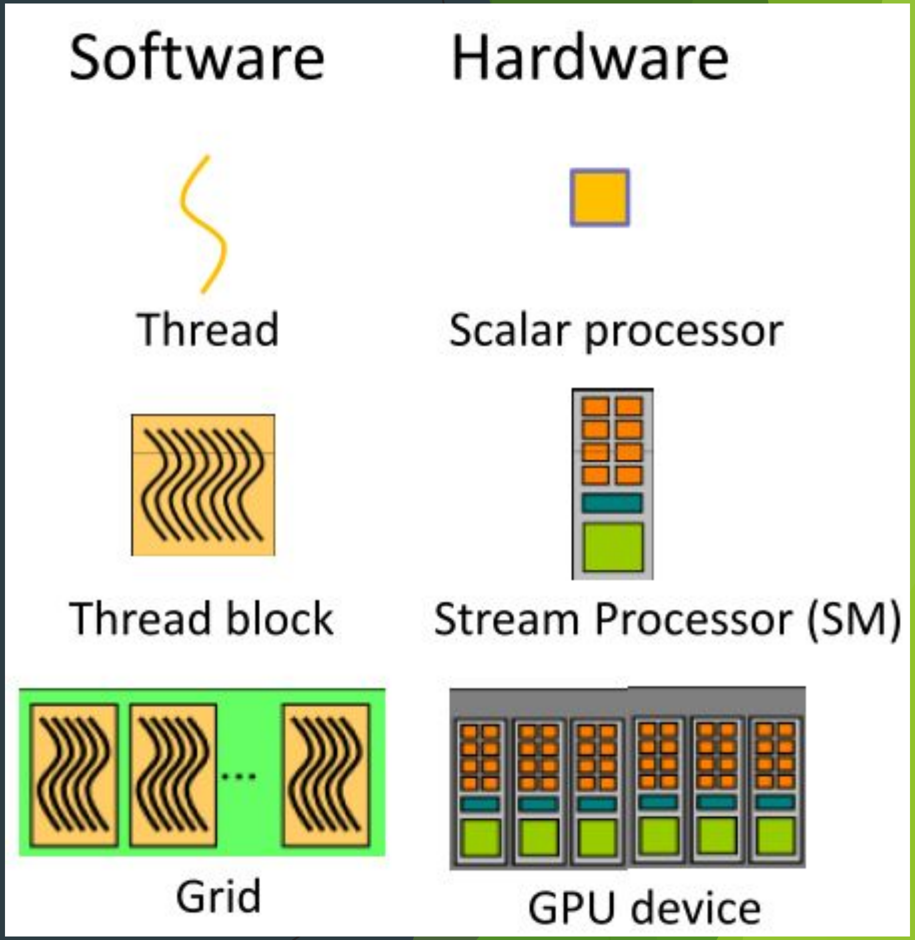
GUDA Introduction II

林政宏

國立台灣師範大學電機系

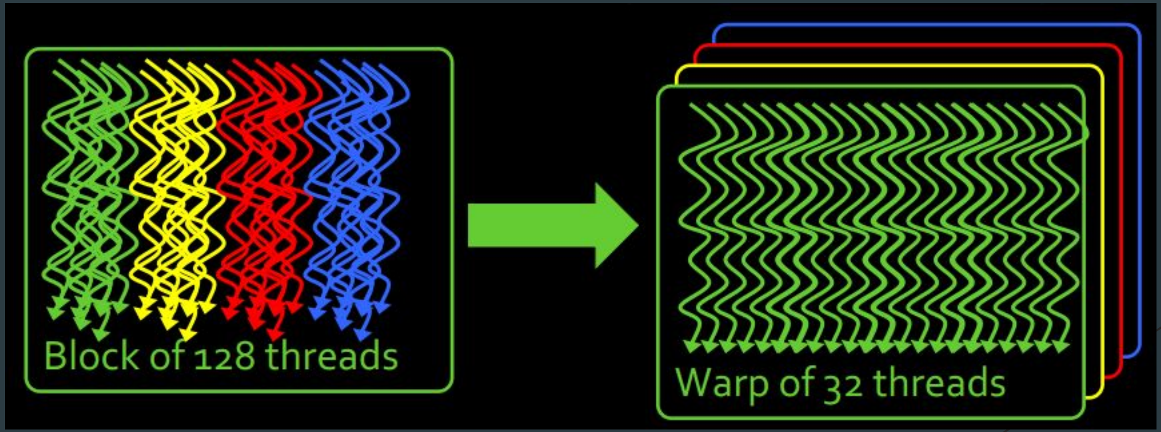
Execution Model

- Threads are executed by scalar processor
- Thread blocks are executed on SM
  - Several concurrent thread block can reside on one SM
- A kernel is launched as a grid of thread blocks



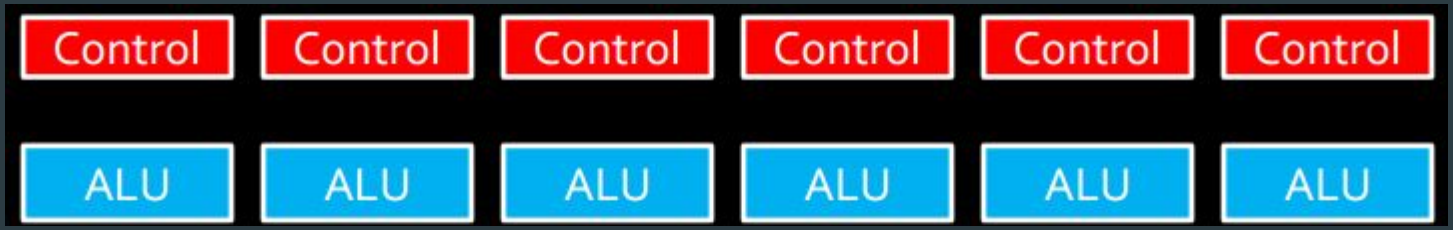
Thread Blocks are Executed as Warps

- Each thread block is mapped to one or more warps
  - When the thread block size is not a multiple of the warp size, unused threads within the last warp are disabled automatically
- The hardware schedules each warp independently
  - Warps within a thread block can execute independently

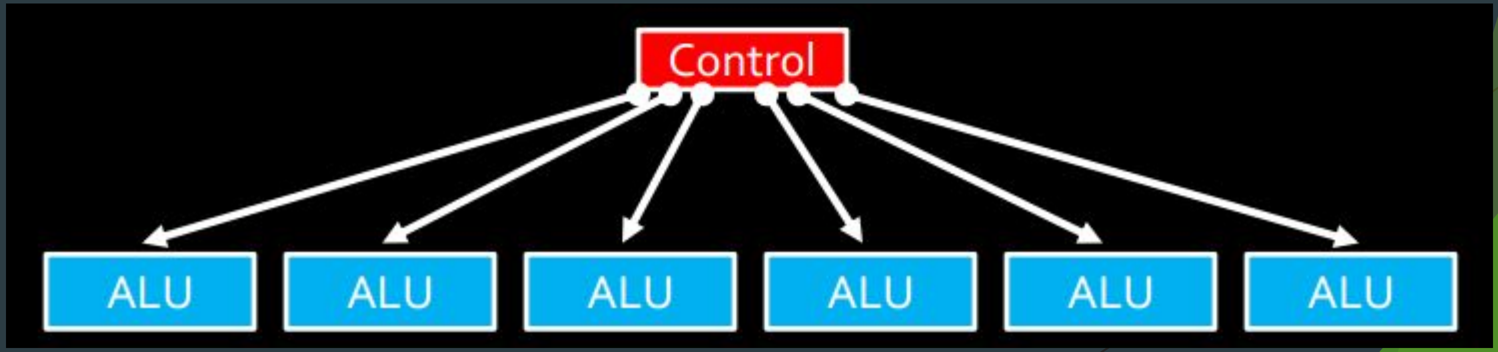


Warps and SIMT

- A warp is a group of threads within a block that are launched together and (usually) execute together
- Conceptual Programming Model



- Conceptual SIMT Execution Model



Warps and SIMT

- SIMT = Single Instruction Multiple Threads
  - Within CUDA context, refers to issuing a single instruction to the (multiple) threads in a warp.
- The warp size is currently 32 threads
- The warp size could change in future GPUs

Filling Warps

- Prefer thread block sizes that result in mostly full warps
  - Bad: kernel<<<N, 1>>> (...)
  - Okay: kernel<<<N/32, 32>>>(...)
  - Better: kernel<<<N/128, 128>>(...)

- Prefer to have enough threads per block to provide hardware with many warps to switch between
  - This is how the GPU hides memory access latency
  - 128 or 256 threads per block are good choices

Filling Warps

- When number of threads is not a multiple of preferred block size, insert bounds test into kernel

```
__global__ void kn( int n, int* x) {
    int i= threadIdx.x + blockIdx.x * blockDim.x;
    if (i < n) {
        //very important
    }
}
```

- Otherwise, threads may access memory outside of arrays
- Do not launch a second grid to process residual elements 12

```
kernel<<<n/128, 128>>>(...);
kernel<<<1, n%128>>>(...); // !!! very bad !!!
kernel<<<(n+127)/128, 128>>>(...); // !!! better!!!
```

Control Flow Divergence

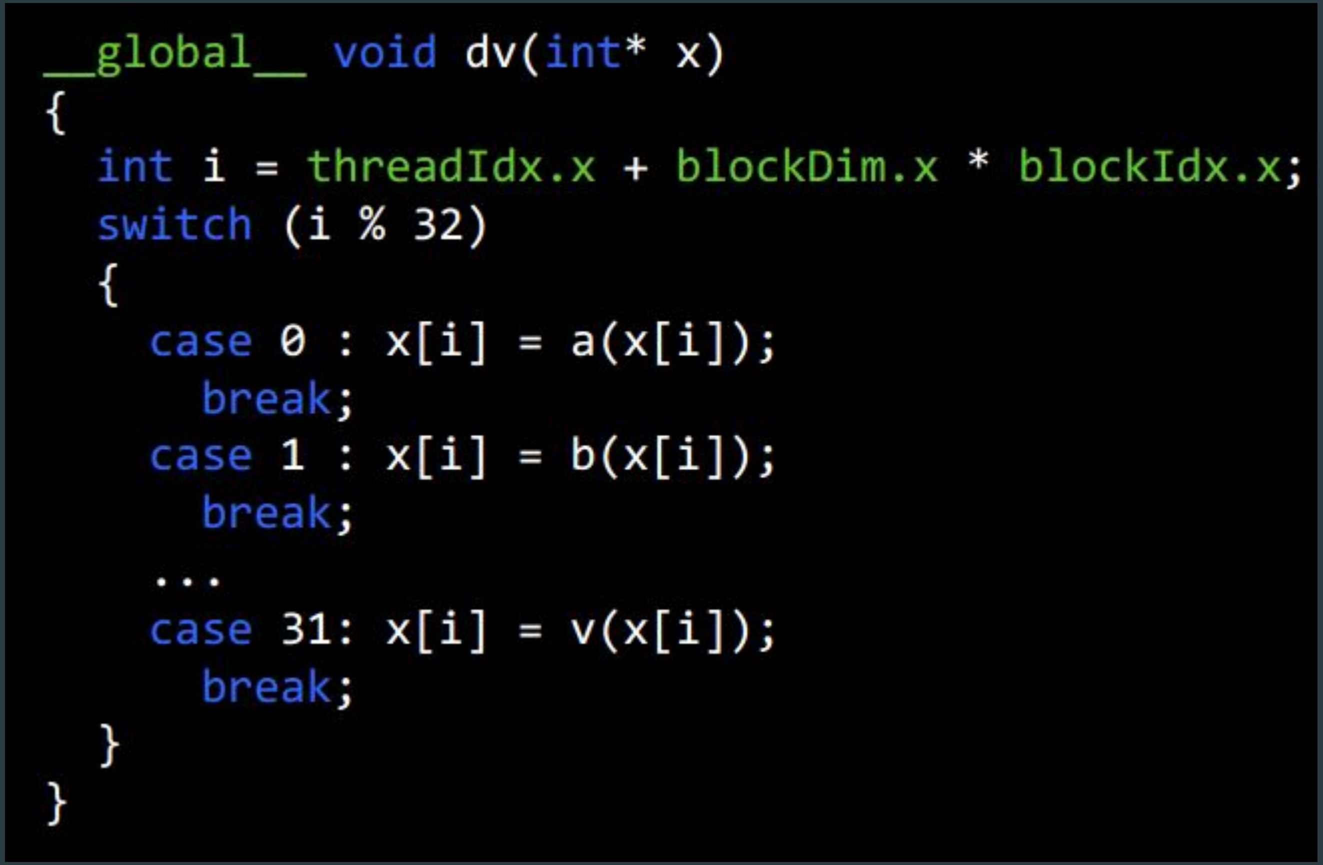
- Consider the following code

```
__global__ void odd_even(int n, int* x) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if ((i & 0x01) == 0) {
        x[i] = x[i] + 1;
    }
    else {
        x[i] = x[i] + 2;
    }
}
```

- Half the threads in the warp must execute the if clause, the other half the else clause

Performance of Divergent Code

- Performance decreases with degree of divergence in warps



```
__global__ void dv(int* x)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  switch (i % 32)
  {
    case 0 : x[i] = a(x[i]);
      break;
    case 1 : x[i] = b(x[i]);
      break;
    ...
    case 31: x[i] = v(x[i]);
      break;
  }
}
```

Performance of Divergence

- Compiler and hardware can detect when all threads in a warp branch in the same direction
  - For example, all take the if clause, or all take the else clause
  - The hardware is optimized to handle these cases without loss of performance
- The compiler can also compile short conditional clauses to use predicates (bits that conditional convert instructions into null ops)
  - Avoids some branch divergence overheads, and is more efficient
  - Often acceptable performance with short conditional clauses

Data Address Divergence

- Concept is similar to control divergence and often conflated
- Hardware is optimized for accessing contiguous blocks of global memory when performing loads and stores
  - Global memory blocks are aligned to multiples of 32,64,128 bytes
  - If requests from a warp span multiple data blocks, multiple data blocks will be fetched from memory, called memory coalesce.
  - Entire block is fetched even if only a single byte is accesses, which can waste bandwidth
- Hardware handles divergence within __shared__ memory more efficiently
  - Designed to support parallel accesses from all threads in warp
  - Still need to worry about addresses that map to the same bank

Data Address Divergence

- Hardware may need to issue multiple loads and stores when a warp accesses addresses that are far apart
  - Conceptually similar to executing the load or store multiple times

- Global memory accesses are most efficient when all load and store addresses generated within a warp are within the same memory block
  - For example, when addresses of loads and stores have stride 1 within a warp
  - Common when array index is a linear function of threadIdx.x

- Consider both address and control divergence when designing algorithms and optimizing code

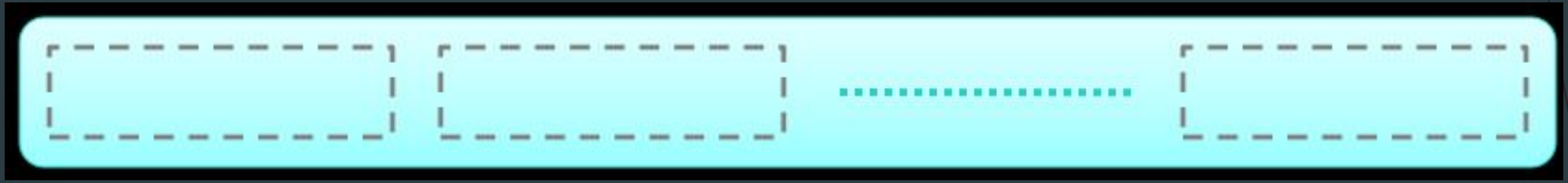
Blocking

- Partition data to operate in well-sized blocks
  - Small enough to be staged in shared memory
  - Assign each data partition to a thread block
  - No different from cache blocking!

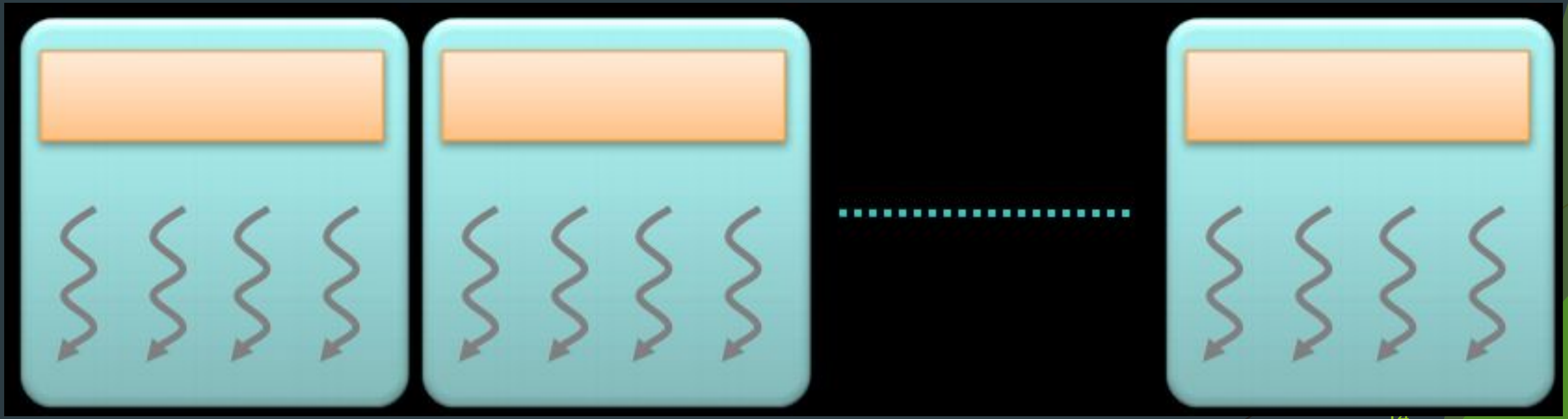
- Provides several significant performance benefits
  - Have enough thread blocks to keep processors busy
  - Working in shared memory reduces memory latency dramatically
  - More likely to have address access patterns that coalesce well on load/store to shared memory

Blocking

- Partition data into subsets that fit into __shared__ memory

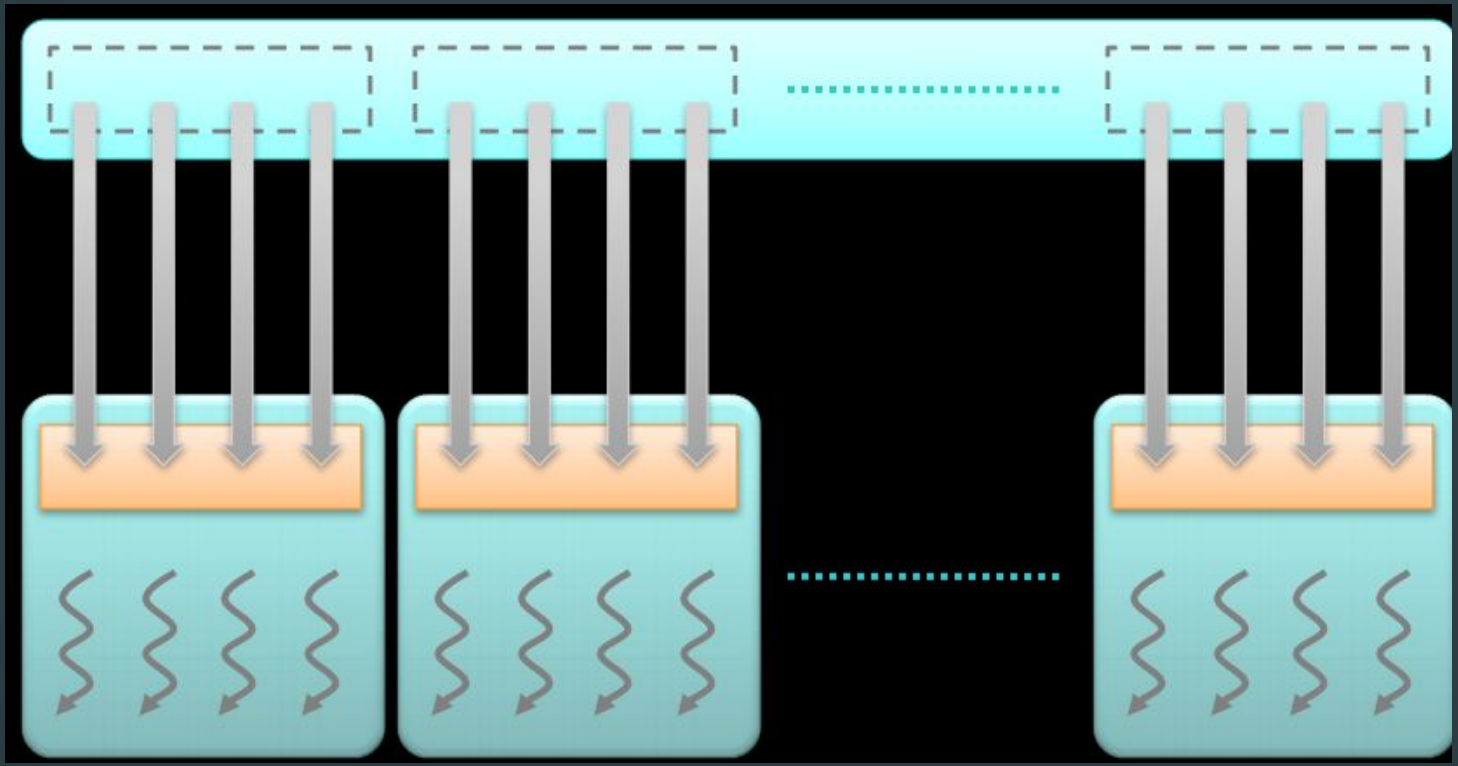


- Process each data subset with one thread block



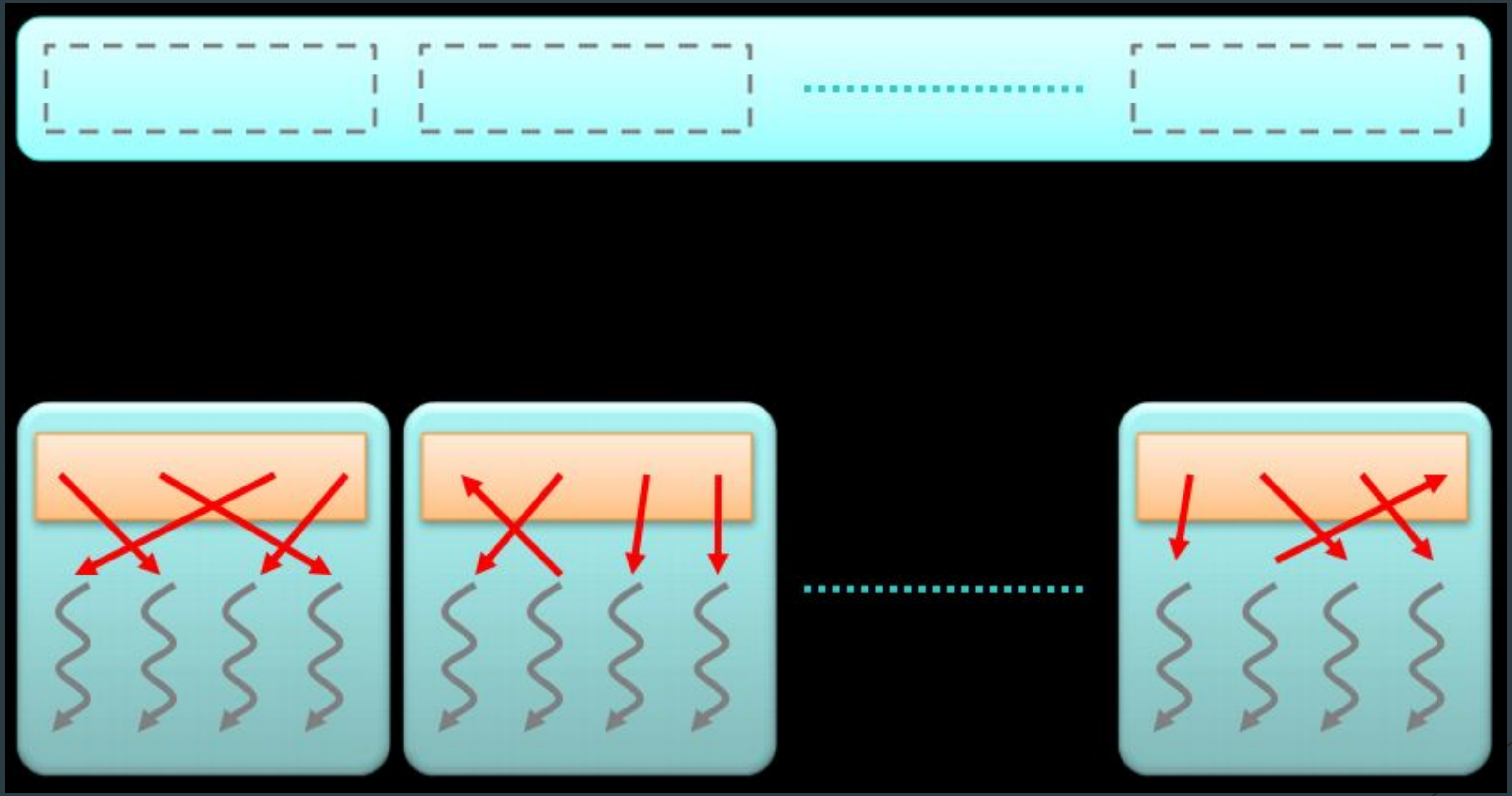
Blocking

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



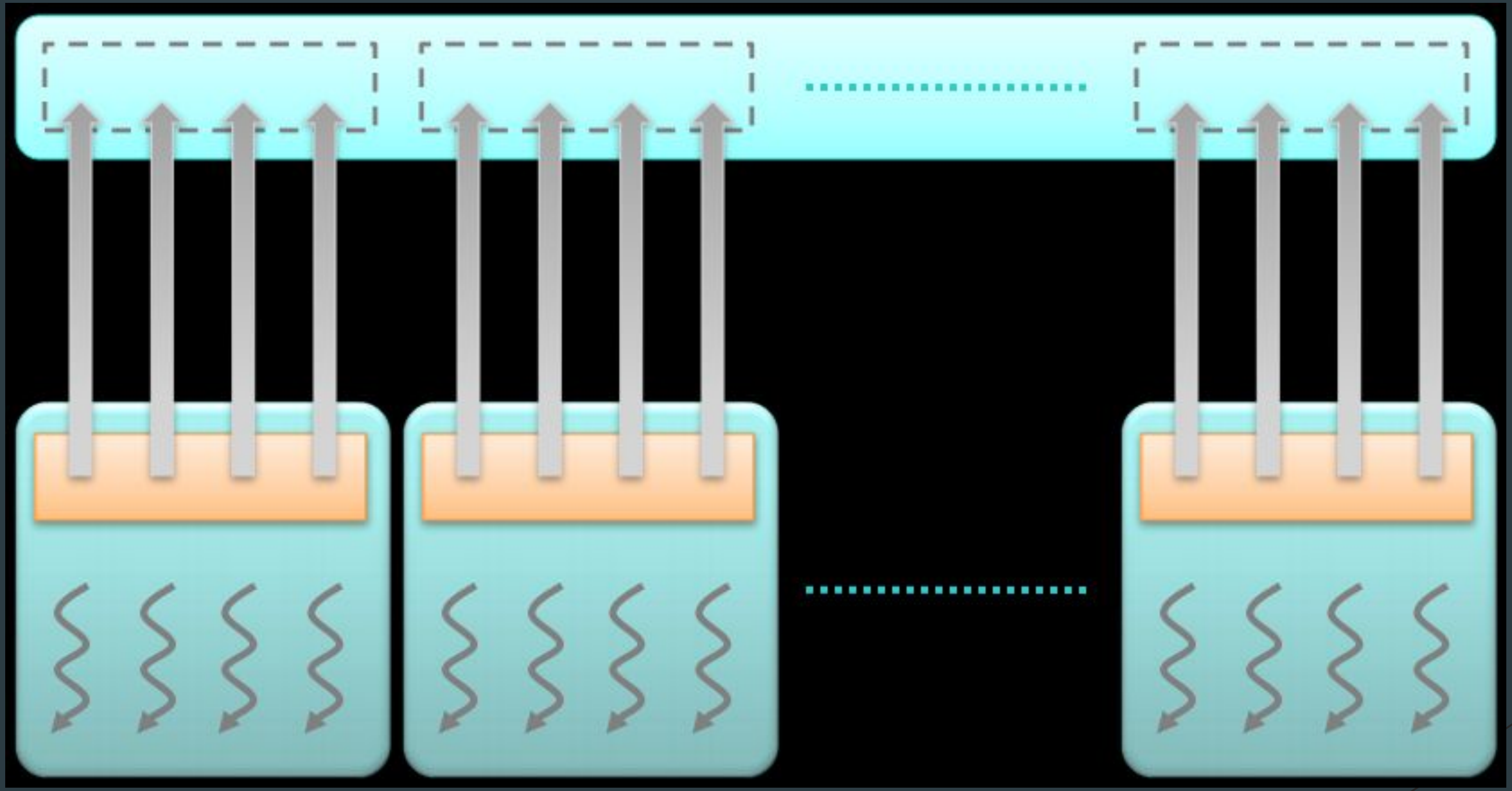
Blocking

- Perform the computation on the subset from shared memory



Blocking

- Copy the result from __shared__ memory back to global memory



Blocking

- Almost all CUDA kernels are built this way
  - Blocking may not impact the performance of a particular problem, but one is still forced to think about it
  - Not all kernels require __shared__ memory
  - All kernels do require registers

Synchronization

- Communication
- Race conditions
- Synchronizing accesses to shared data

Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using __shared__, allocated per block
- Data is not visible to threads in other blocks

__syncthreads()

- void __syncthreads();
- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

Global Communication

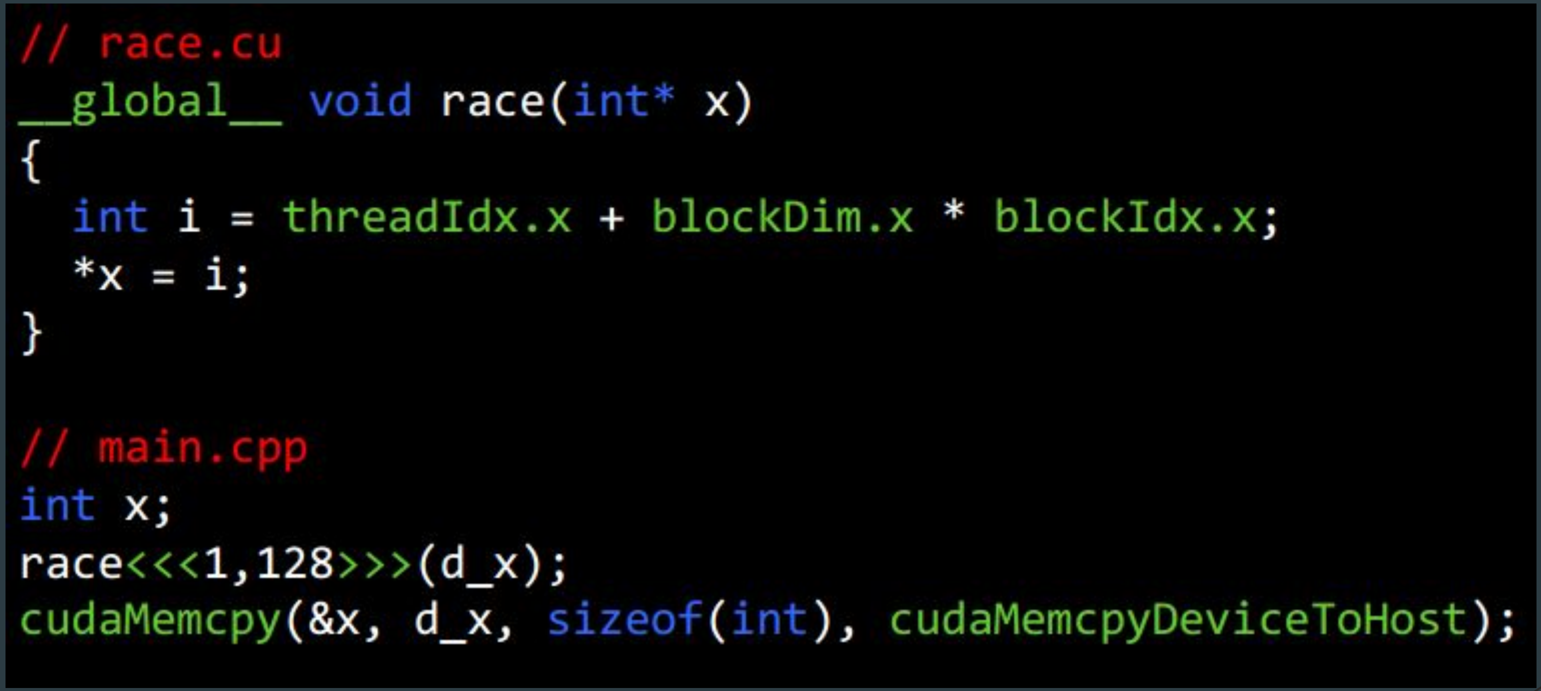
- Device threads communicate through shared memory locations

- Threads in different blocks and different grids
  - Locations in global memory (global variables)

- Threads in same blocks
  - Locations in global memory
  - Locations in shared memory (__shared__ variables)

Race Conditions

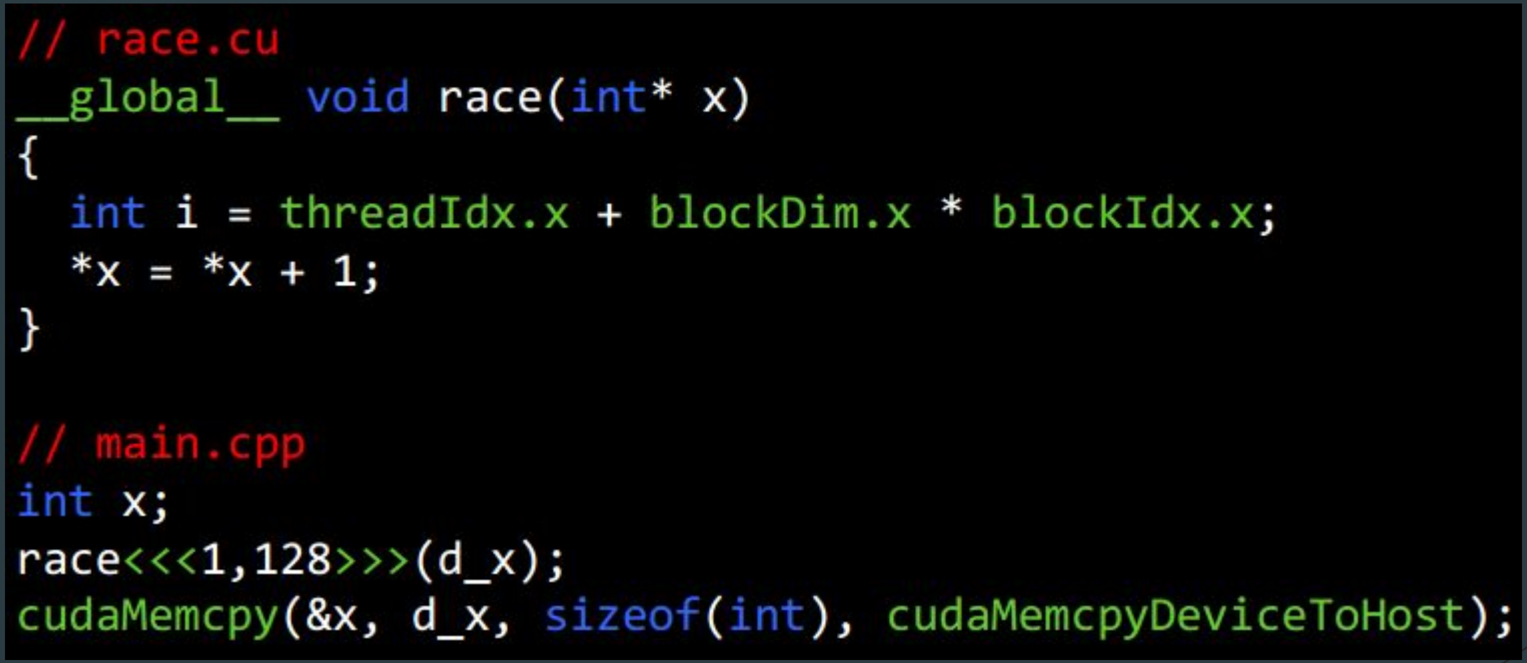
- Race conditions arise when 2+ threads attempt to access the same memory location concurrently and at least one access is a write.



```
// race.cu
__global__ void race(int* x)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  *x = i;
}

// main.cpp
int x;
race<<<1,128>>>(d_x);
cudaMemcpy(&x, d_x, sizeof(int), cudaMemcpyDeviceToHost);
```

Race Conditions

- Programs with race conditions may produce unexpected, seemingly arbitrary results
  - Updates may be missed, and updates may be lost



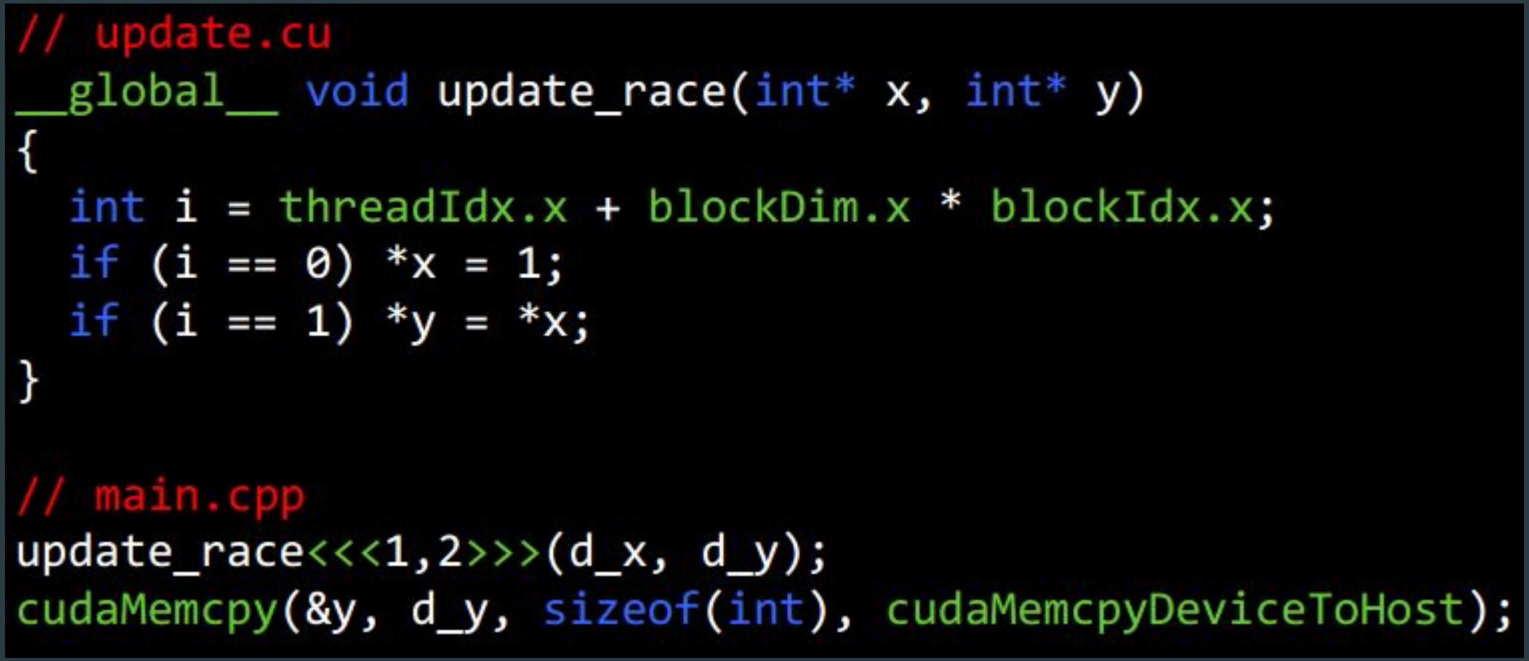
```
// race.cu
__global__ void race(int* x)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  *x = *x + 1;
}

// main.cpp
int x;
race<<<1,128>>>(d_x);
cudaMemcpy(&x, d_x, sizeof(int), cudaMemcpyDeviceToHost);
```

Synchronization

- Accesses to shared locations need to be correctly synchronized (coordinated) to avoid race conditions
- In many common shared memory multithreaded programming models, one uses coordination objects such as locks to synchronize accesses to shared data
- CUDA provides several scalable synchronization mechanisms, such as efficient barriers and atomic memory operations.
- In general, always most efficient to design algorithms to avoid synchronization whenever possible.

Synchronization

- Assume thread T1 reads a value defined by thread T0

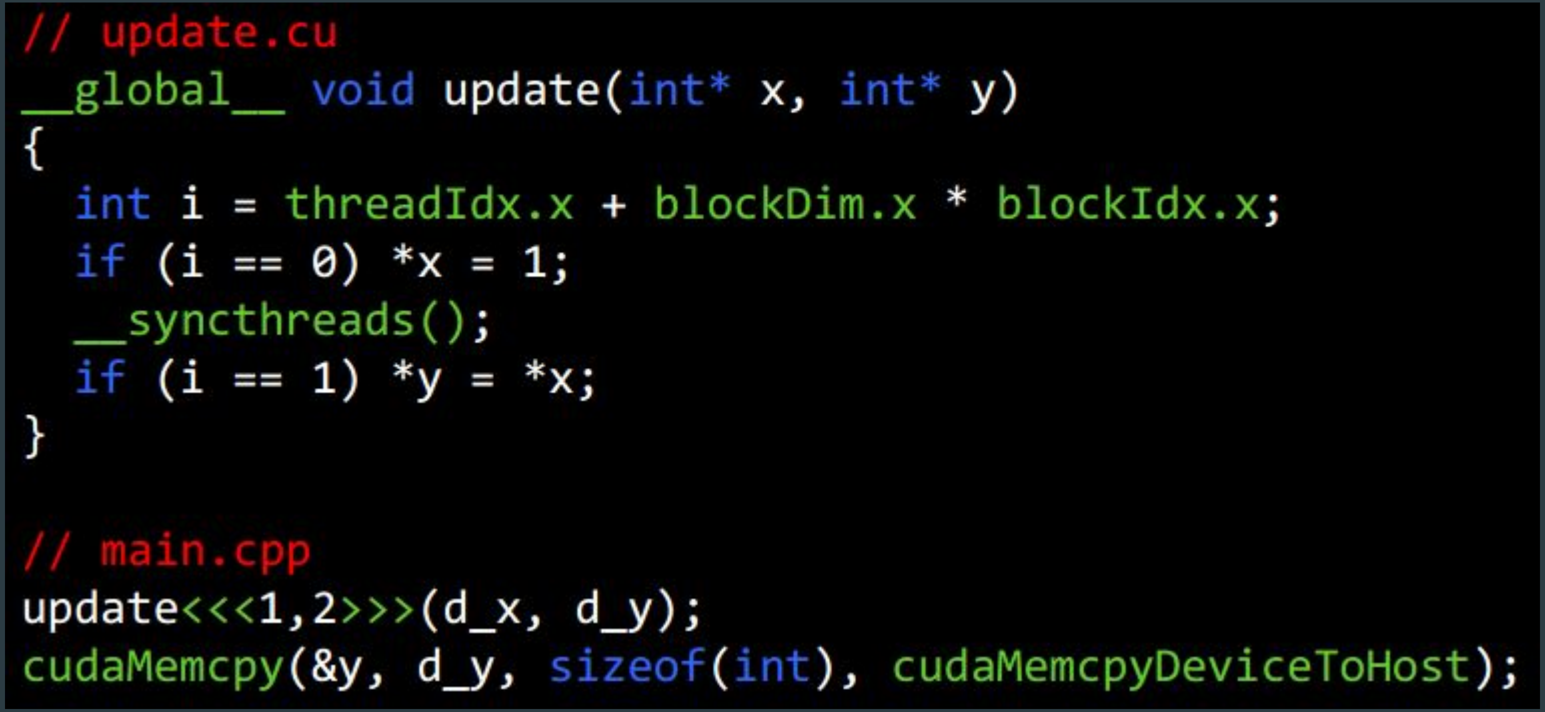


```
// update.cu
__global__ void update_race(int* x, int* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i == 0) *x = 1;
  if (i == 1) *y = *x;
}

// main.cpp
update_race<<<1,2>>>(d_x, d_y);
cudaMemcpy(&y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

- Program needs to ensure that thread T1 reads location after thread T0 has written location.

Synchronization within Block

- Threads in same block: can use __synchthreads() to specify synchronization point that orders accesses

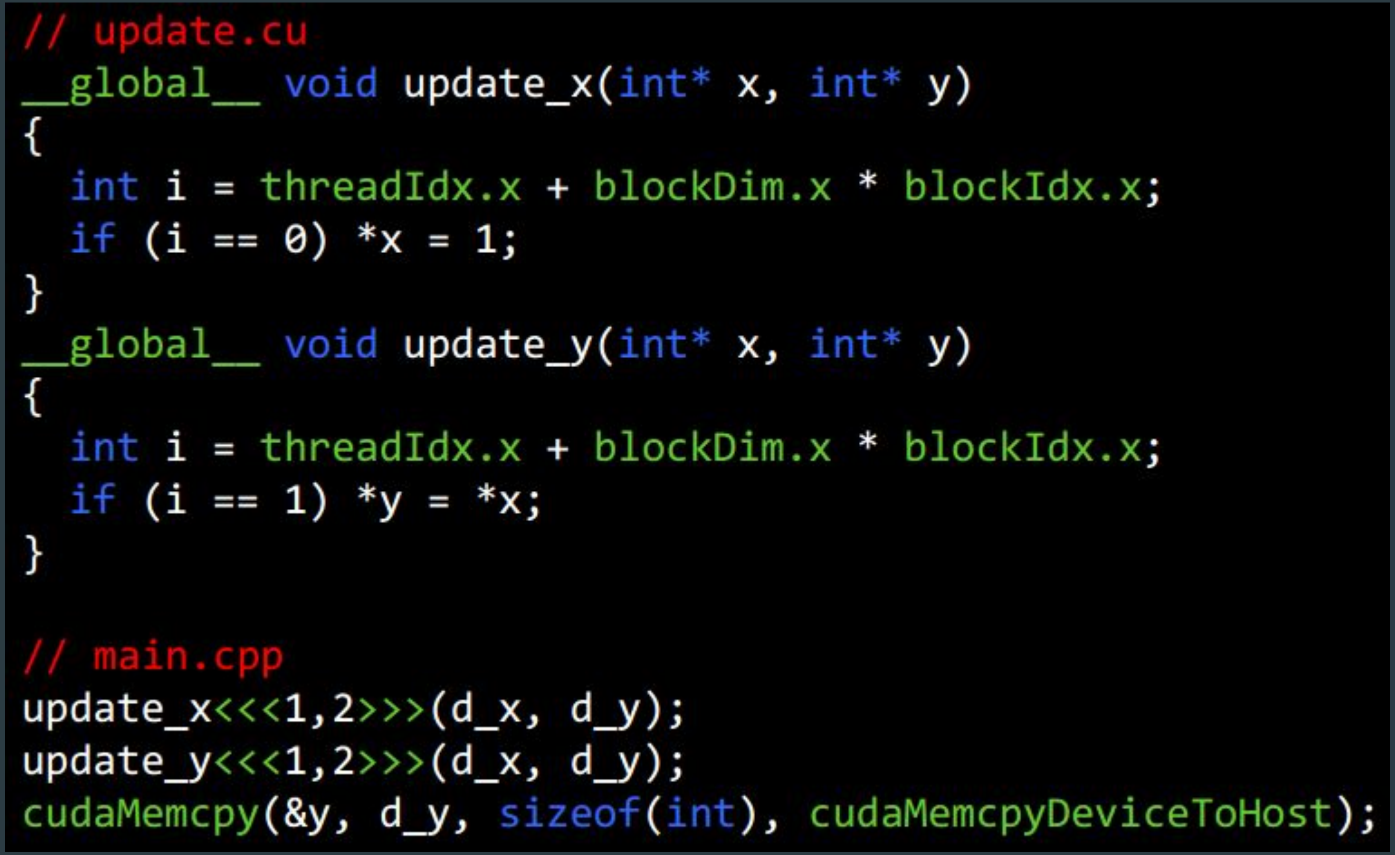


```
// update.cu
__global__ void update(int* x, int* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i == 0) *x = 1;
  __syncthreads();
  if (i == 1) *y = *x;
}

// main.cpp
update<<<1,2>>>(d_x, d_y);
cudaMemcpy(&y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

- Important: all threads within the block must reach the __synchthreads() statement

Synchronization between Grids

- Threads in different grids: system ensures writes from kernel happen before reads from subsequent grid launches.



```
// update.cu
__global__ void update_x(int* x, int* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i == 0) *x = 1;
}
__global__ void update_y(int* x, int* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i == 1) *y = *x;
}

// main.cpp
update_x<<<1,2>>>(d_x, d_y);
update_y<<<1,2>>>(d_x, d_y);
cudaMemcpy(&y, d_y, sizeof(int), cudaMemcpyDeviceToHost);
```

Synchronization within Grid

- Often not reasonable to split kernels to synchronize reads and writes from different threads to common locations
  - Values of __shared__ variables are lost unless explicitly saved
  - Kernel launch overhead is non-trivial, and introducing extra launches can degrade performance

- CUDA provides atomic functions (commonly called atomic memory operations) to enforce atomic accesses to shared variables that may be accessed by multiple threads

- Programmers can synthesize various coordination objects and synchronization schemes using atomic functions.

Introduction to Atomics

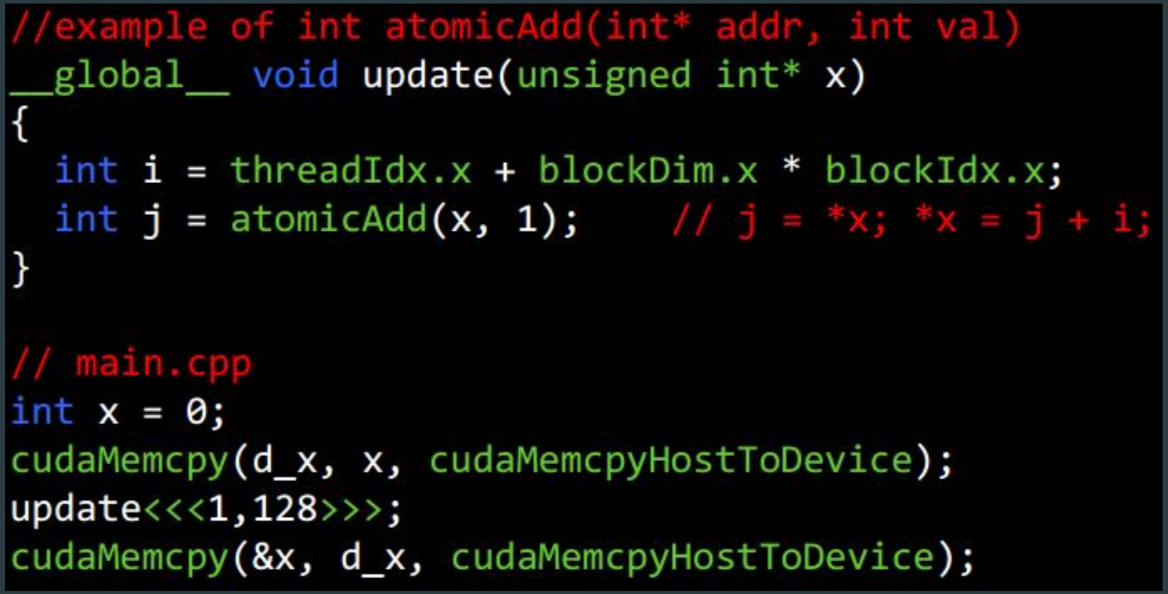
- Atom memory operations (atomic functions) are used to solve all kinds of synchronization and coordination problems in parallel computer systems.

- General concept is to provide a mechanism for a thread to update a memory location such that the update appears to happen atomically (without interruption) with respect to other threads.

- This ensures that all atomic updates issued concurrently are performed (often in some unspecified order) and that all threads can observe all updates.

Atomic Functions

- Atomic functions perform read-modify-write operations on data residing in global and shared memory



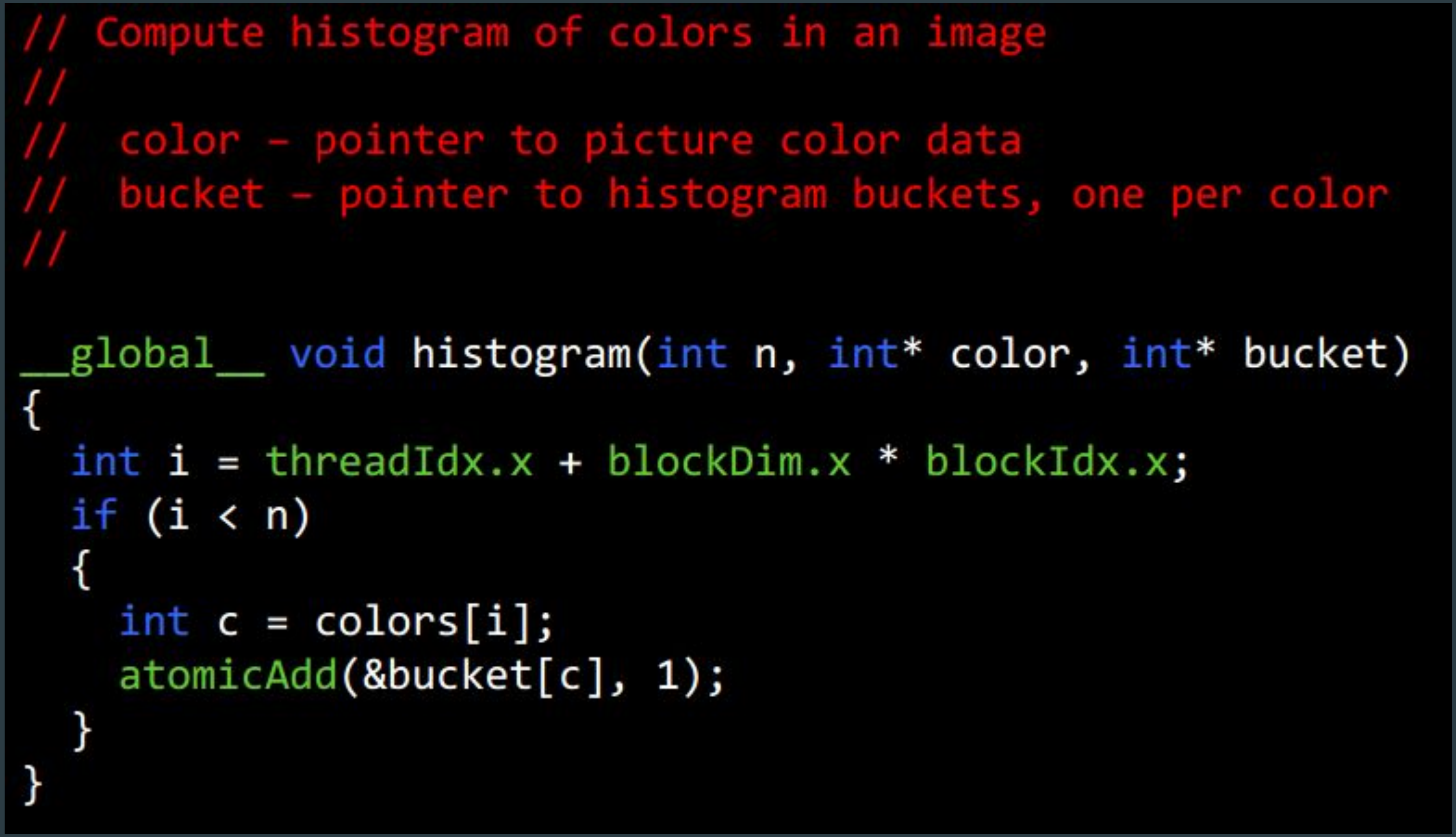
```
//example of int atomicAdd(int* addr, int val)
__global__ void update(unsigned int* x)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int j = atomicAdd(x, 1);    // j = *x; *x = j + i;
}

// main.cpp
int x = 0;
cudaMemcpy(d_x, x, cudaMemcpyHostToDevice);
update<<<1,128>>>;
cudaMemcpy(&x, d_x, cudaMemcpyHostToDevice);
```

- Atomic functions guarantee that only one thread may access a memory location while the operation completes

Atomic Functions

- The name atomic is used because the update is performed atomically: it cannot be interrupted by other atomic updates.
- The order in which concurrent atomic updates are performed is not defined, and may appear arbitrary. However, none of the atomic updates will be lost.
- Many different kinds of atomic operations
  - Add (add), Sub (subtract), Inc (increment), Dec (decrement)
  - And (bit-wise and), Or (bit-wise or) , Xor (bit-wise exclusive or)
  - Exch (Exchange)
  - Min (Minimum), Max (Maximum)
  - Compare-and-Swap

Histogram Example



```
// Compute histogram of colors in an image
//
//  color - pointer to picture color data
//  bucket - pointer to histogram buckets, one per color
//

__global__ void histogram(int n, int* color, int* bucket)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if (i < n)
  {
    int c = colors[i];
    atomicAdd(&bucket[c], 1);
  }
}
```

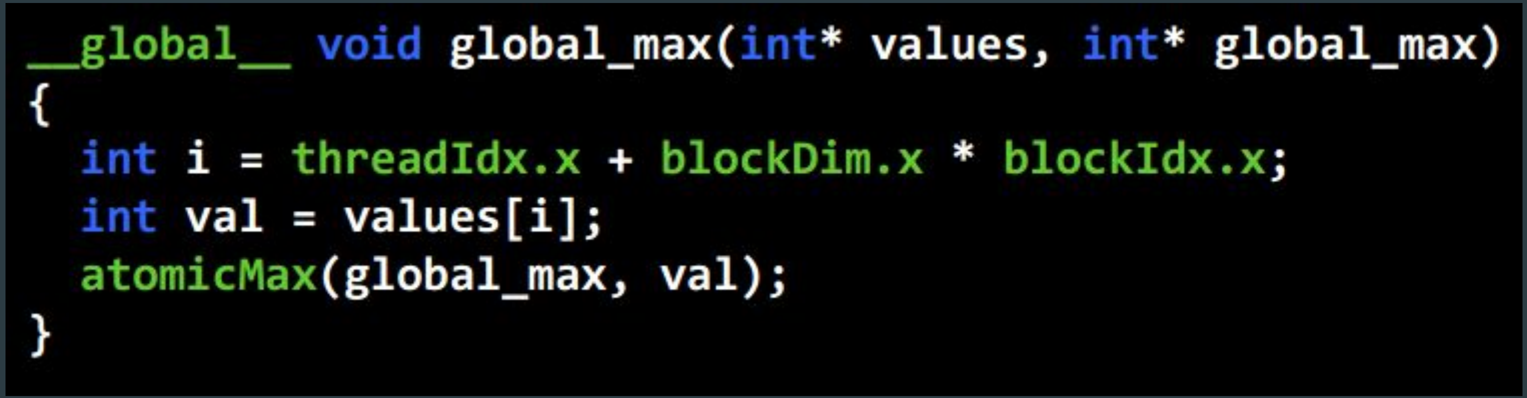
Performance Notes

- Atomics are slower than normal accesses (loads, stores)

- Performance can degrade when many threads attempt to perform atomic operations on a small number of locations

Example: Global Min/Max (Naive)

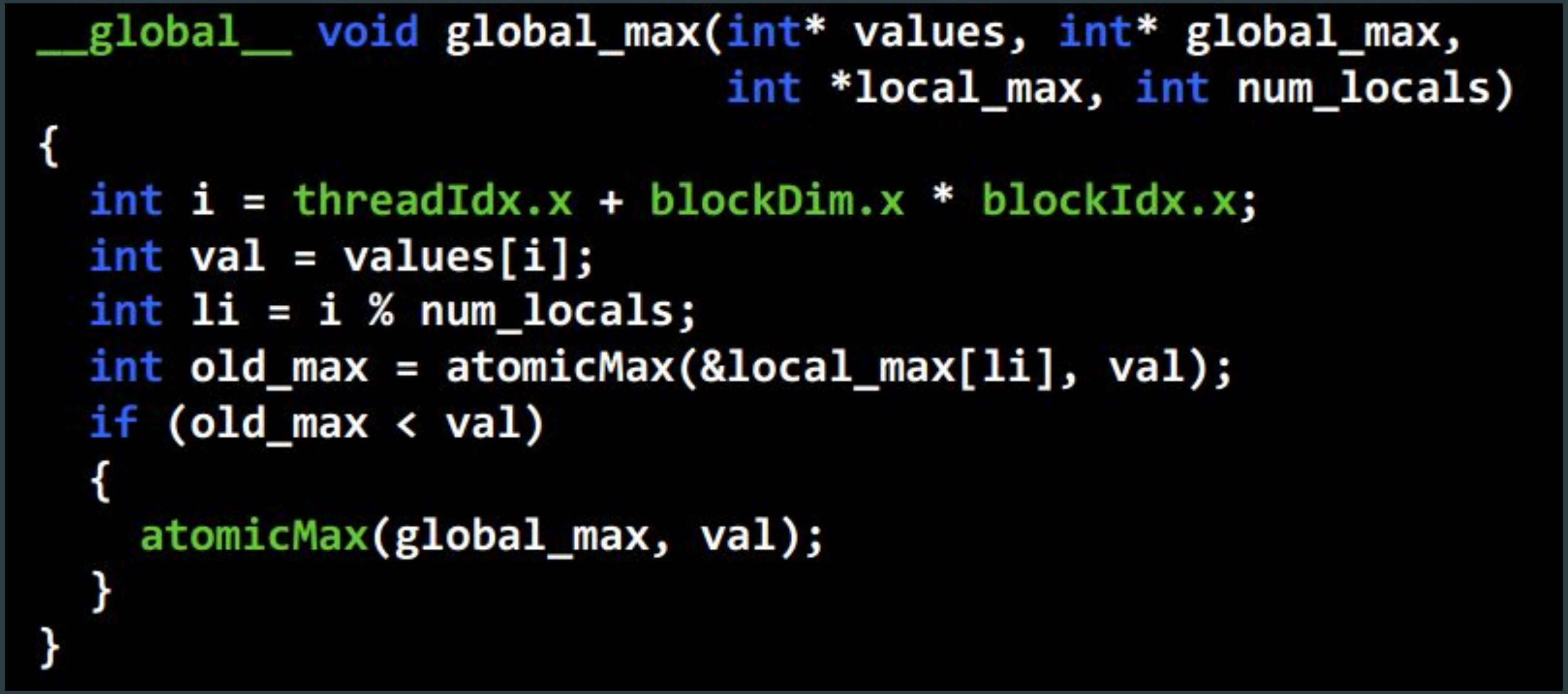
- Compute maximum across all threads in a grid
- One can use a single global maximum value, but it will be VERY slow.



```
__global__ void global_max(int* values, int* global_max)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int val = values[i];
  atomicMax(global_max, val);
}
```

Example: Global Min/Max (Better)

- Introduce local maximums and update global only when new local maximum found.



```
__global__ void global_max(int* values, int* global_max,
                           int *local_max, int num_locals)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  int val = values[i];
  int li = i % num_locals;
  int old_max = atomicMax(&local_max[li], val);
  if (old_max < val)
  {
    atomicMax(global_max, val);
  }
}
```

- Reduces frequency at which threads attempt to update the global maximum, reducing competition access to location.

Lessons from global Min/Max

- Many updates to a single value causes serial bottleneck
- One can create a hierarchy of values to introduce more parallelism and locality into algorithm
- 但是，性能仍然可能很慢，因此請謹慎使用