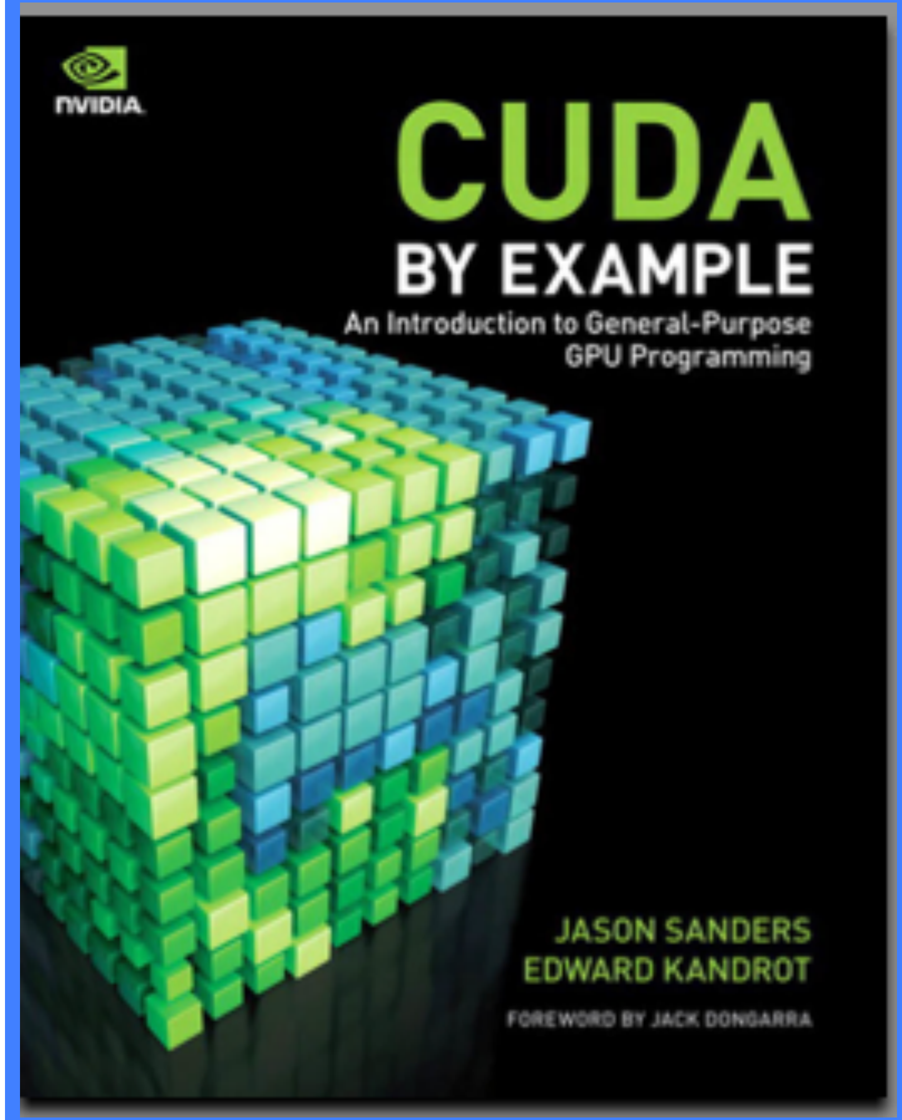
# A crash course on C-CUDA programming for GPU

**Massimo Bernaschi**

*m.bernaschi@iac.cnr.it*

http://www.iac.cnr.it/~massimo

## A class based on

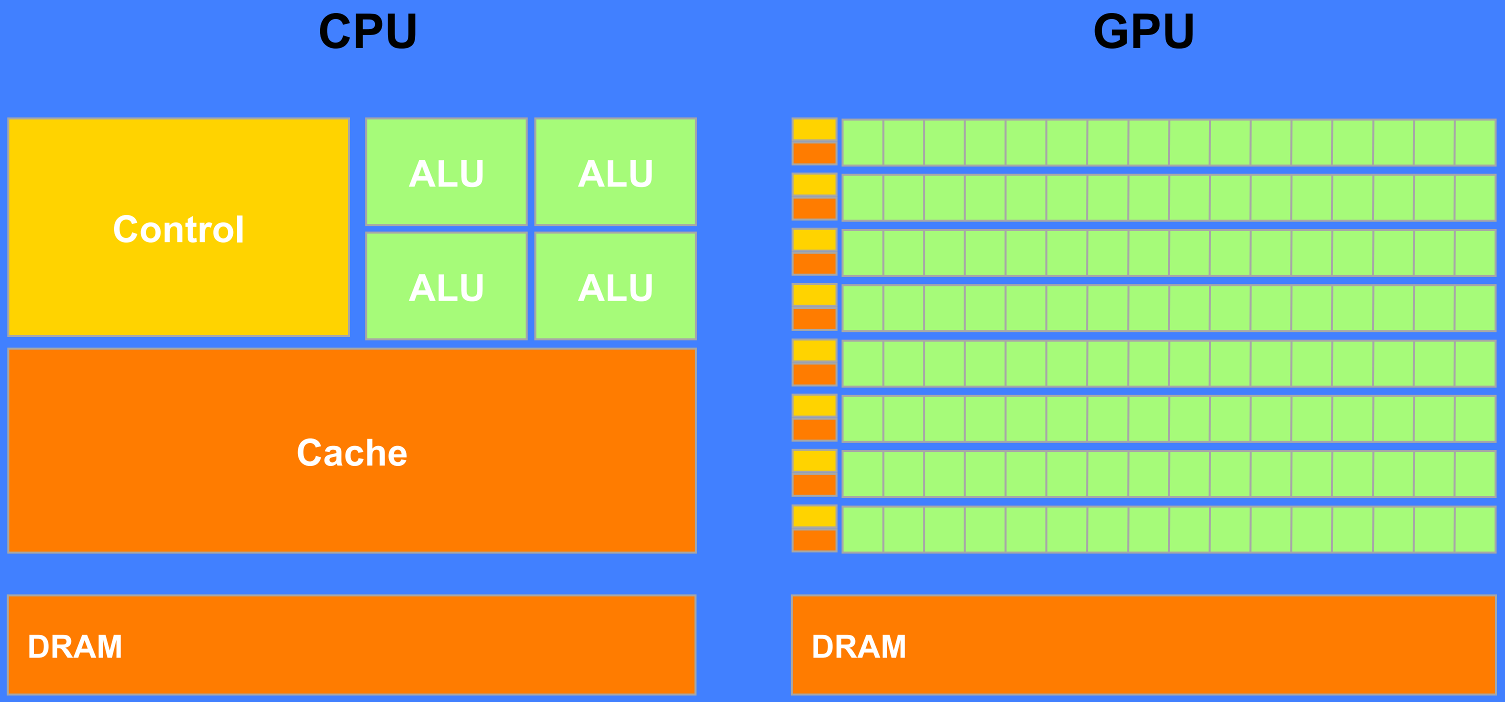


Original slides by Jason Sanders
by courtesy of Massimiliano Fatica.

Code samples available from
http://twin.iac.rm.cnr.it/ccc.tgz
Slides available from
**wget** http://twin.iac.rm.cnr.it/CCC.pdf

Let us start with few basic info

## A Graphics Processing Unit is not a CPU!



And there is no GPU without a CPU…

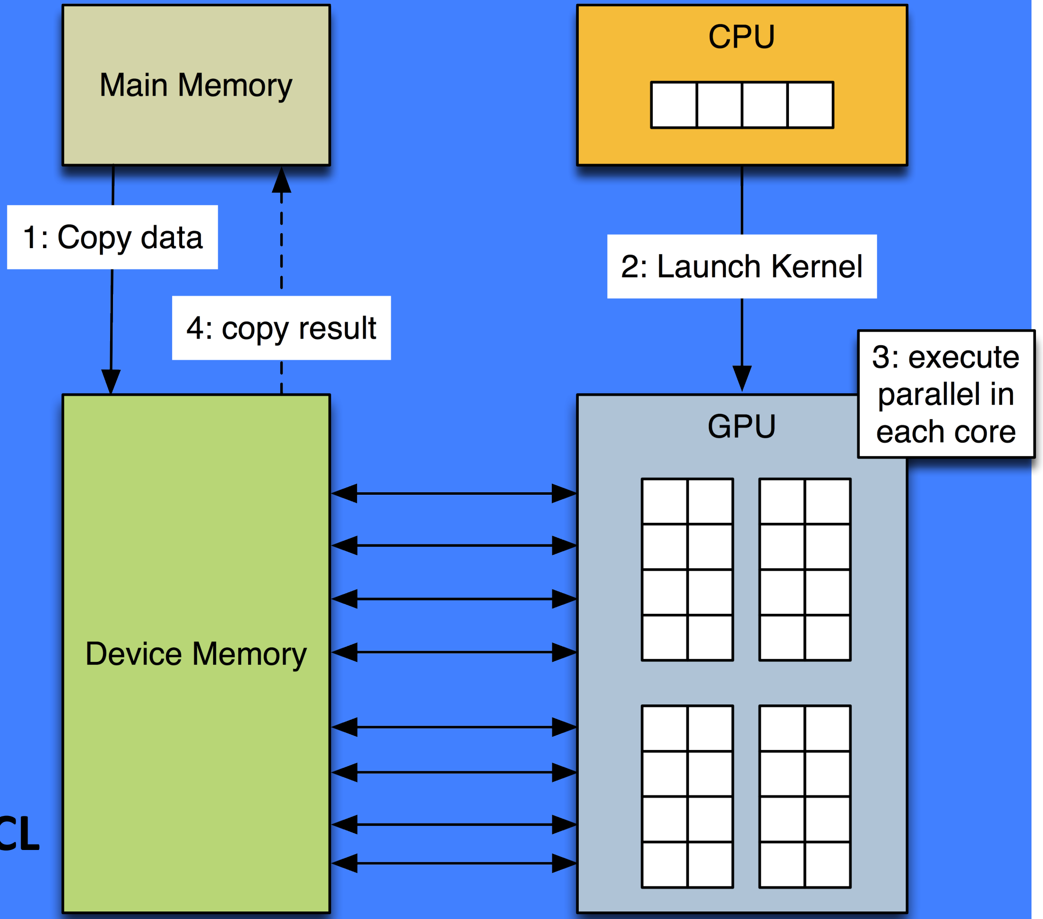
## Graphics Processing Unit

✓Originally dedicated to specific operations required by video cards for accelerating graphics, GPU have become flexible **general-purpose** computational engines (GPGPU):

- ✓Optimized for **high memory throughput**
- ✓Very suitable to **data-parallel processing**

✓Three vendors: Chipzilla (*a.k.a*. Intel), ADI (*i.e.,* AMD), NVIDIA.

✓Traditionally GPU programming has been tricky but **CUDA** and **OpenCL** made it affordable.



# What is CUDA?

- CUDA=Compute Unified Device 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.

**Why C?**

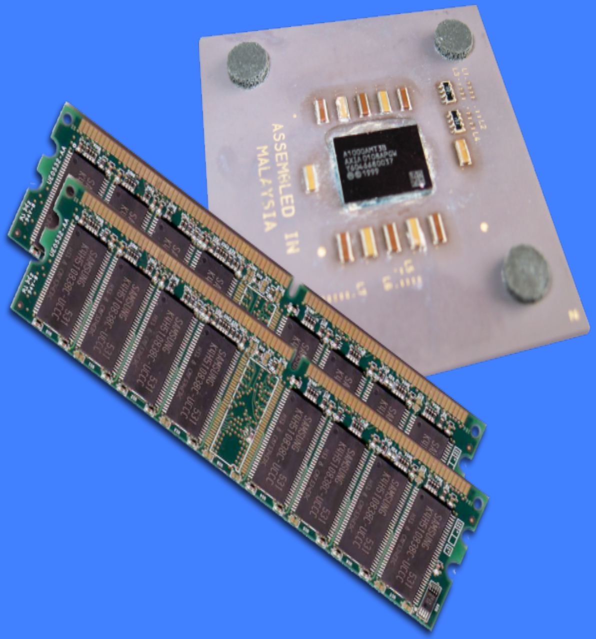
# CUDA Programming Model

- The GPU is viewed as a compute device that:
  - has its own RAM (**device memory**)
  - runs many **threads in parallel**
- Data-parallel portions of an application are executed on the device as **kernels** which run using many threads
- GPU *vs.* CPU threads
  - GPU threads are **extremely lightweight**
    - Very little creation overhead
  - GPU needs **1000s of threads for full efficiency**
    - A multi-core CPU needs only a few

## CUDA C: The Basics

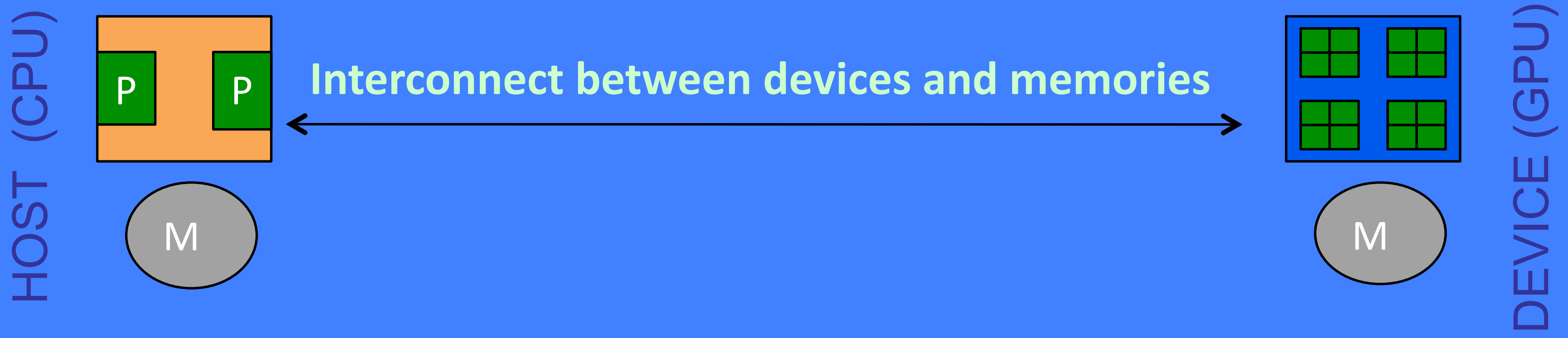
- *Host* – The CPU and its memory (host memory)
- *Device* – The GPU and its memory (device memory)

Host



Device

# What Programmer Expresses in CUDA



- ✓ Computation partitioning (where does computation occur?)
  - ✓ Declarations on functions __host__, __global__, __device__
  - ✓ Mapping of thread programs to device: **compute <<<gs, bs>>>(<args>)**
- ✓ Data partitioning (where does data reside, who may access it and how?)
  - ✓ Declarations on data __shared__, __device__, __constant__, …
- ✓ Data management and orchestration
  - ✓ Copying to/from host:
    *e.g.*, cudaMemcpy(h_obj,d_obj, cudaMemcpyDevicetoHost)
- ✓ Concurrency management
  - ✓ *e.g.* __synchthreads()

# Code Samples

1. **Connect to the front-end system:**
   **ssh –X s_hpc_XX@louis.caspur.it where XX goes from 01 to 30**
2. **Connect to the interactive nodes:**
   **ssh –X ella00Y where Y goes from 1 to 5**

   ```
   s_hpc_01, …, s_hpc_06 -> ella001
   s_hpc_07, …, s_hpc_12 -> ella002
   s_hpc_13, …, s_hpc_18 -> ella003
   s_hpc_19, …, s_hpc_24 -> ella004
   s_hpc_25, …, s_hpc_30 -> ella005
   ```

3. **Get the code samples:**
   **wget** http://twin.iac.rm.cnr.it/ccc.tgz
   or
   **cp** /work/corsihpc/shared/ccc.tgz     ccc.tgz
4. **Unpack the samples:**
   **tar zxvf ccc.tgz**
5. **Go to source directory:**
   **cd cccsample/source**
6. **Get the CUDA C Quick Reference**
   **wget** http://twin.iac.rm.cnr.it/CUDA_C_QuickRef.pdf
   or
   **cp** /work/corsihpc/shared/CUDA_C_QuickRef.pdf   CUDA_C_QuickRef.pdf

## Hello, World!

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

- To compile: **nvcc –o hello_world hello_world.cu**
- To execute: **./hello_world**
- 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!"

To compile: **nvcc –o simple_kernel simple_kernel.cu**

To execute: **./simple_kernel**

## 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 let us run something a bit more useful:

## A kernel to make an addition

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

- Look at line 28 of the *addint.cu* *code*
  How do we pass the first two arguments?
- Compile:
  **nvcc –o addint addint.cu**
- Run:
  **./addint**

# A More Complex Example

- Another 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 now we use ***pointers*** for all our variables:

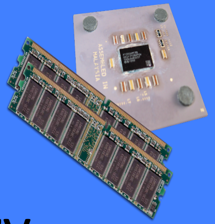
```
 __global__ void add( int *a, int *b, int *c ) {
        *c = *a + *b;
}
```

- **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
    - (In general) May not be dereferenced from host code
  - Host pointers point to CPU memory
    - May be passed to and from device code
    - (In general) 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: `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;
// 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: `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

- Exercise: look at, compile and run the *add_simple_blocks.cu* code

# 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()`

```
        // 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;
}
```

Exercise: compile and run the *add_simple_threads.cu* code

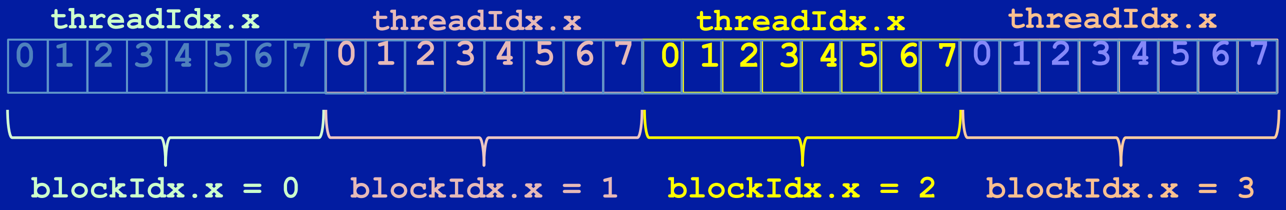
## Using Threads *And* Blocks

- 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 & 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)

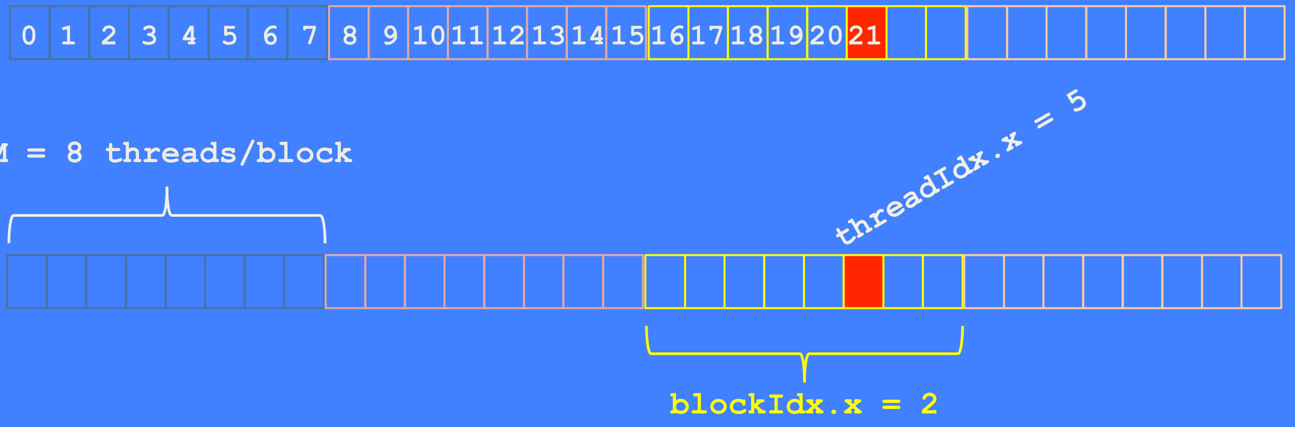


- If we have `M` threads/block, a unique array index for each entry is given by

```
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

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

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

- `gridDim.x` is a built-in variable for blocks in a *grid*;

- 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)

```
#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)

```
    // 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;
}
```

Exercise: compile and run the *add_simple.cu* code

## Exercise: array reversal

- Start from the ***arrayReversal.cu*** code
  The code must fill an array *d_out* with the contents of an input array *d_in* in reverse order so that if *d_in* is [100, 110, 200, 220, 300] then *d_out* must be [300, 220, 200, 110, 100].
- You must complete line 13, 14 and 34.
- Remember that:
  - **`blockDim.x`** is the number of threads per block;
  - **`gridDim.x`** is the number of blocks in a grid;

## Array Reversal: Solution

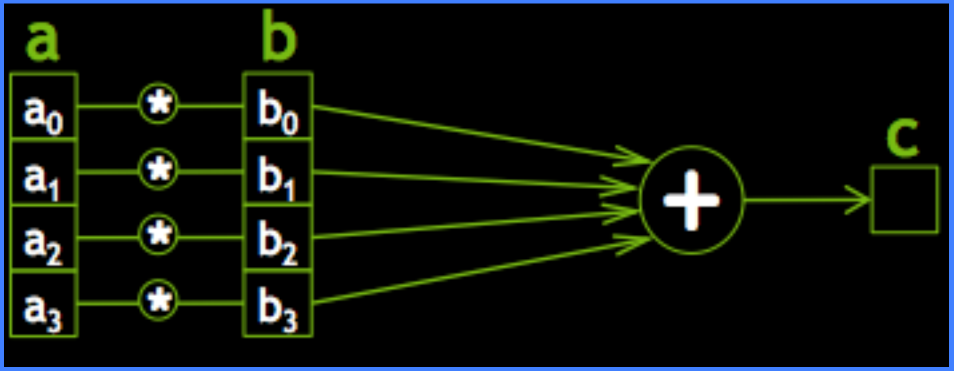
```
__global__ void reverseArrayBlock(int *d_out, int *d_in){
    int inOffset  = blockDim.x * blockIdx.x ;
    int outOffset =  blockDim.x * (gridDim.x - 1 - blockIdx.x)  ;
    int in  = inOffset +   threadIdx.x    ;
    int out = outOffset +  (blockDim.x - 1 - threadIdx. x)  ;
    d_out[out] = d_in[in];
}
```

## 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



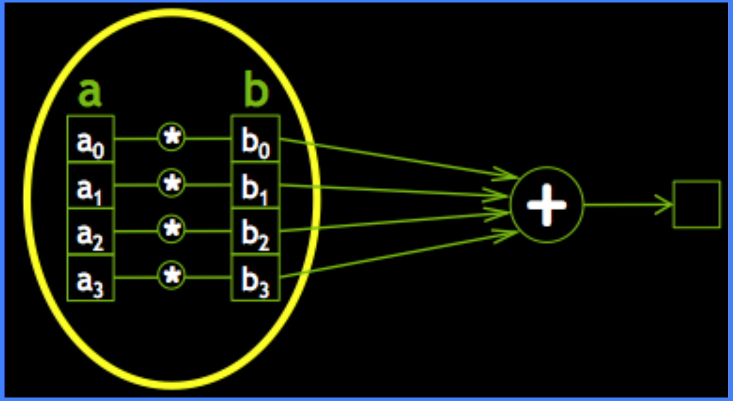
$$c = \vec{\mathbf{a}} \cdot \vec{\mathbf{b}}$$

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

$$c = a_0\, b_0 + a_1\, b_1 + a_2\, b_2 + a_3\, b_3$$

# Dot Product

- Parallel threads have no problem computing the pairwise products:

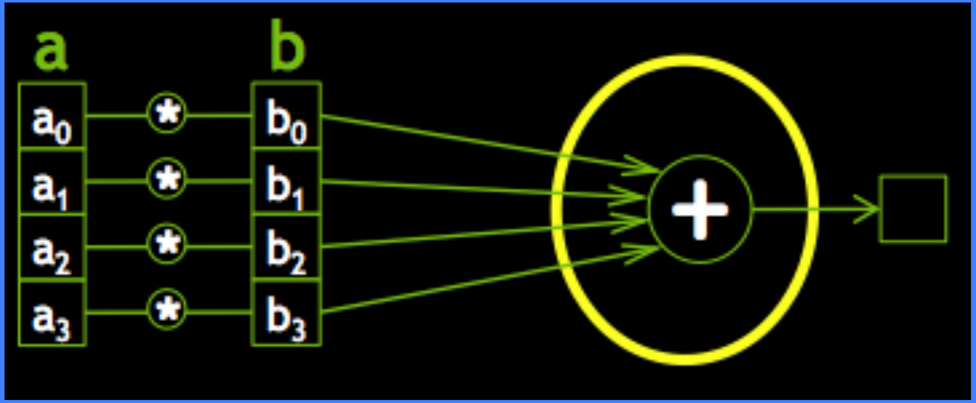


- So we can start a dot product CUDA kernel by doing just that:

```
__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:



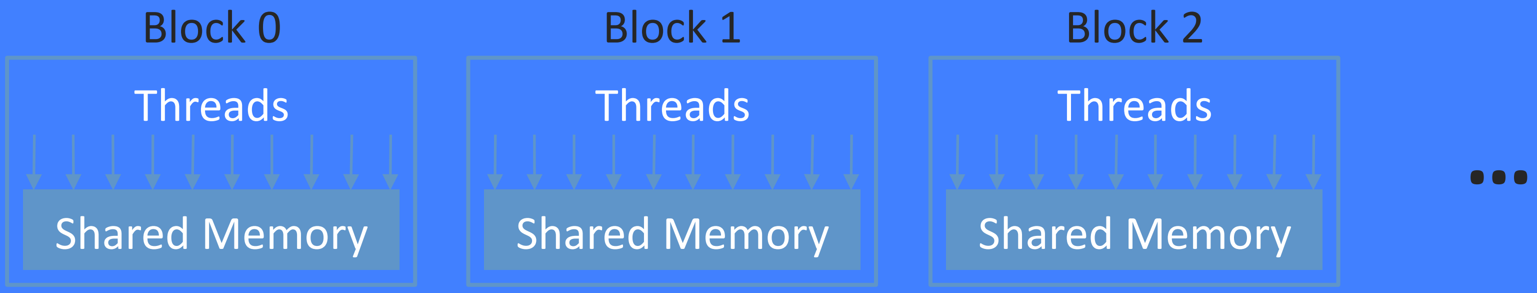
```
__global__ void dot( int *a, int *b, int *c )    {
        // Each thread computes a pairwise product

        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 = N-1; i >= 0; i-- ){
           sum += temp[i];
      }
      *c = sum;
  }
}
```

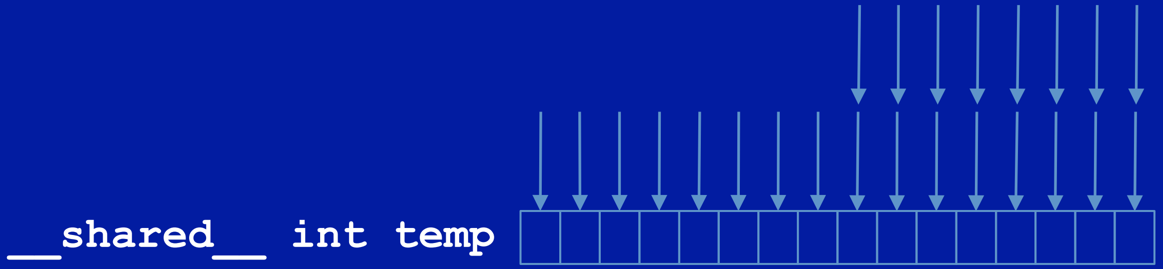
# Parallel Dot Product Recap

- We perform parallel, pairwise multiplications
- Shared memory stores each thread's result
- We sum these pairwise products from a single thread
- Sounds good… But…

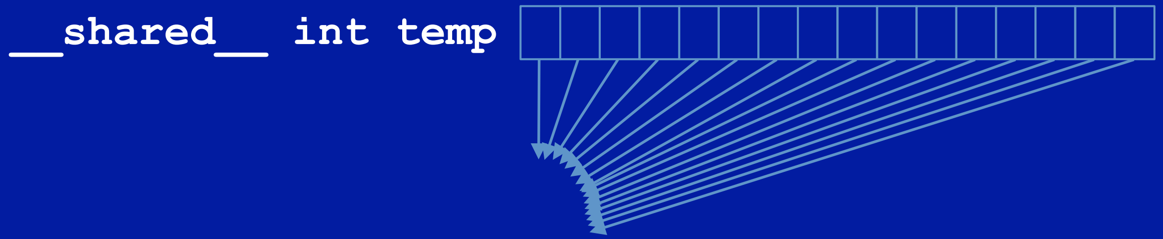
Exercise: Compile and run ***dot_simple_threads.cu***.
Does it work as expected?

# Faulty Dot Product Exposed!

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



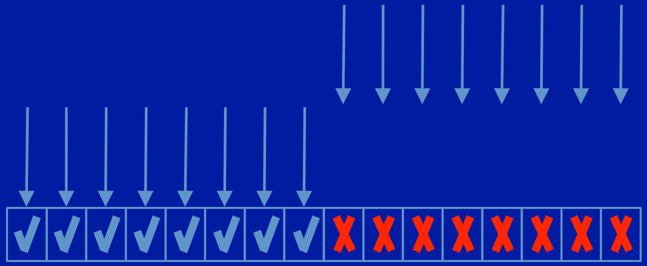
- Step 2: Thread 0 reads and sums the products



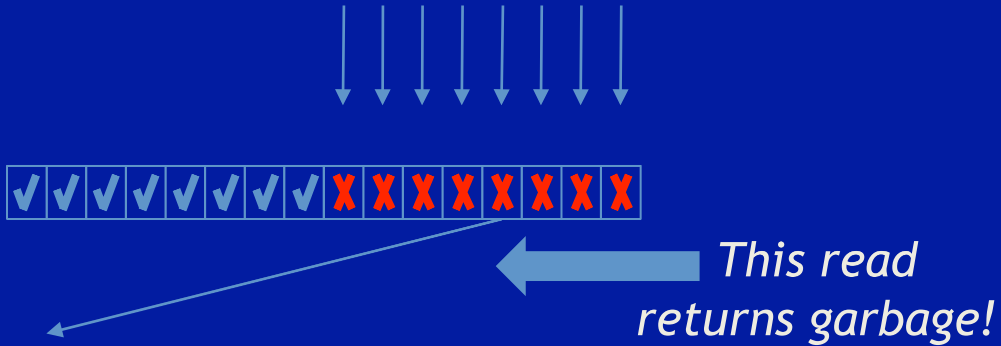
- But there’s an assumption hidden in Step 1…

# Read-Before-Write Hazard

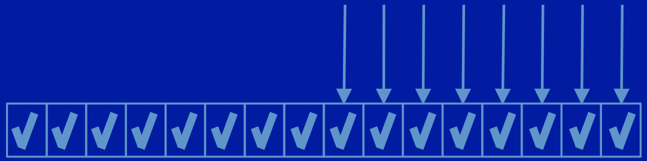
- Suppose thread 0 finishes its write in step 1



- Then thread 0 reads index 12 in step 2



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



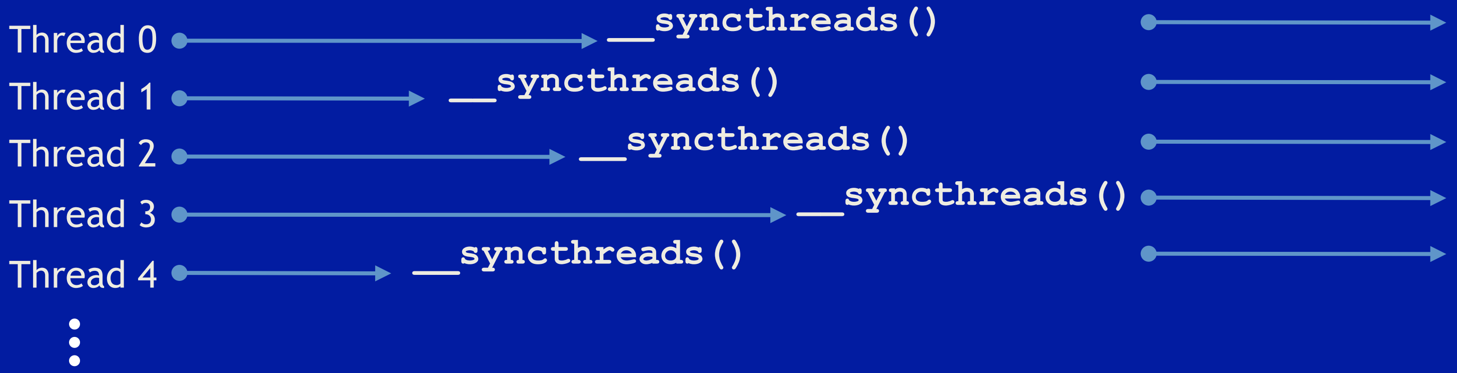
# Synchronization

- 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 = N-1; i >= 0; 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 = N-1; i >= 0; 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 N 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;
}
```

Exercise: insert __syncthreads() in the dot kernel. Compile and run.

## 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:

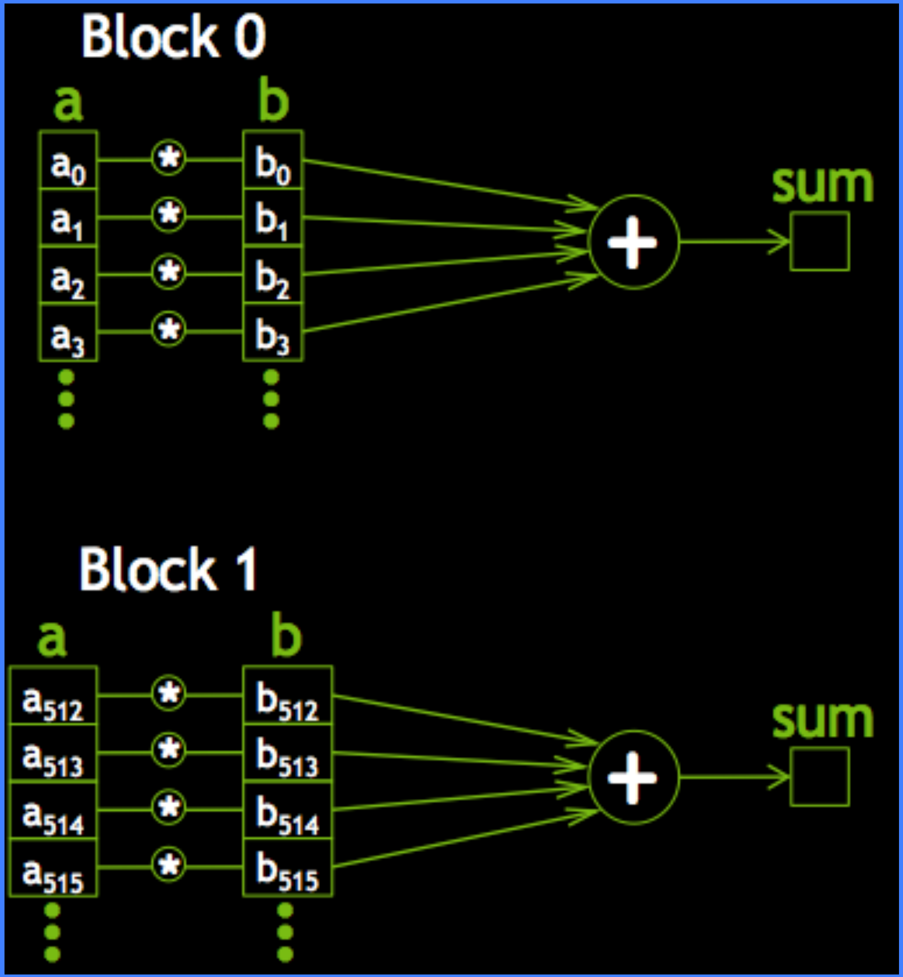
```
// launch dot() kernel with 1 block and N threads
    dot<<< 1, N >>>( dev_a, dev_b, dev_c );
```

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

- Let's write a *multiblock* version of dot product

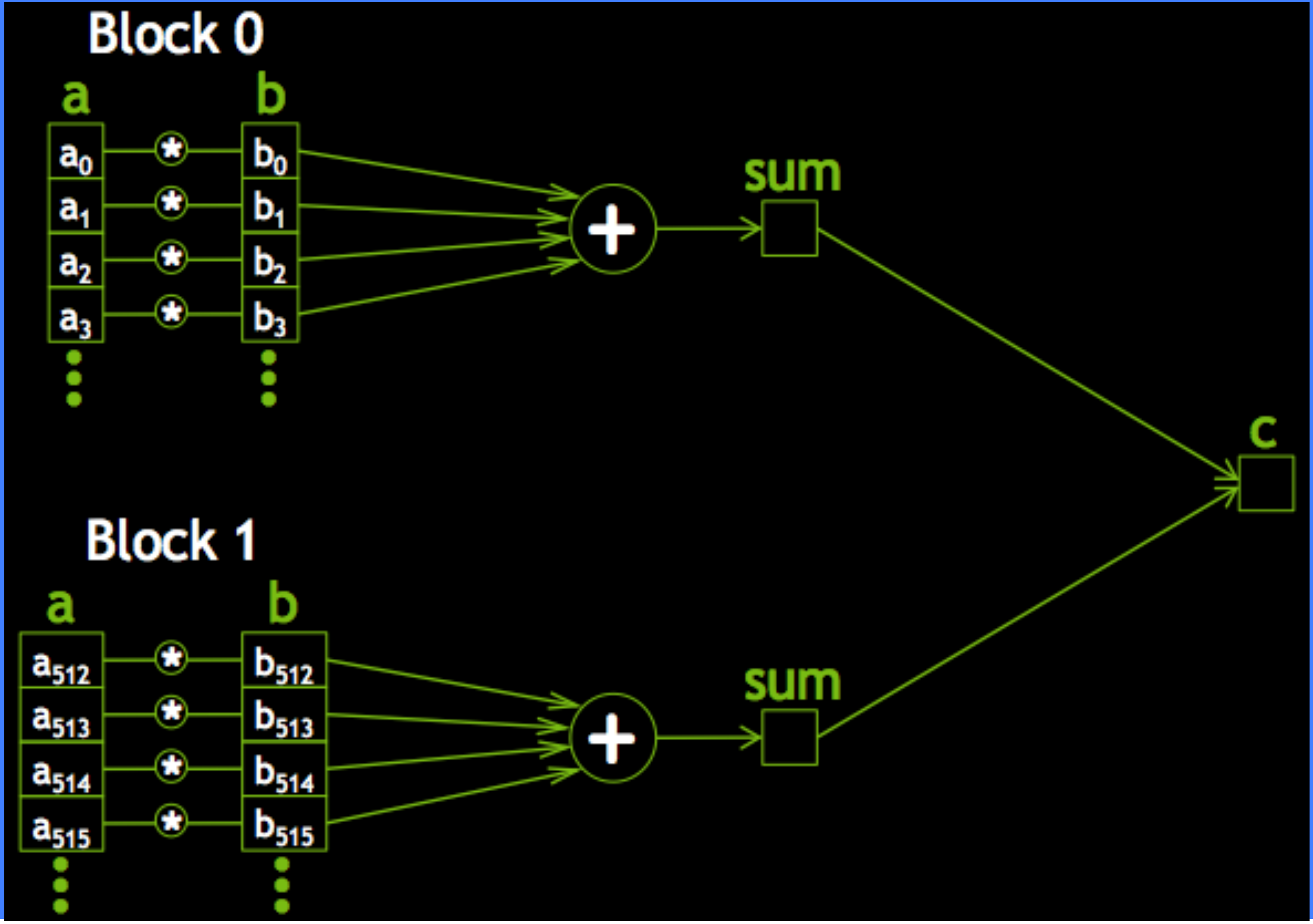
# Multiblock Dot Product: Algorithm

- 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 THREADS_PER_BLOCK  512
#define N  (1024*THREADS_PER_BLOCK)
     __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… Compile and run ***dot_simple_multiblock.cu***
- We can fix it with one of CUDA's atomic operations.
  Use *atomicAdd* in the dot kernel. Compile with
  **nvcc –o dot_simple_multiblock dot_simple_multiblock.cu –arch=sm_20**

# Race Conditions

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

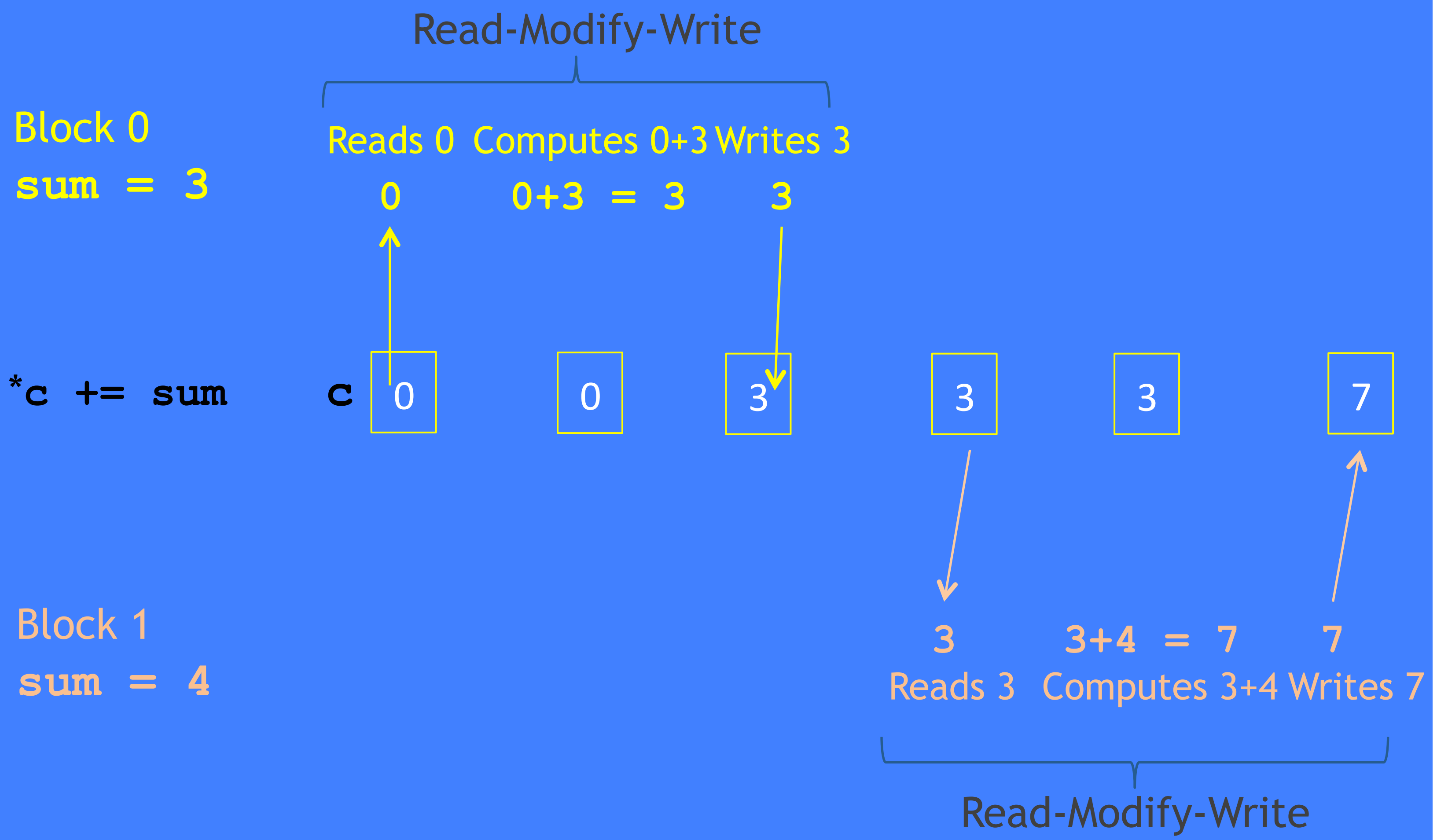
- What actually takes place to execute the line in question: `*c += sum;`
  - Read value at address `c`
  - Add `sum` to value
  - Write result to address `c`

  $\Big\}$ Terminology: *Read-Modify-Write*

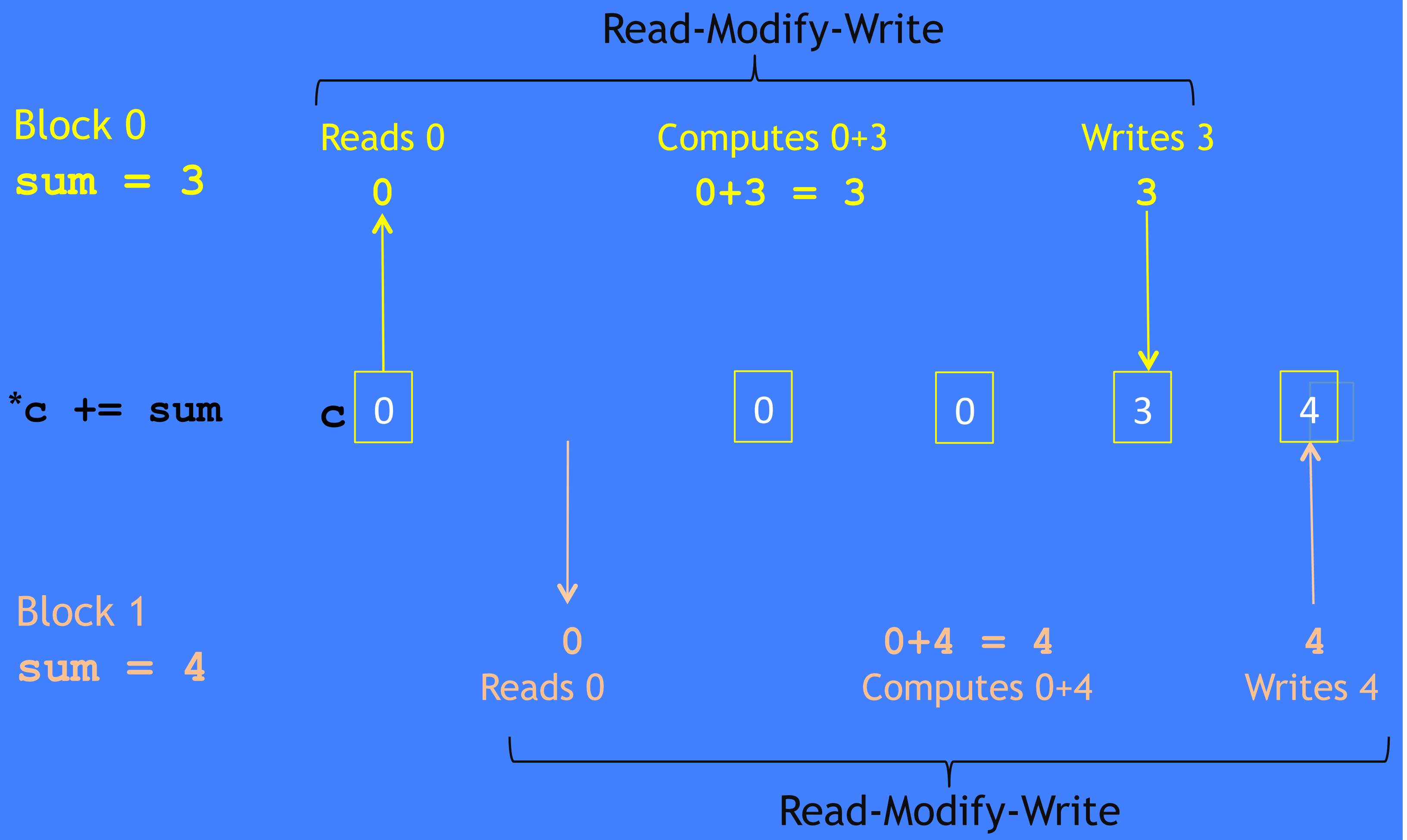
- What if two threads are trying to do this at the same time?

  - 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()` | ▪ `atomicInc()` |
| ▪ `atomicSub()` | ▪ `atomicDec()` |
| ▪ `atomicMin()` | ▪ `atomicExch()` |
| ▪ `atomicMax()` | ▪ `atomicCAS()` `old == compare ? val : old` |

- 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 THREADS_PER_BLOCK 512
#define N  (1024*THREADS_PER_BLOCK)
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;
}
```

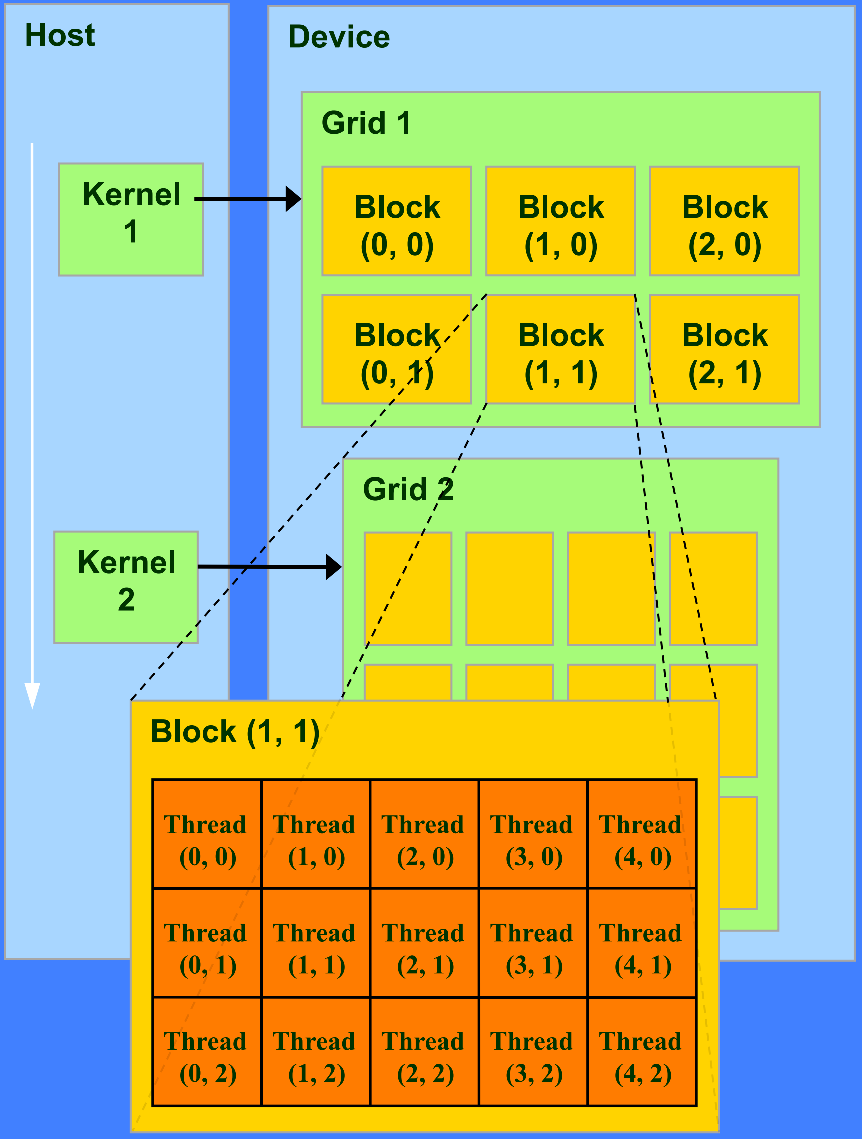
## Review

- 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 use atomic operations a new option (-arch=…) must be used at compile time

## CUDA Thread organization: Grids and Blocks

- A kernel is executed as a 1D or 2D grid of thread blocks
  - All threads share a *global* memory
- A thread block is a 1D, 2D or 3D batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Threads blocks are independent each other and can be executed **in any order**.



Courtesy: NVIDIA

## Built-in Variables to manage grids and blocks

**`dim3`**: a new datatype defined by CUDA;
three unsigned ints where any unspecified component defaults to 1.

- **`dim3 gridDim;`**
  - Dimensions of the grid in blocks (**`gridDim.z`** unused)
- **`dim3 blockDim;`**
  - Dimensions of the block in threads
- **`dim3 blockIdx;`**
  - Block index within the grid
- **`dim3 threadIdx;`**
  - Thread index within the block

## Bi-dimensional threads configuration by example: set the elements of a square matrix (assume the matrix is a single block of memory!)

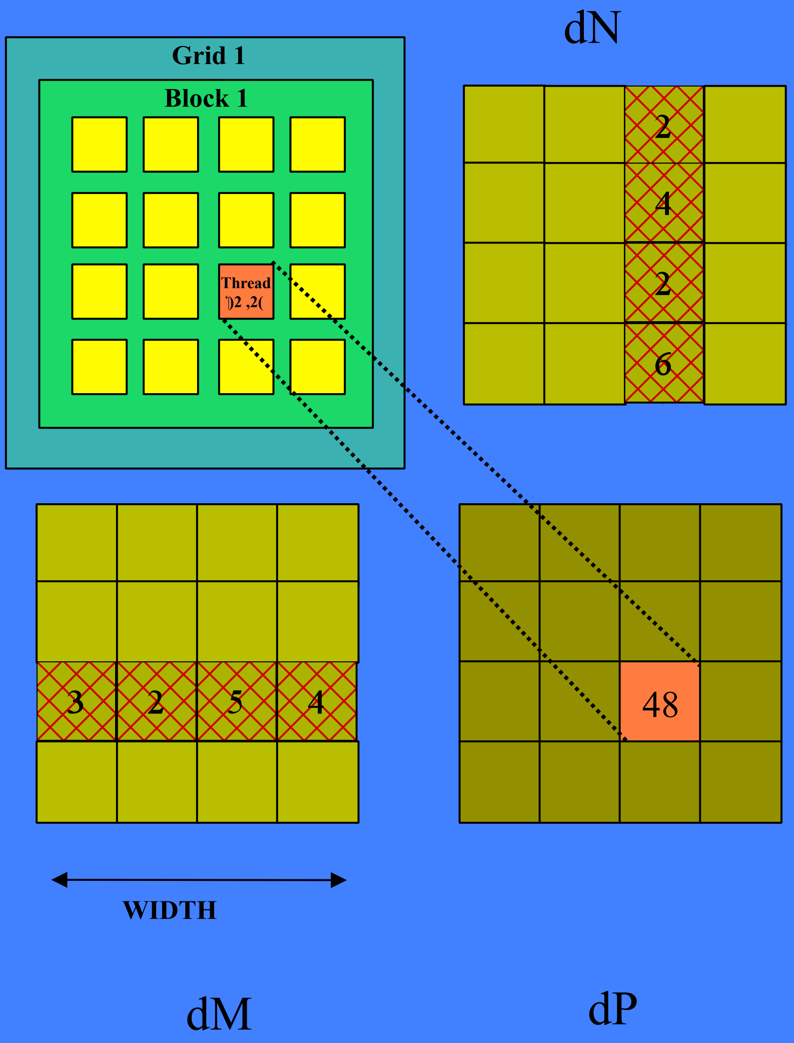
```
__global__ void kernel( int *a, int dimx, int dimy ) {
    int ix   = blockIdx.x*blockDim.x + threadIdx.x;
    int iy   = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx]  = idx+1;
}
```

```
int main() {
    int dimx = 16;
    int dimy = 16;
    int num_bytes = dimx*dimy*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    dim3 grid, block;
    block.x = 4;
    block.y = 4;
    grid.x  = dimx / block.x;
    grid.y  = dimy / block.y;

    kernel<<<grid, block>>>( d_a, dimx, dimy );

    cudaMemcpy(h_a,d_a,num_bytes,
                        cudaMemcpyDeviceToHost);

    for(int row=0; row<dimy; row++) {
       for(int col=0; col<dimx; col++)
          printf("%d ", h_a[row*dimx+col] );
       printf("\n");
    }

    free( h_a );
    cudaFree( d_a );
    return 0;
}
```

Exercise: compile and run: **setmatrix.cu**

# Matrix multiply with one thread block

- One block of threads compute matrix dP
  - Each thread computes one element of dP
- Each thread
  - Loads a row of matrix dM
  - Loads a column of matrix dN
  - Perform one multiply and addition for each pair of dM and dN elements
- Size of matrix limited by the number of threads allowed in a thread block



# Exercise: matrix-matrix multiply

```
__global__ void MatrixMulKernel(DATA* dM,
                              DATA* dN, DATA* dP, int Width) {
DATA Pvalue = 0.0;
  for (int k = 0; k < Width; ++k) {
    DATA Melement = dM[ threadIdx.y*Width+k   ];
    DATA Nelement = dN[  threadIdx.x+Width*k   ];
    Pvalue += Melement * Nelement;
  }
  dP[threadIdx.y*Width+threadIdx.x ] = Pvalue;
}
```

- ✓Start from MMGlob.cu
- ✓Complete the function MatrixMulOnDevice and the kernel MatrixMulKernel
- ✓Use one bi-dimensional block
  - ✓Threads will have an x (threadIdx.x) and an y identifier (threadIdx.y)
- ✓Max size of the matrix: 16

Compile:

**nvcc –o MMGlob MMglob.cu**

Run:

**./MMGlob**

# CUDA Compiler: nvcc basic options

| | |
|---|---|
| ✓ -arch=sm_13 | (or sm_20) Enable double precision (on compatible hardware) |
| ✓ -G | Enable debug for device code |
| ✓ --ptxas-options=-v | Show register and memory usage |
| ✓ --maxrregcount <N> | Limit the number of registers |
| ✓ -use_fast_math | Use fast math library |

✓ -O3 Enables compiler optimization

Exercises:

1. re-compile the MMGlob.cu program and check how many registers the kernel uses:
   ```
   nvcc –o MMGlob MMGlob.cu --ptxas-options=-v
   ```
2. Edit MMGlob.cu and modify DATA from float to double, then compile for double precision and run

## CUDA Error Reporting to CPU

- All CUDA calls return an error code:
  - except kernel launches
  - cudaError_t type

- cudaError_t cudaGetLastError(void)
  - returns the code for the last error (cudaSuccess means “no error”)

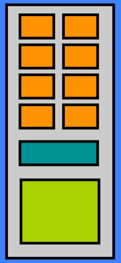
- char* cudaGetErrorString(cudaError_t code)
  - returns a null-terminated character string describing the error

```
printf(“%s\n”,cudaGetErrorString(cudaGetLastError() ) );
```

# Execution Model

**Software** **Hardware**

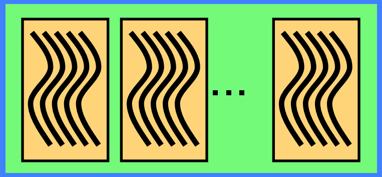
Threads are executed by scalar processors

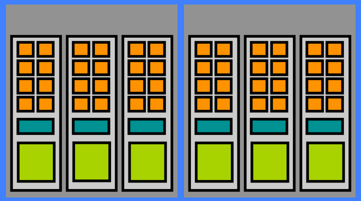


Thread blocks are executed on multiprocessors

Thread blocks **do not migrate**

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)



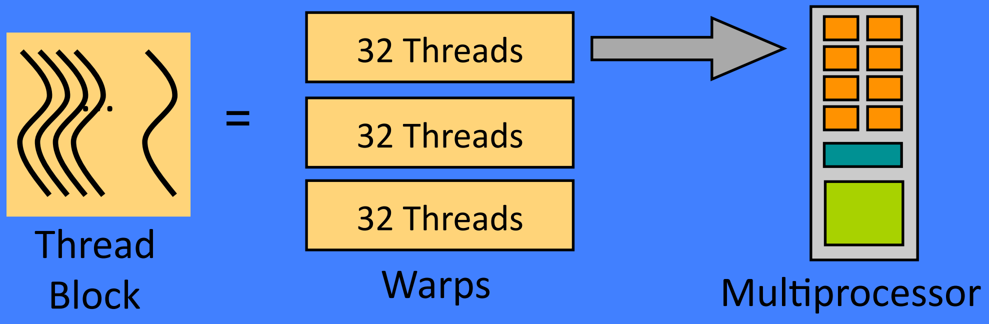


A kernel is launched as a grid of thread blocks

# Fermi 2050 vs. Tesla 1060 & G80

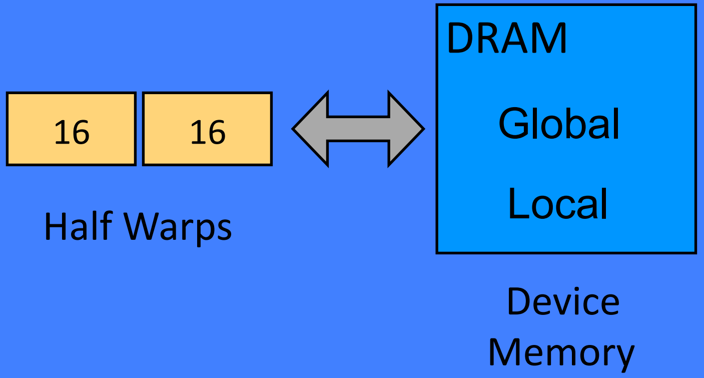
| GPU` | G80 (May 2007) | T 1060 (Nov 2008) | T 2050 (Apr 2010) |
|---|---|---|---|
| **Transistors** | 681 million | 1.4 billion | 3.0 billion |
| **CUDA Cores** | 128 | 240 | 512 |
| **Double Precision Floating Point** | None | 30 FMA ops / clock | 256 FMA ops /clock |
| **Single Precision Floating Point** | 128 MAD ops/clock | 240 MAD ops / clock | 512 FMA ops /clock |
| **Special Function Units / SM** | 2 | 2 | 4 |
| **Warp schedulers (per SM)** | 1 | 1 | 2 |
| **Shared Memory (per SM)** | 16 KB | 16 KB | Configurable 48 KB or 16 KB |
| **L1 Cache (per SM)** | None | None | Configurable 16 KB or 48 KB |
| **L2 Cache** | None | None | 768 KB |
| **ECC Memory Support** | No | No | Yes |
| **Concurrent Kernels** | No | No | Up to 16 |
| **Load/Store Address Width** | 32-bit | 32-bit | 64-bit |

## Warps and Half Warps



A thread block consists of 32-thread warps

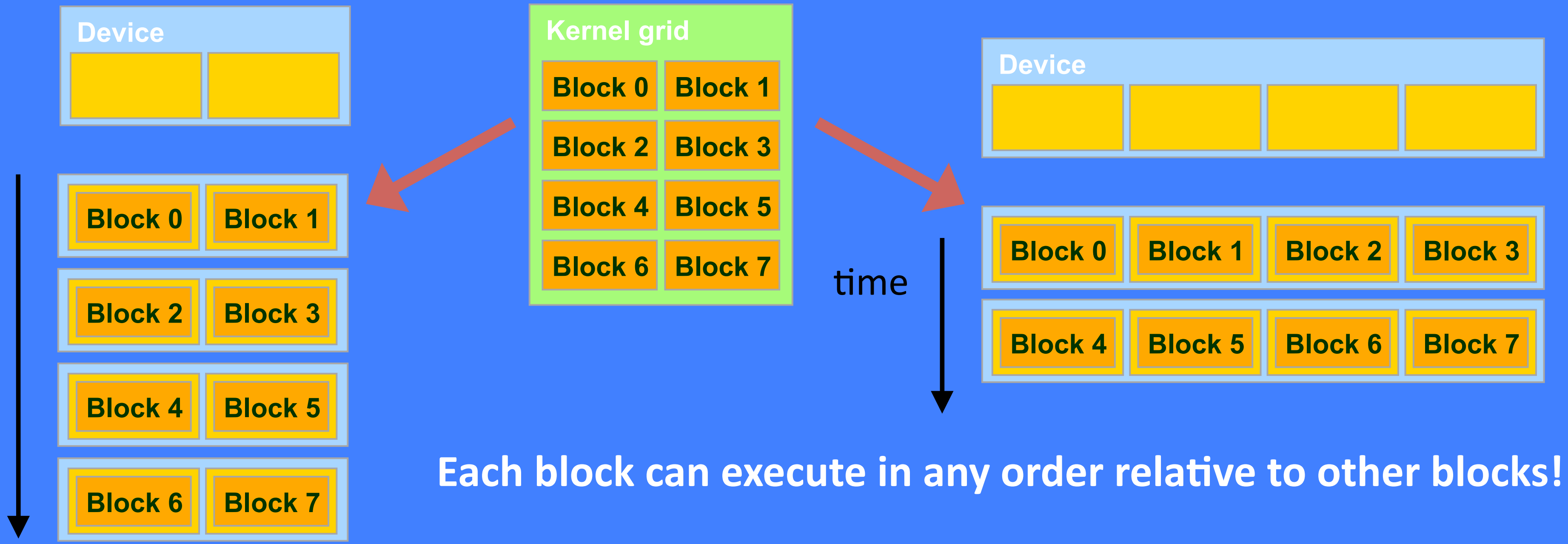
A warp is executed physically in parallel (SIMD) on a multiprocessor



A half-warp of 16 threads can coordinate global memory accesses into a single transaction

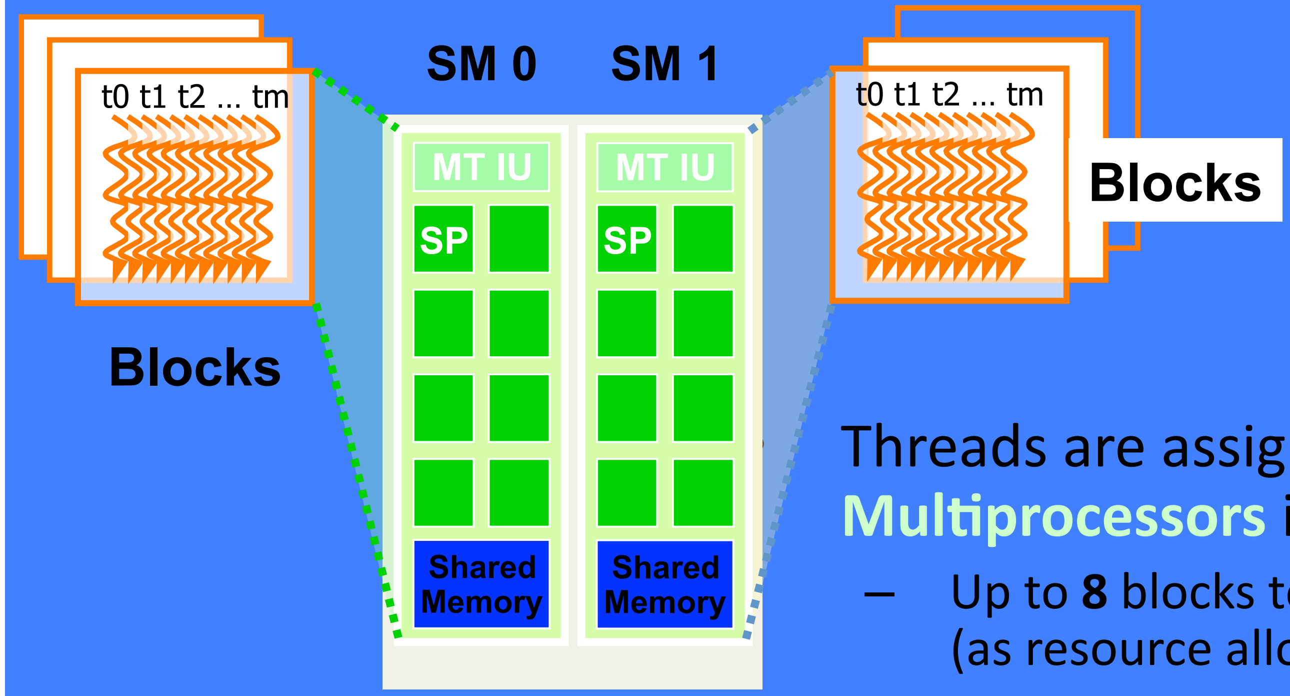
## Transparent Scalability

- Hardware is free to assigns blocks to any processor at any time
  - A kernel scales across any number of parallel processors



**Each block can execute in any order relative to other blocks!**

# Executing Thread Blocks



- Threads are assigned to **Streaming Multiprocessors** in block granularity
  - Up to **8** blocks to each SM (as resource allows)
  - A Fermi SM can take up to 1536 threads
    - Could be 256 (threads/block) * 6 blocks or

      192 (threads/block) * 8 blocks but

      128 (threads/block) * 12 blocks not allowed!
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

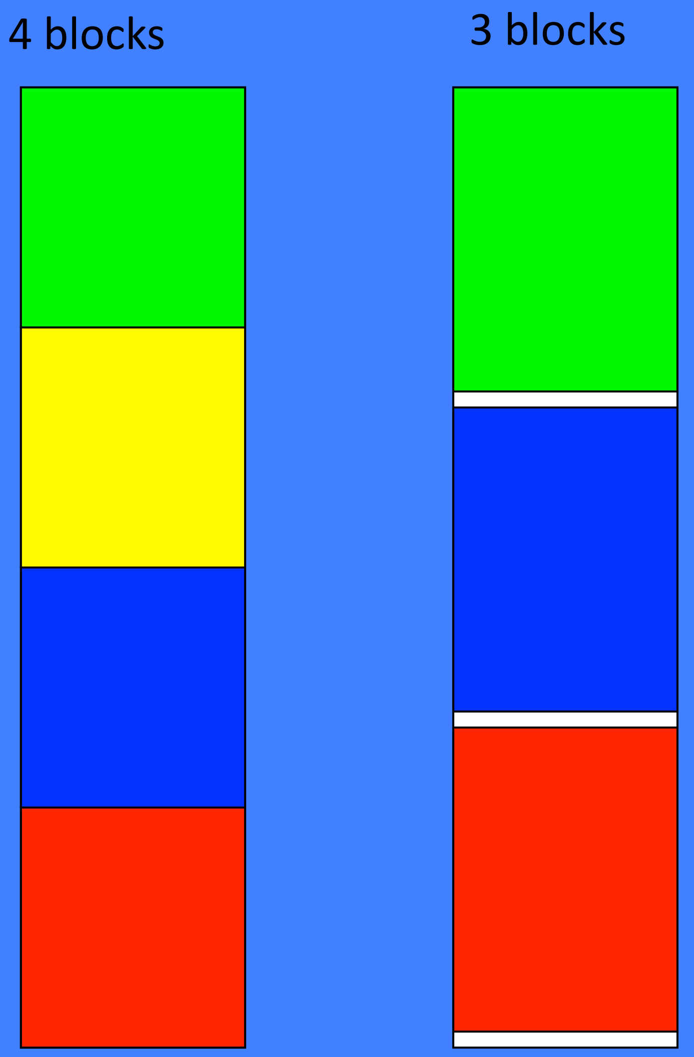
The number of threads in block depends on the capability (*i.e.* version) of the GPU. With Fermi GPU a block may have up to 1024 threads.

## Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should we use 8X8, 16X16 or 32X32 blocks?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 1536 threads, there are 24 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 1536 threads, it can take up to 6 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we have 1024 threads per Block.
    Only one block can fit into a SM!

# Programmer View of Register File

- There are up to 32768 (32 bit) registers in each (Fermi) SM
  - This is an implementation decision, not part of CUDA
  - Registers are dynamically partitioned across all blocks assigned to the SM
  - Once assigned to a block, the register is NOT accessible by threads in other blocks
  - Each thread in the same block only access registers assigned to itself



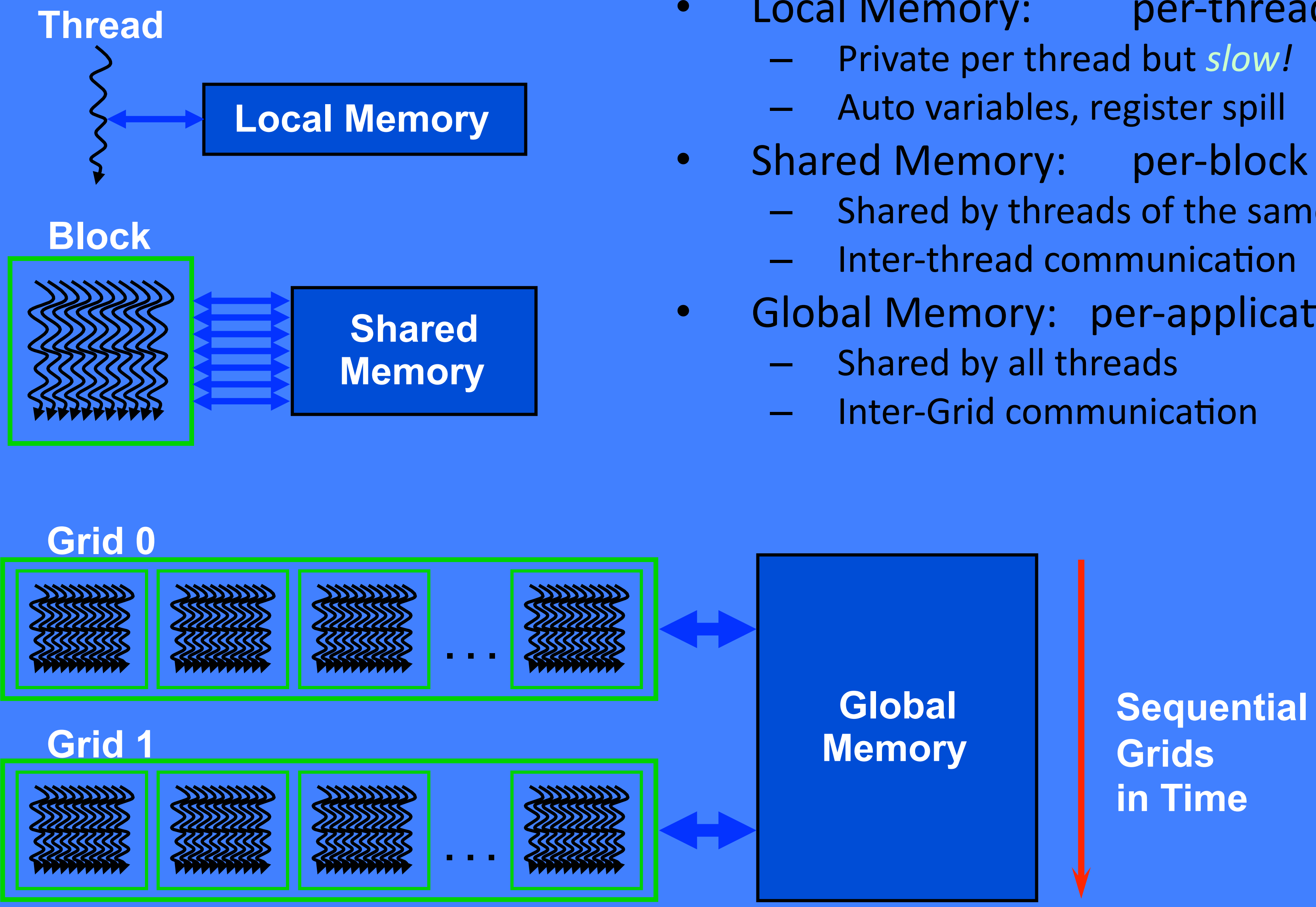
## Registers “occupancy” example

- If a Block has 16*x*16 threads and each thread uses 30 registers, how many threads can run on each SM?
  - Each block requires 30*256 = 7680 registers
  - 32768/7680 = 4 + change
  - So, 4 blocks can run on an SM as far as registers are concerned
- How about if each thread increases the use of registers by 10%?
  - Each  Block now requires 33*256 = 8448 registers
  - 32768/8448 = 3 + change
  - Only three Blocks can run on an SM, **25% reduction of parallelism**!!!

## Optimizing threads per block

- ✓ Choose threads per block as a multiple of warp size (32)
  - ✓ Avoid wasting computation on under-populated warps
  - ✓ Facilitates *coalescing* (efficient memory access)
- ✓ Want to run as many warps as possible per multiprocessor (hide latency)
  - ✓ Multiprocessor can run up to 8 blocks at a time
- ✓ Heuristics
  - ✓ Minimum: 64 threads per block
    - ✓ Only if multiple concurrent blocks
  - ✓ 192 or 256 threads a better choice
    - ✓ Usually still enough registers to compile and invoke successfully
  - ✓ This all depends on your computation, so **experiment, experiment, experiment**!!!

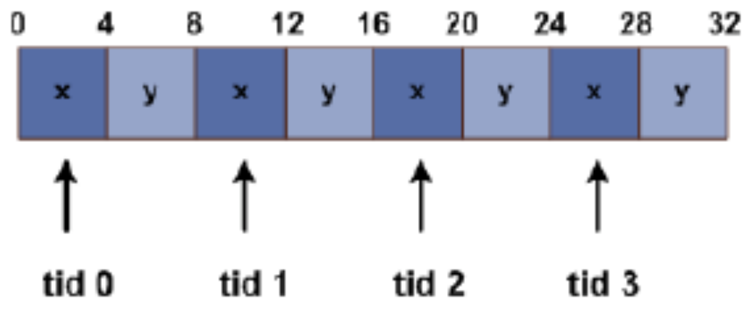
# Parallel Memory Sharing

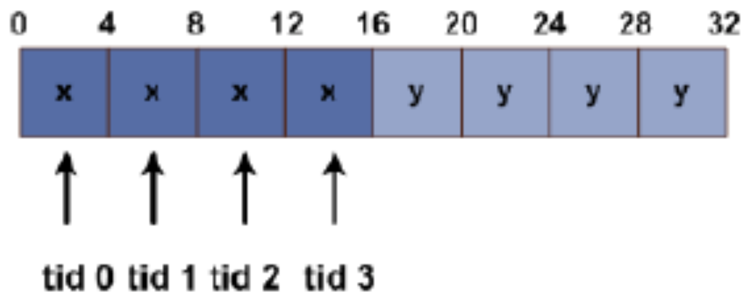


- Local Memory: per-thread
  - Private per thread but *slow!*
  - Auto variables, register spill
- Shared Memory: per-block
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication

# Optimizing Global Memory Access

- Memory Bandwidth is very high (~ 150 Gbytes/sec.) but…
- Latency is also vey high (hundreds of cycles)!
  - *Local* memory has the same latency
- Reduce number of memory transactions by proper memory access pattern.

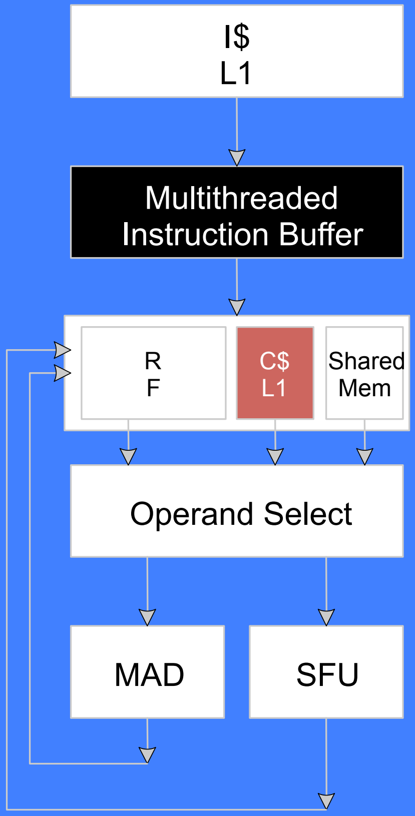




**Thread locality is better than data locality on GPU!**

# Constant Memory

- ✓ Both scalar and array values can be stored
- ✓ Constants stored in DRAM, and cached on chip
  - L1 per SM
- ✓ A constant value can be broadcast to all threads in a Warp
- ✓ Extremely efficient way of accessing a value that is common for all threads in a block!



**cudaMemcpyToSymbol("d_side", &h_side, sizeof(int), 0, cudaMemcpyHostToDevice);**

# Control Flow Instructions

- Main performance concern with branching is divergence
  - Threads within a single warp (group of 32) take different paths
  - Different execution paths are serialized
    - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- A common case: avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - `if (threadIdx.x > 2) { }`
    - This creates two different control paths for threads in a block
    - Branch granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
  - Example without divergence:
    - `if (threadIdx.x / WARP_SIZE > 2) { }`
    - Also creates two different control paths for threads in a block
    - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

# Synchronizing GPU and CPU

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel starts executing once all previous CUDA calls have completed
- Memcopies are synchronous
  - control returns to CPU once the copy is complete
  - copy starts once all previous CUDA calls have completed
- **cudaThreadSynchronize()**
  - CPU code
  - blocks until all previous CUDA calls complete
- Asynchronous CUDA calls provide:
  - non-blocking memcopies
  - ability to overlap memcopies and kernel execution

# Device Management

- CPU can query and select GPU devices
  - cudaGetDeviceCount( int* count )
  - cudaSetDevice( int device )
  - cudaGetDevice( int  *current_device )
  - cudaGetDeviceProperties( cudaDeviceProp* prop,  int device )
  - cudaChooseDevice( int *device, cudaDeviceProp* prop )
- Multi-GPU setup
  - device 0 is used by default
  - Starting on CUDA 4.0 a CPU thread can control more than one GPU
  - multiple CPU threads can control the same GPU
    - calls are serialized by the driver.

## Device Management (sample code)

```
int cudadevice;
struct cudaDeviceProp prop;
cudaGetDevice( &cudadevice );
cudaGetDeviceProperties  (&prop, cudadevice);
mpc=prop.multiProcessorCount;
mtpb=prop.maxThreadsPerBlock;
shmsize=prop.sharedMemPerBlock;
printf("Device %d: number of multiprocessors %d\n"
        "max number of threads per block %d\n"
        "shared memory per block %d\n",
         cudadevice, mpc, mtpb, shmsize);
```

Exercise: compile and run the ***enum_gpu.cu*** code

## Host-Device Data Transfers

- ✓ Device to host memory bandwidth much lower than device to device bandwidth
  - ✓ 8 GB/s peak (PCI-e x16 Gen 2) vs. ~ 150 GB/s peak
- ✓ Minimize transfers
  - ✓ Intermediate data can be allocated, operated on, and deallocated without ever copying them to host memory
- ✓ Group transfers
  - ✓ One large transfer much better than many small ones

## CUDA libraries (1)

- ✓ There are two types of runtime math operations
  - – __func(): direct mapping to hardware ISA
    - Fast but low accuracy
    - Examples: *__sin(x), __exp(x), __pow(x,y)*
  - – func() : compile to multiple instructions
    - Slower but higher accuracy (5 ulp, units in the least place, or less)
    - Examples: *sin(x), exp(x), pow(x,y)*

- ✓ The **`-use_fast_math`** compiler option forces every `funcf()` to compile to `__funcf()`

## CUDA libraries (2)

- ✓ CUDA includes a number of widely used libraries
  - CUBLAS: BLAS implementation
  - CUFFT: FFT implementation
  - CURAND: Random number generation
  - *Etc.*
- ✓ CUDPP (Data Parallel Primitives), available from http://www.gpgpu.org/developer/cudpp:
  - Reduction
  - Scan
  - Sort

## CUFFT

- ✓ CUFFT is the CUDA parallel FFT library
- ✓ CUFFT API is modeled after FFTW. Based on plans, that completely specify the optimal configuration to execute a particular size of FFT
- ✓ Once a plan is created, the library stores whatever state is needed to execute the plan multiple times without recomputing the configuration
  - ✓ Works very well for CUFFT, because different kinds of FFTs require different thread configurations and GPU resources

## Supported Features

- ✓ 1D, 2D and 3D transforms of complex and real-valued data
- ✓ For 2D and 3D transforms, CUFFT performs transforms in row-major (C-order)
  - ✓ If calling from FORTRAN or MATLAB, remember to change the order of size parameters during plan creation
- ✓ In-place and out-of-place transforms for real and complex data.
- ✓ Directions
  - ✓ CUFFT_FORWARD (-1) and CUFFT_INVERSE (1)
    - ✓ According to sign of the complex exponential term
- ✓ Real and imaginary parts of complex input and output arrays are interleaved
  - ✓ cufftComplex type is defined for this

## Code sample: 2D complex to complex transform

```
#define NX 256
#define BATCH 10

cufftHandle plan;
cufftComplex *devPtr;
cudaMalloc((void**)&devPtr, sizeof(cufftComplex)*NX*BATCH);

/* Create a 1D FFT plan. */
 cufftPlan1d(&plan, NX, CUFFT_C2C,BATCH);
 /* Use the CUFFT plan to transform the signal out of place. */
 cufftExecC2C(plan, devPtr, devPtr, CUFFT_FORWARD);
/* Inverse transform the signal in place. */
 cufftExecC2C(plan, devPtr, devPtr, CUFFT_INVERSE);
/* Different pointers to input and output arrays implies out of place transformation */
 /* Destroy the CUFFT plan. */
   cufftDestroy(plan);
   cudaFree(devPtr);
```

Exercise: look at the *cufftsample.cu* code. To compile it:

```
nvcc -o cufftsample cufftsample.cu -lcufft
```

## Multi-GPU Memory

- ✓ GPUs do not share global memory
  - ✓ But starting on CUDA 4.0 one GPU can copy data from/to another GPU memory directly
- ✓ Inter-GPU communication
  - ✓ Application code is responsible for copying/moving data between GPU
  - ✓ Data travel across the PCIe bus
    - ✓ Even when GPUs are connected to the same PCIe switch

## Multi-GPU Environment

- ✓ GPU have consecutive integer IDs, starting with 0
- ✓ Starting on CUDA 4.0, a host thread can maintain more than one GPU context at a time
  - ✓ CudaSetDevice allows to change the “active” GPU (and so the context)
  - ✓ Device 0 is chosen when cudaSetDevice is not called
  - ✓ Note that multiple host threads can establish contexts with the same GPU
    - ✓ Driver handles time-sharing and resource partitioning unless the GPU is in *exclusive* mode

# General inter-GPU Communication

- Application is responsible for moving data between GPUs:
  - Copy data from GPU to host thread A
  - Copy data from host thread A to host thread B
    - Use any CPU library (MPI, ...)
  - Copy data from host thread B to its GPU
- Use asynchronous memcopies to overlap kernel execution with data copies!
  - Require the creation of multiple *streams:* **cudaStreamCreate**

Example:

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(dst, src, size, dir,stream1);
kernel<<<grid, block, 0, stream2>>>(…);
```

overlapped

## Major GPU Performance Detractors

- ✓ Sometimes it is better to recompute than to cache
  - ✓ GPU spends its transistors on ALUs, not memory!
- ✓ Do more computation on the GPU to avoid costly data transfers
  - ✓ Even low parallelism computations can sometimes be faster than transferring back and forth to host

## Major GPU Performance Detractors

- ✓ Partition your computation to keep the GPU multiprocessors equally busy
  - ✓ Many threads, many thread blocks

- ✓ Keep resource usage low enough to support multiple active thread blocks per multiprocessor
  - ✓ Registers, shared memory

# A useful tool: the CUDA **Profiler**

- ✓ What it does:
  - ✓ Provide access to hardware performance monitors
  - ✓ Pinpoint performance issues and compare across implementations
- ✓ Two interfaces:
  - ✓ Text-based:
    - ✓ Built-in and included with compiler

```
setenv CUDA_PROFILE 1
setenv CUDA_PROFILE_LOG XXXXX
```

where XXXXX is any file name you want

  - ✓ GUI: specific for each platform

Exercise: run the MMGlob program under the control of the profiler

## Final CUDA Highlights

- The CUDA API is a minimal extension of the ANSI C programming language

  ➡ Low learning curve

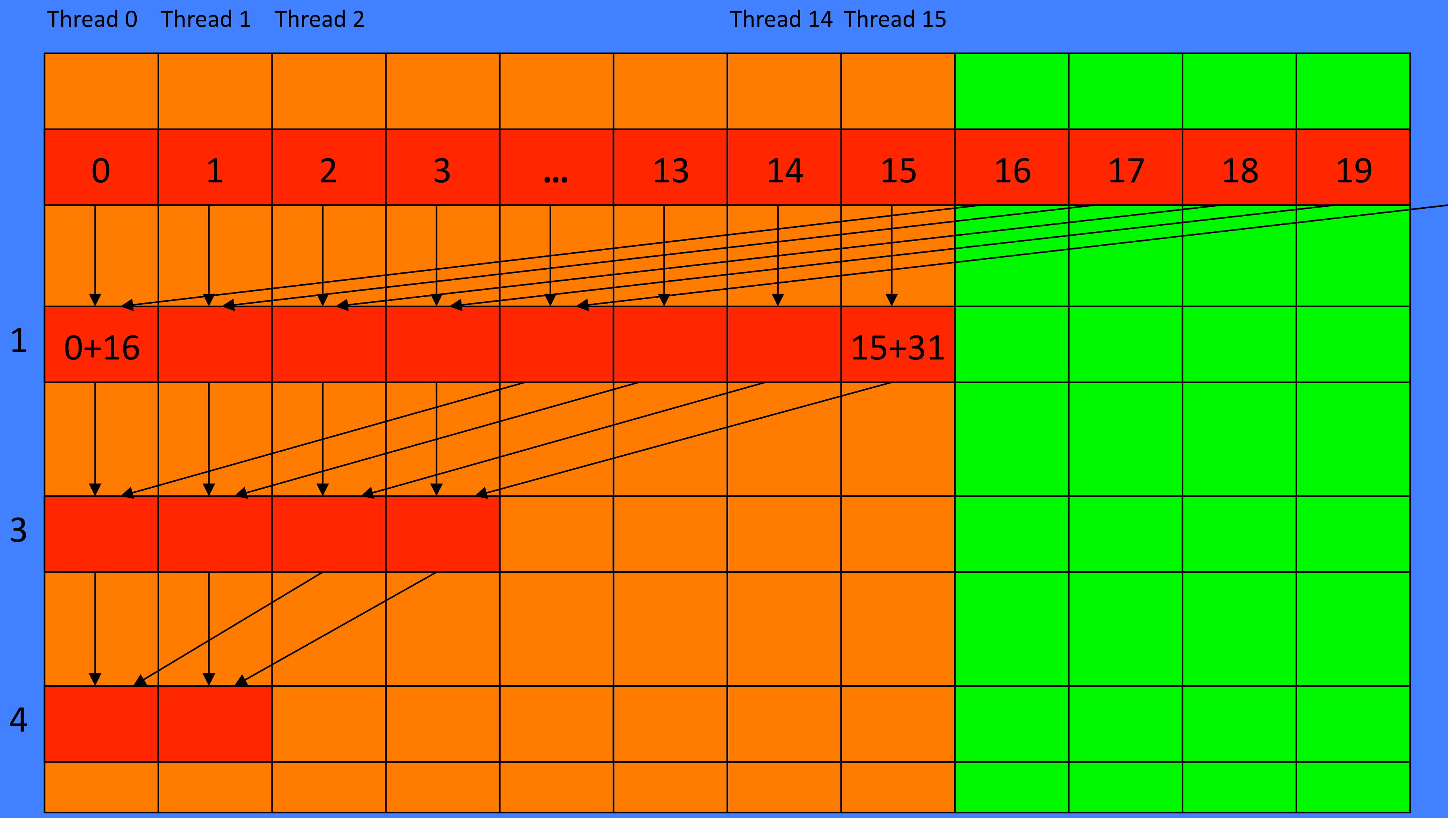
- The memory hierarchy is directly *exposed* to the programmer to allow optimal control of data flow

  ➡ Huge differences in performance with, apparently, ***tiny*** changes to the data organization.

## Homework

- Write a multi block version of the matrix-matrix product
- Write a version of the dot product that computes the sum in each block by using multiple threads.
  Use the scheme in the next slide.
  The final result can be computed either by summing the partial results of each block on the cpu or by calling a new kernel to do that final sum.

# How to compute the sum required by the dot product with multiple threads



## What is missing?

- Texture
- Voting functions (__all(), __any())
- Cuda Events
- Memory Coalescing
- Shared memory bank conflicts
- Variants of cudaMemcpy()

**Thanks!**