 1D stencil to a 1D array of elements

- Each output element is the sum of input elements within a radius

If radius is 3, then each output element is the sum of 7 input elements:

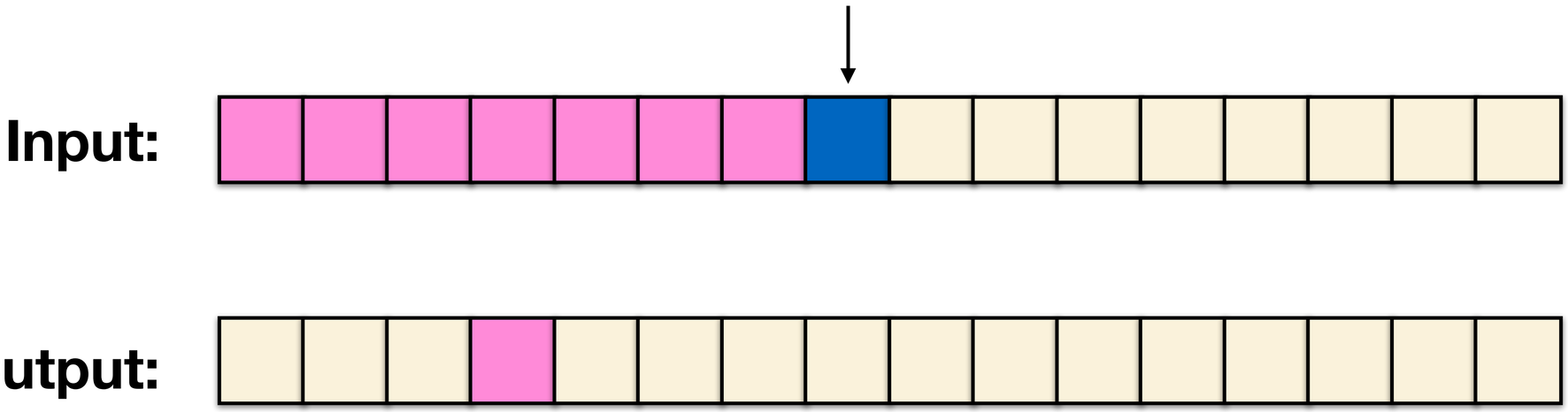
Implementing Within a Block

Each thread processes one output element

- `blockDim.x` elements per block

Input elements are read several times

- With radius 3, each input element is read seven times



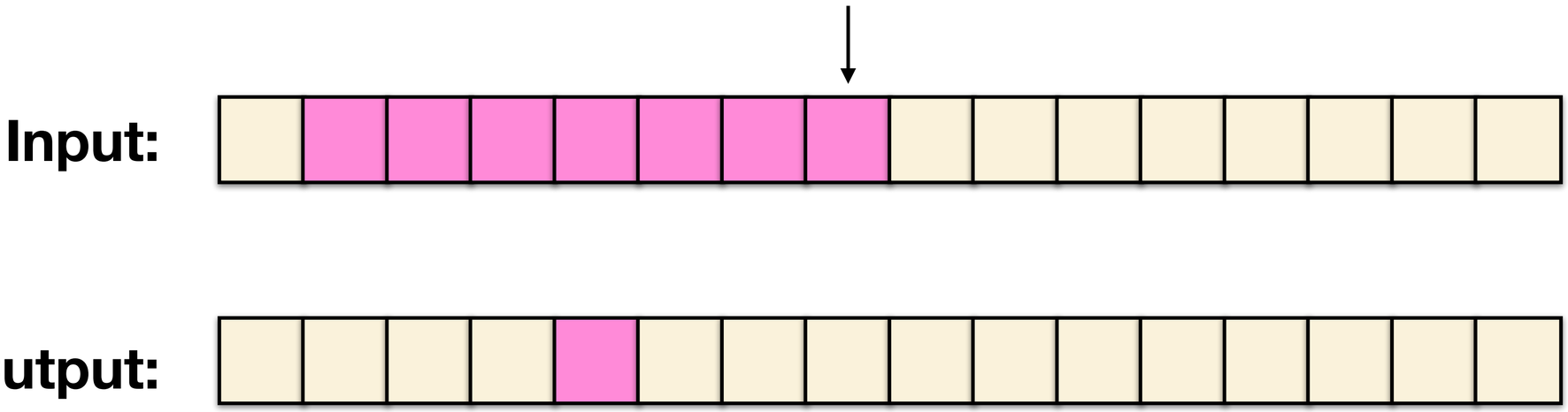
Implementing Within a Block

Each thread processes one output element

- `blockDim.x` elements per block

Input elements are read several times

- With radius 3, each input element is read seven times



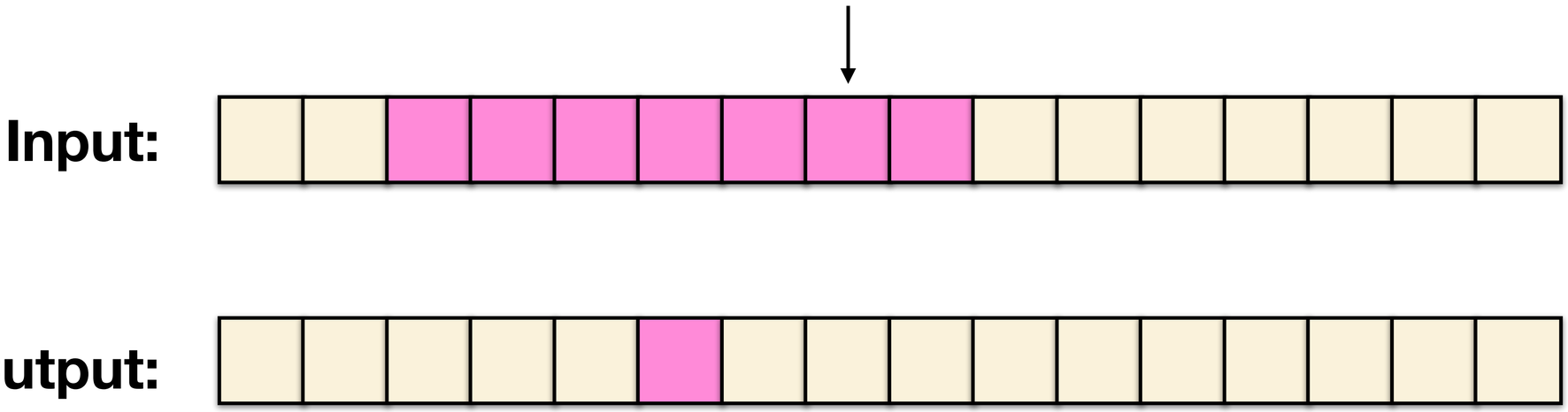
Implementing Within a Block

Each thread processes one output element

- `blockDim.x` elements per block

Input elements are read several times

- With radius 3, each input element is read seven times



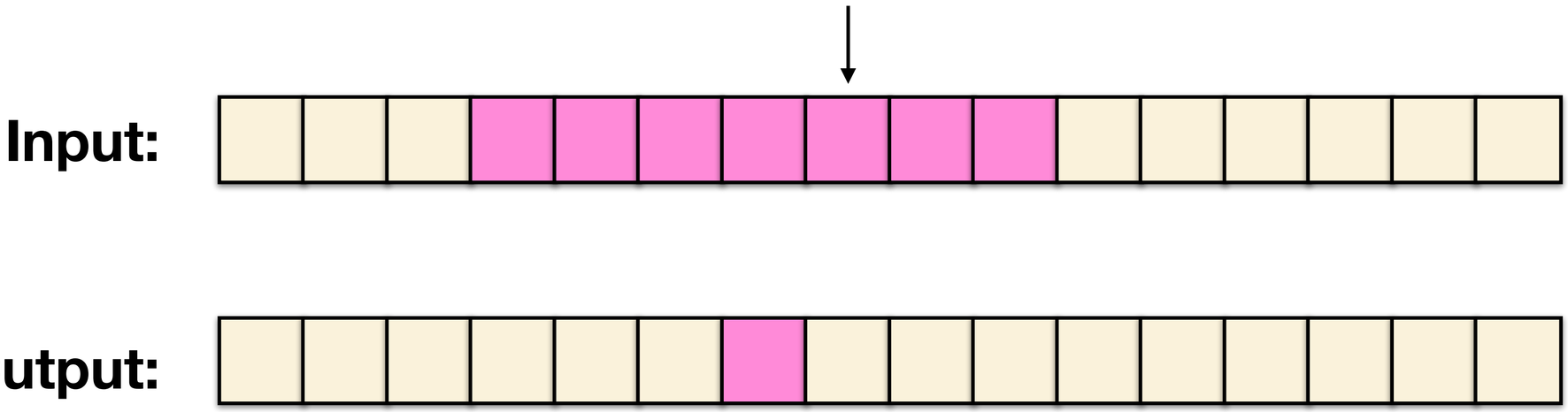
Implementing Within a Block

Each thread processes one output element

- `blockDim.x` elements per block

Input elements are read several times

- With radius 3, each input element is read seven times



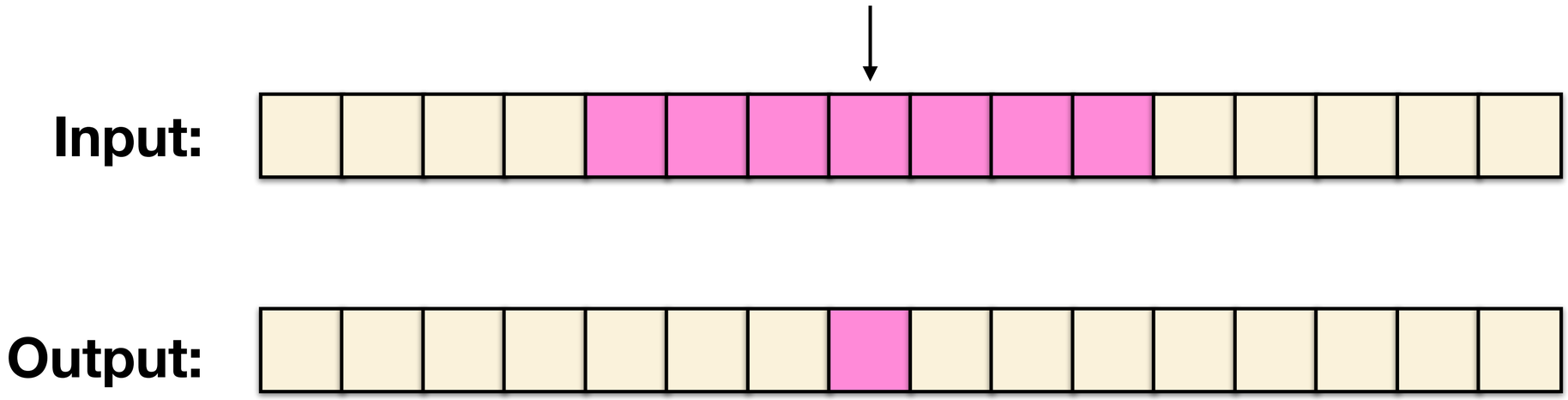
Implementing Within a Block

Each thread processes one output element

- `blockDim.x` elements per block

Input elements are read several times

- With radius 3, each input element is read seven times



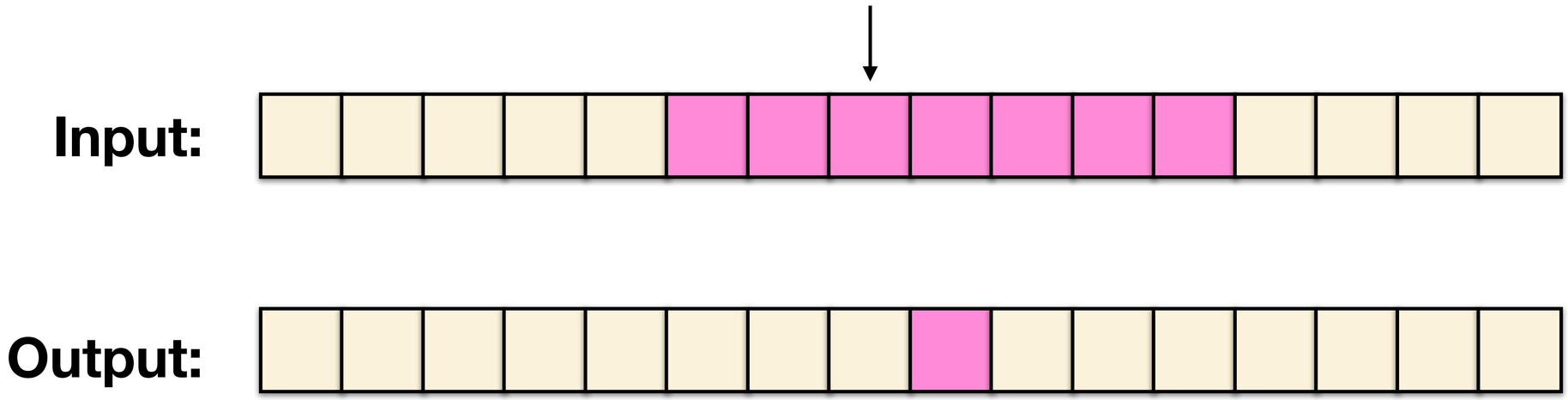
Implementing Within a Block

Each thread processes one output element

- `blockDim.x` elements per block

Input elements are read several times

- With radius 3, each input element is read seven times



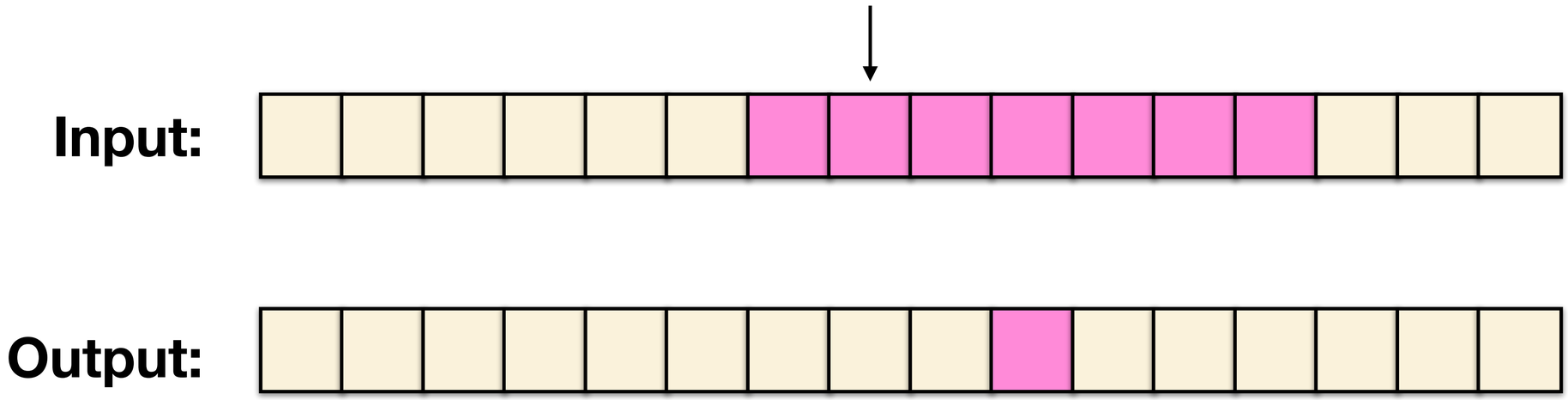
Implementing Within a Block

Each thread processes one output element

- `blockDim.x` elements per block

Input elements are read several times

- With radius 3, each input element is read seven times



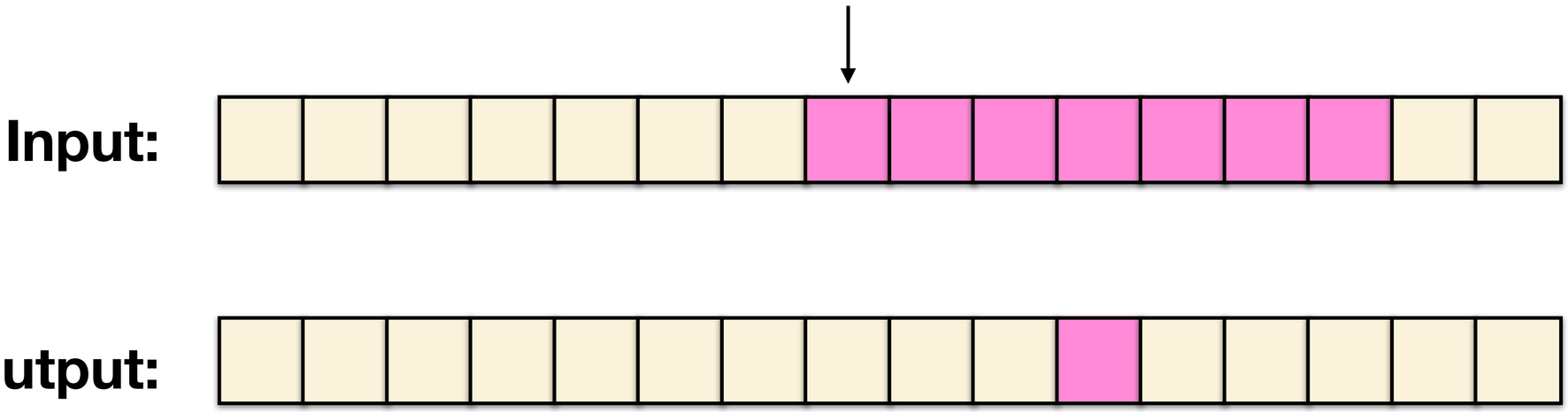
Implementing Within a Block

Each thread processes one output element

- `blockDim.x` elements per block

Input elements are read several times

- With radius 3, each input element is read seven times



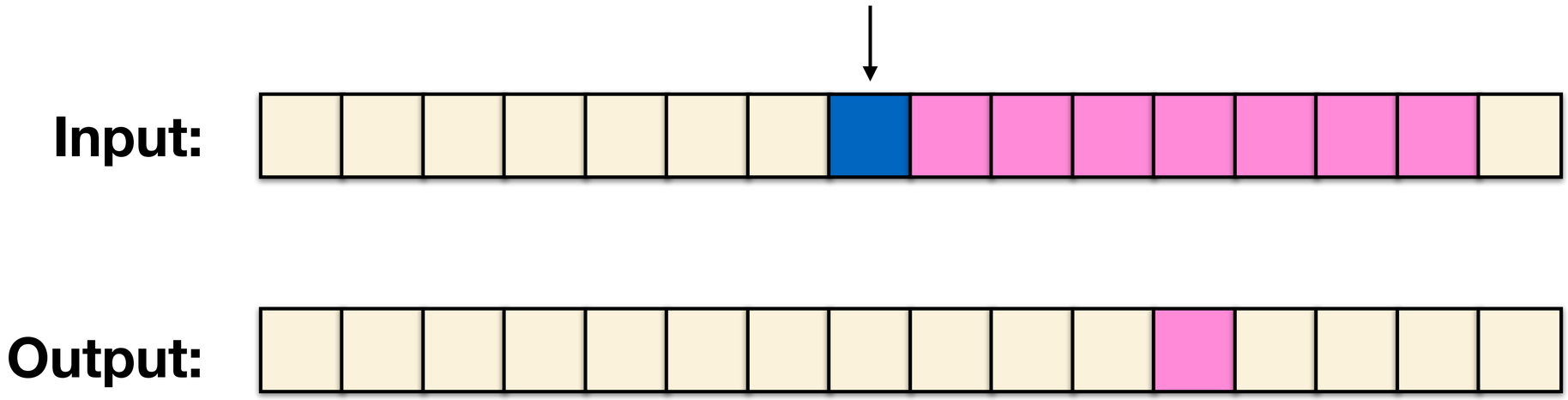
Implementing Within a Block

Each thread processes one output element

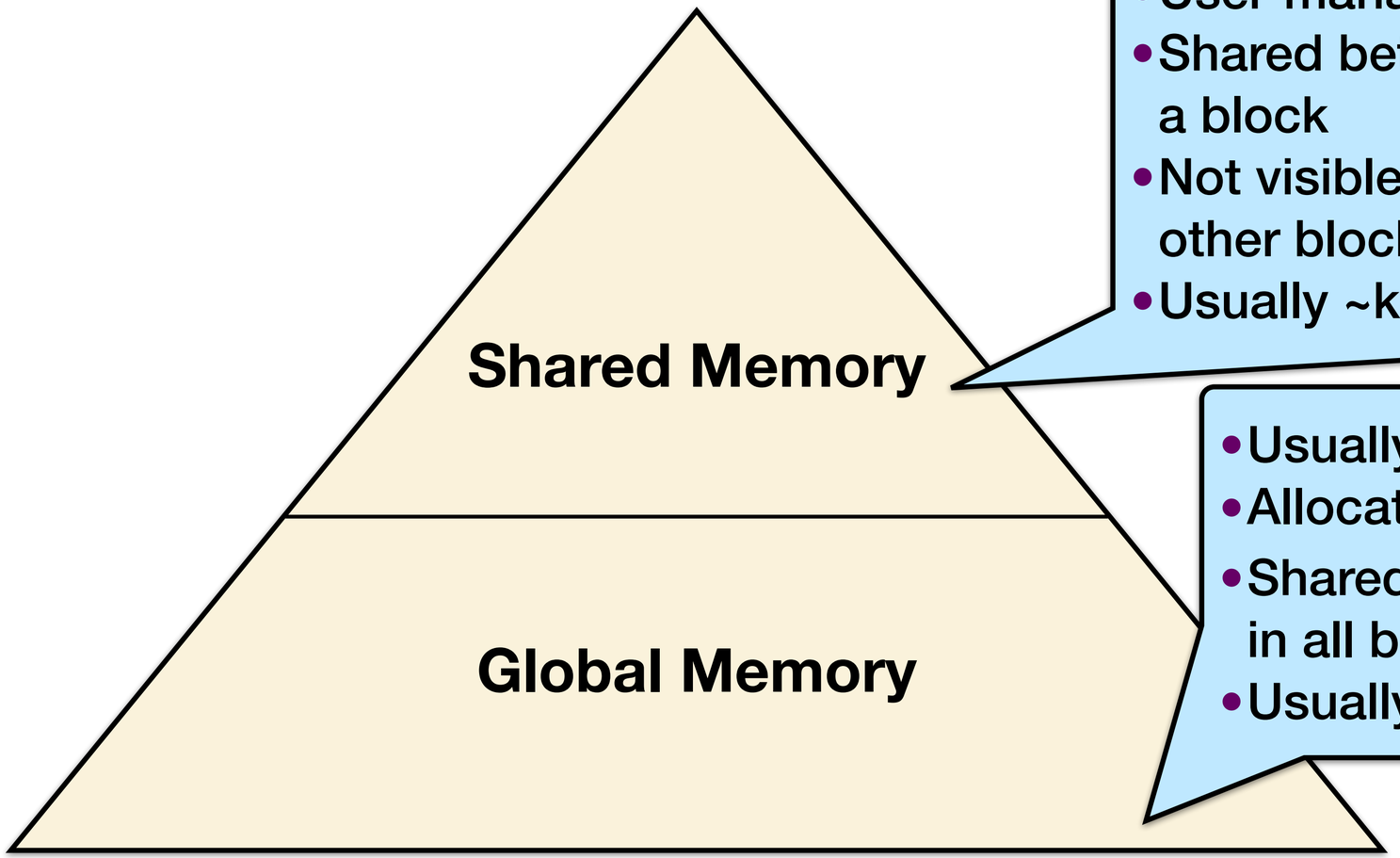
- `blockDim.x` elements per block

Input elements are read several times

- With radius 3, each input element is read seven times



GPU Memory Hierarchy



- Fast on-chip memory
- User-managed
- Shared between threads in a block
- Not visible to threads in other blocks
- Usually ~kB size

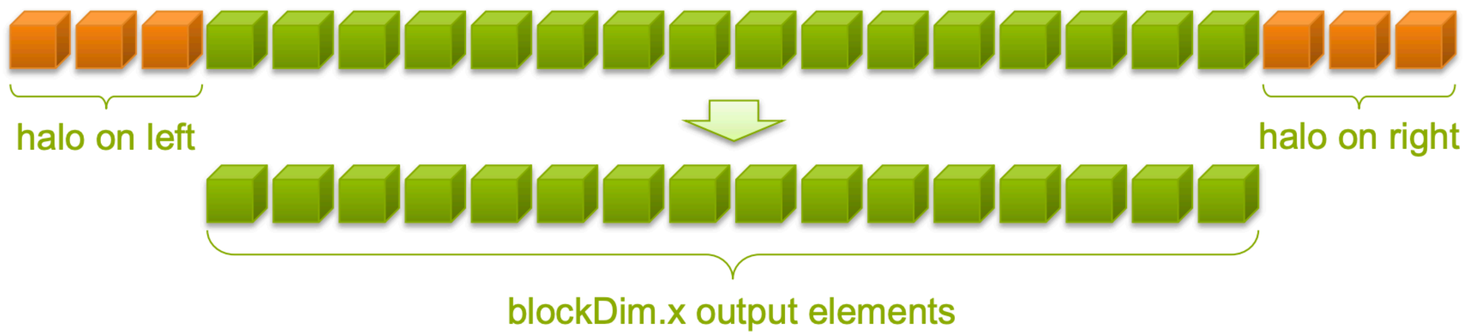
- Usually in GPU DRAM
- Allocate with `cudaMalloc`
- Shared across all threads in all blocks
- Usually ~GB size

Implementing With Shared Memory

Cache data in shared memory

- Read (`blockDim.x + 2 * radius`) input elements from global memory to shared memory
- Compute `blockDim.x` output elements
- Write `blockDim.x` output elements to global memory

Each block needs a halo of `radius` elements at each boundary



Stencil Kernel

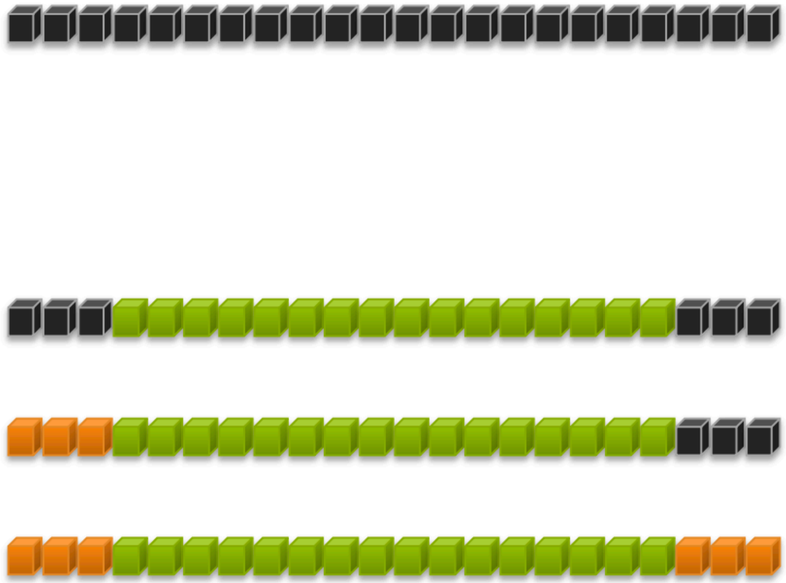
```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into sha
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
```

Num threads

All threads execute this line

Fill in the halo



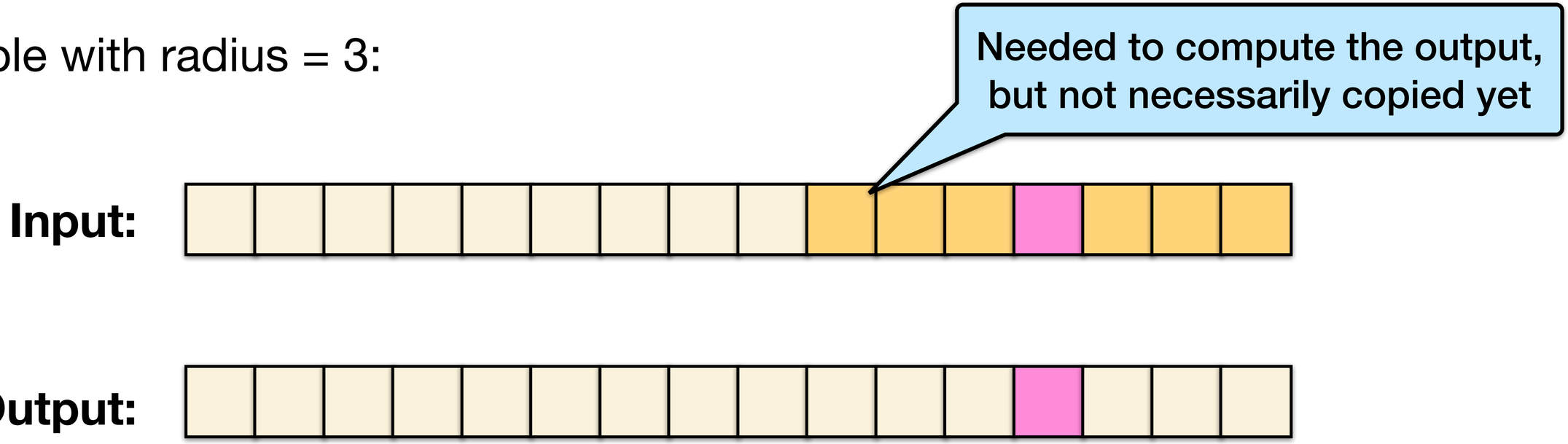
Stencil Kernel

```
  // Apply the stencil
 int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result;
}
```

Issue: Data Race

- The previous code so far will not work because threads are not guaranteed to execute in any specific order.
- Therefore, some threads may run ahead of others. As the code is currently written, a thread may read uninitialized values from shared memory.
- Example with radius = 3:



__syncthreads()

```
void __syncthreads();
```

Synchronizes all threads within a block

All threads must reach the barrier before any proceed beyond the barrier

- In conditional code, the condition must be uniform across the block

Stencil Kernel With Barrier

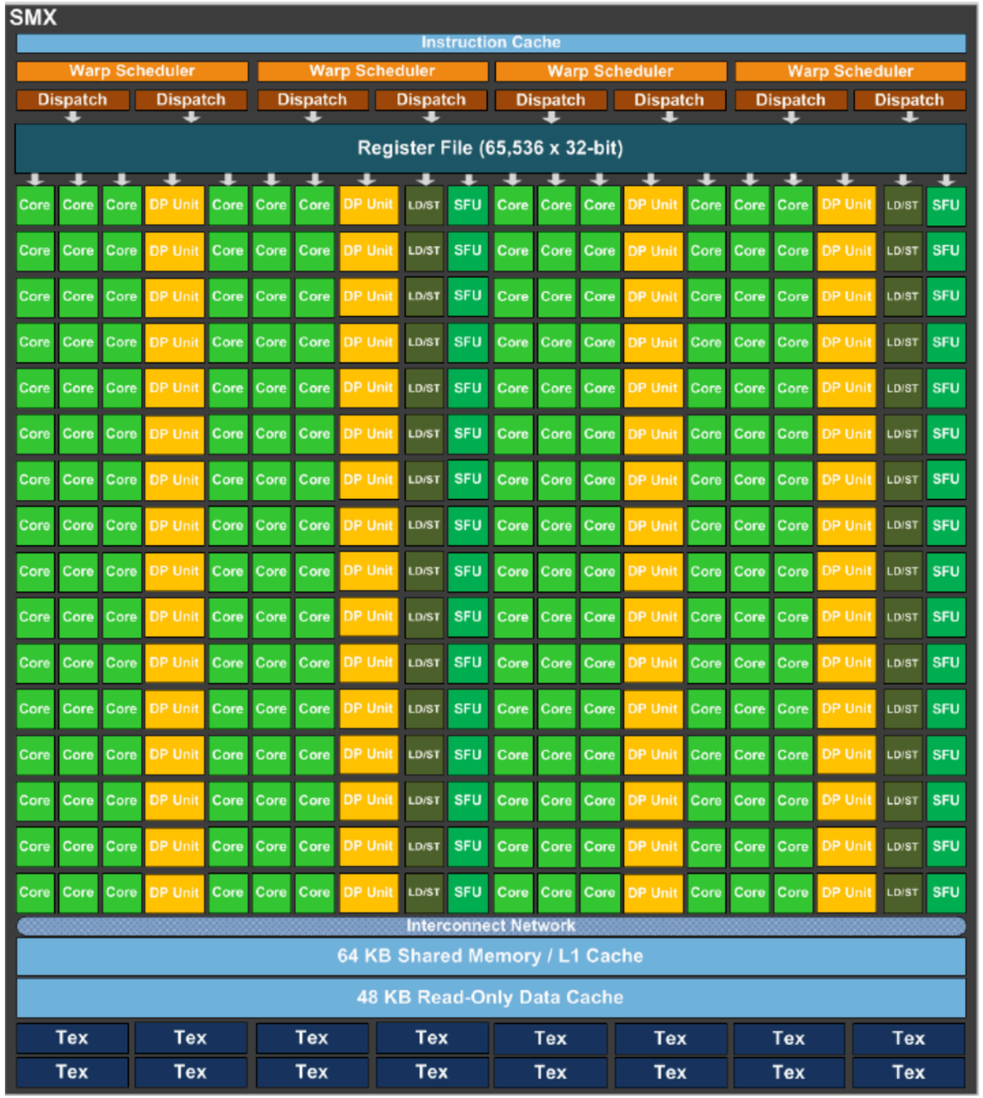
```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // Synchronize (ensure all the data is available)
    __syncthreads();
```

Optimizing for Parallelism

Example: Kepler CC 3.6 SM

- "SMX" (enhanced SM)
- 192 SP units ("cores")
- 64 DP units
- LD/ST units, 64K registers
- 4 warp schedulers
- Each warp scheduler is dual-issue capable
- K20: 13 SMX's, 5GB
- K20X: 14 SMX's, 6GB
- K40: 15 SMX's, 12GB

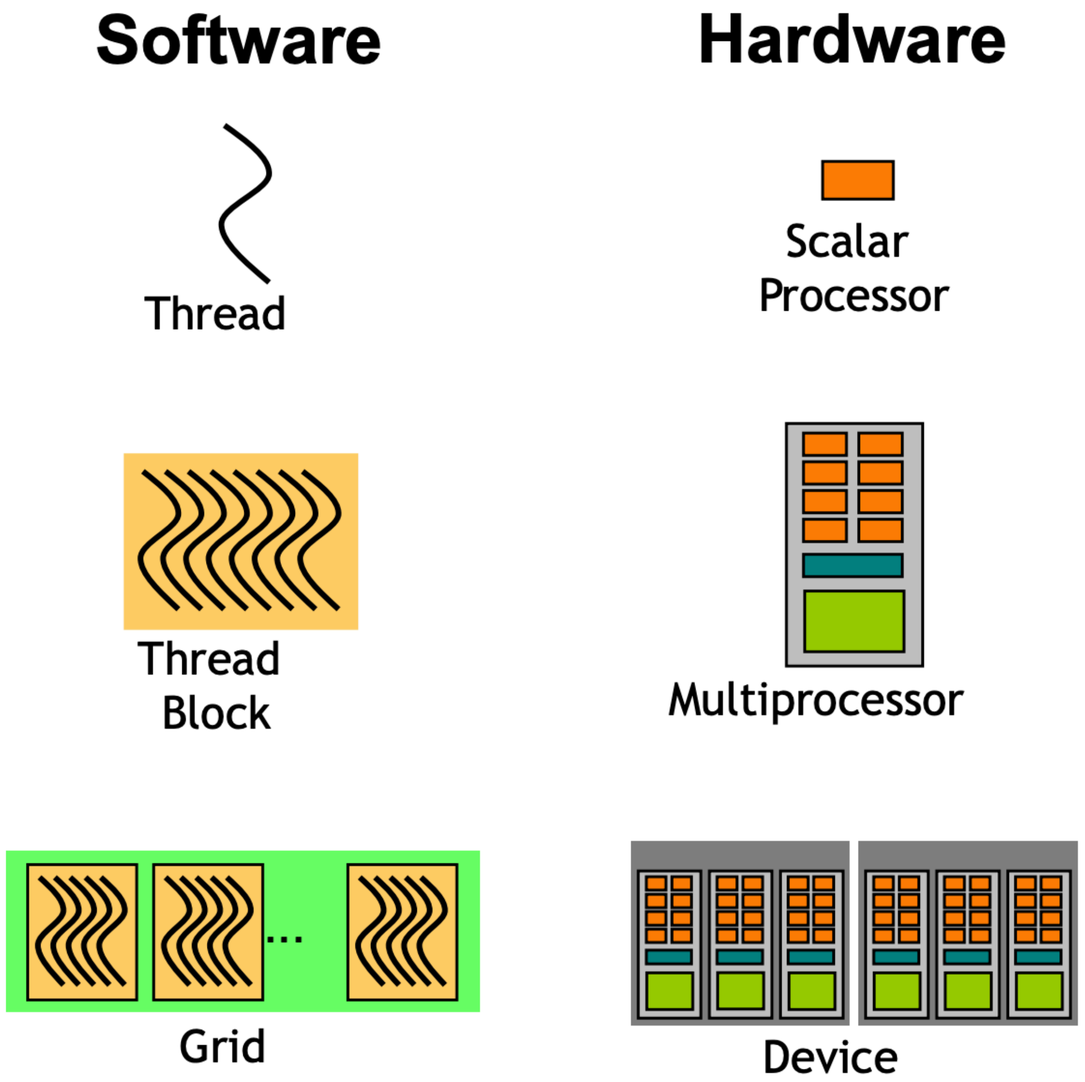


From OLCF Cuda Training Series

Pascal/Volta CC6.0/7.0

- 64 SP units (“cores”)
- 32 DP units
- LD/ST units
- FP16 @ 2x SP rate
- cc7.0: TensorCore
- P100/V100 2/4 warp schedulers
- Volta adds separate int32 units
- P100: 56 SM’s, 16GB
- V100: 80 SM’s, 16/32GB

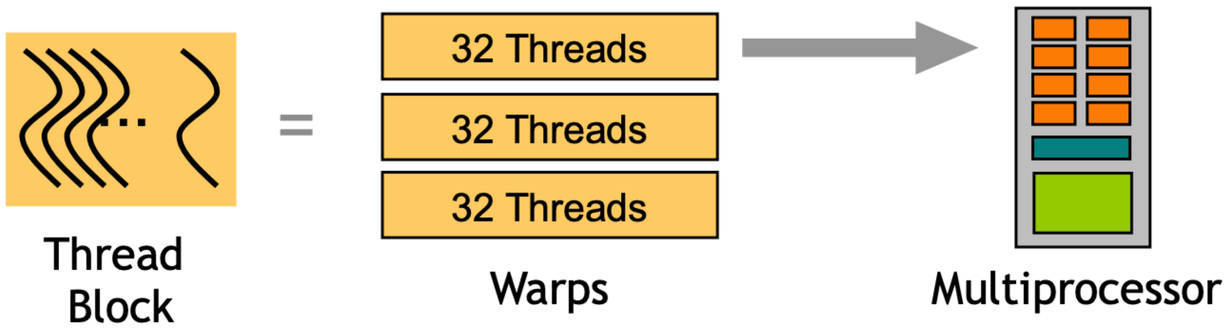
Execution Model



- Threads are executed by scalar processors
- Thread blocks are executed on multiprocessors
- Thread blocks do not migrate
- Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)
- A kernel is launched as a grid of thread blocks

Warps

- A thread block consists of 32-thread warps
- A warp is executed physically in parallel (SIMD) on a multiprocessor



Launch Configuration

Key to understanding:

- Instructions are issued **in order**
- A thread stalls when one of the operands isn't ready:
  - Memory read by itself doesn't stall execution

- Latency is hidden by switching threads
  - **GMEM latency:** >100 cycles (varies by architecture/design)
  - **Arithmetic latency:** <100 cycles (varies by architecture/design)

How many threads/threadblocks to launch?

Objective: Need enough threads to **hide latency**

GPU Latency Hiding

In CUDA C source code:

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
c[idx] = a[idx] + b[idx];
```

In machine code:

```
I0: LD R0, a[idx];
I1: LD R1, b[idx];
I2: MPY R2, R0, R1
```

GPU Latency Hiding - Inside the SM

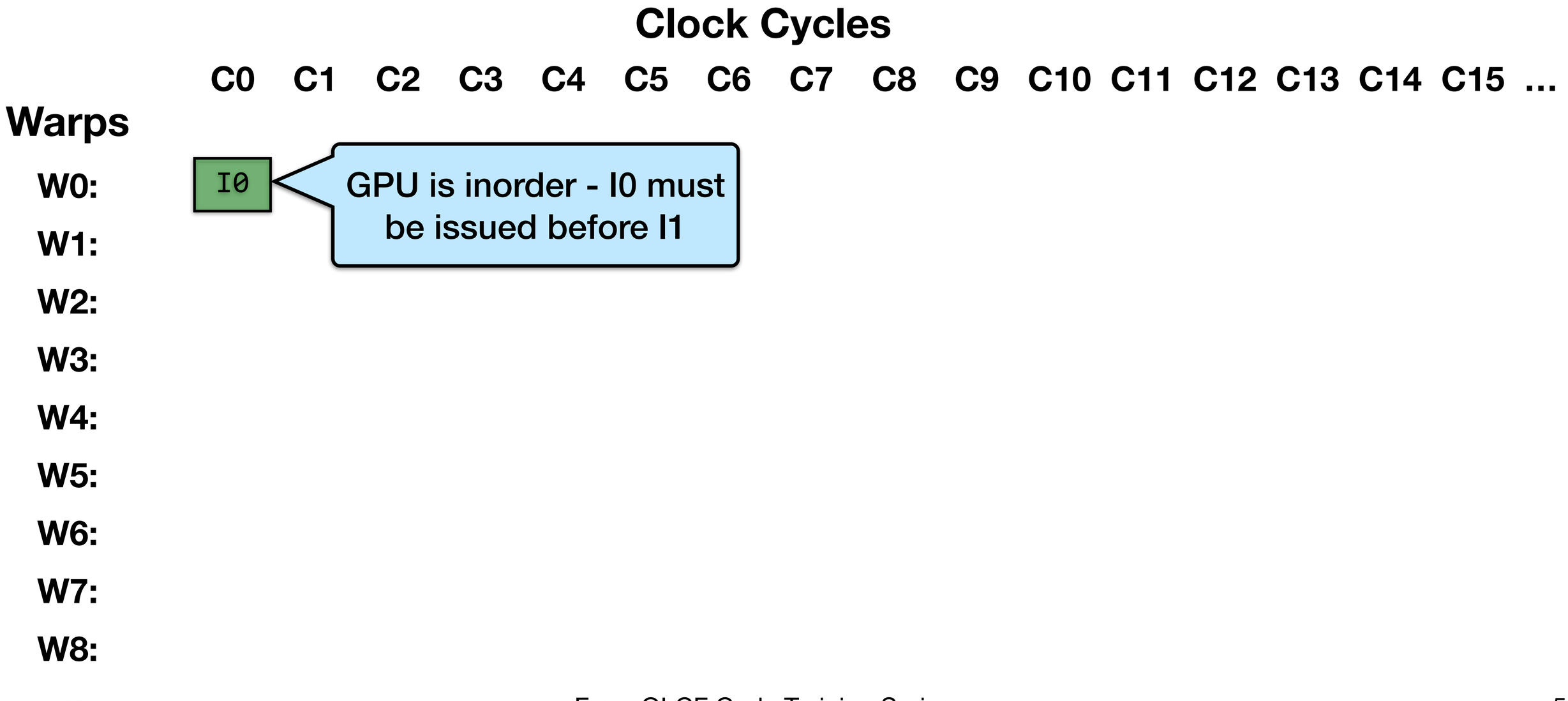
```
I0: LD R0, a[idx];
I1: LD R1, b[idx];
I2: MPY R2, R0, R1
```

Clock Cycles

Warps	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	…
W0:																	
W1:																	
W2:																	
W3:																	
W4:																	
W5:																	
W6:																	
W7:																	
W8:																	
…																	

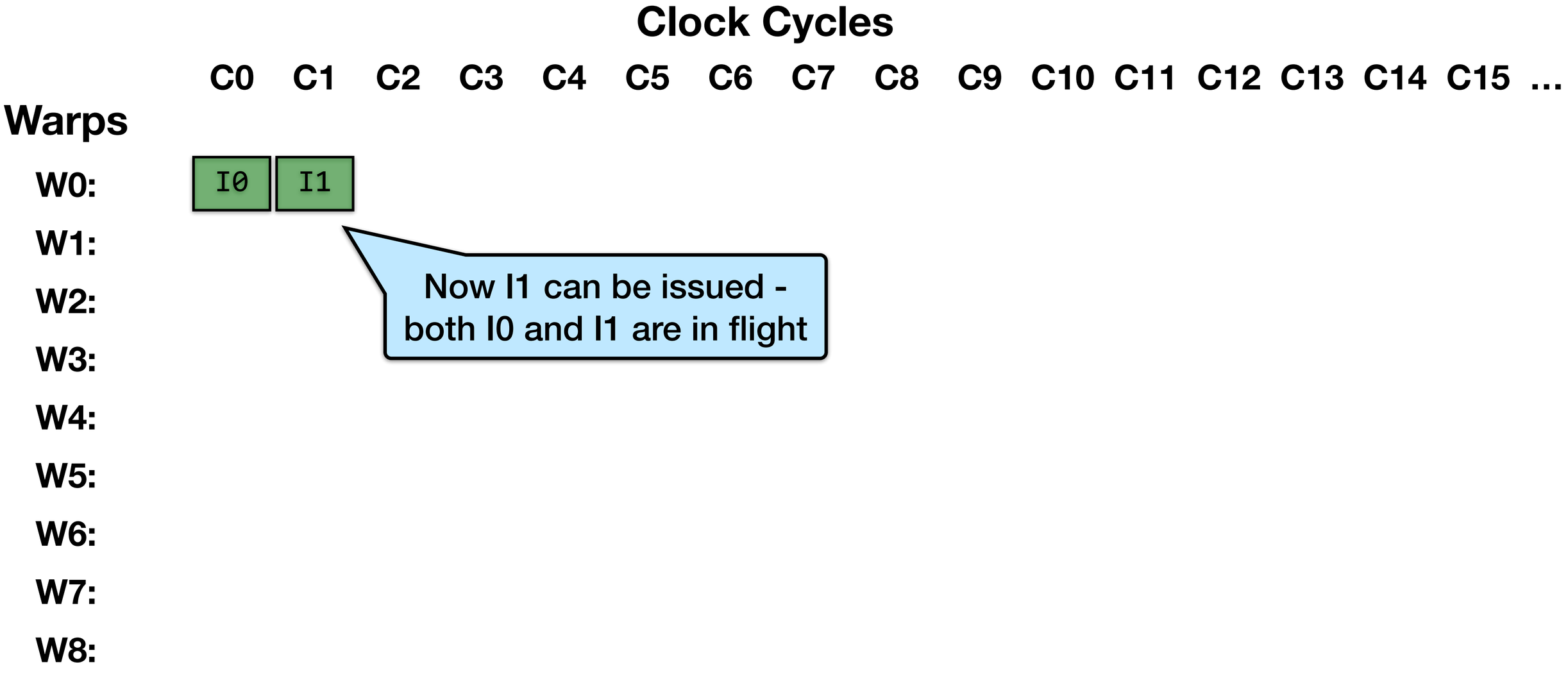
GPU Latency Hiding - Inside the SM

```
I0: LD R0, a[idx];
I1: LD R1, b[idx];
I2: MPY R2, R0, R1
```



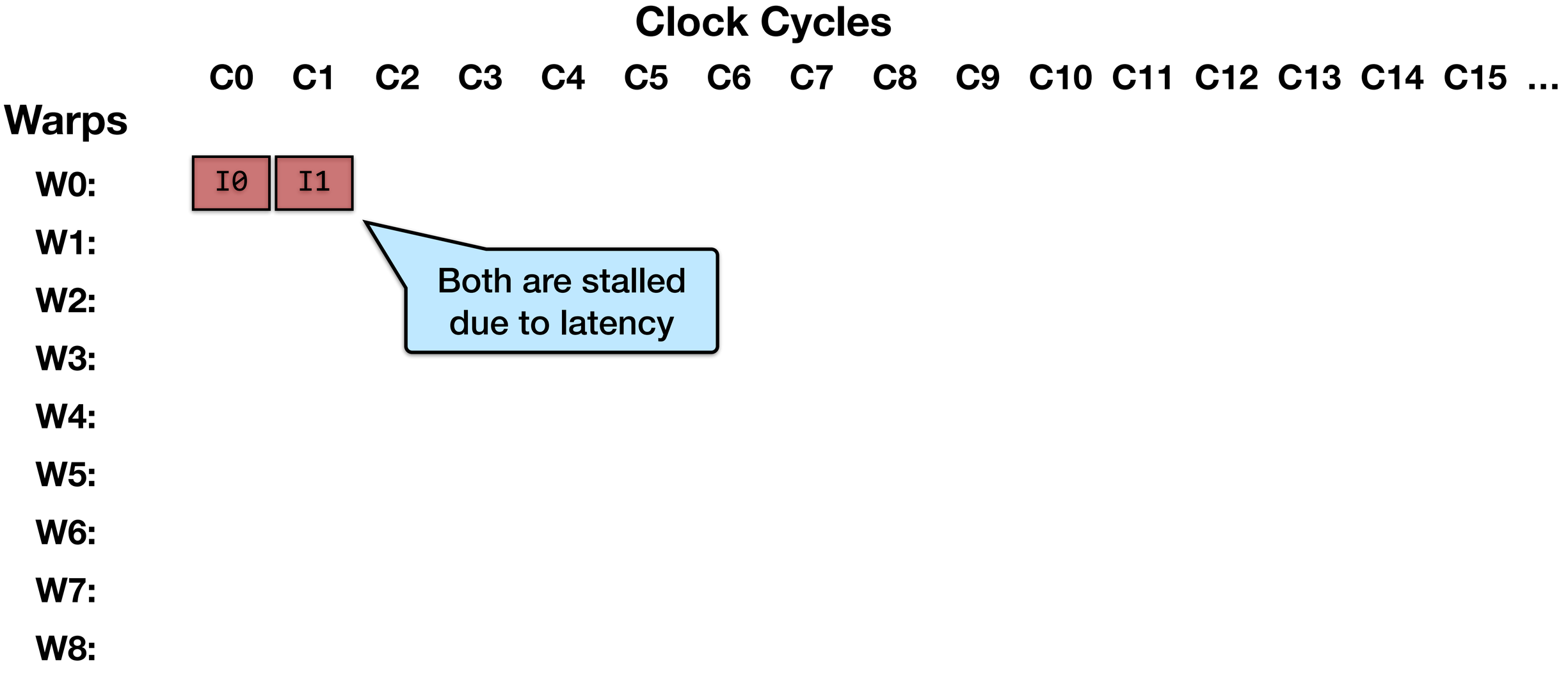
GPU Latency Hiding - Inside the SM

```
I0: LD R0, a[idx];
I1: LD R1, b[idx];
I2: MPY R2, R0, R1
```



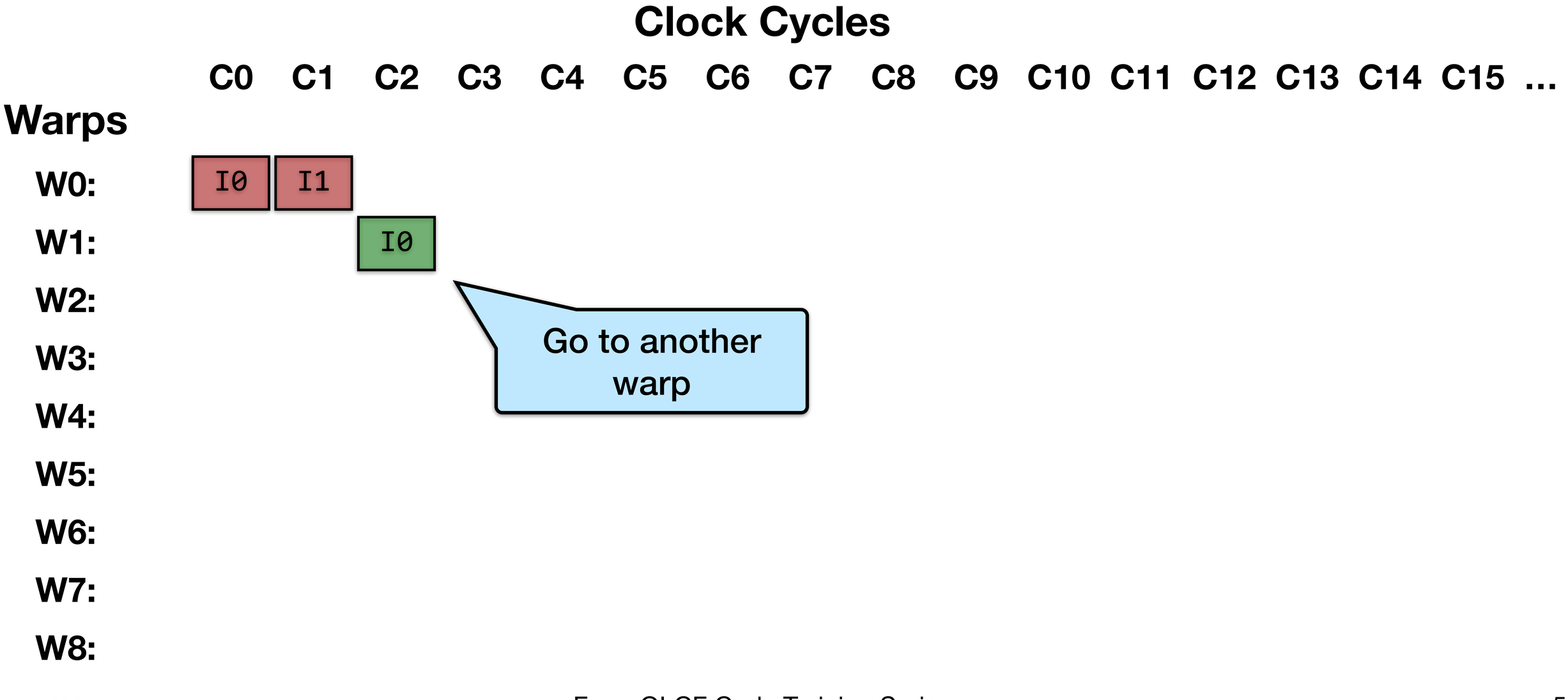
GPU Latency Hiding - Inside the SM

```
I0: LD R0, a[idx];
I1: LD R1, b[idx];
I2: MPY R2, R0, R1
```



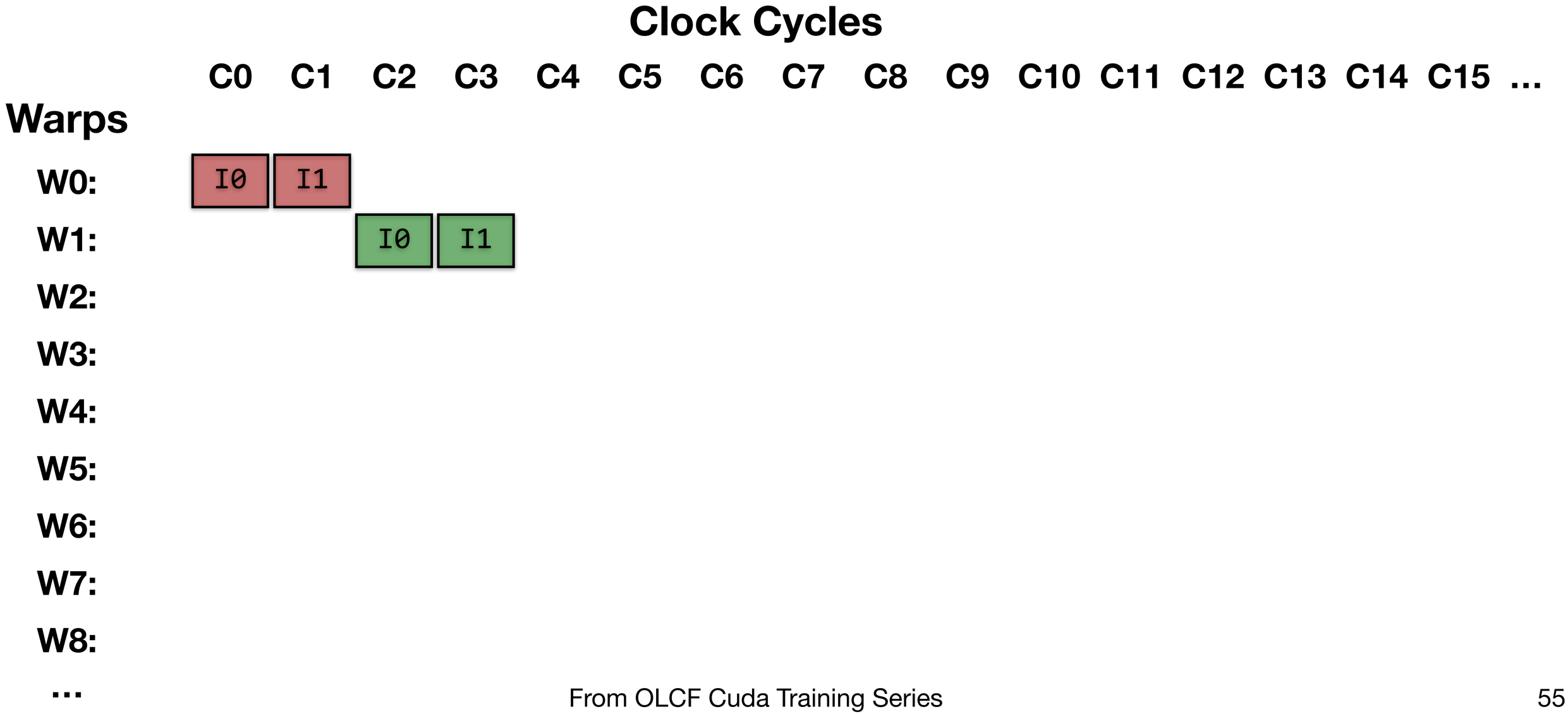
GPU Latency Hiding - Inside the SM

```
I0: LD R0, a[idx];
I1: LD R1, b[idx];
I2: MPY R2, R0, R1
```



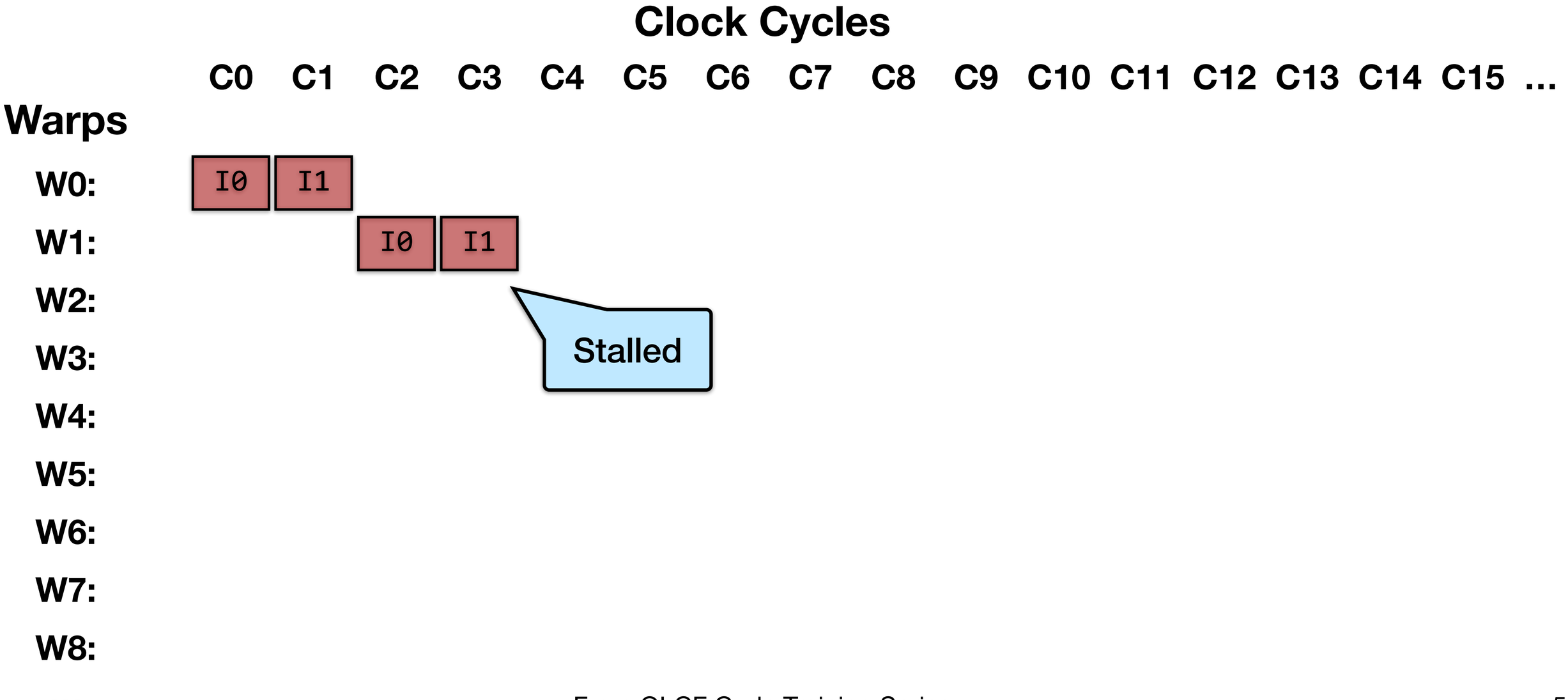
GPU Latency Hiding - Inside the SM

```
I0: LD R0, a[idx];
I1: LD R1, b[idx];
I2: MPY R2, R0, R1
```



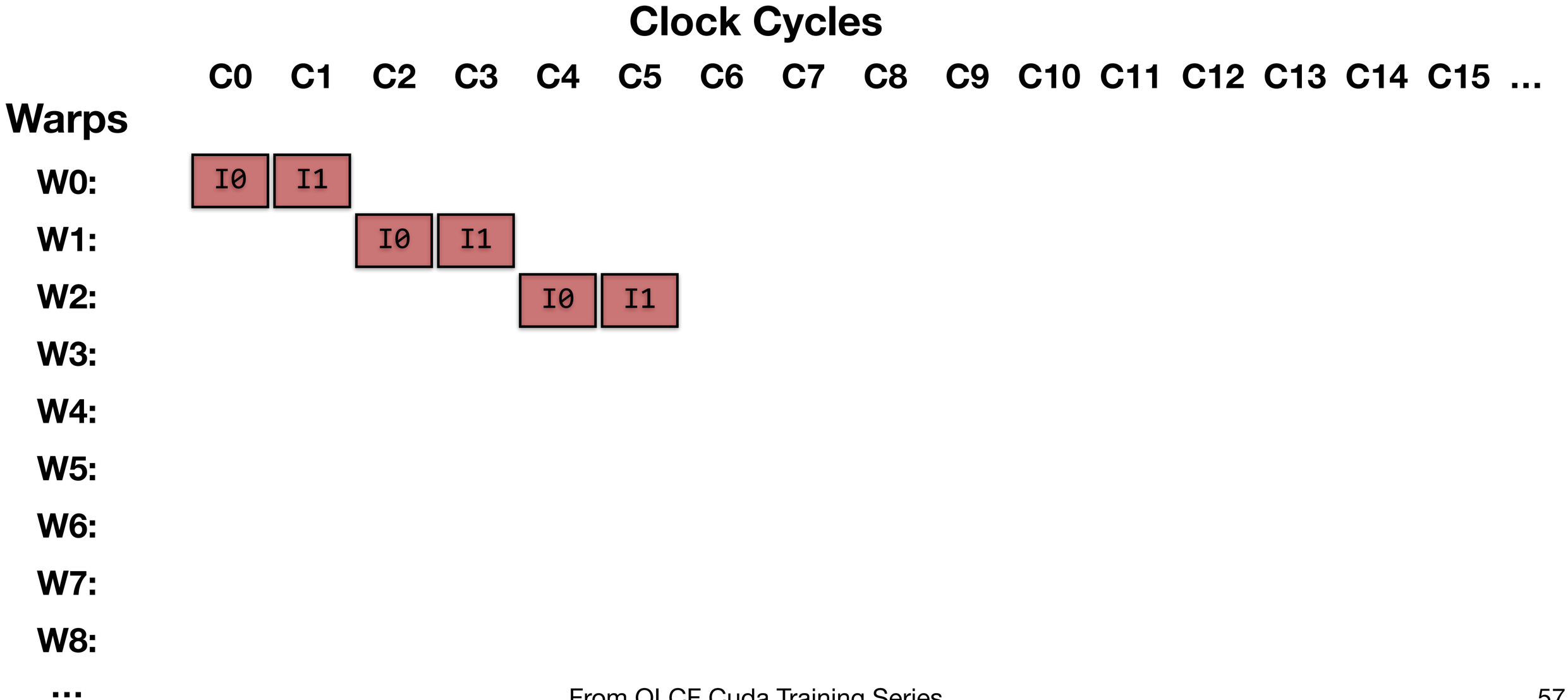
GPU Latency Hiding - Inside the SM

```
I0: LD R0, a[idx];
I1: LD R1, b[idx];
I2: MPY R2, R0, R1
```



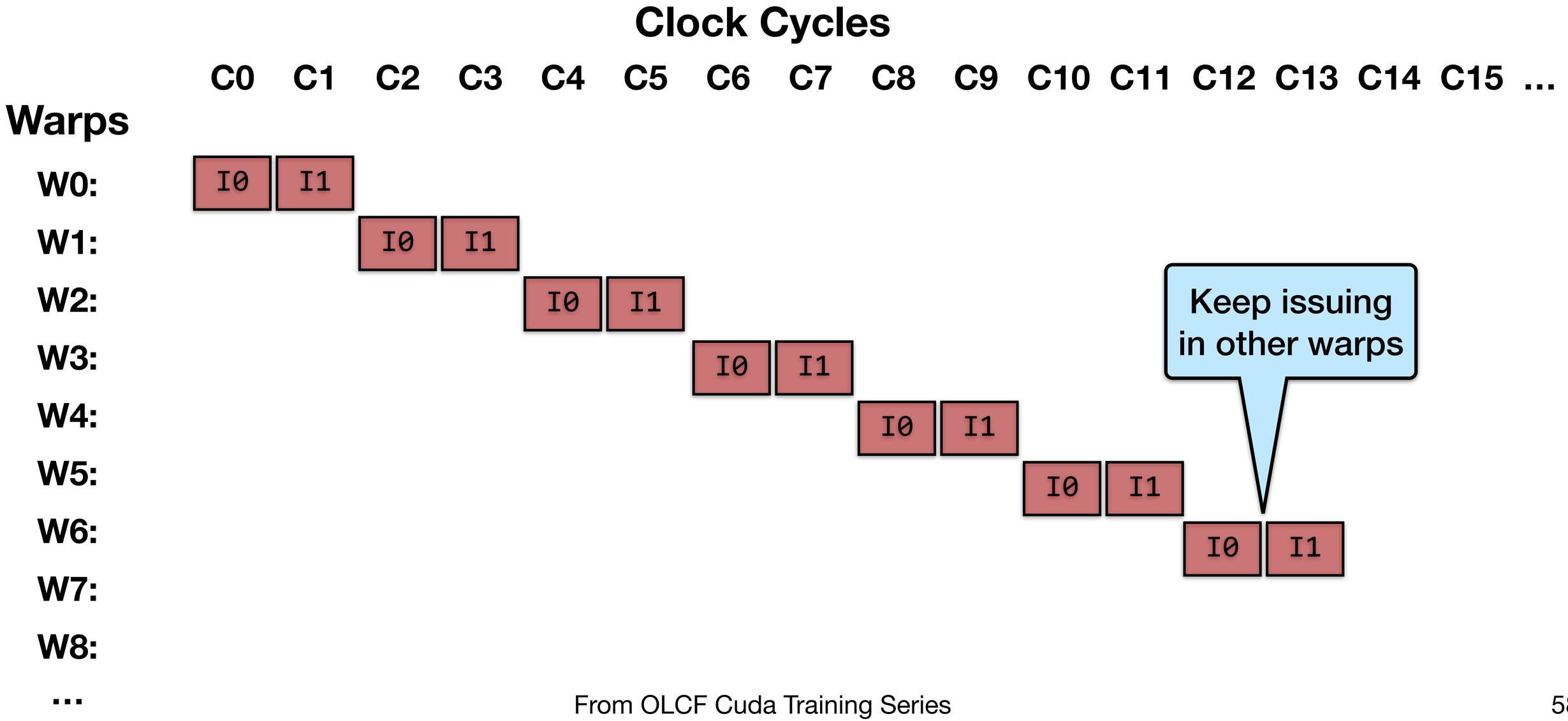
GPU Latency Hiding - Inside the SM

```
I0: LD R0, a[idx];
I1: LD R1, b[idx];
I2: MPY R2, R0, R1
```



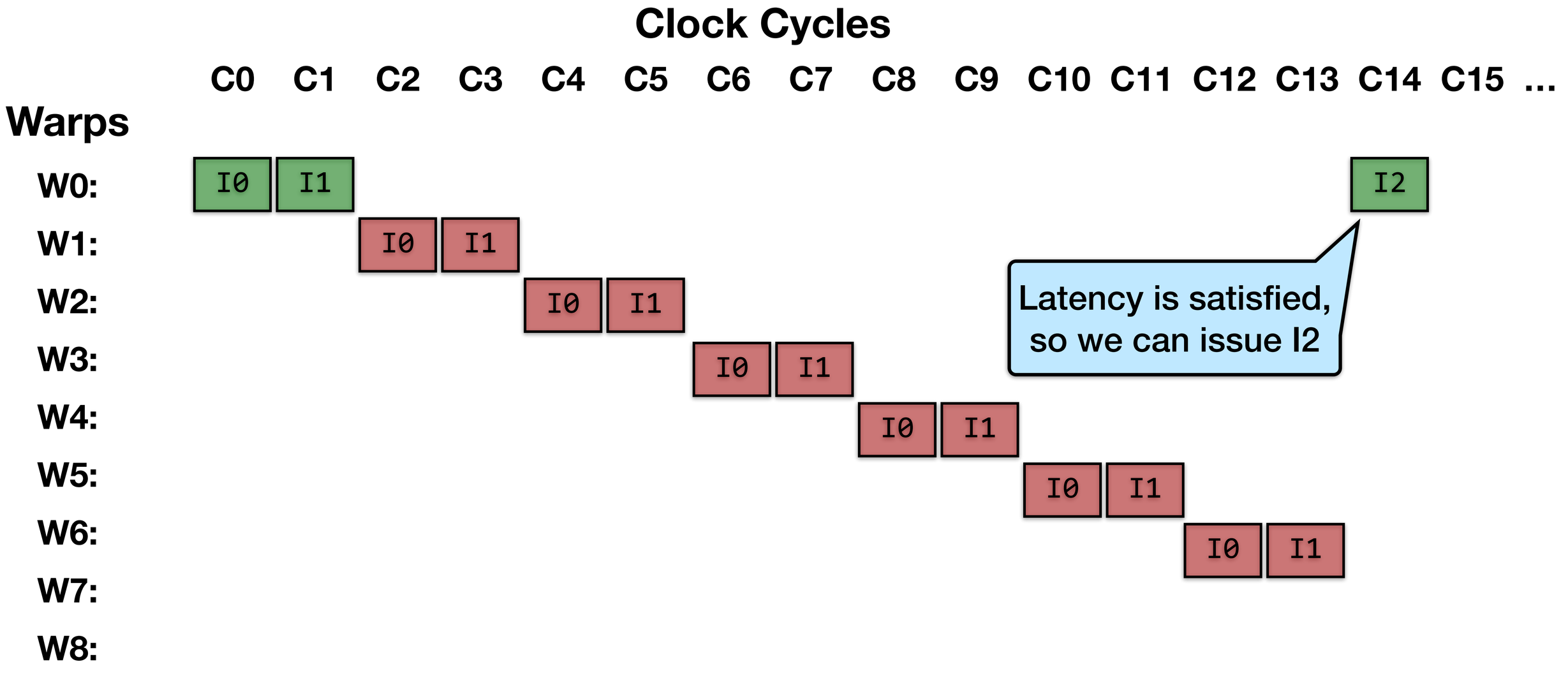
GPU Latency Hiding - Inside the SM

```
I0: LD R0, a[idx];
I1: LD R1, b[idx];
I2: MPY R2, R0, R1
```



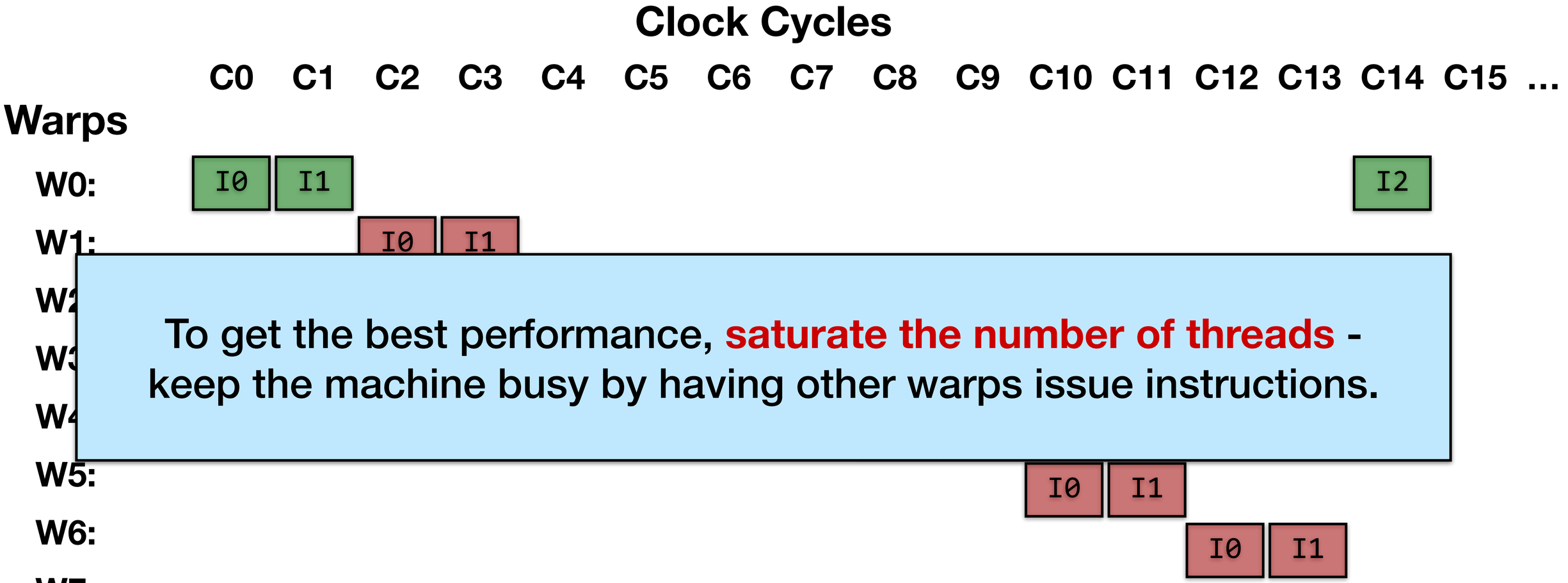
GPU Latency Hiding - Inside the SM

```
I0: LD R0, a[idx];
I1: LD R1, b[idx];
I2: MPY R2, R0, R1
```



GPU Latency Hiding - Inside the SM

```
I0: LD R0, a[idx];
I1: LD R1, b[idx];
I2: MPY R2, R0, R1
```

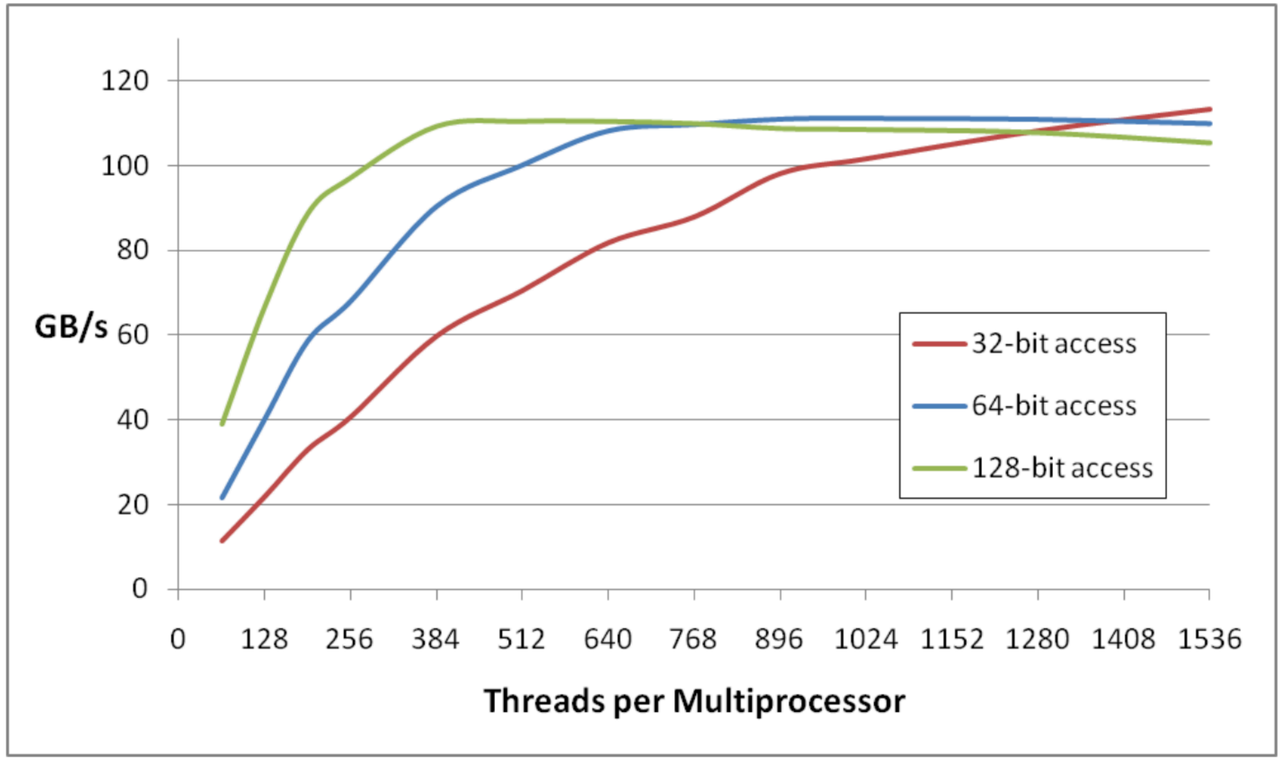


Maximizing Memory Throughput

Maximizing global memory throughput - need enough memory transactions in flight to saturate the bus

- Depends on access pattern and word size

Theoretical bandwidth: ~120 GB/s



From OLCF Cuda Training Series

Launch Configuration

Need enough total threads to keep GPU busy

- Typically, you'd like 512+ threads per SM (aim for 2048 - maximum "occupancy")
  - More if processing one fp32 element per thread

Threadblock configuration

- Threads per block should be a **multiple of warp size** (32)
- SM can concurrently execute at least 16 thread blocks (Maxwell/Pascal/Volta: 32)
  - Really small thread blocks prevent achieving good occupancy
  - Really large thread blocks are less flexible
  - Could generally use 128-256 threads/block, but use whatever is best for the application

What is Occupancy?

A measure of the **actual thread load** in an SM, vs. peak theoretical/peak achievable

CUDA includes an occupancy calculator spreadsheet

Achievable occupancy is affected by limiters to occupancy

Primary limiters:

- Registers per thread (can be reported by the profiler, or can get at compile time)
- Threads per threadblock
- Shared memory usage

Summary

- GPUs gain efficiency from simpler cores and more parallelism.
- Heterogeneous programming with manual offload - GPU for compute
- Massive (mostly data) parallelism required – Not as strict as CPU-SIMD
- Non-contiguous
- CUDA documentation: https://docs.nvidia.com/cuda/cuda-c-programming-guide/contents.html

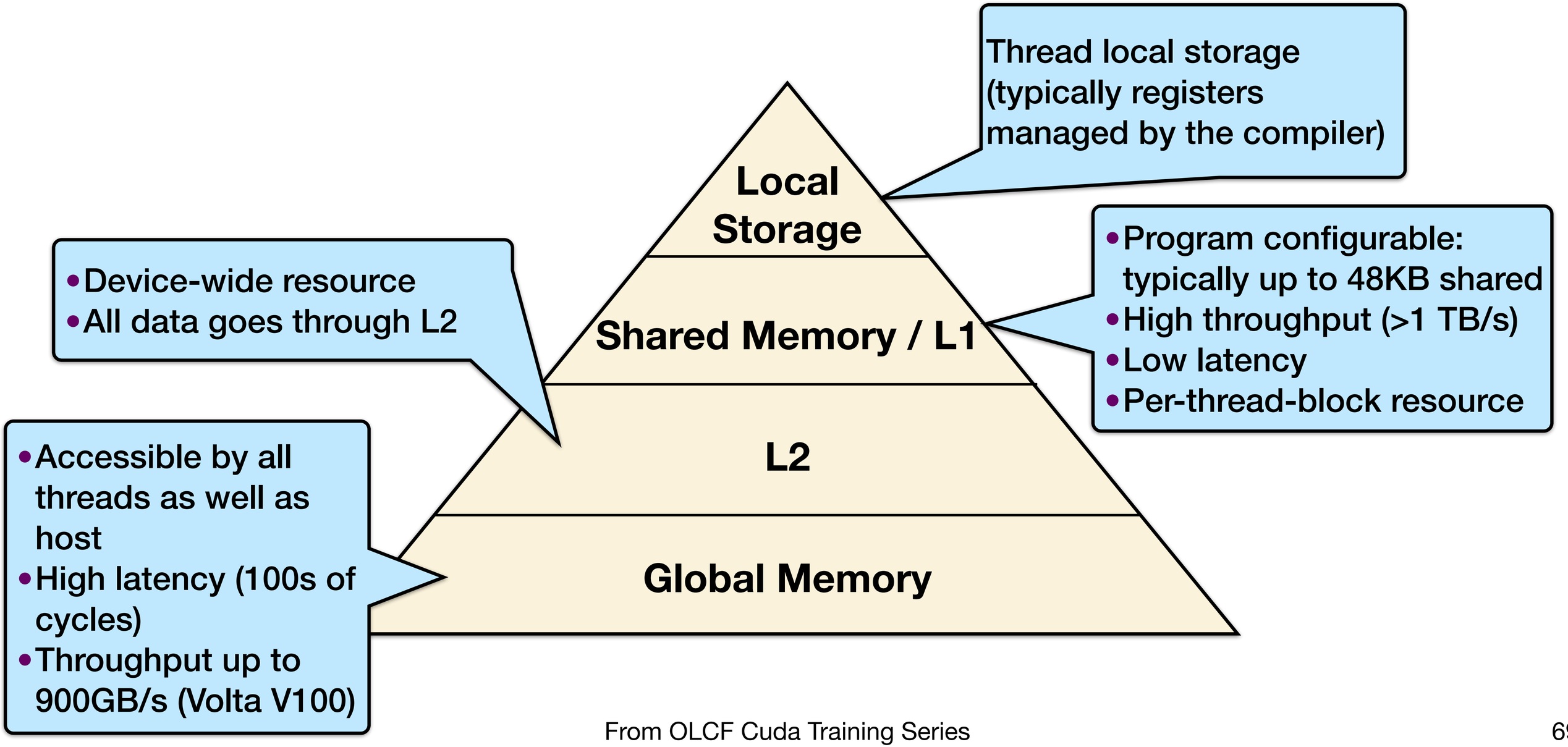
Resources

- https://www.olcf.ornl.gov/cuda-training-series/
- https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/
- https://devblogs.nvidia.com/even-easier-introduction-cuda/
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- https://docs.nvidia.com/cuda/index.html
- https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

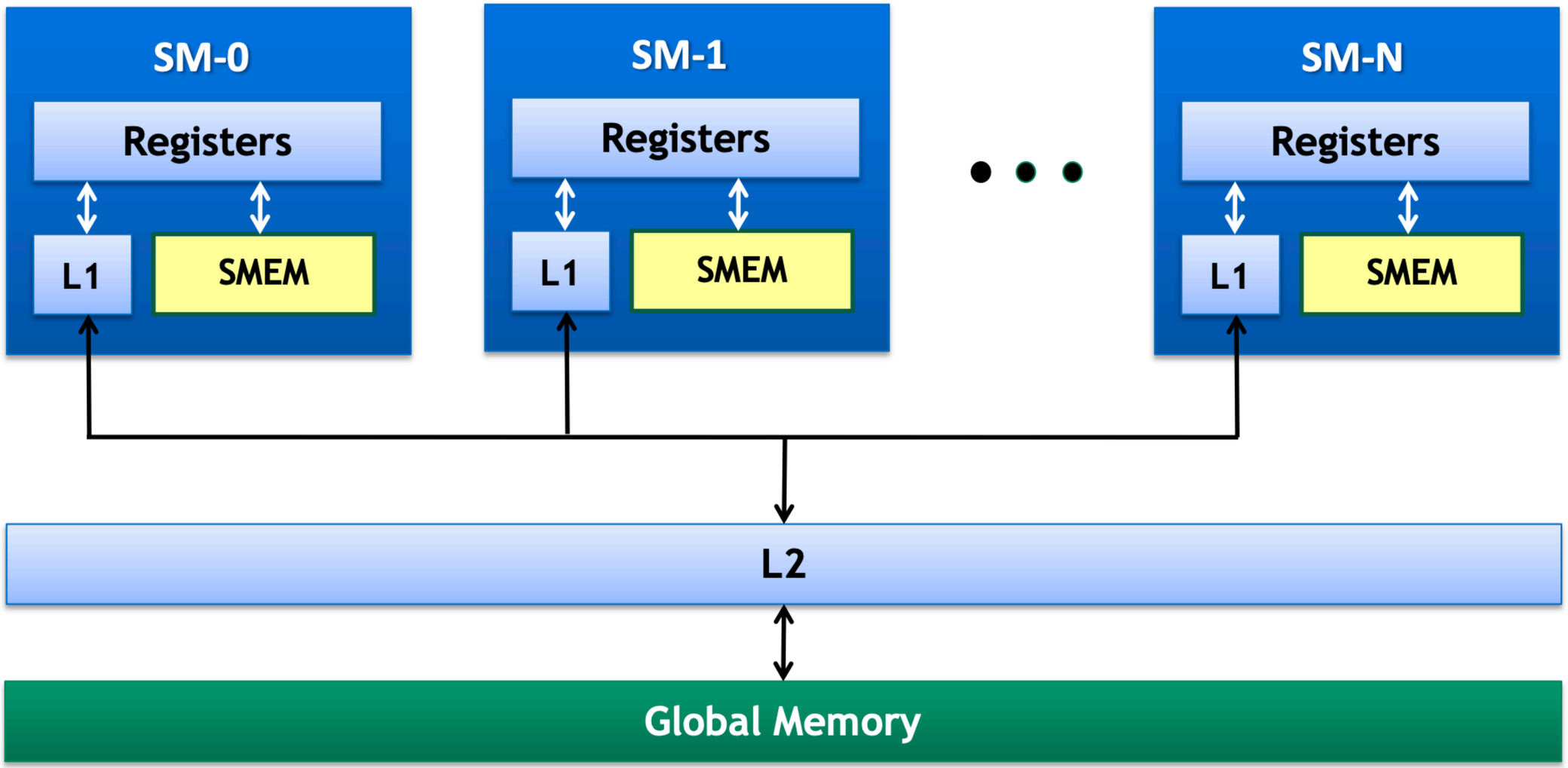
BACKUP

Optimizing for Memory Subsystem

Memory Hierarchy Review



Memory Hierarchy Review



GMEM Operations

Loads:

- Caching
  - Default mode
  - Attempts to hit in L1, then L2, then GMEM
  - Load granularity is 128-byte line

Stores:

- Invalidate L1, write-back for L2

Load Operation

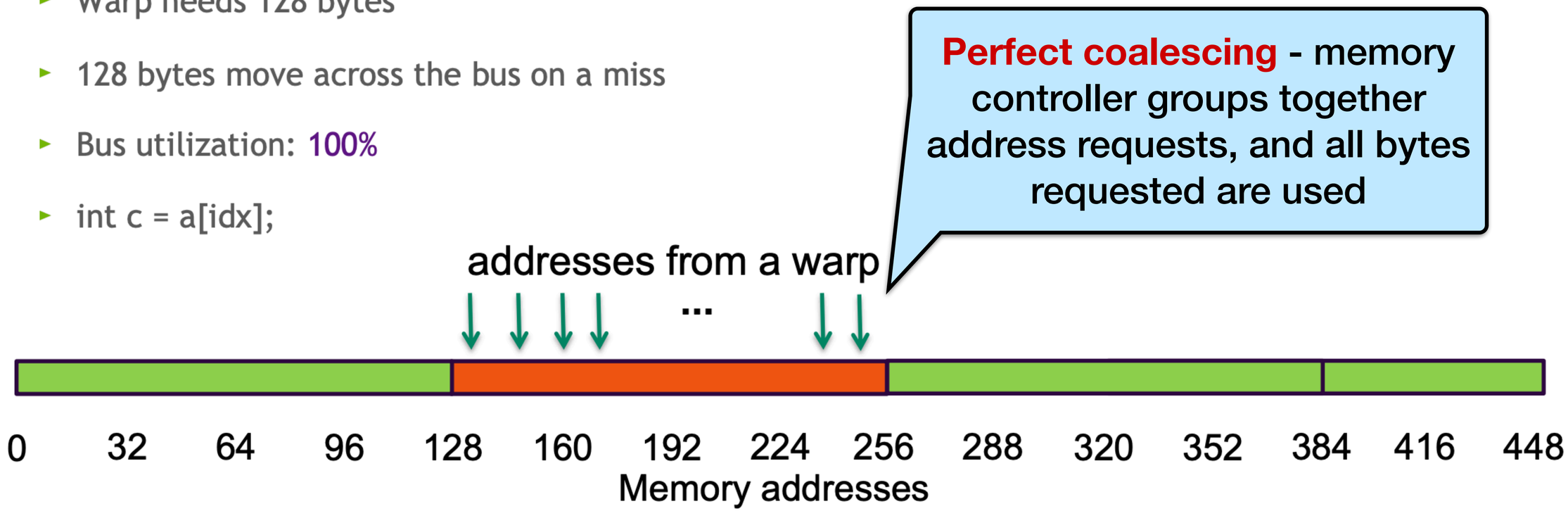
Memory operations are issued per warp (32 threads) - just like all other instructions

Operation:

- Threads in a warp provide memory addresses
- Determine which lines/segments are needed
- Request the needed lines/segments

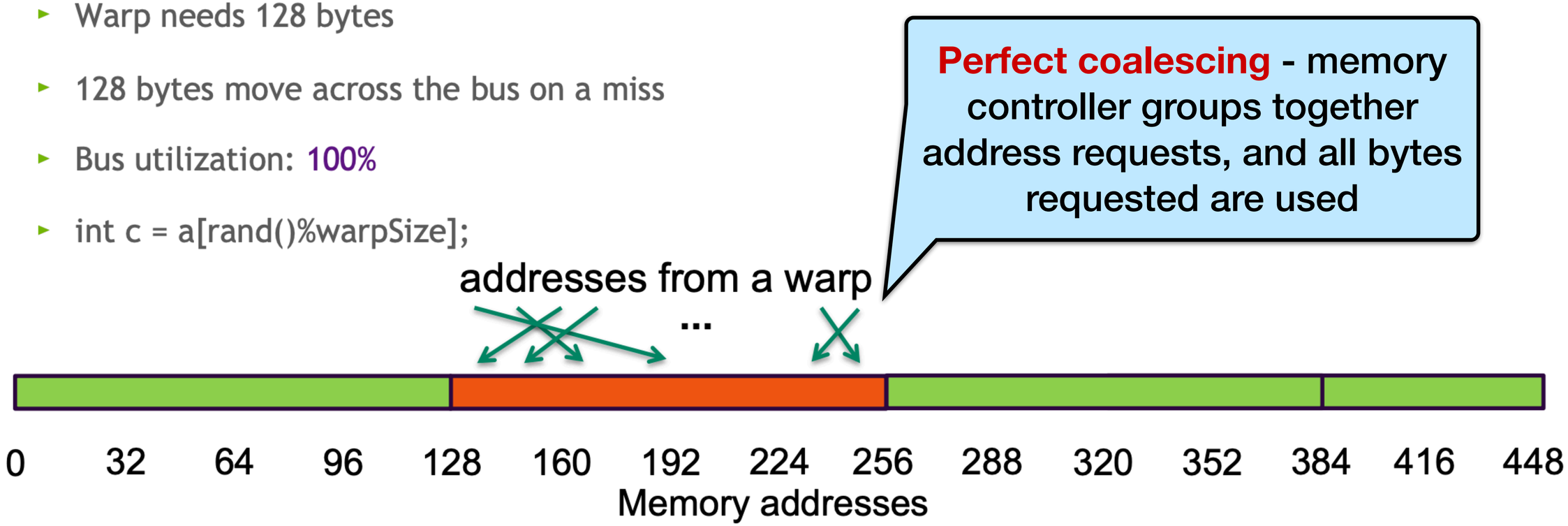
Caching Load

- Warp requests 32 aligned, consecutive 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%
  - int c = a[idx];



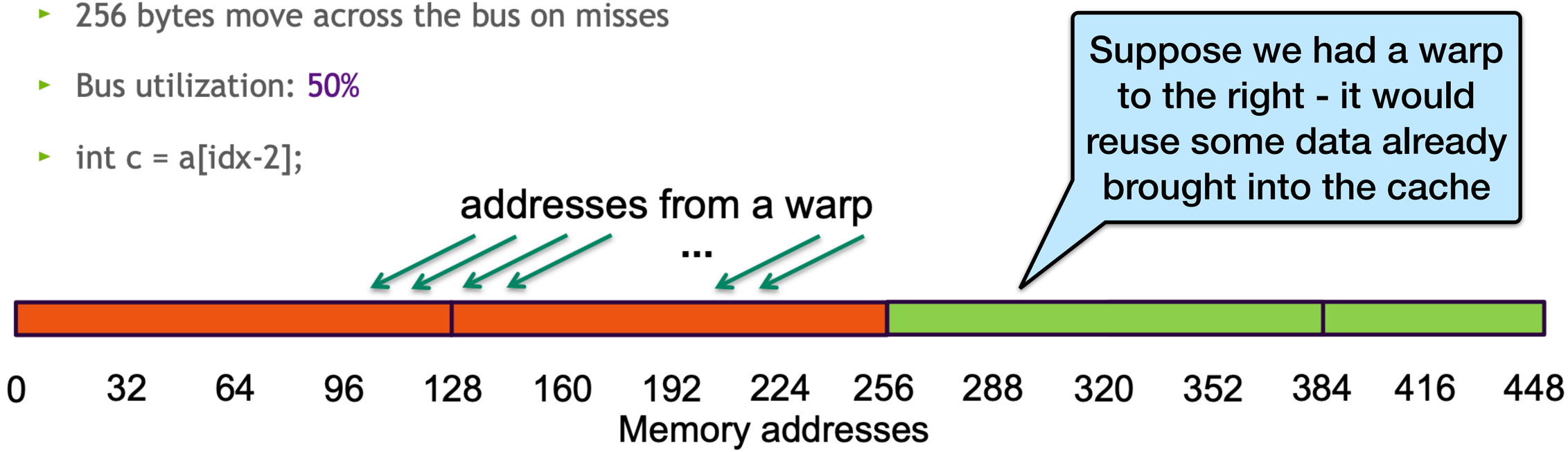
Caching Load

- Warp requests 32 aligned, permuted 4-byte words
- Addresses fall within 1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%
  - int c = a[rand()%warpSize];



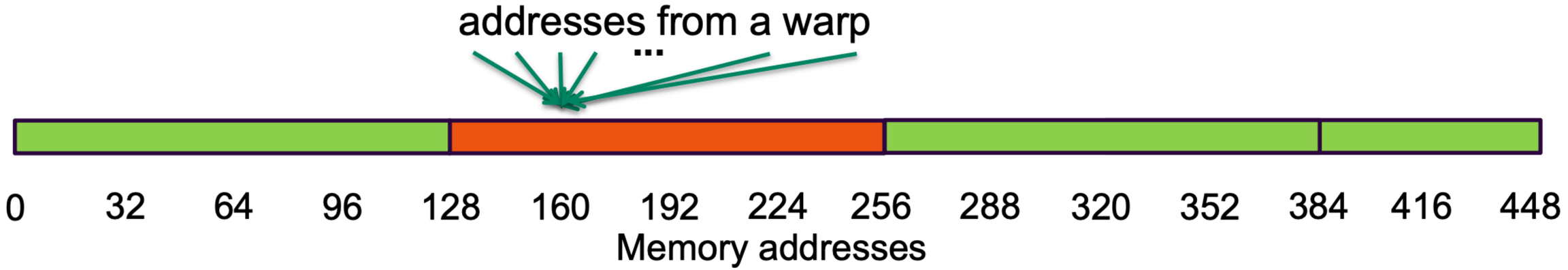
Caching Load

- Warp requests 32 misaligned, consecutive 4-byte words
- Addresses fall within 2 cache-lines
  - Warp needs 128 bytes
  - 256 bytes move across the bus on misses
  - Bus utilization: 50%
  - int c = a[idx-2];



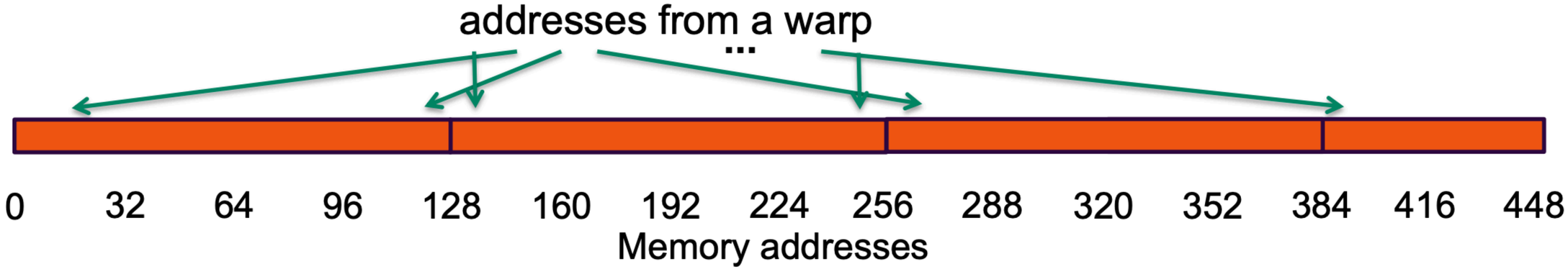
Caching Load

- All threads in a warp request the same 4-byte word
- Addresses fall within a single cache-line
  - Warp needs 4 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 3.125%
  - int c = a[40];



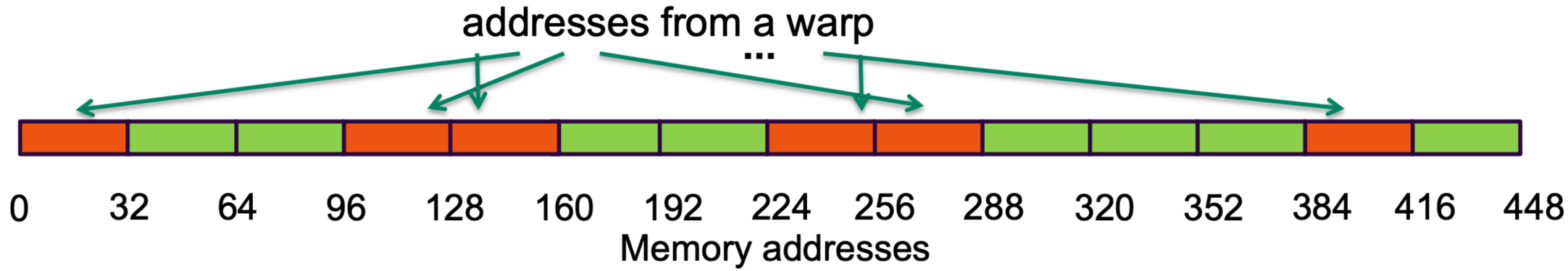
Caching Load

- 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) (3.125% worst case N=32)
  - int c = a[rand()];



Non-Caching Load

- Warp requests 32 scattered 4-byte words
- Addresses fall within N segments
  - Warp needs 128 bytes
  - N*32 bytes move across the bus on a miss
  - Bus utilization: 128 / (N*32) (12.5% worst case N = 32)
  - int c = a[rand()]; -Xptxas -dlcm=cg



Summary

- GPUs gain efficiency from simpler cores and more parallelism –
- Heterogeneous programming with manual offload - GPU for compute
- Massive (mostly data) parallelism required – Not as strict as CPU-SIMD
- Coalescing data accesses is necessary for performance
- CUDA documentation: https://docs.nvidia.com/cuda/cuda-c-programming-guide/contents.html

Launch Configuration

Hiding arithmetic latency:

- Need ~10's warps (~320 threads) per SM
- Or, latency can also be hidden with independent instructions from the same warp
  - ->if instructions never depends on the output of preceding instruction, then only 5 warps are needed, etc.

Maximizing global memory throughput:

- Depends on the access pattern, and word size
- Need enough memory transactions in flight to saturate the bus
  - Independent loads and stores from the same thread
  - Loads and stores from different threads
  - Larger word sizes can also help (float2 is twice the transactions of float, for example)

BACKUP SLIDES

Shared Memory

Uses:

- Inter-thread communication within a block
- Cache data to reduce redundant global memory accesses
- Use it to improve global memory access patterns

Organization: 32 banks, 4-byte wide banks

- Successive 4-byte words belong to different banks

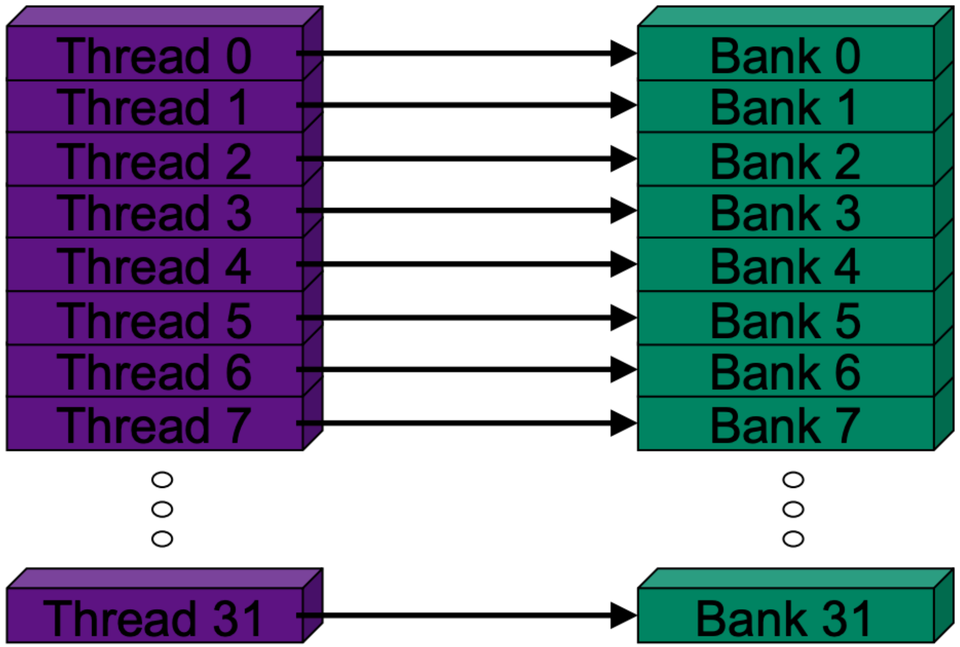
Shared Memory

Performance:

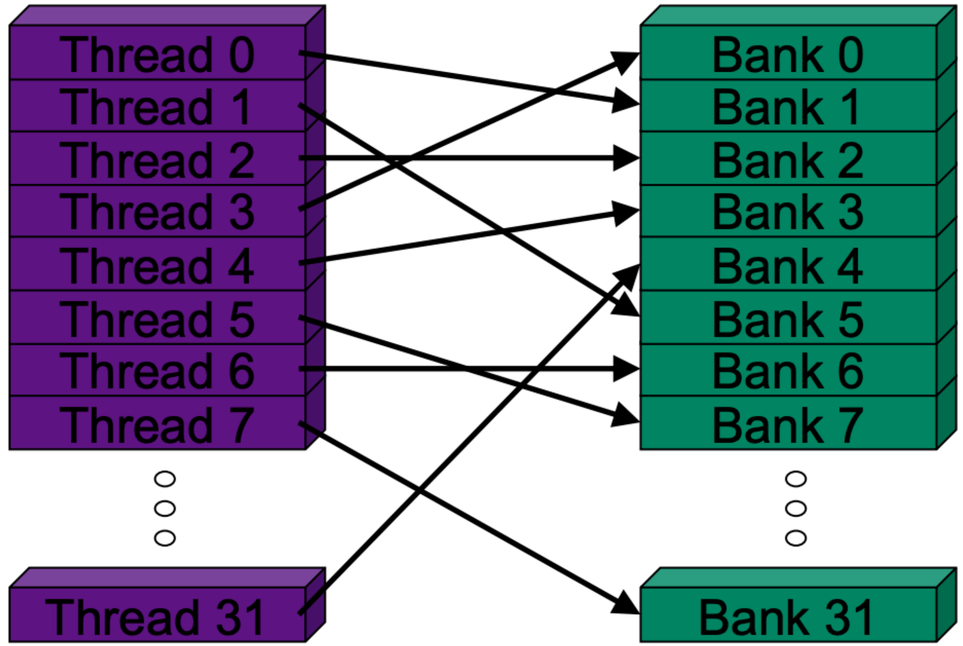
- Typically: 4 bytes per bank per 1 or 2 clocks per multiprocessor
- shared accesses are issued per 32 threads (warp)
- serialization: if N threads of 32 access different 4-byte words in the same bank, N accesses are executed serially
- multicast: N threads access the same word in one fetch
  - Could be different bytes within the same word

Bank Addressing Examples

No Bank Conflicts

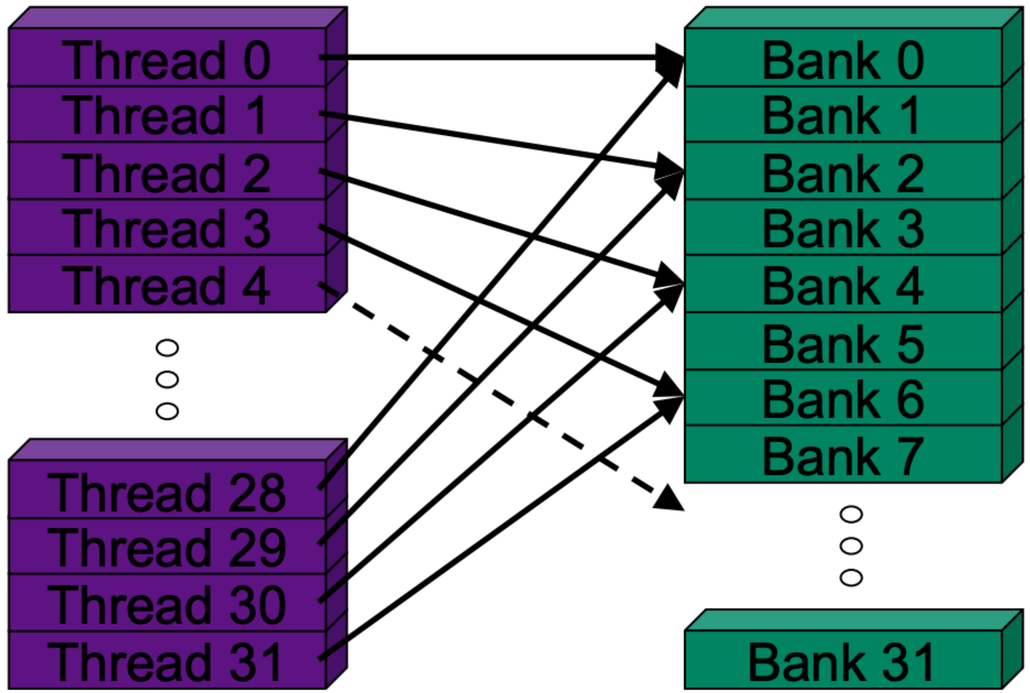


No Bank Conflicts

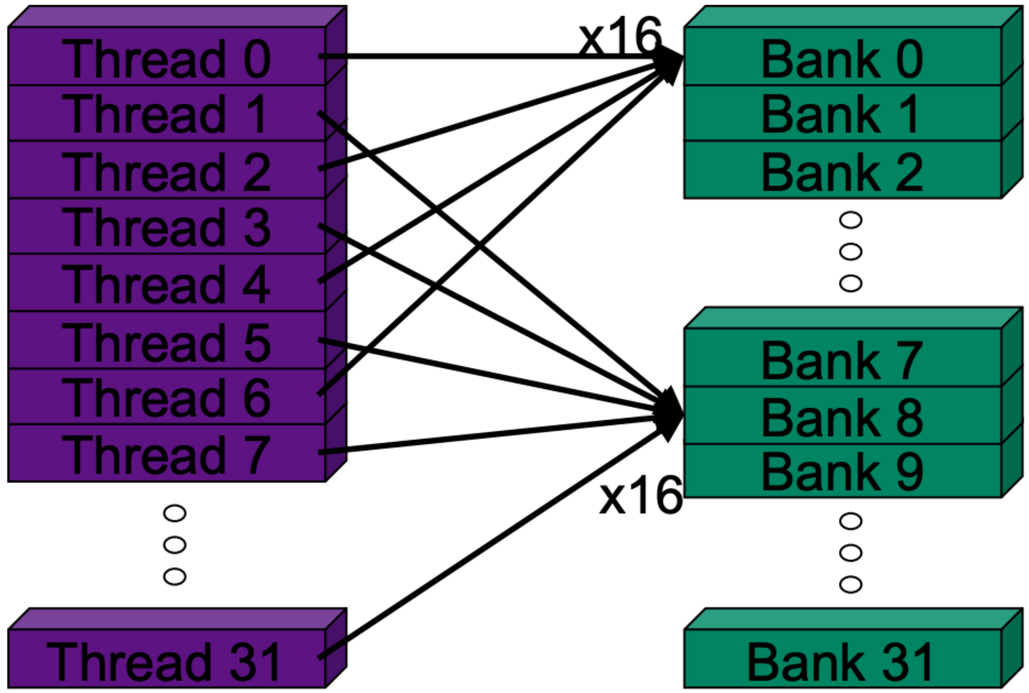


Bank Addressing Examples

2-way Bank Conflicts

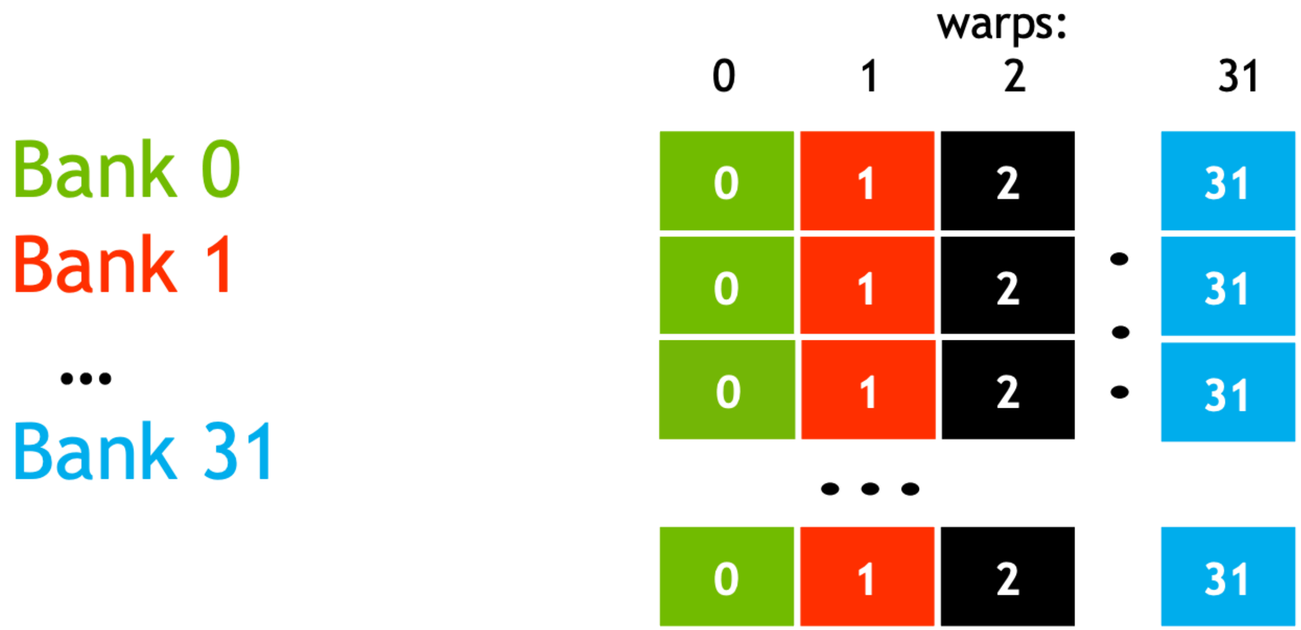


16-way Bank Conflicts



Shared Memory: Avoiding Bank Conflicts

- 32x32 SMEM array
- Warp accesses a column:
  - 32-way bank conflicts (threads in a warp access the same bank



Shared Memory: Avoiding Bank Conflicts

- Add a column for padding:
  - 32x33 SMEM array
- Warp accesses a column:
  - 32 different banks, no bank conflicts

Bank 0
Bank 1

Bank 31