

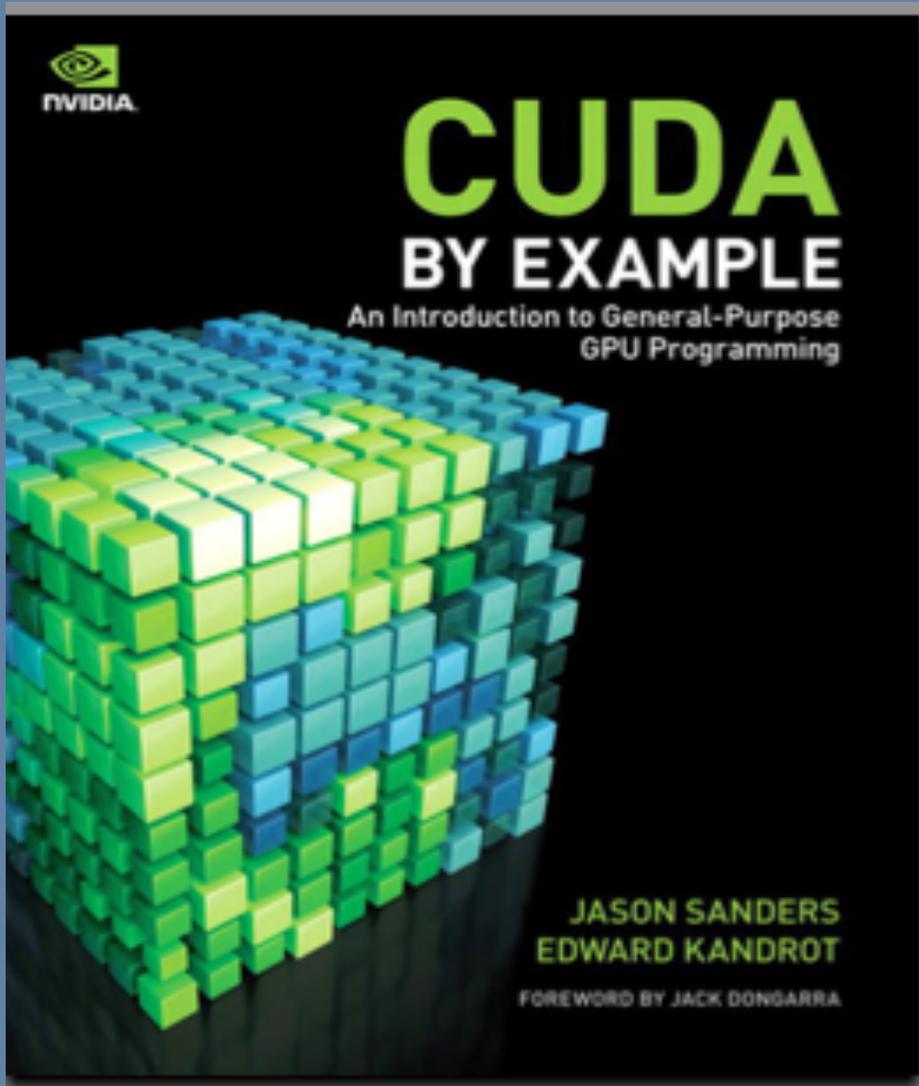
A crash course on  
(C-) CUDA programming for (multi) GPU

**Massimo Bernaschi**

*[m.bernaschi@iac.cnr.it](mailto:m.bernaschi@iac.cnr.it)*

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

# Single GPU part of the class based on



Original slides by Jason Sanders  
by courtesy of Massimiliano Fatica.

Code samples available from

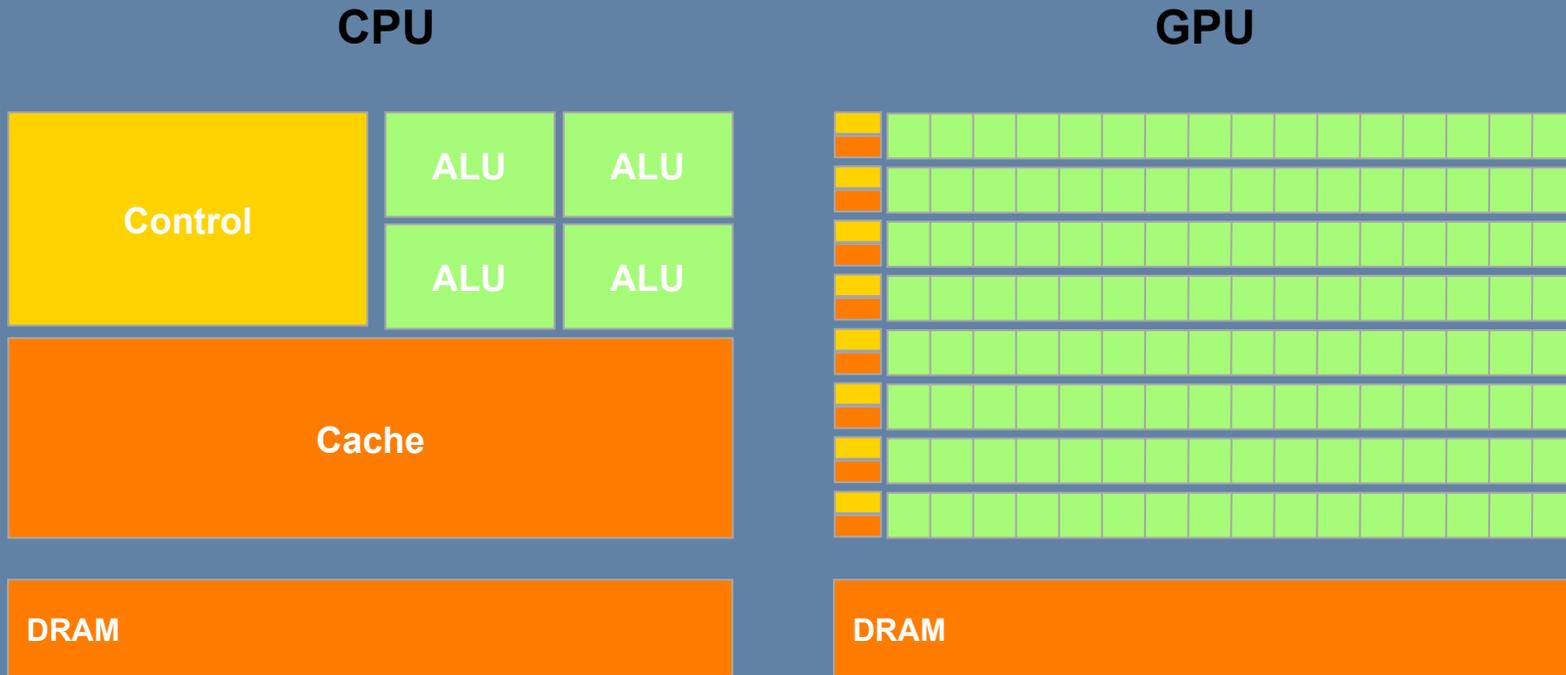
(**wget**) <http://twin.iac.rm.cnr.it/ccc.tgz>

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!



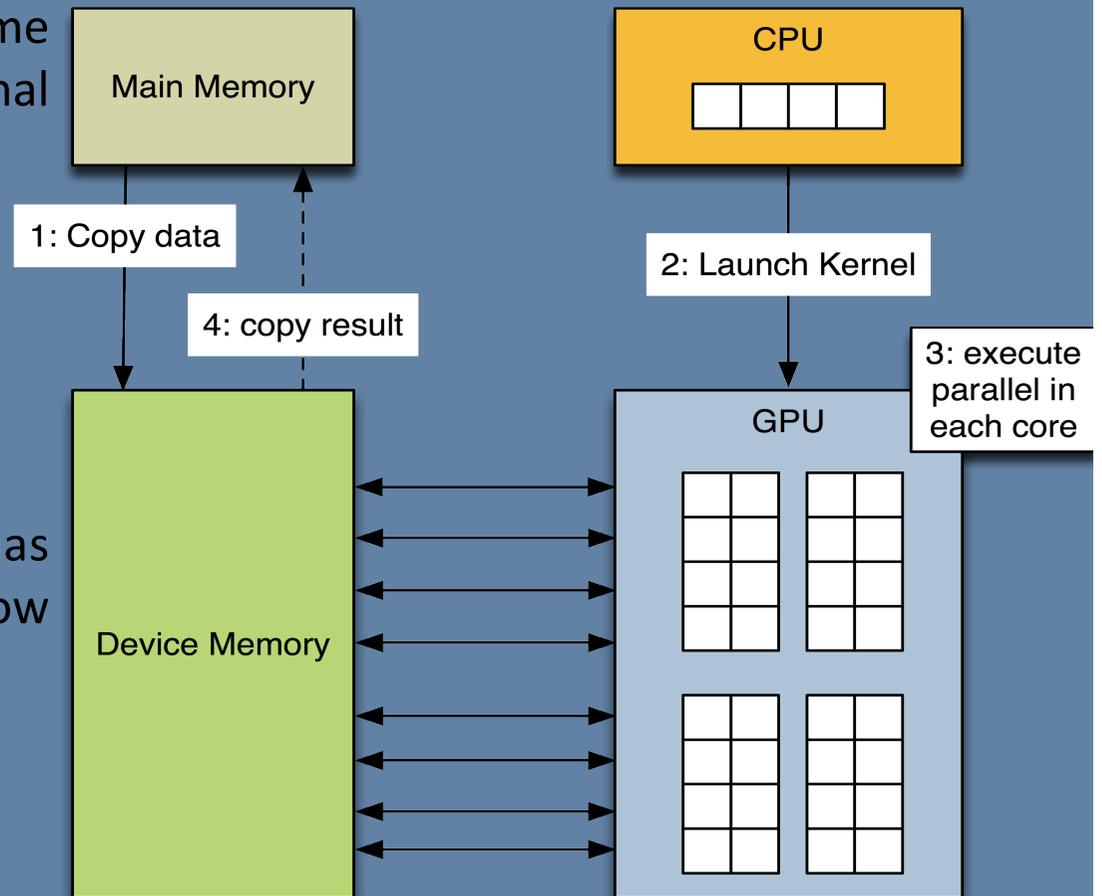
And there is no GPU without a CPU...

# Graphics Processing Unit

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

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

✓ Traditionally GPU programming has been tricky but **CUDA, OpenCL** and now **OpenACC** 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.

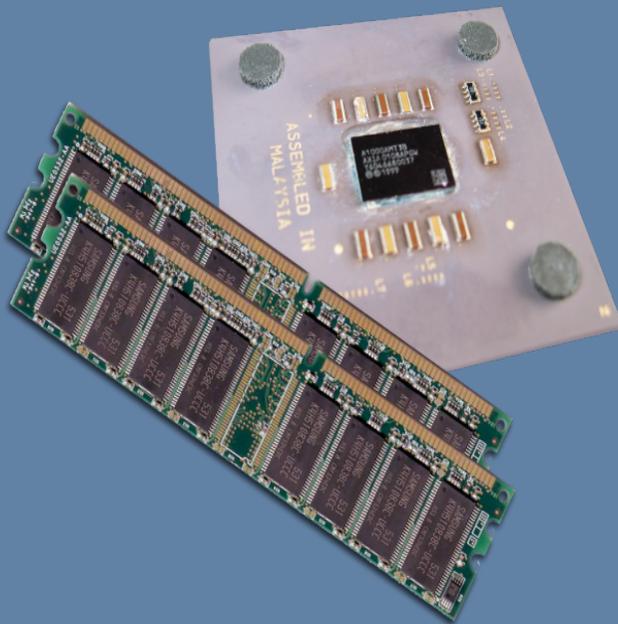
# CUDA Programming Model

- The GPU is viewed as a compute device that:
  - has its own RAM (**device memory**)
  - runs data-parallel portions of an application as **kernels** by 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 (basically one thread *per core*)

# CUDA C Jargon: The Basics

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

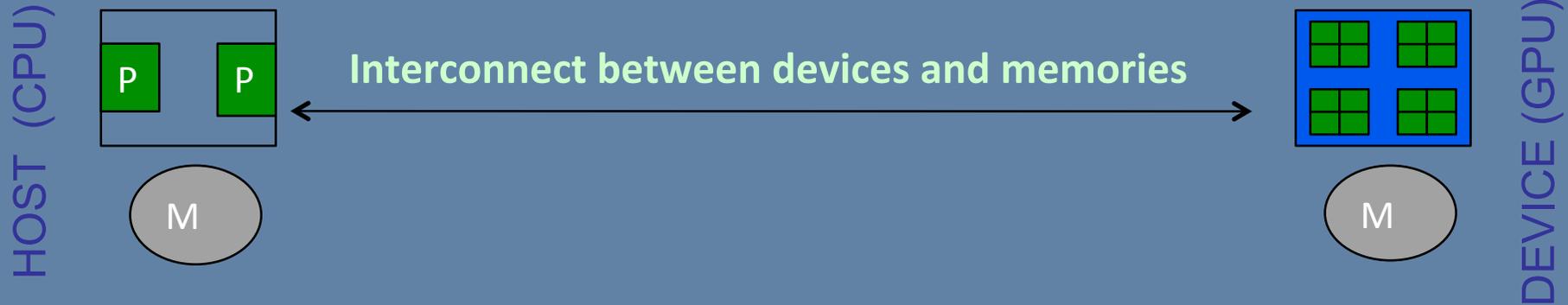
Host



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.* `__syncthreads()`

# Code Samples

## 1. Login:

```
ssh -o TCPKeepAlive=yes -o ServerAliveInterval=15 -p 55567  
cudaXX@94.82.133.184
```

**dove XX va da 1 a 27**

**(controllare la lista per vedere l'utenza assegnata)**

## 2. Unpack the samples:

```
tar zxvf ccc.tgz
```

## 3. Go to source directory:

```
cd cccsample/source
```

## 4. Get the CUDA C Quick Reference

```
wget http://twin.iac.rm.cnr.it/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;  
    }  
}
```

To compile: `nvcc -o simple_kernel simple_kernel.cu`

To execute: `./simple_kernel`

- 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, icc, ...`
    - **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
  - A “kernel launch” in CUDA jargon
  - 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;  
}
```

- Compile:

```
nvcc -o addint addint.cu
```

- Run:

```
./addint
```

- Look at line 28 of the *addint.cu* code

How do we pass the first two arguments?

# 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

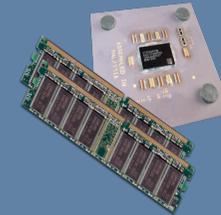
- Notice that now we use *pointers* for all our variables:

```
__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

- Up to CUDA 4.0 host and device memory were distinct entities from the programmers' viewpoint
  - 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



Starting on CUDA 4.0 there is a **Unified Virtual Addressing** feature.

- Basic CUDA API for dealing with device memory
  - `cudaMalloc (&p, size), cudaFree (p),`  
`cudaMemcpy (t, s, size, direction)`
  - Similar to their C equivalents: `malloc ()`, `free ()`, `memcpy ()`

**pointer to pointer**

# 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 a different index
- `blockIdx.x` is the first example of a CUDA predefined variable.

# 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 >>>()` ;
  - `blockIdx.x` allows 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 blocks
add<<< N, 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;
}
```

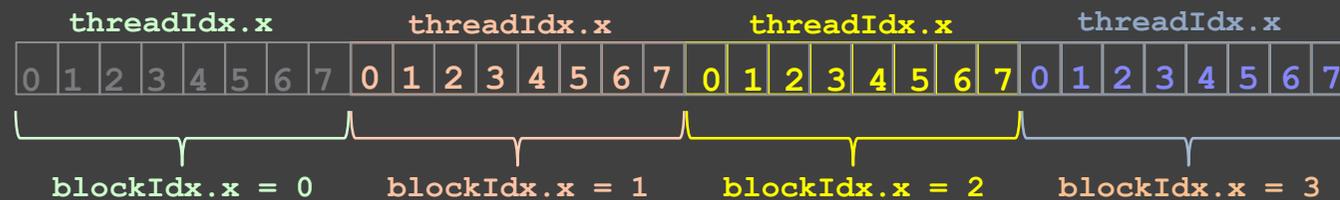
Exercise: compile and run the *add\_simple\_threads.cu* code

# 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)

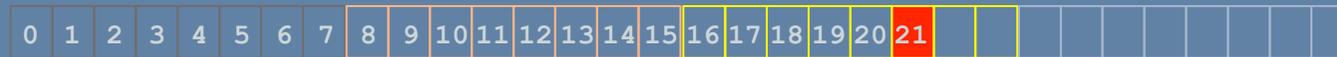


- If we have  $M$  threads/block, a unique array index for each entry is given by

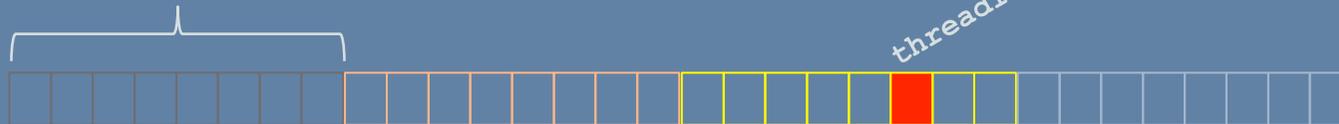
```
int index = threadIdx.x + blockIdx.x * M;  
int index = x + y * width;
```

# Indexing Arrays: Example

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



`M = 8 threads/block`



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

# Indexing Arrays: other examples (4 blocks with 4 threads *per* block)

```
__global__ void kernel( int *a )  
{  
    int idx = blockIdx.x*blockDim.x + threadIdx.x;  
    a[idx] = 7;  
}
```

Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```
__global__ void kernel( int *a )  
{  
    int idx = blockIdx.x*blockDim.x + threadIdx.x;  
    a[idx] = blockIdx.x;  
}
```

Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

```
__global__ void kernel( int *a )  
{  
    int idx = blockIdx.x*blockDim.x + threadIdx.x;  
    a[idx] = threadIdx.x;  
}
```

Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

# 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

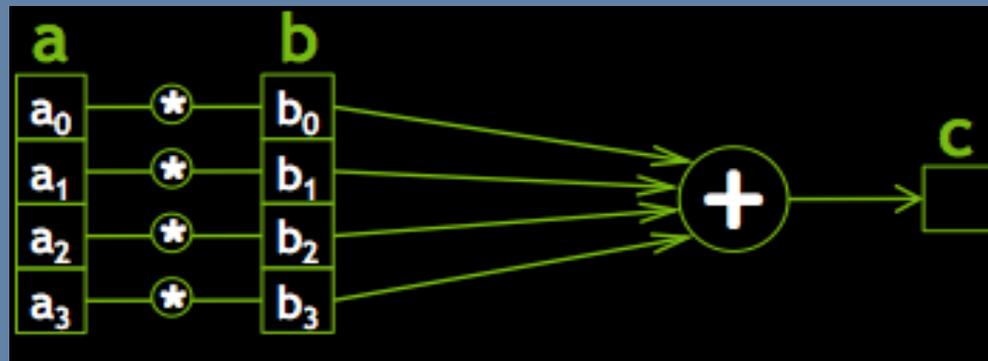
```
__global__ void reverseArrayBlock(int *d_out, int *d_in) {  
    int in = ;  
    int out = ;  
    d_out[out] = d_in[in];  
}
```

# 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



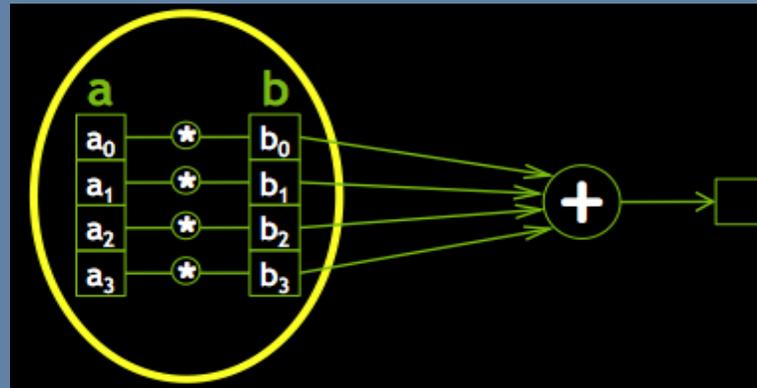
$$c = \vec{a} \cdot \vec{b}$$

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

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

# Dot Product

- Parallel threads have no problem computing the pairwise products:

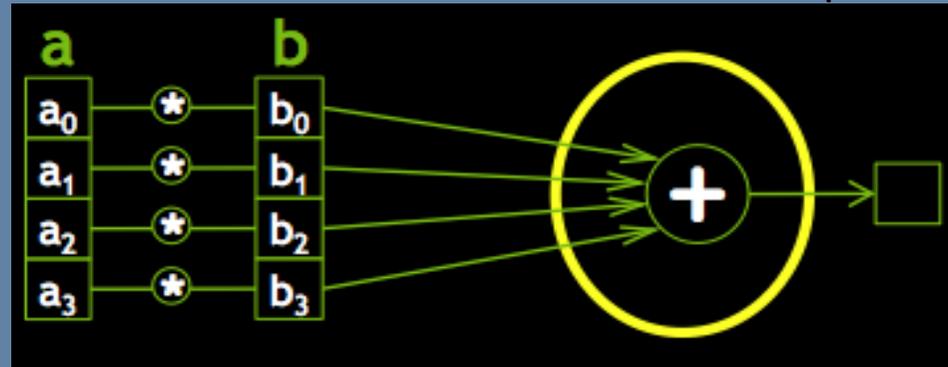


- So we can start a dot product CUDA kernel by doing just that:

```
__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

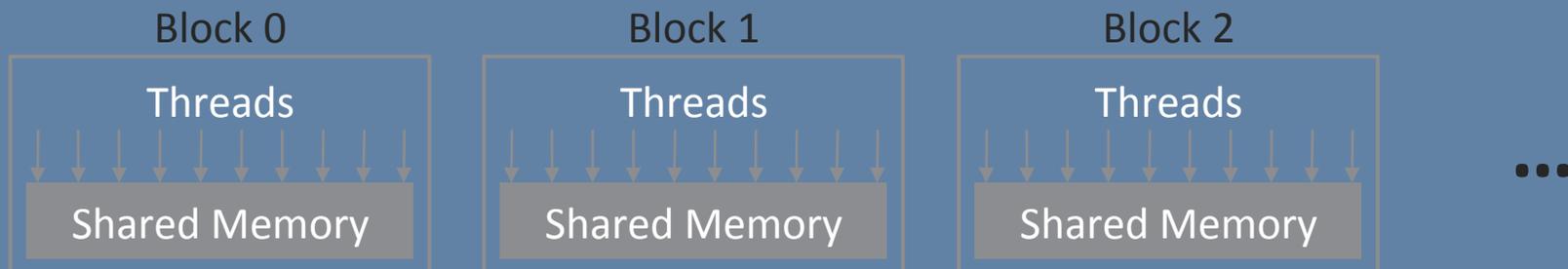
- But we need to share data between threads to compute the final sum:



```
__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;
    }
}
```

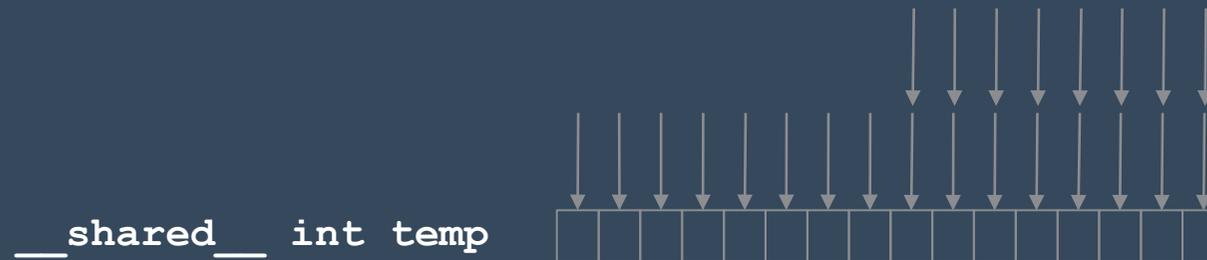
# Parallel Dot Product Recap

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

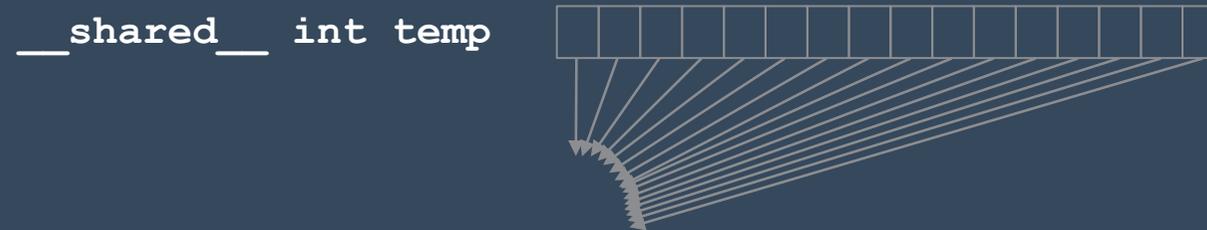
Exercise: Compile and run *dot\_simple\_threads.cu*.  
Does it work as expected?

# Faulty Dot Product Exposed!

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



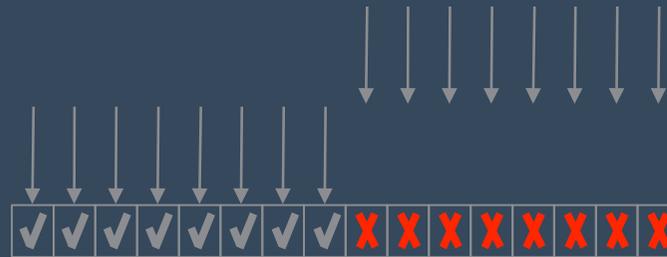
- Step 2: Thread 0 reads and sums the products



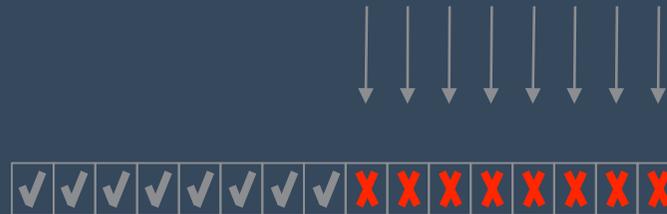
- But there's an assumption hidden in Step 1...

# Read-Before-Write Hazard

- Suppose thread 0 finishes its write in step 1

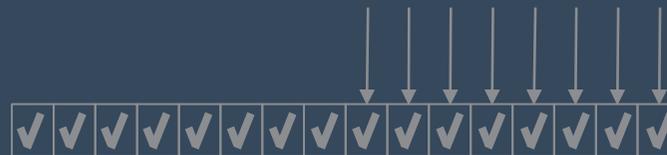


- Then thread 0 reads index 12 in step 2



← This read returns garbage!

- Before thread 12 writes to index 12 in step 1



# Synchronization

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

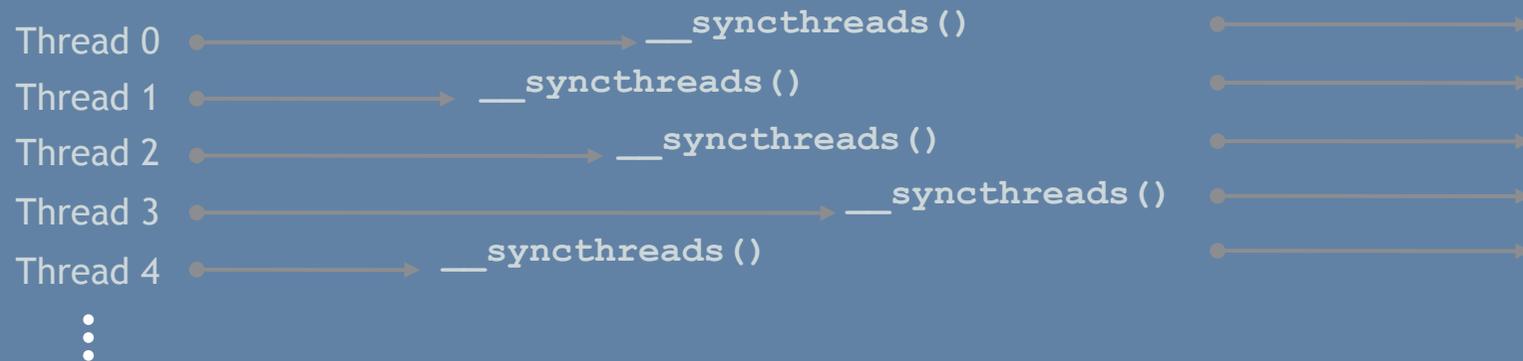
```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // * 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

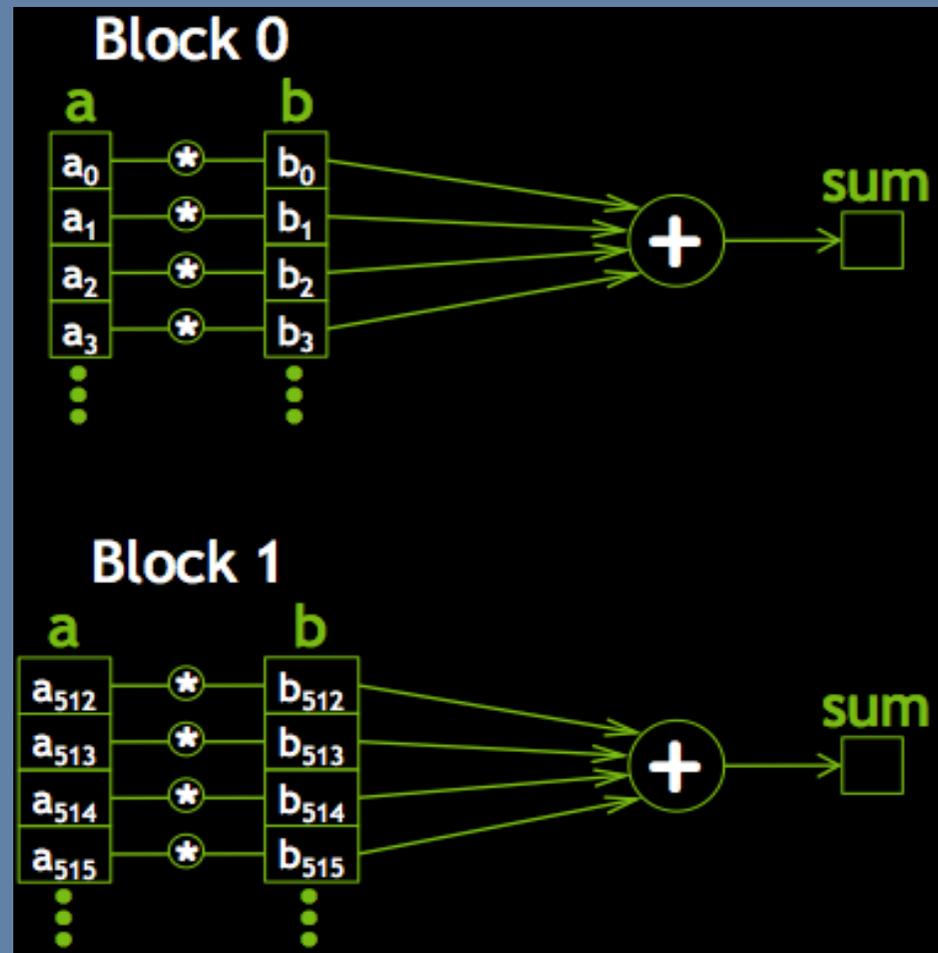
- Recall our dot product launch:

```
// 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

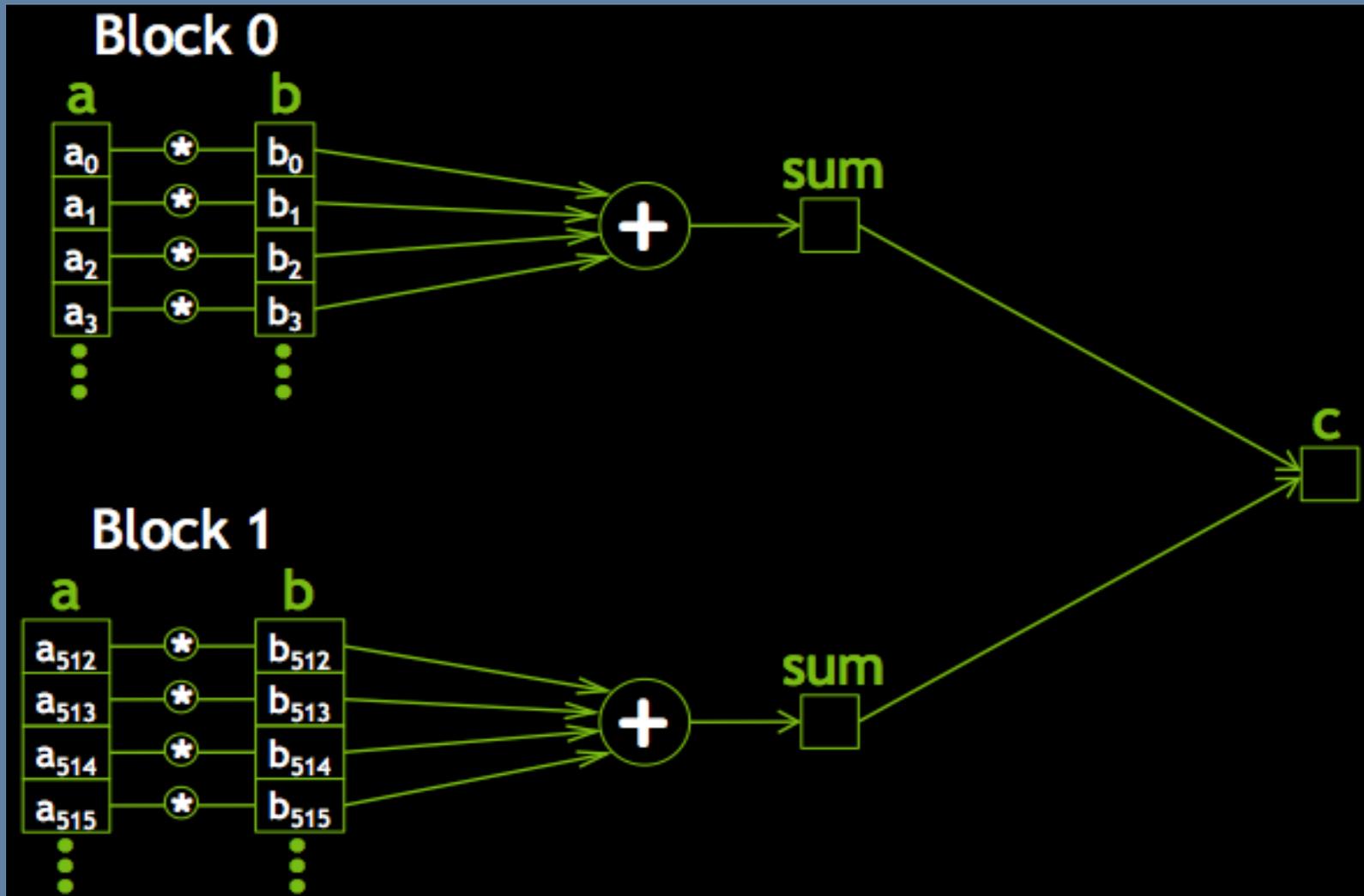
# Multiblock Dot Product: Algorithm

- 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];
        }
        atomicAddatomicAdd( c , sum );
    }
}
```

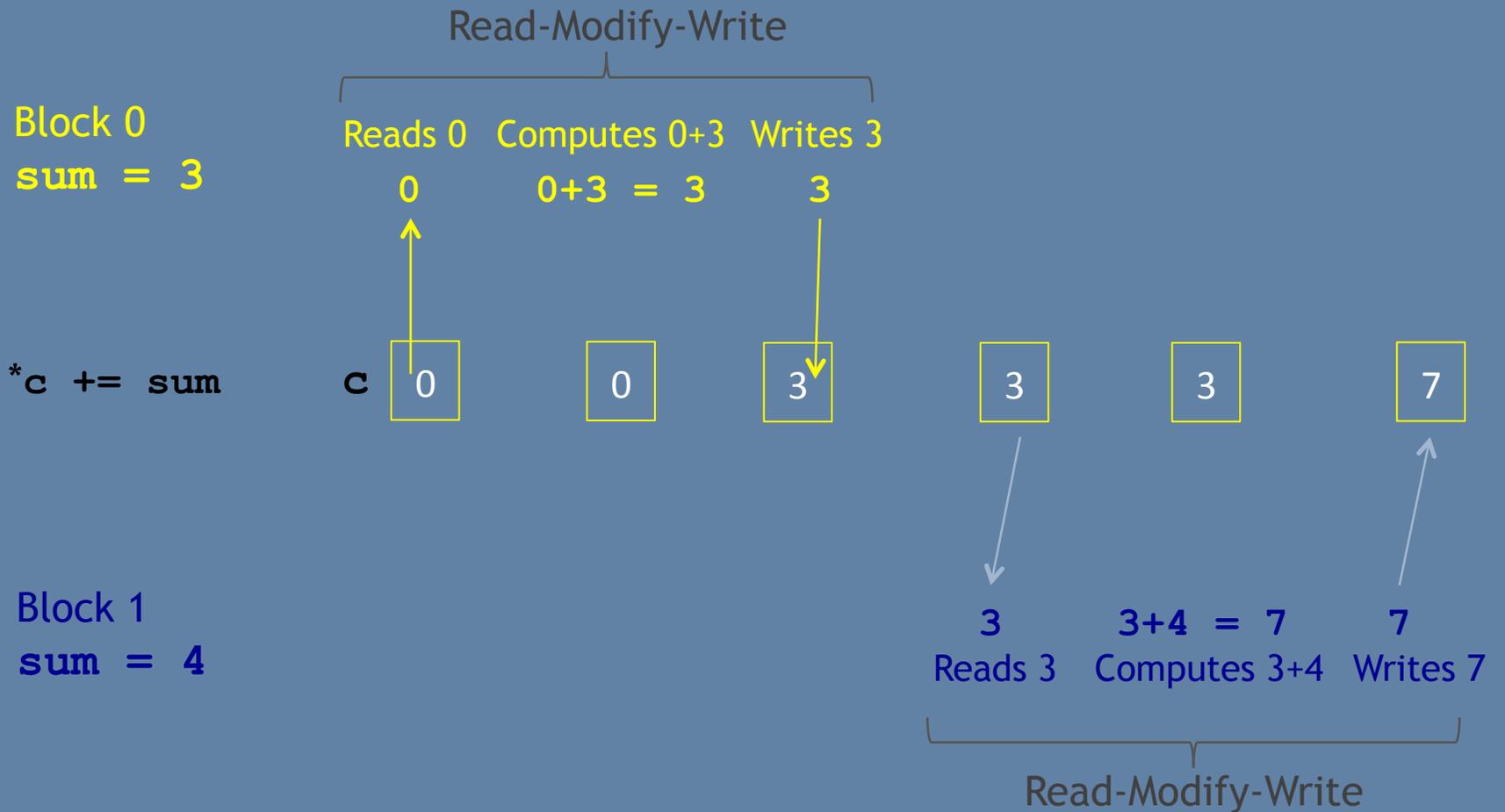
- But we have a race condition... Compile and run *dot\_simple\_multiblock.cu*
- We can fix it with one of CUDA's atomic operations. Use *atomicAdd* in the dot kernel. Compile with `nvcc -o dot_simple_multiblock dot_simple_multiblock.cu -arch=sm_35`

# Race Conditions

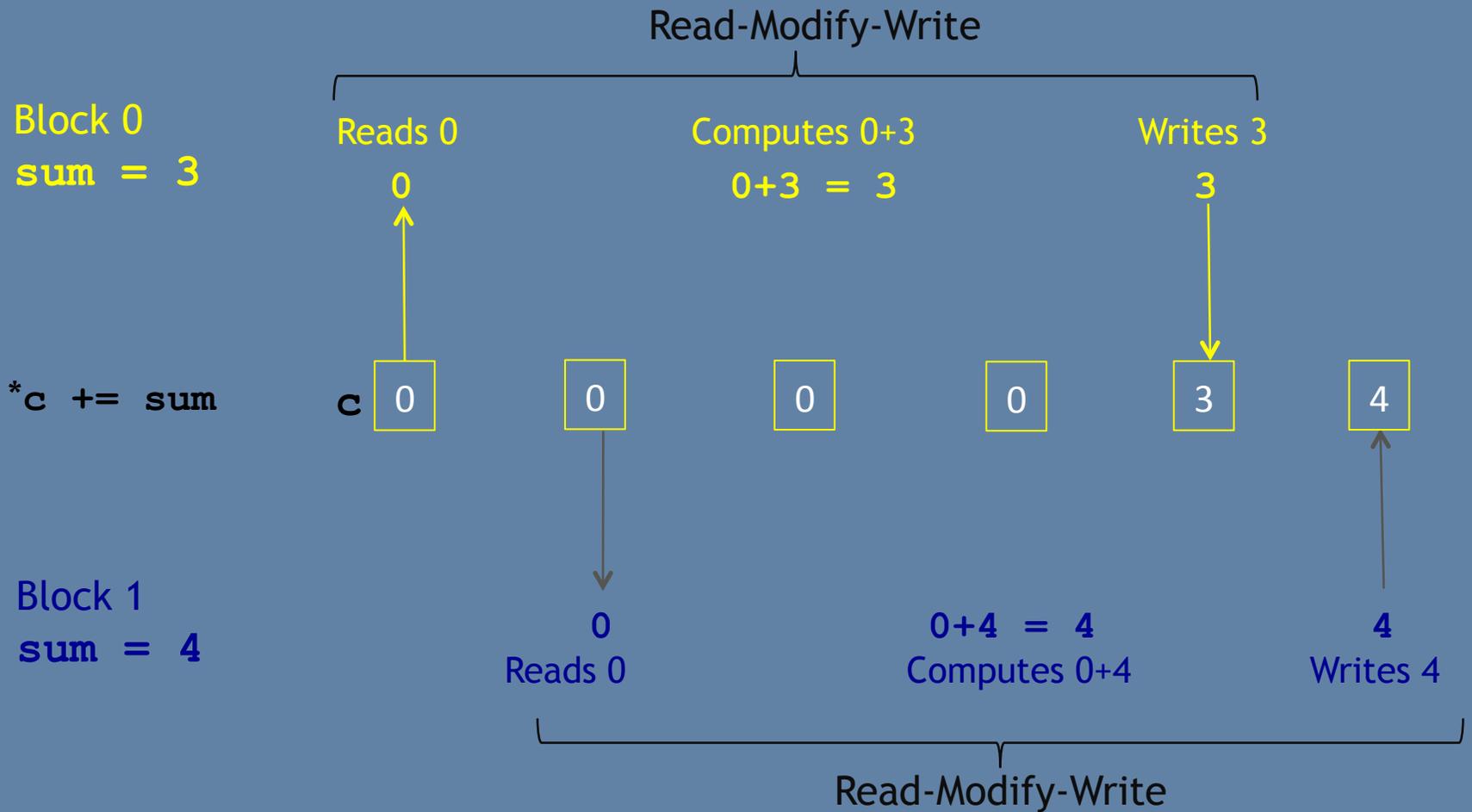
- Terminology: A *race condition* occurs when program behavior depends upon relative timing of two (or more) event sequences
- What actually takes place to execute the line in question: `*c += sum;`
  - Read **value** at address `c`
  - Add `sum` to **value**
  - Write result to address `c`

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()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;
```

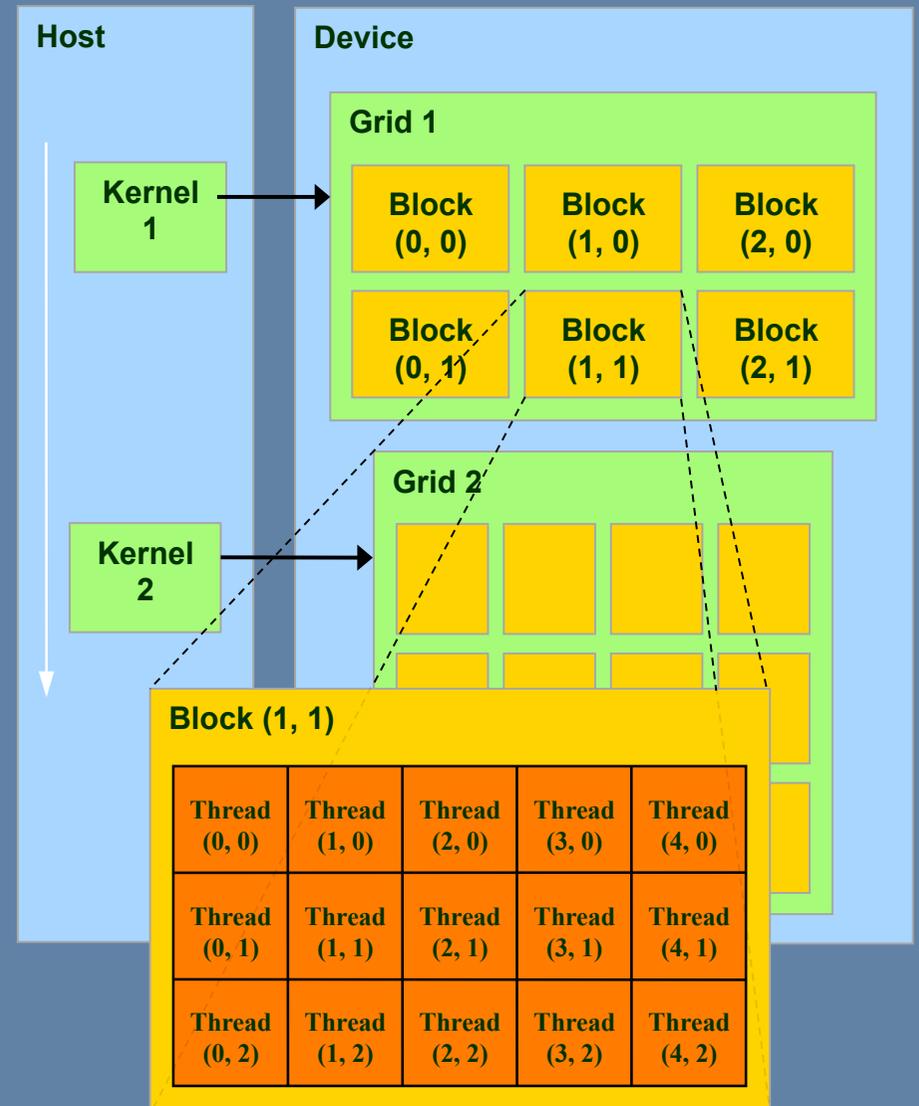
```
}
```

# 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, 2D or 3D grid of thread blocks.
  - All threads share the *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 of 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: `struct dim3 { unsigned int x, y, z };` three unsigned ints where any unspecified component defaults to 1.

- `dim3 gridDim;`
  - Dimensions of the grid in blocks
- `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!)

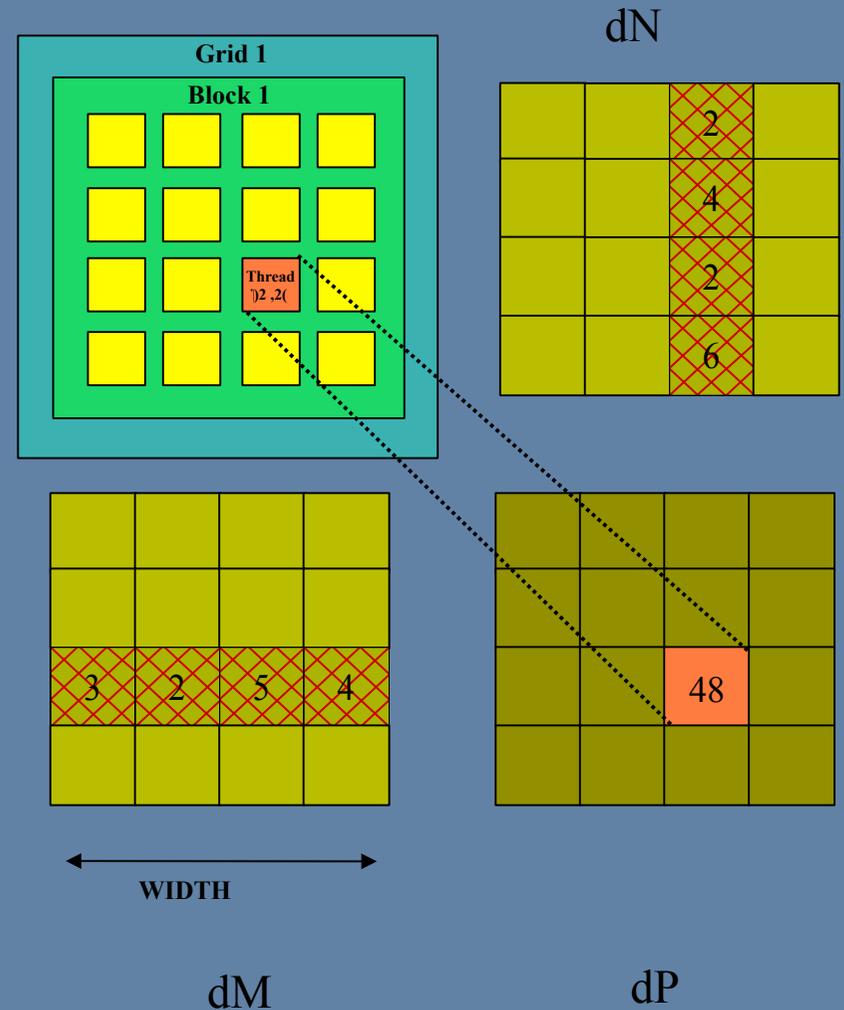
```
__global__ void kernel( int *a, int dimx, int dimy ) {  
    int ix = blockIdx.x*blockDim.x + threadIdx.x;  
    int iy = blockIdx.y*blockDim.y + threadIdx.y;  
    int idx = iy*dimx + ix;  
  
    a[idx] = idx+1;  
}
```

Exercise: compile and run: [setmatrix.cu](#)

```
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;  
}
```

# Matrix multiply with one thread block

- One block of threads computes 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[                ];  
        DATA Nelement = dN[                ];  
        Pvalue += Melement * Nelement;  
    }  
    dP[                ] = Pvalue;  
}
```

WIDTH

threadIdx.x=0 threadIdx.y=0	threadIdx.x=1 threadIdx.y=0	threadIdx.x=2 threadIdx.y=0	threadIdx.x=3 threadIdx.y=0
threadIdx.x=0 threadIdx.y=1	threadIdx.x=1 threadIdx.y=1	threadIdx.x=2 threadIdx.y=1	threadIdx.x=3 threadIdx.y=1
threadIdx.x=0 threadIdx.y=2	threadIdx.x=1 threadIdx.y=2	threadIdx.x=2 threadIdx.y=2	threadIdx.x=3 threadIdx.y=2
threadIdx.x=0 threadIdx.y=3	threadIdx.x=1 threadIdx.y=3	threadIdx.x=2 threadIdx.y=3	threadIdx.x=3 threadIdx.y=3

✓ 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 N**

# CUDA Compiler: nvcc basic options

- ✓ `-arch=sm_13` (or `sm_20`, `sm_35`) 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
- ✓ `-ccbin compiler_path` use a different C compiler (*e.g.*, Intel instead of gcc)

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() ) );
```

# CUDA Event API

- Events are inserted (recorded) into CUDA call streams
- Usage scenarios:
  - measure elapsed time (in milliseconds) for CUDA calls (resolution ~0.5 microseconds)
  - query the status of an asynchronous CUDA call
  - block CPU until CUDA calls prior to the event are completed

```
cudaEvent_t start, stop;  
float et;  
cudaEventCreate(&start);  
cudaEventCreate(&stop);  
cudaEventRecord(start, 0);  
kernel<<<grid, block>>>(…);  
cudaEventRecord(stop, 0);  
cudaEventSynchronize(stop);  
cudaEventElapsedTime(&et, start, stop);  
cudaEventDestroy(start);  
cudaEventDestroy(stop);  
printf(“Kernel execution time=%f\n”,et);
```

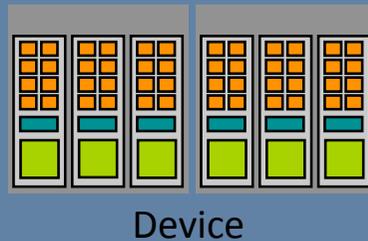
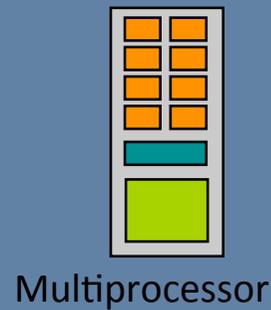
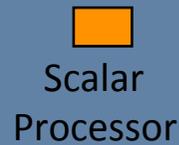
Read the function MatrixMulOnDevice of the MMGlobWT.cu program. Compile and run

# Execution Model

## Software



## Hardware



Threads are executed by scalar processors

Thread blocks are executed on multiprocessors

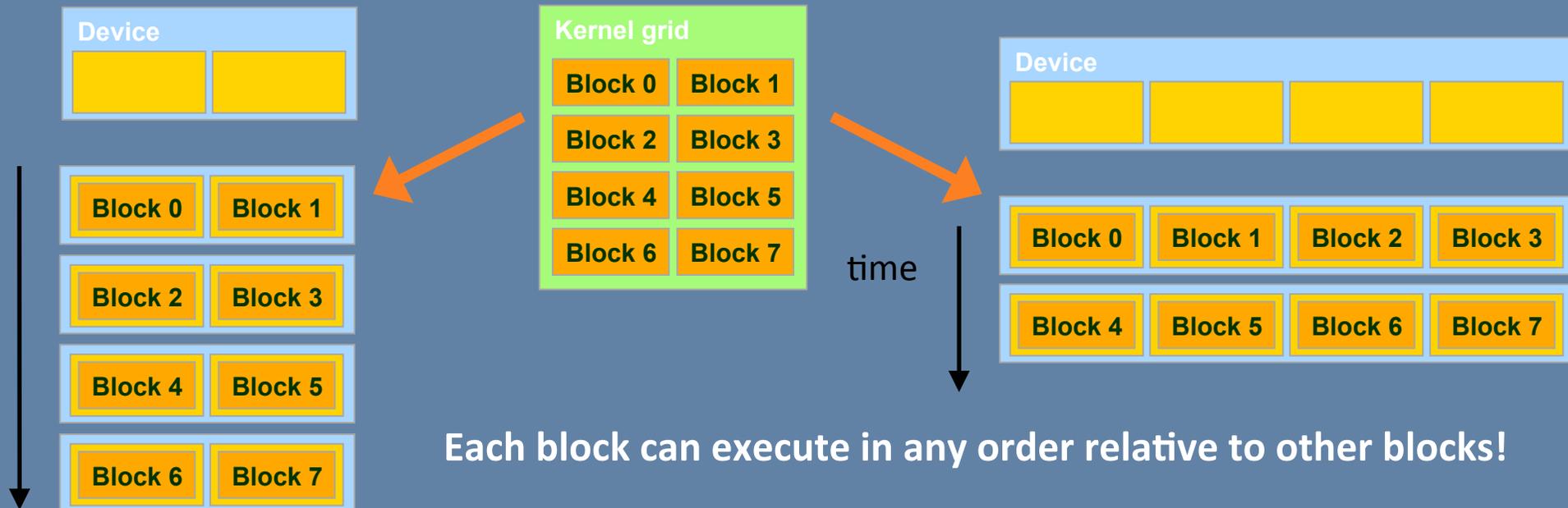
Thread blocks **do not migrate**

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

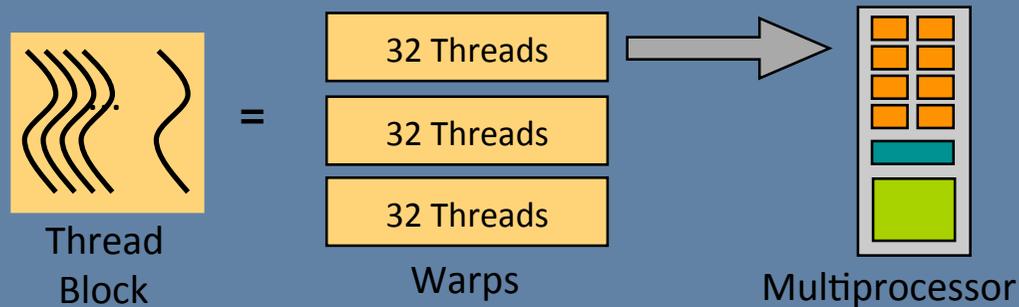
A kernel is launched as a grid of thread blocks

# Transparent Scalability

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

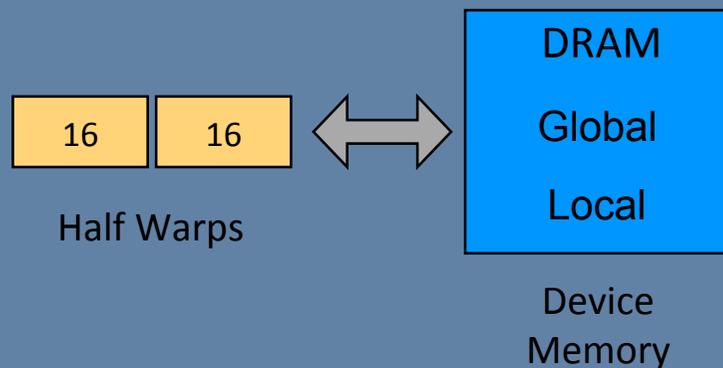


# Warps and Half Warps



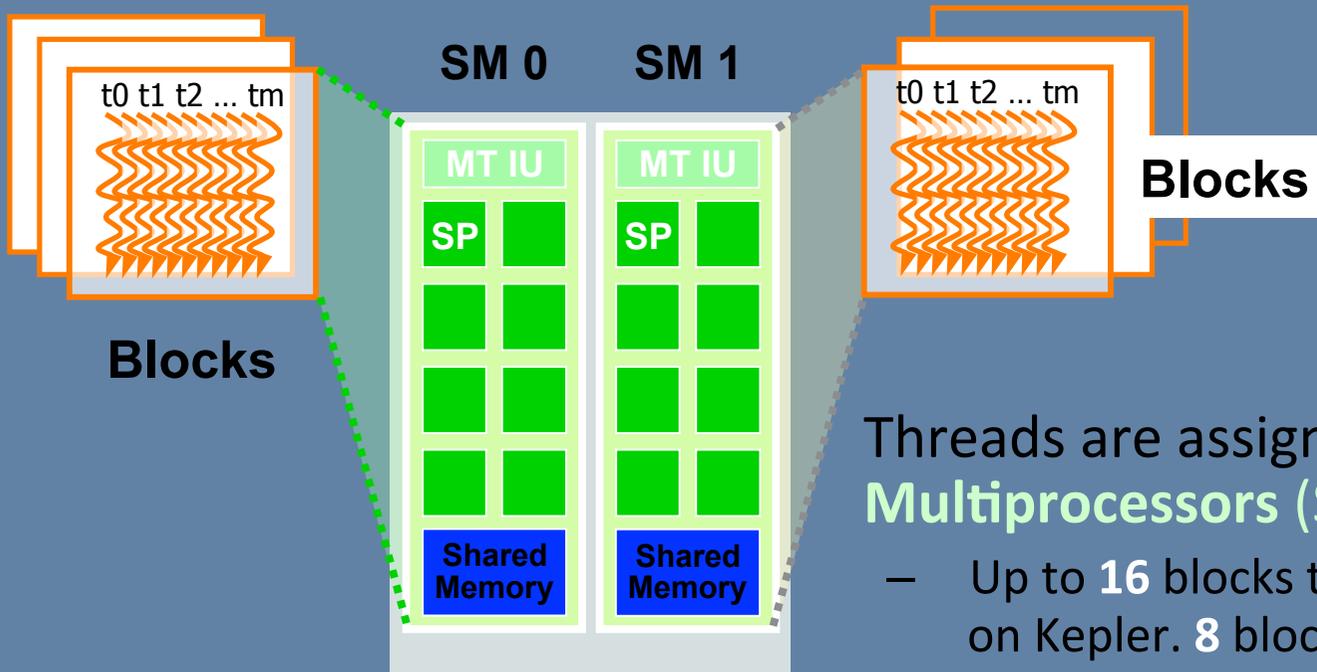
A thread block consists of 32-thread warps

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

# Executing Thread Blocks



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

Threads are assigned to **Streaming Multiprocessors (SM)** in block granularity

- Up to **16** blocks to each SM on Kepler. **8** blocks on Fermi SM.
- A Fermi SM can take up to 1536 threads
  - Could be 256 (threads/block) \* 6 blocks or 192 (threads/block) \* 8 blocks but, for instance, 128 (threads/block) \* 12 blocks not allowed!
- A Kepler SM can take up to **2048** threads
- Threads run concurrently
  - SM manages/schedules thread execution

# Block Granularity Considerations

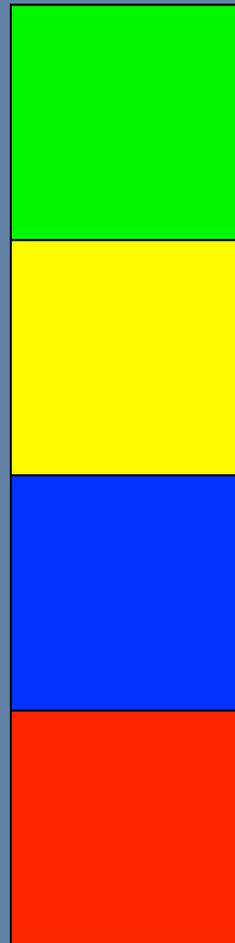
- For a kernel running on a Fermi and using multiple blocks, should we use 8X8, 16X16 or 32X32 blocks?
  - For 8X8, we have 64 threads per Block. Since each Fermi SM can take up to 1536 threads, there are 24 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!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

Repeat the analysis for a Kepler card!

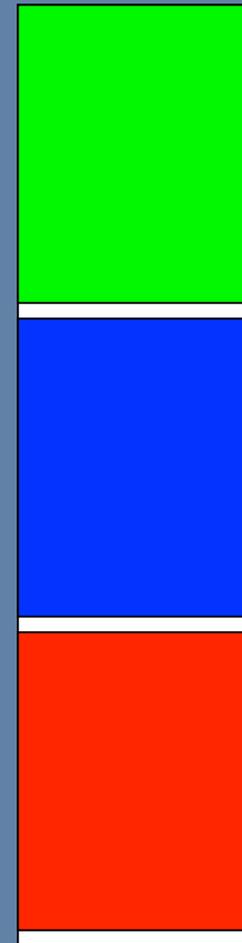
# Programmer View of Register File

- There are up to **32768** (32 bit) registers in each Fermi SM
- There are up to **65536** (32 bit) registers in each Kepler 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

4 blocks



3 blocks



# Registers “occupancy” example

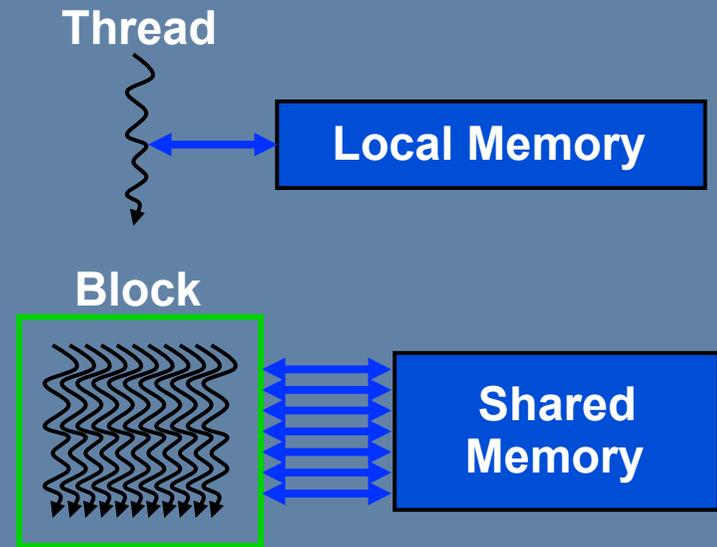
- If a Block has 16x16 threads and each thread uses 30 registers, how many blocks can run on each Fermi SM?
  - Each block requires  $30 * 256 = 7680$  registers
  - $32768 / 7680 = 4 + \text{change } (2048)$
  - So, 4 blocks can run on an SM as far as registers are concerned
- How about if each thread increases the use of registers by 10%?
  - Each Block now requires  $33 * 256 = 8448$  registers
  - $32768 / 8448 = 3 + \text{change } (7424)$
  - Only three Blocks can run on an SM, **25% reduction of parallelism!!!**

Repeat the analysis for a Kepler card!

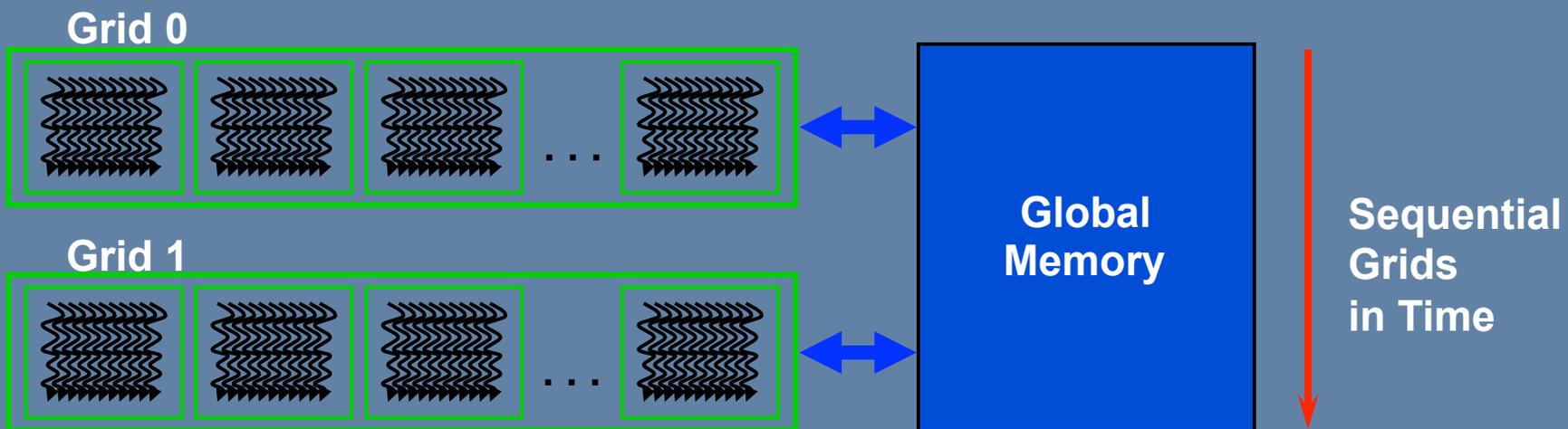
# Optimizing threads per block

- ✓ Choose threads per block as a multiple of warp size (32)
  - ✓ Avoid wasting computation on under-populated warps
  - ✓ Facilitates efficient memory access (*coalescing*)
- ✓ Run as many warps as possible per multiprocessor (hide latency)
  - ✓ SM can run up to 16 (on Kepler) blocks at a time
- ✓ Heuristics
  - ✓ Minimum: 64 threads per block
  - ✓ 192 or 256 threads a better choice
    - ✓ Usually still enough registers to compile and invoke successfully
  - ✓ The right tradeoff depends on your computation, so **experiment, experiment, experiment!!!**

# Review on Memory Hierarchy



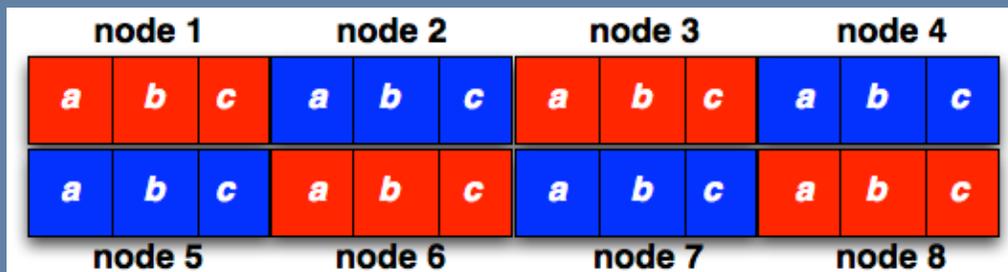
- Local Memory: per-thread
  - Private per thread but *slow!*
  - Auto variables, register spill
- Shared Memory: per-block
  - Shared by threads of the same block
  - *Fast* inter-thread communication
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication



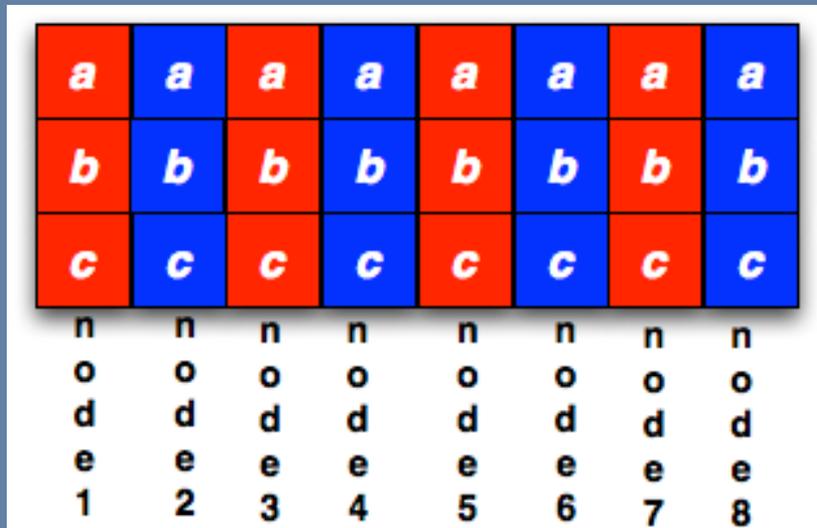
# Optimizing Global Memory Access

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

Array of structures (data locality!)



Structure of arrays (thread locality!)



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

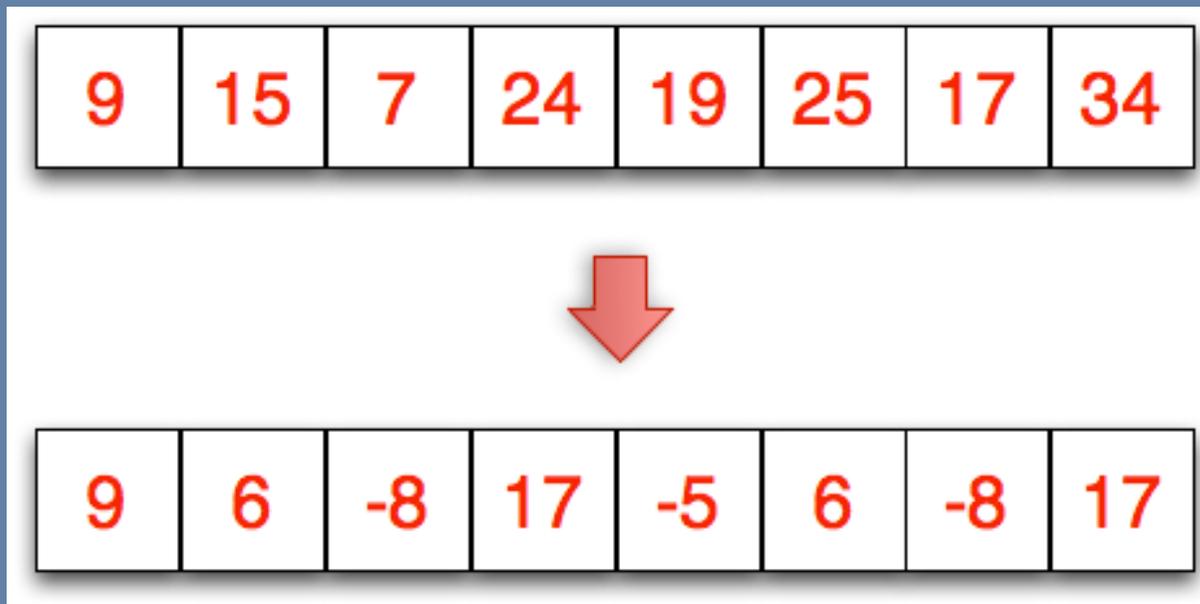
# Shared Memory

- On-chip memory
  - 2 orders of magnitude lower latency than global memory
  - Much higher bandwidth than global memory
  - Up to **48 KByte** per multiprocessor (on Fermi)
- Allocated per thread-block
- Accessible by any thread in the thread-block
  - Not accessible to other thread-blocks
- Several uses:
  - Sharing data among threads in a thread-block
  - User-managed cache (reducing global memory accesses)

```
__shared__ float As[16][16];
```

# Shared Memory Example

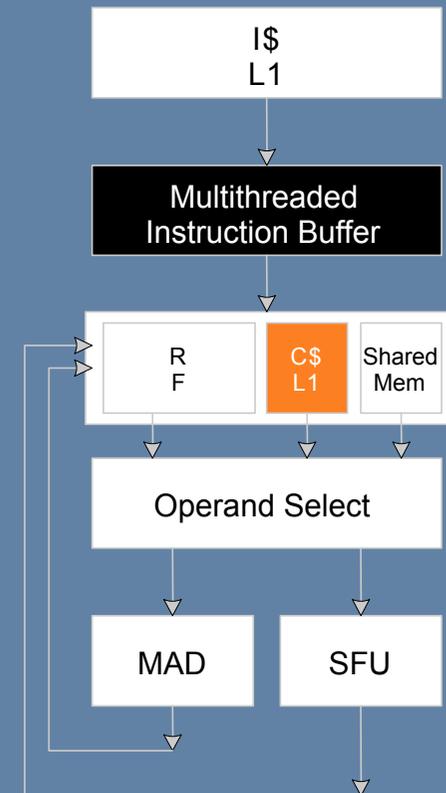
- Compute adjacent\_difference
  - Given an array d\_in produce a d\_out array containing the difference:  
d\_out[0]=d\_in[0];  
d\_out[i]=d\_in[i]-d\_in[i-1]; /\* i>0 \*/



**Exercise:** Complete the shared\_variables.cu code

# 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(ds, &hs, 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.
- Avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - `if (threadIdx.x > 7) { }`
    - This creates two different control paths for threads in a block
    - Branch granularity < warp size; threads 0, 1, ..., 7 follow different path than the rest of the threads 8, 9, ..., 31 in the same 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

# 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 is usually much better than many small ones

# Synchronizing GPU and CPU

- All kernel launches are asynchronous!
  - control returns to CPU immediately
    - Be careful measuring times!
  - 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**

# Page-Locked Data Transfers

- ✓ **cudaMallocHost()** allows allocation of page-locked (“pinned”) host memory
  - ✓ Same syntax of **cudaMalloc()** but works with CPU pointers and memory
- ✓ Enables highest **cudaMemcpy** performance
  - ✓ 3.2 GB/s on PCI-e x16 Gen1
  - ✓ 6.5 GB/s on PCI-e x16 Gen2
- ✓ Use with caution!!!
  - ✓ Allocating too much page-locked memory can reduce overall system performance

See and test the *[bandwidth.cu](#)* sample

# Overlapping Data Transfers and Computation

- ✓ Async and Stream API allow overlap of H2D or D2H data transfers with computation
  - ✓ CPU computation can overlap data transfers on all CUDA capable devices
  - ✓ Kernel computation can overlap data transfers on devices with “Concurrent copy and execution” (compute capability  $\geq 1.1$ )
- ✓ Stream = sequence of operations that execute in order on GPU
  - ✓ Operations from different streams can be interleaved
  - ✓ Stream ID used as argument to async calls and kernel launches

# Asynchronous Data Transfers

- ✓ Asynchronous host-device memory copy returns control immediately to CPU
  - ✓ `cudaMemcpyAsync` (`dst`, `src`, `size`, `dir`, `stream`);
  - ✓ requires *pinned* host memory (allocated with `cudaMallocHost`)
- ✓ Overlap CPU computation with data transfer
  - ✓ `0` = default stream

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);  
kernel<<<grid, block>>>(a_d);  
cpuFunction();
```

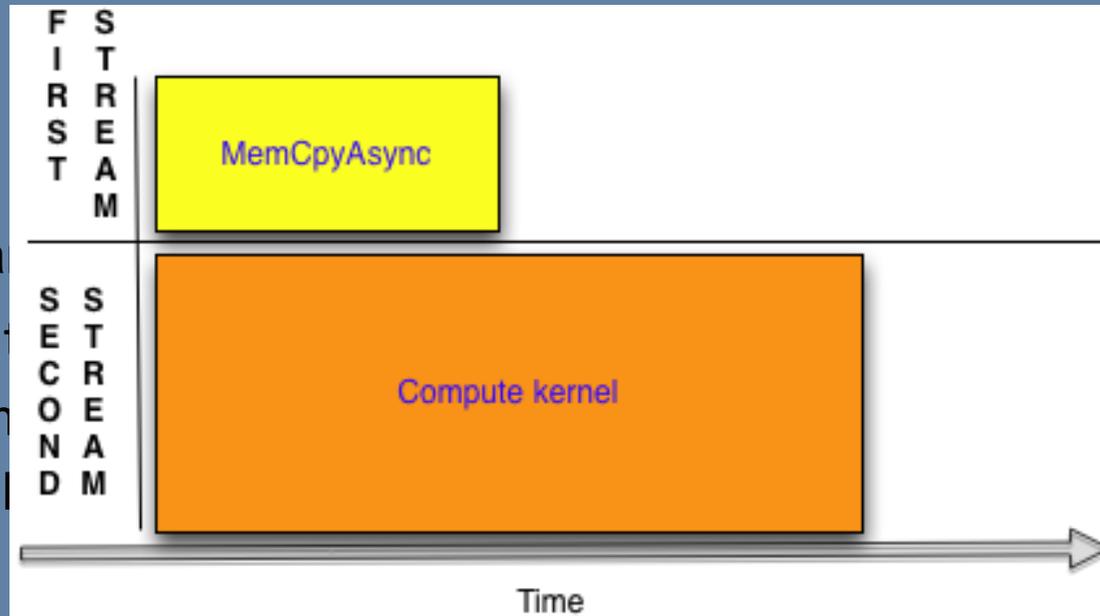
overlapped



# Overlapping kernel and data transfer

## ✓ Requires:

- ✓ Kernel and data transfer must be on different streams
- ✓ For each stream, the kernel must be launched before the data transfer
- ✓ Remember to call cudaDeviceSynchronize() at the end of the program



## ✓ Example:

```

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

create  
all previous calls

overlapped

Exercise: read, compile and run **simpleStreams.cu**

# 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
    - the driver regulates the access.

# 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

# CUDA libraries

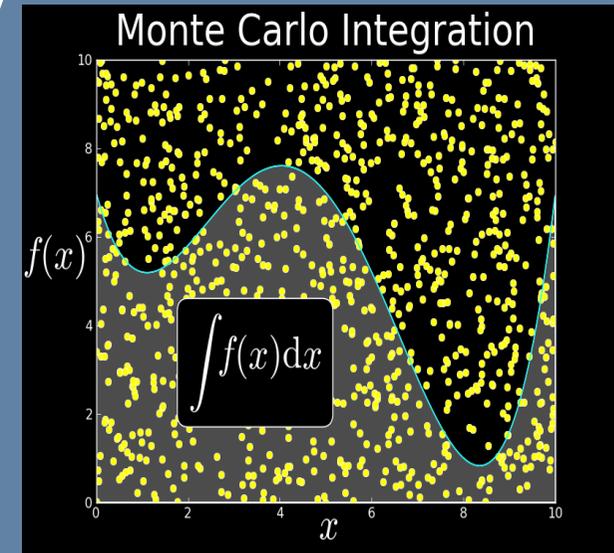
- ✓ CUDA includes a number of widely used libraries
  - CUBLAS: BLAS implementation
  - CUFFT: FFT implementation
  - CURAND: Random number generation
  - *Etc.*
- ✓ Thrust
  - <http://docs.nvidia.com/cuda/thrust>
  - Reduction
  - Scan
  - Sort
  - *Etc.*

# CUDA libraries

- ✓ 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 `func()` to compile to `__func()`

# CURAND Library

- Library for generating random numbers
- Features:
  - XORWOW pseudo-random generator
  - Sobol' quasi-random number generators
  - Host API for generating random numbers in bulk
  - Inline implementation allows use inside GPU functions/kernels
  - Single- and double-precision, uniform, normal and log-normal distributions



# CURAND Features

- Pseudo-random numbers  
George Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14), 2003. Available at <http://www.jstatsoft.org/v08/i14/paper>.
- Quasi-random 32-bit and 64-bit Sobol' sequences with up to 20,000 dimensions.
- Host API: call kernel from host, generate numbers on GPU, consume numbers on host or on GPU.
- GPU API: generate and consume numbers during kernel execution.

# CURAND use

1. Create a generator:

**curandCreateGenerator()**

2. Set a seed:

**curandSetPseudoRandomGeneratorSeed()**

3. Generate the data from a distribution:

**curandGenerateUniform()/curandGenerateUniformDouble():** Uniform

**curandGenerateNormal()/cuRandGenerateNormalDouble():** Gaussian

**curandGenerateLogNormal/curandGenerateLogNormalDouble():** Log-Normal

4. Destroy the generator:

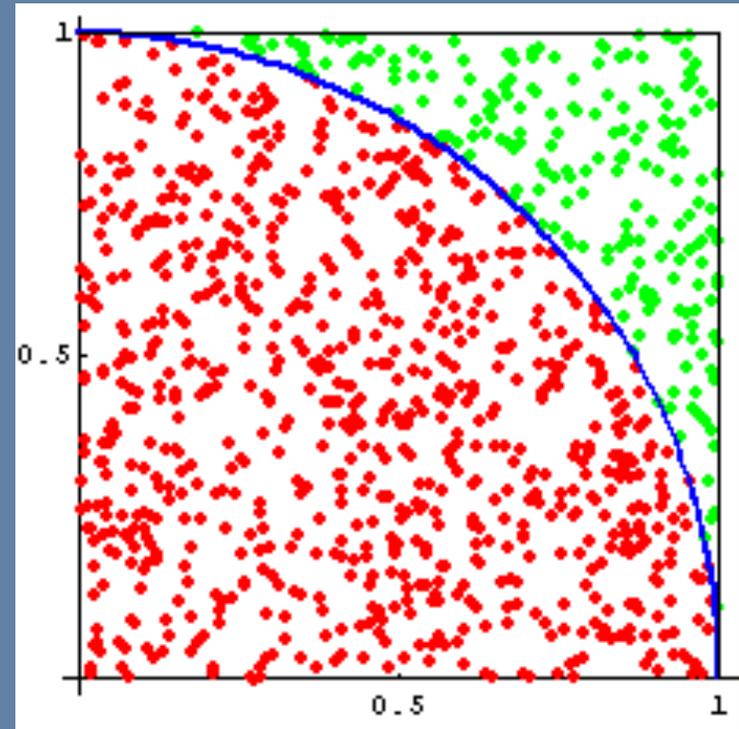
**curandDestroyGenerator()**

# Example CURAND Program: Computing $\pi$ with MonteCarlo method

$$\pi = 4 * (\sum \text{red points}) / (\sum \text{points})$$

1. Generate  $0 \leq x < 1$
2. Generate  $0 \leq y < 1$
3. Compute  $z = x^2 + y^2$
4. If  $z \leq 1$  the point is **red**

Accuracy increases slowly...



# Example CURAND Program: Computing $\pi$ with MonteCarlo method

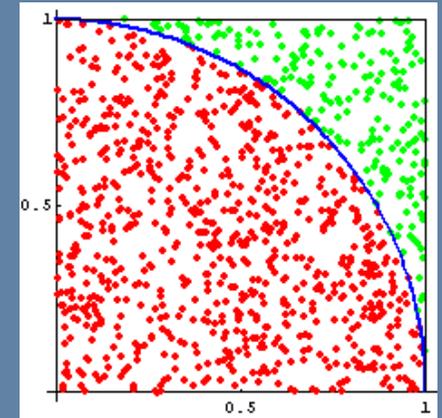
Compare the speed of *pigpu* and *picpu*  
**cd Pi**

**gcc -o picpu -O3 picpu.c -lm**

**nvcc -o pigpu -O3 pigpu.cu -arch=sm\_35 -lm**

Run for 100000 iterations

Look at *pigpu* and try to understand why is it  
so SLOW...?



# 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 if the GPU are connected to the same PCIe switch

- ✓ **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

- ✓ GPUs 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
  - ✓ Remember 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

# Multi-GPU Environment: a simple approach

```
// Run independent kernel on each CUDA device
```

```
int numDevs = 0;
```

```
cudaGetNumDevices(&numDevs); ...
```

```
for (int d = 0; d < numDevs; d++) {
```

```
    cudaSetDevice(d);
```

```
    kernel<<<blocks, threads>>>(args);
```

```
}
```

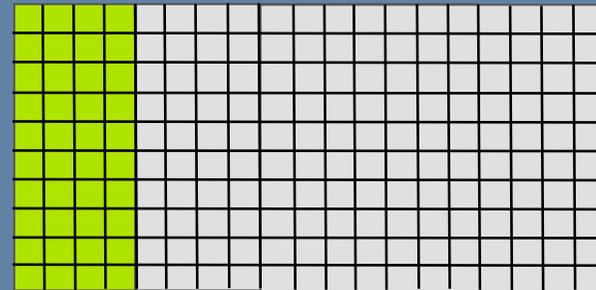
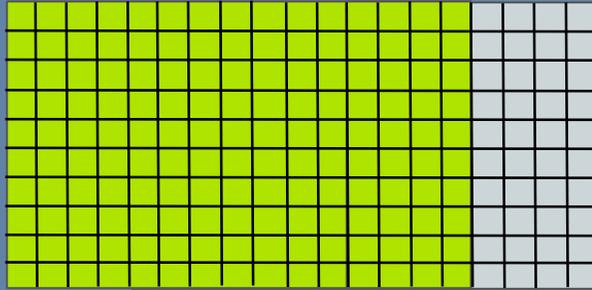
Exercise: Compare the `dot_simple_multiblock.cu` and `dot_multigpu.cu` programs. Compile and run.

Attention! Add the `-arch=sm_35` option: `nvcc -o dot_multigpu dot_multigpu.cu -arch=sm_35`

# General Multi-GPU Programming Pattern

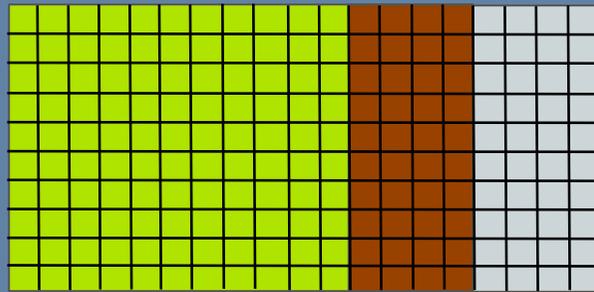
- **The goal is to hide communication cost**
  - Overlap with computation
- **So, every time-step, each GPU should:**
  - Compute parts (*halos*) to be sent to neighbors
  - Compute the internal region (*bulk*)
  - Exchange halos with neighbors
- **Linear scaling as long as internal-computation takes longer than halo exchange**
  - Actually, separate halo computation adds some overhead

# Example: Two Subdomains

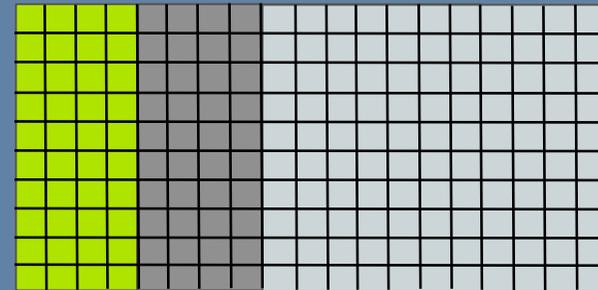


# Example: Two Subdomains

Phase 1

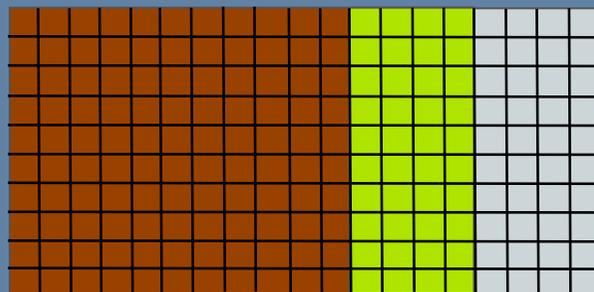


compute



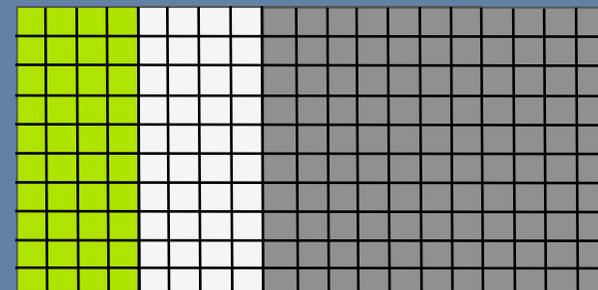
compute

Phase 2



compute

send



send

compute

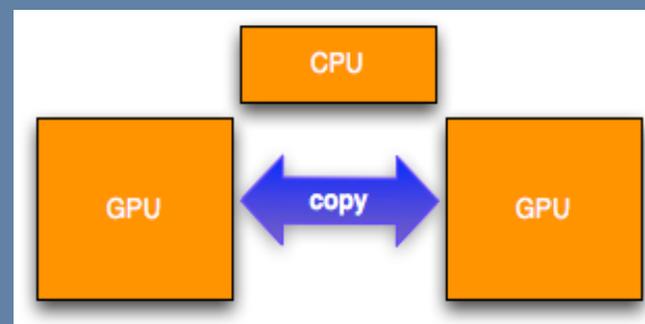
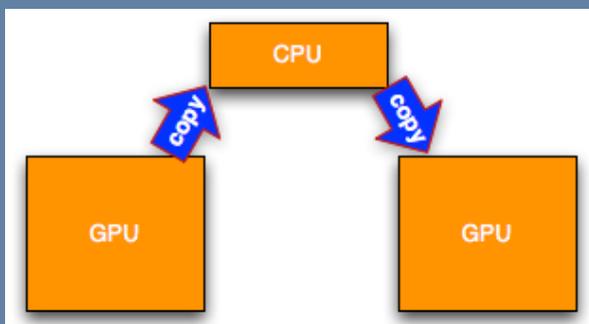
GPU-0: green subdomain  
GPU-1: grey subdomain

# CUDA Features Useful for Multi-GPU

- **Control multiple GPUs with a single CPU thread**
  - Simpler coding: no need for CPU multithreading
- **Streams:**
  - Enable executing kernels and memcopies concurrently
  - Up to 2 concurrent memcopies: to/from GPU
- **Peer-to-Peer (P2P) GPU memory copies**
  - Transfer data between GPUs using PCIe P2P support
  - Done by GPU DMA hardware – host CPU is not involved
    - Data traverses PCIe links, without touching CPU memory
  - Disjoint GPU-pairs can communicate simultaneously

# Peer-to-Peer GPU Communication

- Up to CUDA 4.0, GPU could not exchange data directly.
- With CUDA  $\geq 4.0$  two GPU connected to the same PCI/express switch can exchange data directly.



```
cudaStreamCreate (&stream_on_gpu_0);
cudaSetDevice(0);
cudaDeviceEnablePeerAccess( 1, 0 );
cudaMalloc(d_0, num_bytes); /* create array on GPU 0 */
cudaSetDevice(1);
cudaMalloc(d_1, num_bytes); /* create array on GPU 1 */
cudaMemcpyPeerAsync( d_0, 0, d_1, 1, num_bytes, stream_on_gpu_0 );
/* copy d_1 from GPU 1 to d_0 on GPU 0: pull copy */
```

**A SINGLE TASK!**

Example: read `p2pcopy.cu`, compile `nvcc -o p2pcopy p2pcopy.cu -arch=sm_35`

# What does the code look like?

```
for( int i=0; i<num_gpus-1; i++ )// “right” stage
    cudaMemcpyPeerAsync( d_a[i+1], gpu[i+1], d_a[i], gpu[i], num_bytes, stream[i] );
for( int i=0; i<num_gpus; i++ )
    cudaStreamSynchronize( stream[i] );
for( int i=1; i<num_gpus; i++ ) // “left” stage
    cudaMemcpyPeerAsync( d_b[i-1], gpu[i-1], d_b[i], gpu[i], num_bytes, stream[i] );
```

- Arguments to the **cudaMemcpyPeerAsync()** call:
  - Dest address, dest GPU, src address, src GPU, num bytes, stream ID
- The **middle loop** isn't necessary for correctness
  - Improves performance by preventing the two stages from interfering with each other (15 vs. 11 GB/s using 4-GPU example)

# Code Pattern for Multi-GPU

- **Every time-step each GPU will:**
  - Compute halos to be sent to neighbors (**stream *h***)
  - Compute the internal region (**stream *i***)
  - Exchange halos with neighbors (**stream *h***)

# Code For Looping through Time

```
for( int istep=0; istep<nsteps; istep++)
{
    for( int i=0; i<num_gpus; i++ )
    {
        cudaSetDevice( gpu[i] );
        kernel<<<..., stream_halo[i]>>>( ... );
        kernel<<<..., stream_halo[i]>>>( ... );
        cudaStreamQuery( stream_halo[i] );
        kernel<<<..., stream_internal[i]>>>( ... );
    }

    for( int i=0; i<num_gpus-1; i++ )
        cudaMemcpyPeerAsync( ..., stream_halo[i] );
    for( int i=0; i<num_gpus; i++ )
        cudaStreamSynchronize( stream_halo[i] );
    for( int i=1; i<num_gpus; i++ )
        cudaMemcpyPeerAsync( ..., stream_halo[i] );

    for( int i=0; i<num_gpus; i++ )
    {
        cudaSetDevice( gpu[i] );
        cudaDeviceSynchronize();
        // swap input/output pointers
    }
}
```

Compute halos

Compute internal subdomain

Exchange halos

Synchronize before next step

# Code For Looping through Time

```
for( int istep=0; istep<nsteps; istep++)
{
    for( int i=0; i<num_gpus; i++ )
    {
        cudaSetDevice( gpu[i] );
        kernel<<<..., stream_halo[i]>>>( ... );
        kernel<<<..., stream_halo[i]>>>( ... );
        cudaStreamQuery( stream_halo[i] );
        kernel<<<..., stream_internal[i]>>>( ... );
    }

    for( int i=0; i<num_gpus-1; i++ )
        cudaMemcpyPeerAsync( ..., stream_halo[i] );
    for( int i=0; i<num_gpus; i++ )
        cudaStreamSynchronize( stream_halo[i] );
    for( int i=1; i<num_gpus; i++ )
        cudaMemcpyPeerAsync( ..., stream_halo[i] );

    for( int i=0; i<num_gpus; i++ )
    {
        cudaSetDevice( gpu[i] );
        cudaDeviceSynchronize();
        // swap input/output pointers
    }
}
```

- **Two CUDA streams per GPU**
  - halo and internal
  - Each in an array indexed by GPU
  - Calls issued to different streams can overlap
- **Prior to time-loop**
  - Create streams
  - Enable P2P access

# GPUs in Different Cluster Nodes

- **Add network communication to halo-exchange**
  - Transfer data from “edge” GPUs of each node to CPU
  - Hosts exchange via MPI (or other API)
  - Transfer data from CPU to “edge” GPUs of each node
- **With CUDA  $\geq$  4.0, transparent interoperability between CUDA and Infiniband**

**export** CUDA\_NIC\_INTEROP=1

Let MPI use, for Remote Direct Memory Access operations, memory pinned by GPU

# Inter-GPU Communication with MPI

- **Example: simpleMPI.c**
    - Generate some random numbers on one node.
    - Dispatch them to all nodes.
    - Compute their square root on each node's GPU.
    - Compute the average of the results by using MPI.
- Replace simpleMPI.c with the version in /tmp/simpleMPI.c

To compile:

```
$nvcc -c simpleCUDAMPI.cu
```

```
$mpic++ -o simpleMPI simpleMPI.c simpleCUDAMPI.o  
-L/usr/local/cuda/lib64/ -lcudart
```

To run:

```
$mpirun -np 2 simpleMPI
```

On a single line!

# GPU-aware MPI

- Support GPU to GPU communication through standard MPI interfaces:
  - e.g. enable MPI\_Send, MPI\_Recv from/to GPU memory
  - Made possible by Unified Virtual Addressing (UVA) in CUDA 4.0
- Provide high performance without exposing low level details to the programmer:
  - Pipelined data transfer which *automatically* provides optimizations inside MPI library without user tuning

From: MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters

H Wang, S. Potluri, M. Luo, A.K. Singh, S. Sur, D. K. Panda

# Naive code with “old style” MPI

- **Sender:**

```
cudaMemcpy(s_buf, s_device, size, cudaMemcpyDeviceToHost);
```

```
MPI_Send(s_buf, size, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
```

- **Receiver:**

```
MPI_Recv(r_buf, size, MPI_CHAR, 0, 1, MPI_COMM_WORLD, &req);
```

```
cudaMemcpy(r_device, r_buf, size, cudaMemcpyHostToDevice);
```

# Optimized code with “old style” MPI

- Pipelining at user level with non-blocking MPI and CUDA functions
- **Sender:**

```
for (j = 0; j < pipeline_len; j++)
    cudaMemcpyAsync(s_buf+j*block_sz,s_device+j*block_sz,...);
for (j = 0; j < pipeline_len; j++) {
    while (result != cudaSuccess) {
        results = cudaStreamQuery(..);
        if (j>0) MPI_Test(...);
    }
    MPI_Isend(s_buf+j*block_sz,block_sz,MPI_CHAR,1,1,...)
}
MPI_Waitall();
```

# Code with GPU-aware MPI

- Standard MPI interfaces from device buffers

## Sender:

```
MPI_Send(s_dev, size, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
```

## Receiver:

```
MPI_Recv(r_dev, size, MPI_CHAR, 0, 1, MPI_COMM_WORLD, &req);
```

- MPI library can automatically differentiate between device memory and host memory
- Overlap CUDA copy and RDMA transfer
- Pipeline DMA of data from GPU and InfiniBand RDMA

# Major GPU Performance Detractors

- ✓ Sometimes it is better to recompute than to cache
  - ✓ GPU spends its transistors on ALUs, not on cache!
- ✓ Do more computation on the GPU to avoid costly data transfers
  - ✓ Even low parallelism computations can at times be faster than transferring data 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
  - Max number of blocks per SM: 16 (Kepler)
  - Max number of threads per SM: 2048 (Kepler)
  - Max number of threads per block: 1024 (Fermi & Kepler)
  - Total number of registers per SM: 65536 (Kepler)
  - Size of shared memory per SM: 48 Kbyte (Fermi & Kepler)

# 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
      - export CUDA\_PROFILE=1 (bash) or setenv CUDA\_PROFILE 1 (tcsh)
      - export CUDA\_PROFILE\_LOG=XXXXXX or 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.

# What is missing?

- Texture
- Voting functions (`__all()`, `__any()`)
- ECC effect
- Shared memory bank conflicts
- Single virtual address space and other variants of memory copy operations
- Many GPU libraries

# Thanks!