Introduction to CUDA C

What is CUDA?

- CUDA Architecture
  - Expose general-purpose GPU computing as first-class capability
  - Retain traditional DirectX/OpenGL graphics performance

- CUDA C
  - Based on industry-standard C
  - A handful of language extensions to allow heterogeneous programs
  - Straightforward APIs to manage devices, memory, etc.

- This talk will introduce you to CUDA C

Introduction to CUDA C

- What will you learn today?
  - Start from “Hello, World!”
  - Write and launch CUDA C kernels
  - Manage GPU memory
  - Run parallel kernels in CUDA C
  - Parallel communication and synchronization
  - Race conditions and atomic operations

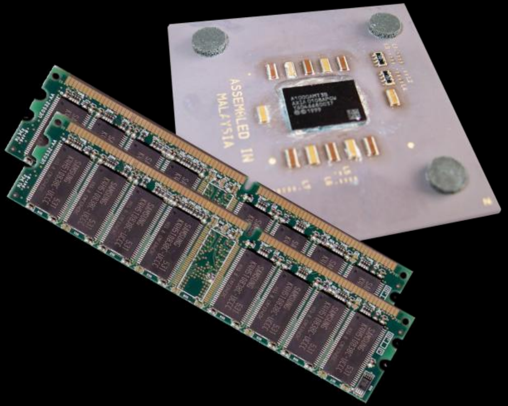
CUDA C Prerequisites

- You (probably) need experience with C or C++
- You do not need any GPU experience
- You do not need any graphics experience
- You do not need any parallel programming experience

CUDA C: The Basics

- Terminology
  - *Host* – The CPU and its memory (host memory)
  - *Device* – The GPU and its memory (device memory)

Host



Device

Note: *Figure Not to Scale*

Hello, World!

```
int main( void ) {
    printf( "Hello, World!\n" );
    return 0;
}
```

- This basic program is just standard C that runs on the *host*
- NVIDIA's compiler (`nvcc`) will not complain about CUDA programs with no *device* code
- At its simplest, CUDA C is just C!

Hello, World! with Device Code

```
__global__ void kernel( void ) {
}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

- Two notable additions to the original “Hello, World!”

Hello, World! with Device Code

```
__global__ void kernel( void ) {
}
```

- CUDA C keyword `__global__` indicates that a function
  - Runs on the device
  - Called from host code
- `nvcc` splits source file into host and device components
  - NVIDIA's compiler handles device functions like `kernel()`
  - Standard host compiler handles host functions like `main()`
    - `gcc`
    - Microsoft Visual C

Hello, World! with Device Code

```
int main( void ) {
    kernel<<< 1, 1 >>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Sometimes called a "kernel launch"
  - We'll discuss the parameters inside the angle brackets later
- This is all that's required to execute a function on the GPU!
- The function `kernel()` does nothing, so this is fairly anticlimactic…

A More Complex Example

- A simple kernel to add two integers:

```
__global__ void add( int *a, int *b, int *c ) {
    *c = *a + *b;
}
```

- As before, `__global__` is a CUDA C keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

A More Complex Example

- Notice that we use pointers for our variables:

```
__global__ void add( int *a, int *b, int *c ) {
    *c = *a + *b;
}
```

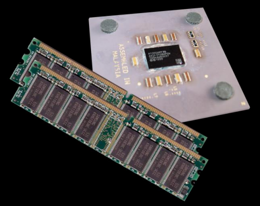
- `add()` runs on the device…so `a`, `b`, and `c` must point to device memory

- How do we allocate memory on the GPU?

Memory Management

- Host and device memory are distinct entities
  - Device pointers point to GPU memory
    - May be passed to and from host code
    - May not be dereferenced from host code

  - Host pointers point to CPU memory
    - May be passed to and from device code
    - May not be dereferenced from device code



- Basic CUDA API for dealing with device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to their C equivalents, `malloc()`, `free()`, `memcpy()`

A More Complex Example: `add()`

- Using our `add()` kernel:

```
__global__ void add( int *a, int *b, int *c ) {
    *c = *a + *b;
}
```

- Let’s take a look at `main()`…

A More Complex Example: main()

```
int main( void ) {
    int a, b, c;                    // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;     // device copies of a, b, c
    int size = sizeof( int );       // we need space for an integer

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = 2;
    b = 7;
```

A More Complex Example: `main()` (cont)

```
    // copy inputs to device
    cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice );

    // launch add() kernel on GPU, passing parameters
    add<<< 1, 1 >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost );

    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Parallel Programming in CUDA C

- But wait…GPU computing is about massive parallelism

- So how do we run code in parallel on the device?

- Solution lies in the parameters between the triple angle brackets:

```
add<<< 1, 1 >>>( dev_a, dev_b, dev_c );
```

↓

```
add<<< N, 1 >>>( dev_a, dev_b, dev_c );
```

- Instead of executing `add()` once, `add()` executed `N` times in parallel

Parallel Programming in CUDA C

- With `add()` running in parallel…let's do vector addition

- Terminology: Each parallel invocation of `add()` referred to as a *block*

- Kernel can refer to its block's index with the variable `blockIdx.x`

- Each block adds a value from `a[]` and `b[]`, storing the result in `c[]`:

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

- By using `blockIdx.x` to index arrays, each block handles different indices

Parallel Programming in CUDA C

- We write this code:

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

- This is what runs in parallel on the device:

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```

Parallel Addition: `add()`

- Using our newly parallelized `add()` kernel:

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

- Let's take a look at `main()`…

Parallel Addition: main()

```
#define N   512
int main( void ) {
    int *a, *b, *c;                   // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;       // device copies of a, b, c
    int size = N * sizeof( int );     // we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition: `main()` (cont)

```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch add() kernel with N parallel blocks
    add<<< N, 1 >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

    free( a ); free( b ); free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Review

- Difference between “host” and “device”
  - Host = CPU
  - Device = GPU

- Using `__global__` to declare a function as device code
  - Runs on device
  - Called from host

- Passing parameters from host code to a device function

Review (cont)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`

- Launching parallel kernels
  - Launch `N` copies of `add()` with: `add<<< N, 1 >>>();`
  - Used `blockIdx.x` to access block's index

Threads

- Terminology: A block can be split into parallel *threads*

- Let’s change vector addition 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` in `add()`

- `main()` will require one change as well…

Parallel Addition (Threads): `main()`

```
#define N  512
int main( void ) {
    int *a, *b, *c;                        //host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;            //device copies of a, b, c
    int size = N * sizeof( int );          //we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition (Threads): `main()` (cont)

```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch add() kernel with N threads
    add<<< 1, N >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

    free( a ); free( b );  free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Using Threads *And* Blocks

- We've seen parallel vector addition using
  - Many blocks with 1 thread apiece
  - 1 block with many threads

- Let's adapt vector addition to use lots of *both* blocks and threads

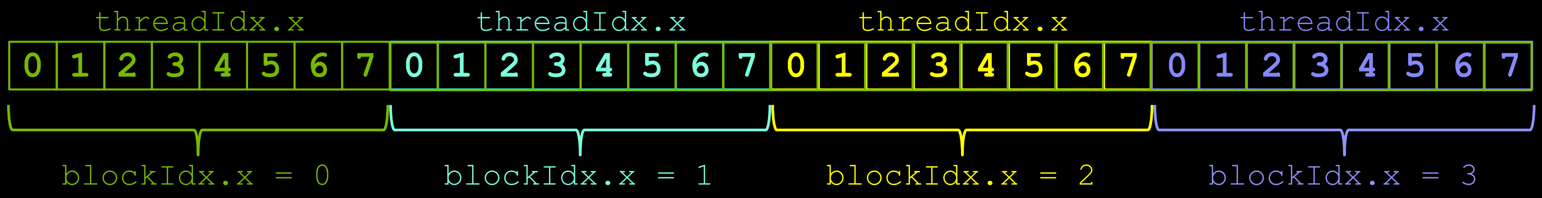
- After using threads and blocks together, we'll talk about *why* threads

- First let's discuss data indexing…

Indexing Arrays With Threads And Blocks

- No longer as simple as just using `threadIdx.x` or `blockIdx.x` as indices

- To index array with 1 thread per entry (using 8 threads/block)

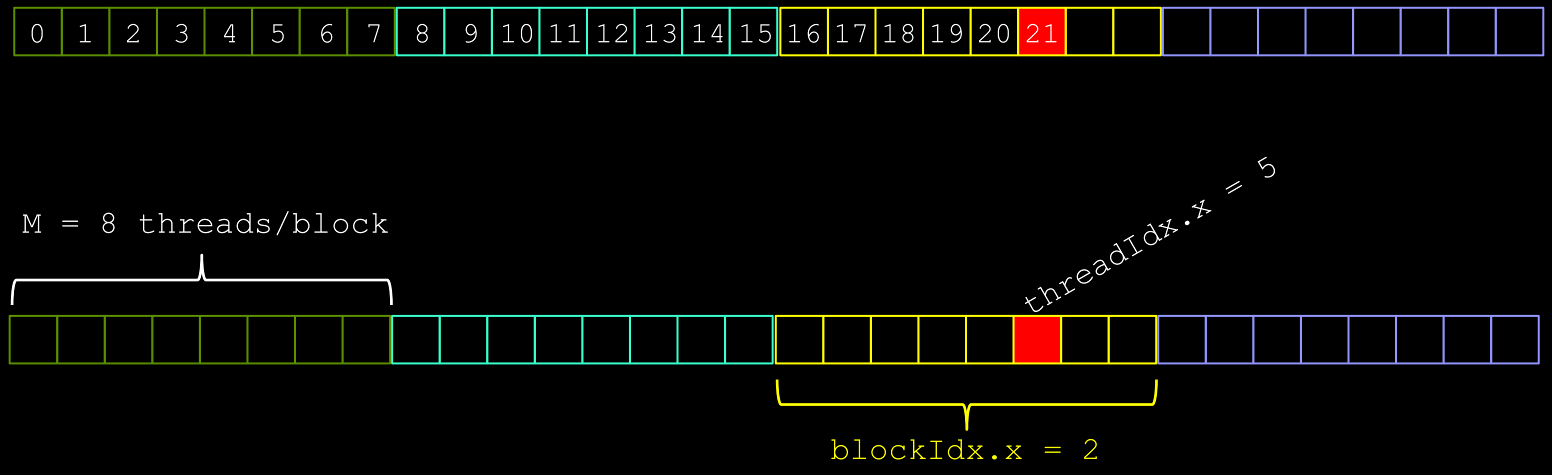


- If we have `M` threads/block, a unique array index for each entry given by

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

int index =      x      +      y     * width;
```

Indexing Arrays: Example

- In this example, the red entry would have an index of 21:




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

Addition with Threads and Blocks

- The `blockDim.x` is a built-in variable for threads per block:

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

- A combined version of our vector addition kernel to use blocks *and* threads:

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

- So what changes in `main()` when we use both blocks and threads?

Parallel Addition (Blocks/Threads): main()

```
#define N  (2048*2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
    int *a, *b, *c;                          // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;              // device copies of a, b, c
    int size = N * sizeof( int );            // we need space for N integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Addition (Blocks/Threads): main()

```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch add() kernel with blocks and threads
    add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

    free( a ); free( b ); free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Why Bother With Threads?

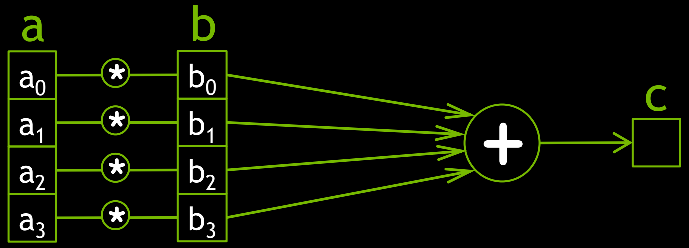
- Threads seem unnecessary
  - Added a level of abstraction and complexity
  - What did we gain?

- Unlike parallel blocks, parallel threads have mechanisms to
  - Communicate
  - Synchronize

- Let’s see how…

Dot Product

- Unlike vector addition, dot product is a *reduction* from vectors to a scalar



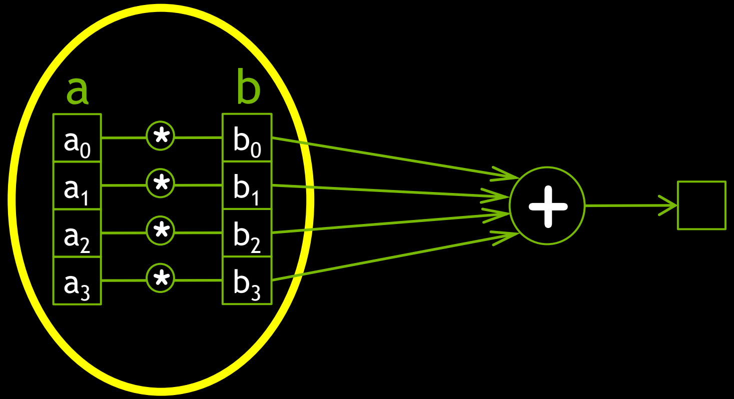
$$c = \vec{a} \cdot \vec{b}$$

$$= (a_0, a_1, a_2, a_3) \cdot (b_0, b_1, b_2, b_3)$$

$$= a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3$$

Dot Product

- Parallel threads have no problem computing the pairwise products:

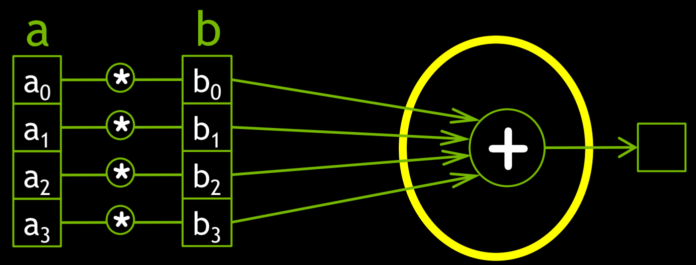


- So we can start a dot product CUDA kernel by doing just that:

```
__global__ void dot( int *a, int *b, int *c )   {
    // Each thread computes a pairwise product
    int temp = a[threadIdx.x] * b[threadIdx.x];
```

Dot Product

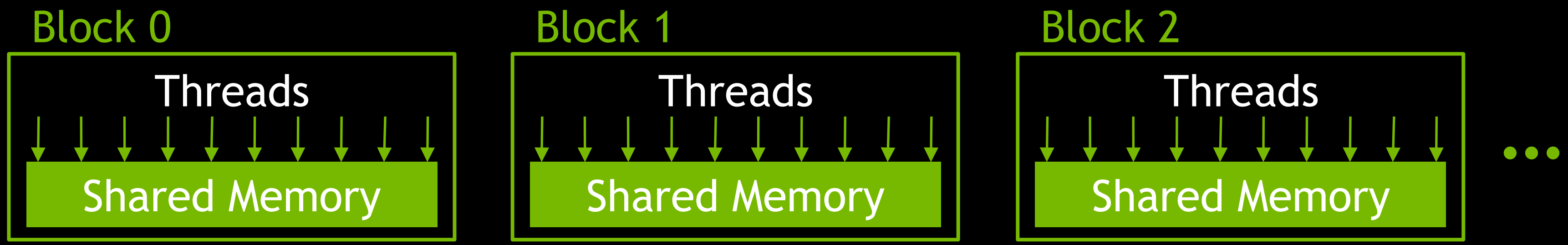
- But we need to share data between threads to compute the final sum:




```
__global__ void dot( int *a, int *b, int *c )    {
    // Each thread computes a pairwise product
    int temp = a[threadIdx.x] * b[threadIdx.x];

    // Can't compute the final sum
    // Each thread's copy of 'temp' is private
}
```

Sharing Data Between Threads

- Terminology: A block of threads shares memory called…*shared memory*
- Extremely fast, on-chip memory (user-managed cache)
- Declared with the `__shared__` CUDA keyword
- Not visible to threads in other blocks running in parallel



Parallel Dot Product: dot()

- We perform parallel multiplication, serial addition:

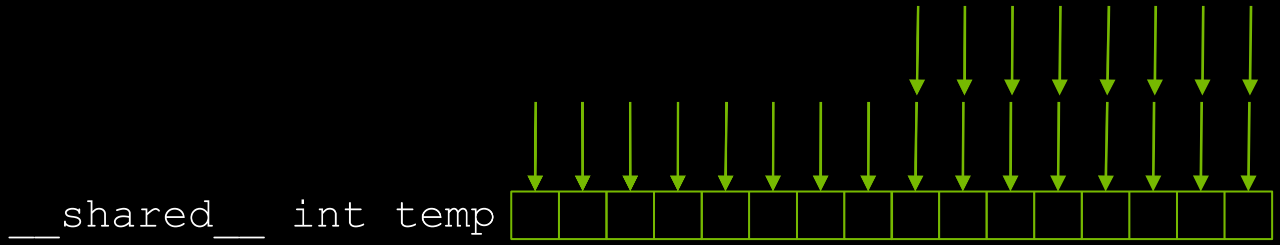
```
#define N  512
__global__ void dot( int *a, int *b, int *c ) {
      // Shared memory for results of multiplication
      __shared__ int temp[N];
      temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

      // Thread 0 sums the pairwise products
      if( 0 == threadIdx.x ) {
          int sum = 0;
          for( int i = 0; i < N; i++ )
              sum += temp[i];
          *c = sum;
      }
}
```

Parallel Dot Product Recap

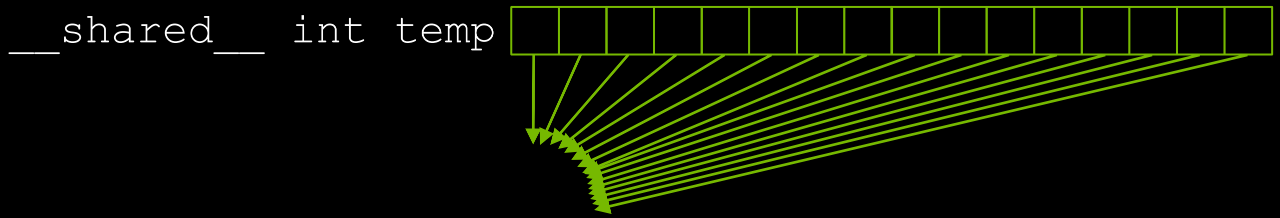
- We perform parallel, pairwise multiplications
- Shared memory stores each thread's result
- We sum these pairwise products from a single thread
- Sounds good…but we've made a huge mistake

Faulty Dot Product Exposed!

- Step 1: In parallel, each thread writes a pairwise product



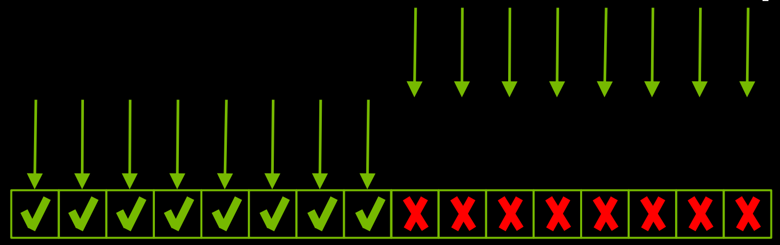
- Step 2: Thread 0 reads and sums the products



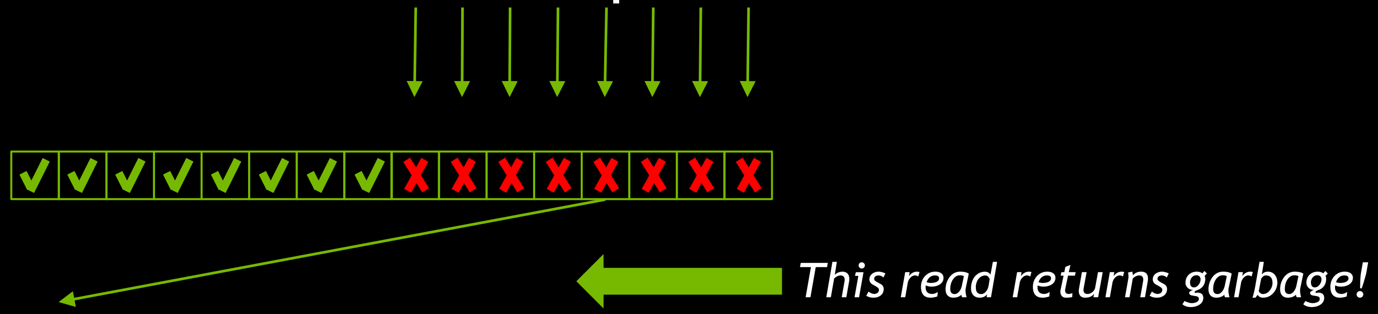
- But there’s an assumption hidden in Step 1…

Read-Before-Write Hazard

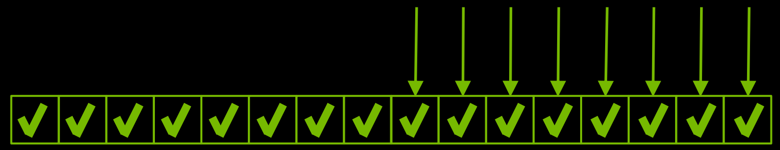
- Suppose thread 0 finishes its write in step 1



- Then thread 0 reads index 12 in step 2



- Before thread 12 writes to index 12 in step 1?



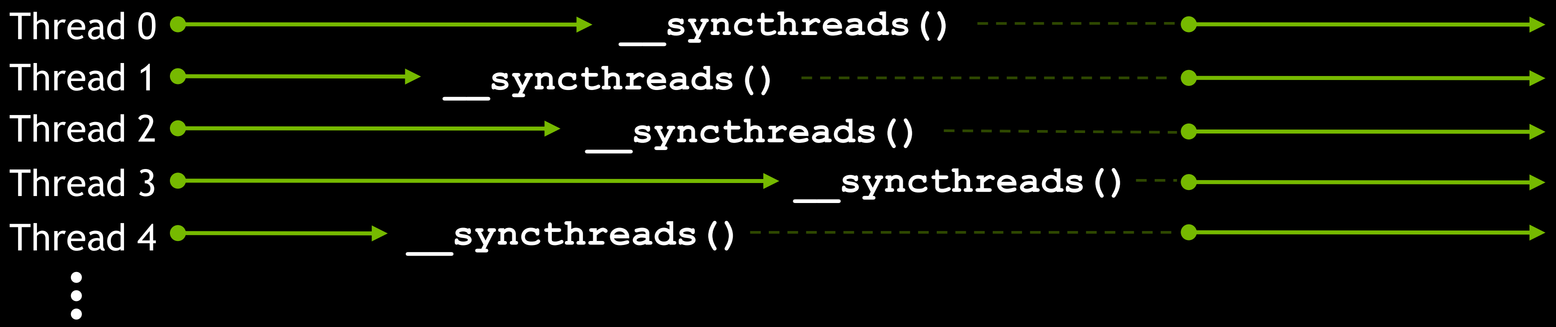
Synchronization

- We need threads to wait between the sections of `dot()`:

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // * NEED THREADS TO SYNCHRONIZE HERE *
    // No thread can advance until all threads
    // have reached this point in the code

    // Thread 0 sums the pairwise products
    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < N; i++ )
            sum += temp[i];
        *c = sum;
    }
}
```

__syncthreads()

- We can synchronize threads with the function `__syncthreads()`
- Threads in the block wait until *all* threads have hit the `__syncthreads()`



- Threads are *only* synchronized within a block

Parallel Dot Product: dot()

```
__global__ void dot( int *a, int *b, int *c ) {
      __shared__ int temp[N];
     temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

      __syncthreads();

      if( 0 == threadIdx.x ) {
          int sum = 0;
          for( int i = 0; i < N; i++ )
              sum += temp[i];
          *c = sum;
      }
}
```

- With a properly synchronized `dot()` routine, let’s look at `main()`

Parallel Dot Product: `main()`

```
#define N   512
int main( void ) {
    int *a, *b, *c;                     // copies of a, b, c
    int *dev_a, *dev_b, *dev_c;         // device copies of a, b, c
    int size = N * sizeof( int );       // we need space for 512 integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, sizeof( int ) );

    a = (int *)malloc( size );
    b = (int *)malloc( size );
    c = (int *)malloc( sizeof( int ) );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Dot Product: `main()`

```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch dot() kernel with 1 block and N threads
    dot<<< 1, N >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, sizeof( int ) , cudaMemcpyDeviceToHost );

    free( a ); free( b ); free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

Review

- Launching kernels with parallel threads
  - Launch `add()` with `N` threads: `add<<< 1, N >>>();`
  - Used `threadIdx.x` to access thread's index

- Using both blocks and threads
  - Used `(threadIdx.x + blockIdx.x * blockDim.x)` to index input/output
  - `N/THREADS_PER_BLOCK` blocks and `THREADS_PER_BLOCK` threads gave us `N` threads total

Review (cont)

- Using `__shared__` to declare memory as shared memory
  - Data shared among threads in a block
  - Not visible to threads in other parallel blocks

- Using `__syncthreads()` as a barrier
  - No thread executes instructions after `__syncthreads()` until all threads have reached the `__syncthreads()`
  - Needs to be used to prevent *data hazards*

Multiblock Dot Product

- Recall our dot product launch:

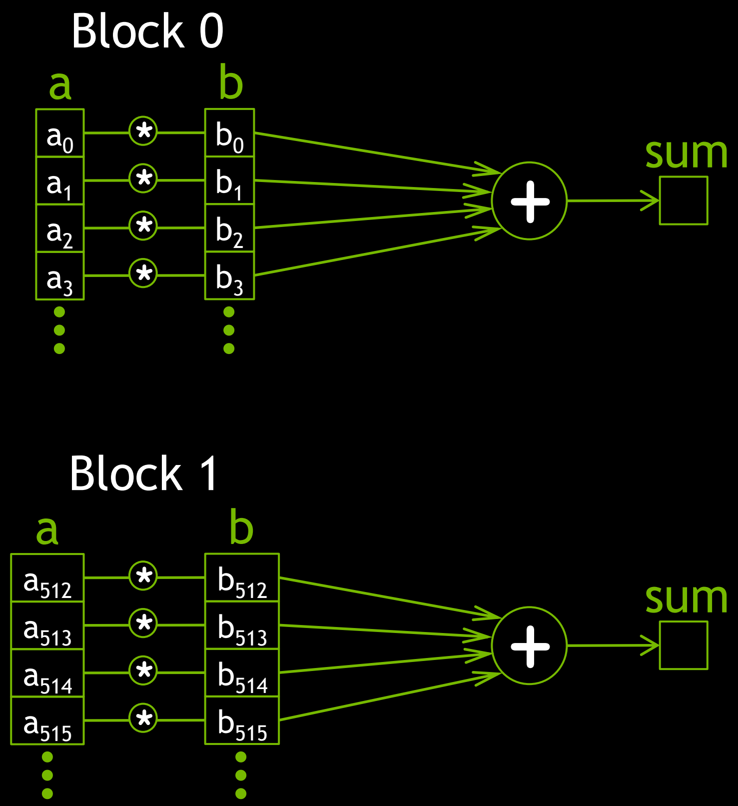
```
// launch dot() kernel with 1 block and N threads
dot<<< 1, N >>>( dev_a, dev_b, dev_c );
```

- Launching with one block will not utilize much of the GPU

- Let’s write a multiblock version of dot product

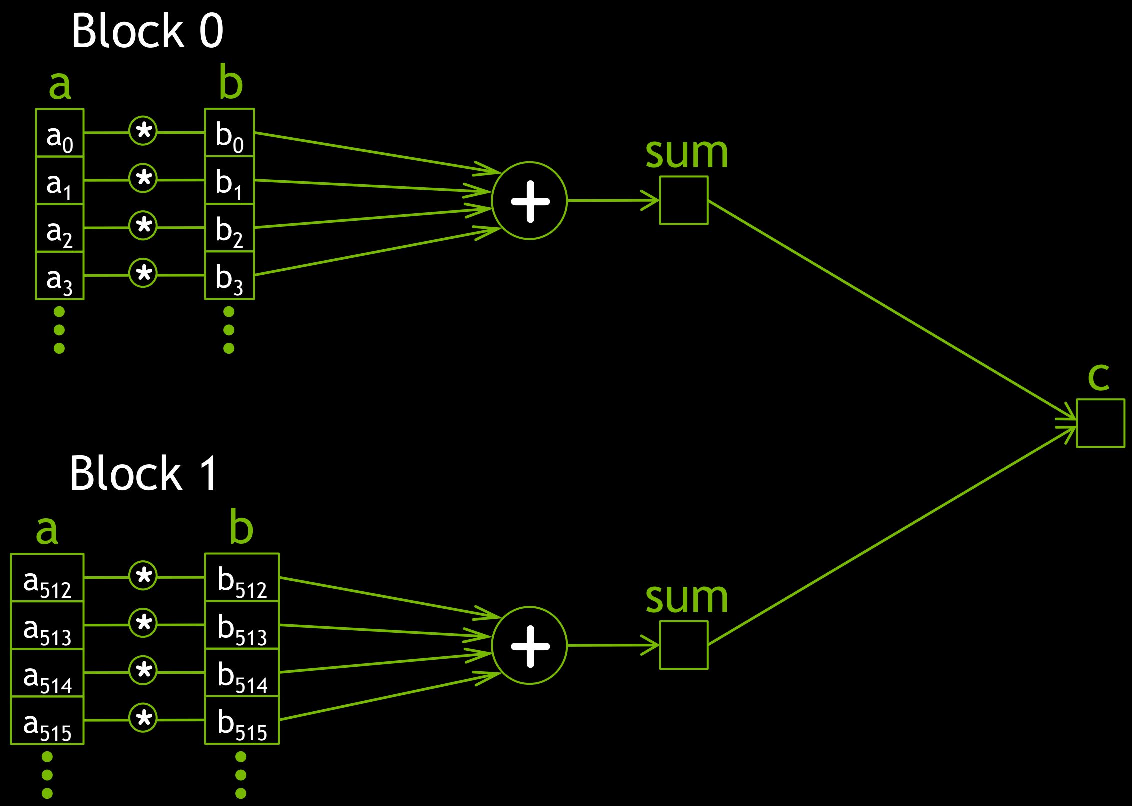
Multiblock Dot Product: Algorithm

- Each block computes a sum of its pairwise products like before:



Multiblock Dot Product: Algorithm

- And then contributes its sum to the final result:



Multiblock Dot Product: dot()

```
#define N  (2048*2048)
#define THREADS_PER_BLOCK  512
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[THREADS_PER_BLOCK];
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    temp[threadIdx.x] = a[index] * b[index];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < THREADS_PER_BLOCK; i++ )
            sum += temp[i];
        *c += sum;
        atomicAdd( c , sum );
    }
}
```

- But we have a race condition…
- We can fix it with one of CUDA’s atomic operations

Race Conditions

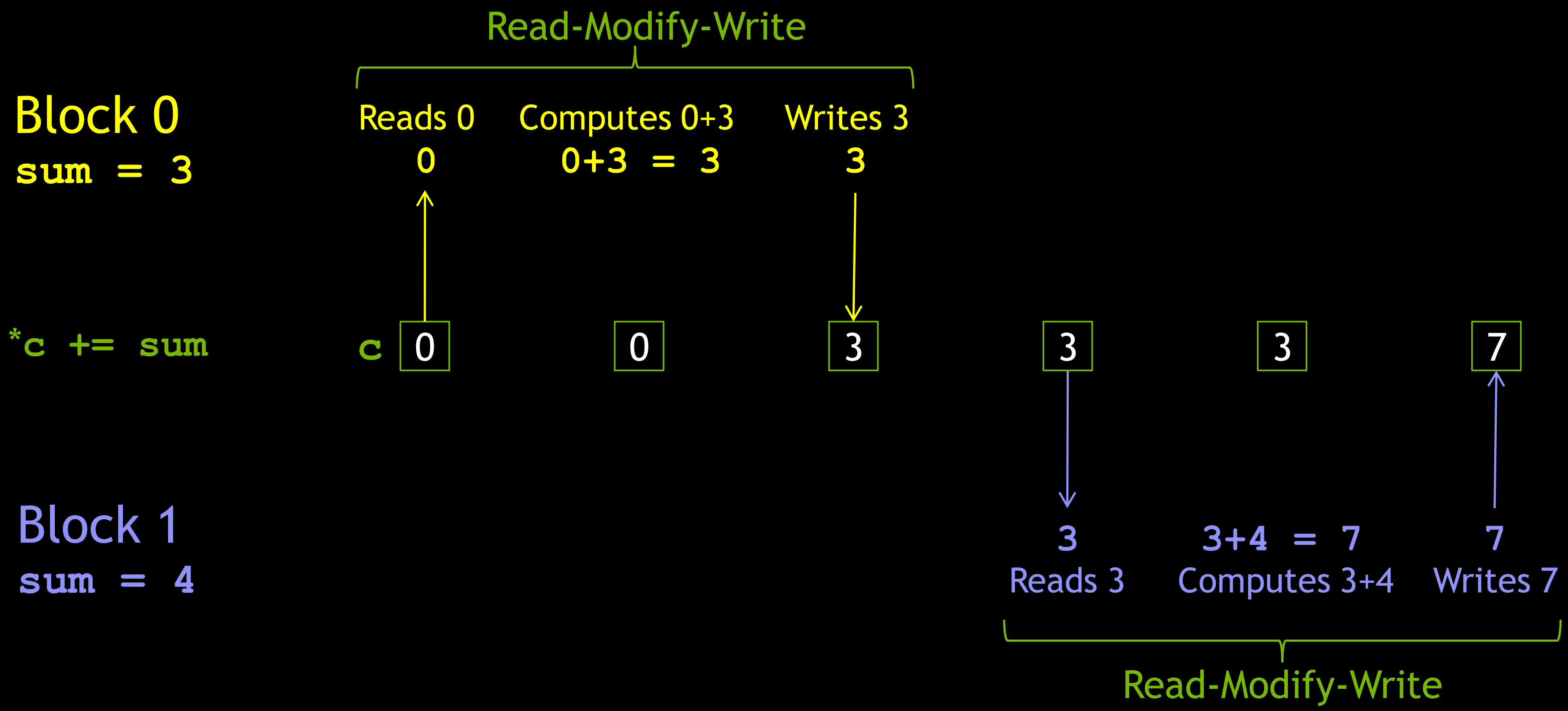
- Terminology: A *race condition* occurs when program behavior depends upon relative timing of two (or more) event sequences

- What actually takes place to execute the line in question: `*c += sum;`
  - Read value at address **`c`**
  - Add **`sum`** to value
  - Write result to address **`c`**

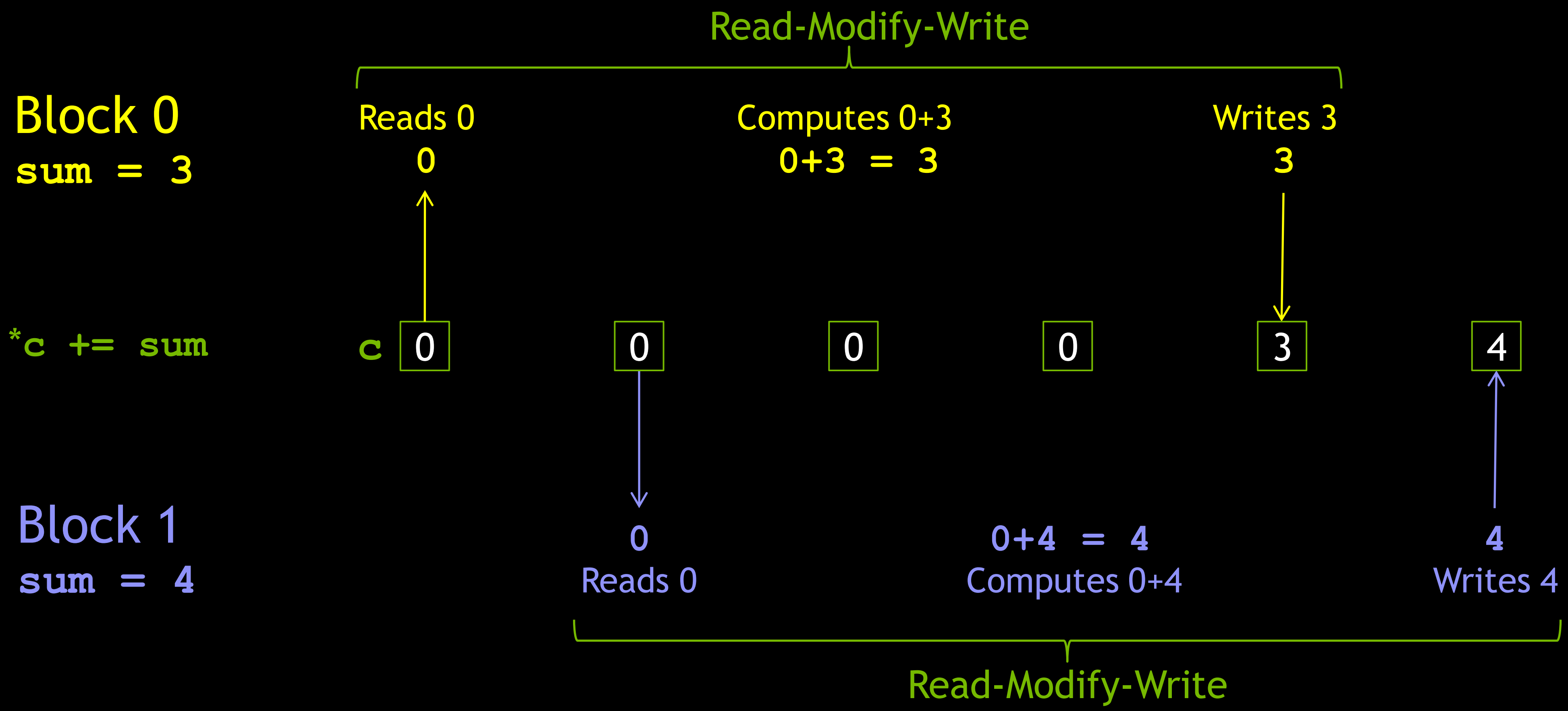
  Terminology: *Read-Modify-Write*

- What if two threads are trying to do this at the same time?
  - Thread 0, Block 0
    - Read value at address **`c`**
    - Add **`sum`** to value
    - Write result to address **`c`**
  - Thread 0, Block 1
    - Read value at address **`c`**
    - Add **`sum`** to value
    - Write result to address **`c`**

Global Memory Contention



Global Memory Contention



Atomic Operations

- Terminology: Read-modify-write uninterruptible when *atomic*

- Many *atomic operations* on memory available with CUDA C
  - `atomicAdd()`
  - `atomicSub()`
  - `atomicMin()`
  - `atomicMax()`
  - `atomicInc()`
  - `atomicDec()`
  - `atomicExch()`
  - `atomicCAS()`

- Predictable result when simultaneous access to memory required

- We need to atomically add `sum` to `c` in our multiblock dot product

Multiblock Dot Product: dot()

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[THREADS_PER_BLOCK];
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    temp[threadIdx.x] = a[index] * b[index];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < THREADS_PER_BLOCK; i++ )
            sum += temp[i];
        atomicAdd( c , sum );
    }
}
```

- Now let’s fix up `main()` to handle a multiblock dot product

Parallel Dot Product: main()

```
#define N  (2048*2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
    int *a, *b, *c;                          // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;              // device copies of a, b, c
    int size = N * sizeof( int );            // we need space for N ints

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, sizeof( int ) );

    a = (int *)malloc( size );
    b = (int *)malloc( size );
    c = (int *)malloc( sizeof( int ) );

    random_ints( a, N );
    random_ints( b, N );
```

Parallel Dot Product: main()

```
    // copy inputs to device
    cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

    // launch dot() kernel
    dot<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( dev_a, dev_b, dev_c );

    // copy device result back to host copy of c
    cudaMemcpy( c, dev_c, sizeof( int ) , cudaMemcpyDeviceToHost );

    free( a ); free( b ); free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

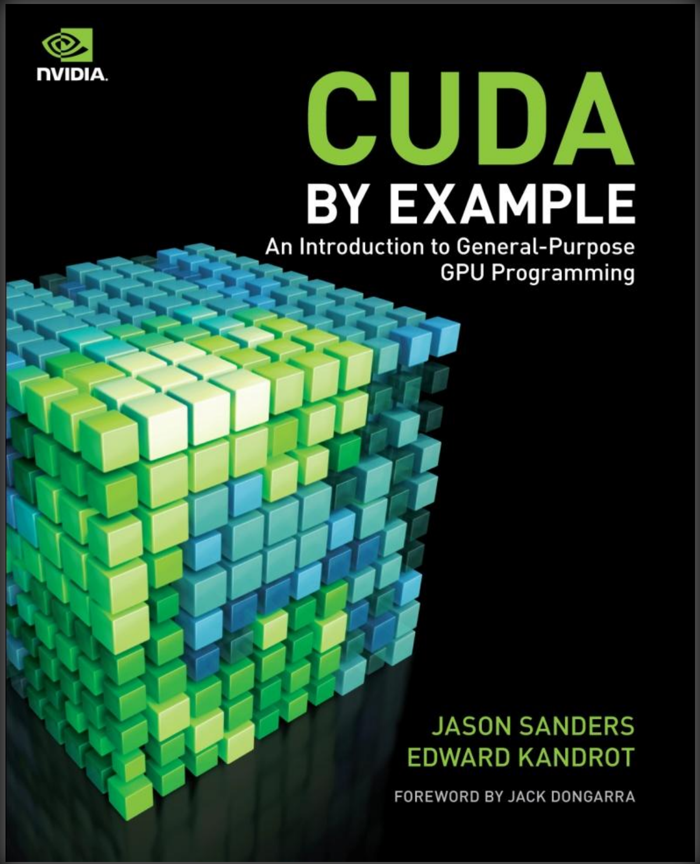
Review

- Race conditions
  - Behavior depends upon relative timing of multiple event sequences
  - Can occur when an implied read-modify-write is interruptible

- Atomic operations
  - CUDA provides read-modify-write operations guaranteed to be atomic
  - Atomics ensure correct results when multiple threads modify memory

To Learn More CUDA C

- Check out *CUDA by Example*
  - Parallel Programming in CUDA C
  - Thread Cooperation
  - Constant Memory and Events
  - Texture Memory
  - Graphics Interoperability
  - Atomics
  - Streams
  - CUDA C on Multiple GPUs
  - Other CUDA Resources



- http://developer.nvidia.com/object/cuda-by-example.html

Questions

- First my questions
- Now your questions…