

MORE PROGRAMMING IN CUDA C: SHARED MEMORY AND THREAD COOPERATION

Will Landau, Prof. Jarad Niemi

REVIEW

BASIC C PROGRAM

```
#include <iostream>

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

BASIC CUDA C PROGRAM

```
#include <iostream>

__global__ void kernel ( void ) {
}

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

`__global__`: Call from CPU and run only on GPU.

`__device__`: Call from GPU and run only on GPU.

(More specifically, call only from within
a `__global__` or another `__device__` function.)

`__host__`: Call from CPU and run only on CPU.

(i.e., a traditional C function.)

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

```
__device__ int dev1( void ){
}

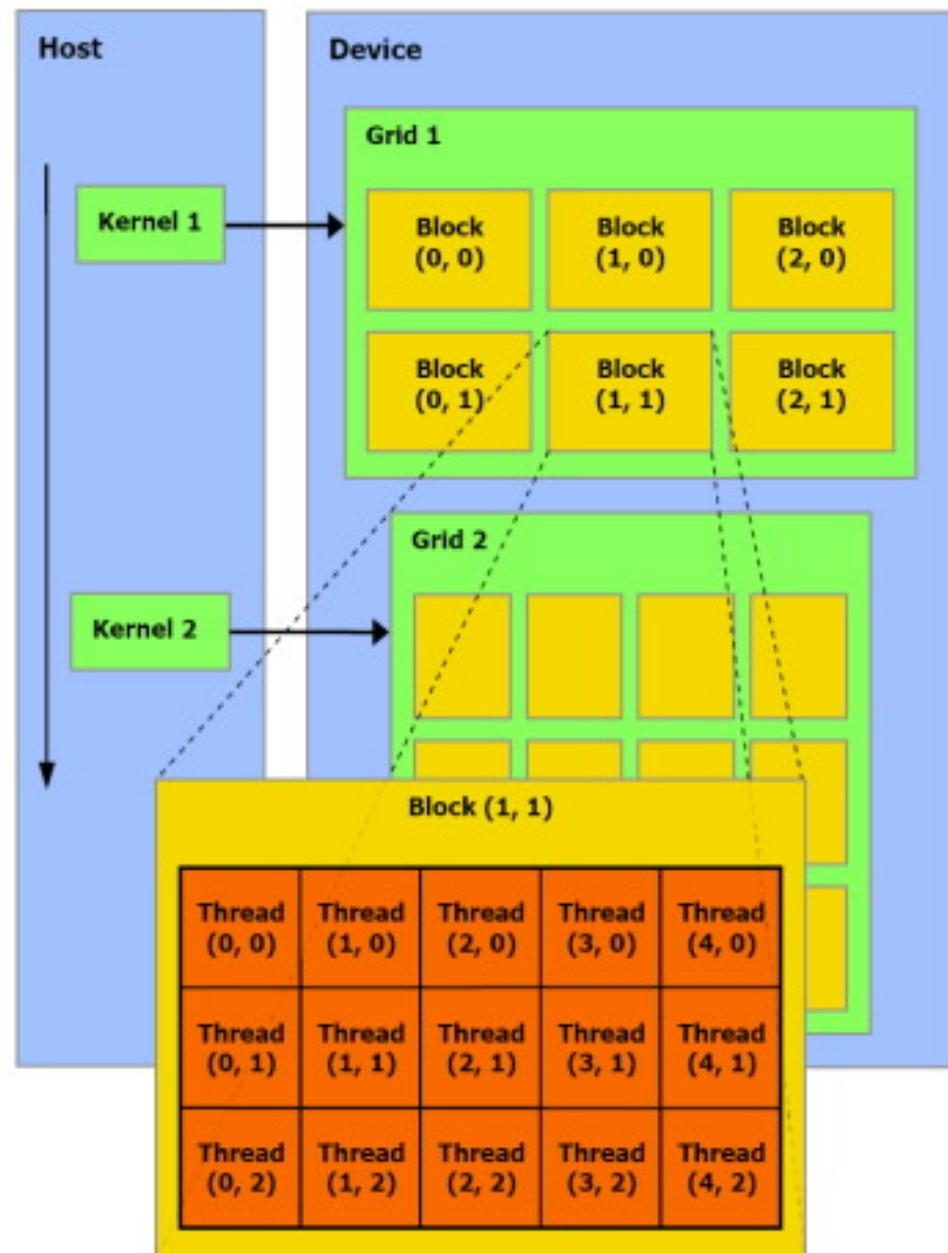
__device__ int dev2( void ){
}

__global__ void kernel ( void ) {
    dev1();
    dev2();
}

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

PARALLELIZING A WORKLOAD ON A GPU

1. The CPU sends a CPU-to-GPU command called a **kernel** to a single GPU core.
2. The GPU core multitasks to execute the command:
 - a. The GPU makes $N \cdot M$ **copies** of the kernel's code, and then runs all those copies simultaneously. Those parallel copies are called **threads**.
 - b. The $N \cdot M$ threads are partitioned into N groups, called **blocks**, of M threads each.
 - c. The sum total of all the threads from a kernel call is a **grid**.



USEFUL VARIABLES WITHIN A KERNEL CALL OF B BLOCKS AND T THREADS PER BLOCK

blockIdx.x: the block ID corresponding to the current thread, an integer from 0 to $B - 1$ inclusive.

threadIdx.x: the thread ID of the current thread within its block, an integer from 0 to $T - 1$ inclusive.

gridDim.x: B , the number of blocks in the grid.

blockDim.x: T , the number of threads per block.

maxThreadsPerBlock: exactly that: 1024 on an impact1 core.

Note: the maximum number of blocks per grid is 65535 on an impact1 core.

END OF REVIEW

EXAMPLE: VECTOR SUMMATION (Sanders, et. al.)

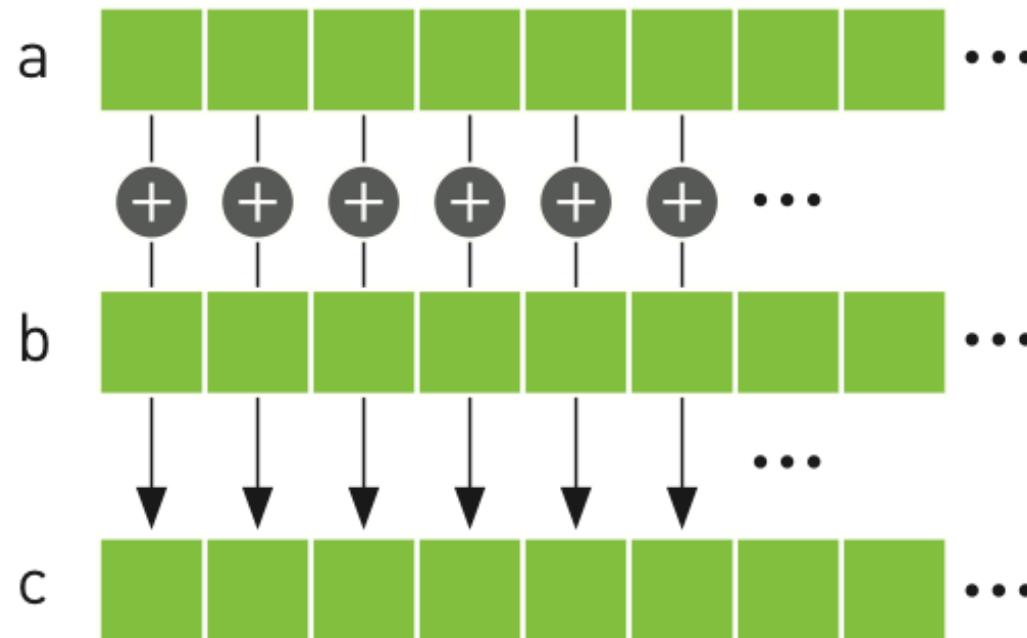


Figure 4.1 Summing two vectors

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;      // handle the data at this index
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

```
#define N    10

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
```

```
// copy the arrays 'a' and 'b' to the GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
                         cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                         cudaMemcpyHostToDevice ) );

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

// copy the array 'c' back from the GPU to the CPU
HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
                         cudaMemcpyDeviceToHost ) );

// display the results
for (int i=0; i<N; i++) {
    printf( "%d + %d = %d\n", a[i], b[i], c[i] );
}
```

```
// free the memory allocated on the GPU
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );

return 0;
}
```

```
[landau@impact1 vectorsums]$ nvcc vectorsums.cu -o vectorsums.out
[landau@impact1 vectorsums]$ ./vectorsums.out
0 + 0 = 0
-1 + 1 = 0
-2 + 4 = 2
-3 + 9 = 6
-4 + 16 = 12
-5 + 25 = 20
-6 + 36 = 30
-7 + 49 = 42
-8 + 64 = 56
-9 + 81 = 72
[landau@impact1 vectorsums]$
```

You can download the code, along with other simple CUDA C examples, at
https://github.com/jarad/gpuIntroduction/tree/master/CUDA_C_sandbox.

VECTOR SUMMATION: MULTIPLE THREADS PER BLOCK

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

USEFUL VARIABLES IN A CALL TO

```
add<<<B, T>>>(dev_a, dev_b, dev_c)
```

blockIdx.x: the block ID corresponding to the current thread, an integer from 0 to $B - 1$ inclusive.

threadIdx.x: the thread ID of the current thread within its block, an integer from 0 to $T - 1$ inclusive.

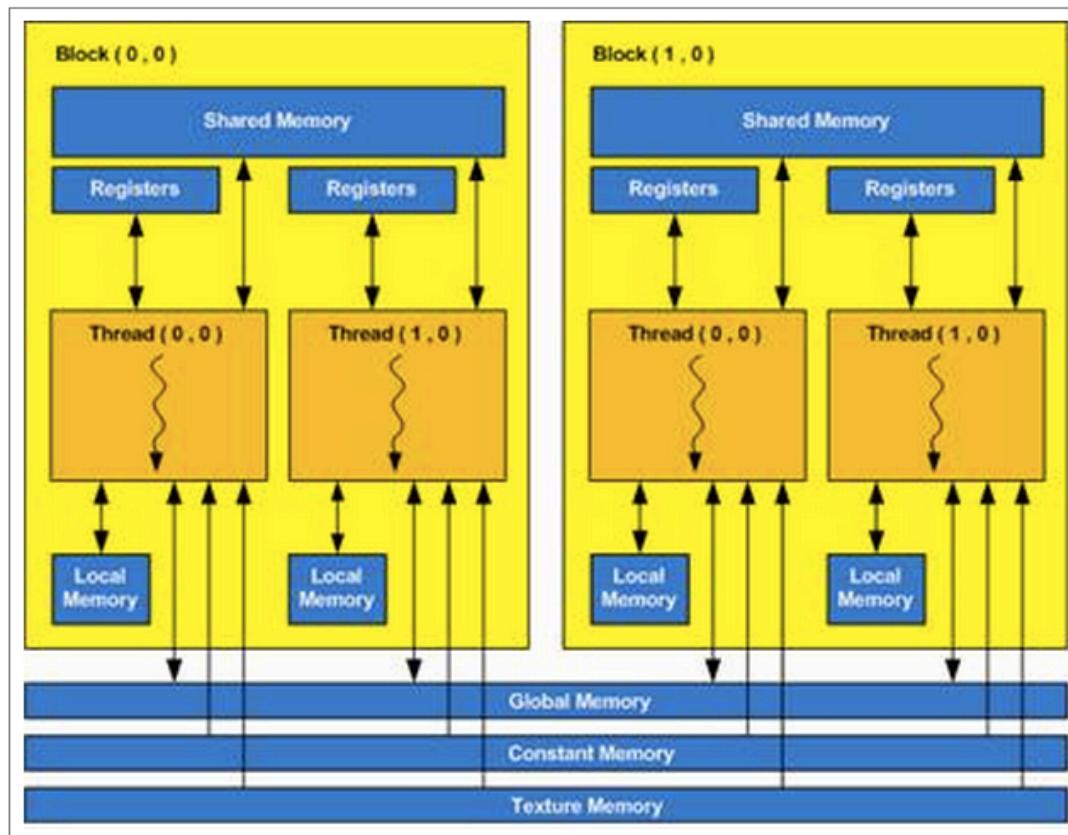
gridDim.x: B , the number of blocks in the grid.

blockDim.x: T , the number of threads per block.

maxThreadsPerBlock: exactly that: 1024 on an impact1 core.

Note: the maximum number of blocks per grid is : 65535 on an impact1 core.

THREAD COOPERATION: SHARED MEMORY AND SYNCHRONIZATION



Let's say we have a kernel:

```
--global__ void kernel( void ){
    int x;
    x = threadIdx.x + blockIdx.x * blockDim.x;
}
```

```
kernel<<<3, 2>>>();
```

→ kernel(blockIdx.x = 0, threadIdx.x = 0);	$x = 0 + 0 * 2 = 0$
→ kernel(blockIdx.x = 0, threadIdx.x = 1);	$x = 1 + 0 * 2 = 1$
→ kernel(blockIdx.x = 1, threadIdx.x = 0);	$x = 0 + 1 * 2 = 2$
→ kernel(blockIdx.x = 1, threadIdx.x = 1);	$x = 1 + 1 * 2 = 3$
→ kernel(blockIdx.x = 2, threadIdx.x = 0);	$x = 0 + 2 * 2 = 4$
→ kernel(blockIdx.x = 2, threadIdx.x = 1);	$x = 1 + 2 * 2 = 5$

Remember: we don't know which thread finishes last!

All the threads will share the same copy of x in **GLOBAL MEMORY**:

If we call:

```
kernel<<3, 2>>();
```

Then we would get:

	Block 0	Block 1	Block 2
Thread 0	x = 3	x = 3	x = 3
Thread 1	x = 3	x = 3	x = 3

If thread 1 block 1 finished last.

If, on the other hand, we define:

```
__global__ void kernel( void ){
    __shared__ int x;
    x = threadIdx.x + blockIdx.x * blockDim.x;
}
```

then each BLOCK will have its own copy of x in **SHARED MEMORY**, shared by all the threads in the block.

If we call:

```
kernel<<3, 2>>();
```

Then we would get:

	Block 0	Block 1	Block 2
Thread 0	x = 0	x = 3	x = 4
Thread 1	x = 0	x = 3	x = 4

If thread 0 finished last in block 0, thread 1 finished last in block 1, and thread 0 finished last in block 2.

NOW, WE'RE READY FOR AN EXAMPLE OF THREAD COOPERATION: THE DOT PRODUCT

$$(x_1, x_2, x_3, x_4) \bullet (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

First part of the code:

```
#include "../common/book.h"

#define imin(a,b) (a<b?a:b)

const int N = 33 * 1024;
const int threadsPerBlock = 256;

__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    // set the cache values
    cache[cacheIndex] = temp;
}
```

What the code does:

```
dot<<2,4>>(a, b, c)  
  
blockDim.x = 4  
gridDim.x = 2
```

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)

Block 0	Block 1
cache[0] =	cache[0] =
cache[1] =	cache[1] =
cache[2] =	cache[2] =
cache[3] =	cache[3] =

```
dot<<2,4>>(a, b, c)
```

```
blockDim.x = 4  
gridDim.x = 2
```

threadIdx.x = 0
blockIdx.x = 0

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)

*

+

*

cache[0] = 47

Block 0	Block 1
cache[0] = 47 cache[1] = cache[2] = cache[3] =	cache[0] = cache[1] = cache[2] = cache[3] =

```
dot<<2,4>>(a, b, c)
```

```
blockDim.x = 4  
gridDim.x = 2
```

threadIdx.x = 1
blockIdx.x = 0

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)

*

+

*

cache[1] = 14

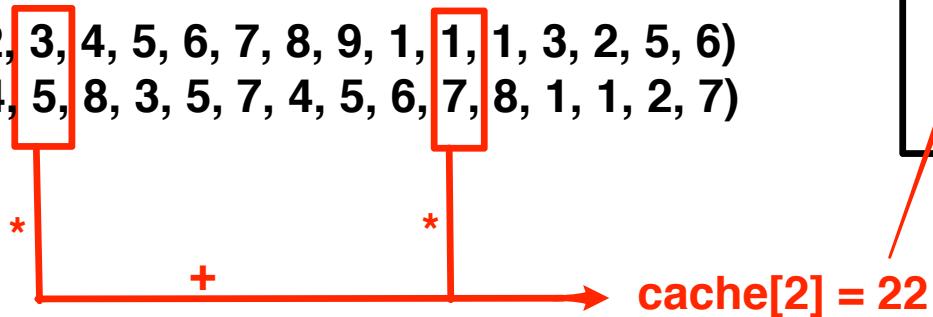
Block 0	Block 1
cache[0] = 47 cache[1] = 14 cache[2] = cache[3] =	cache[0] = cache[1] = cache[2] = cache[3] =

```
dot<<2,4>>(a, b, c)
```

```
blockDim.x = 4  
gridDim.x = 2
```

threadIdx.x = 2
blockIdx.x = 0

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)



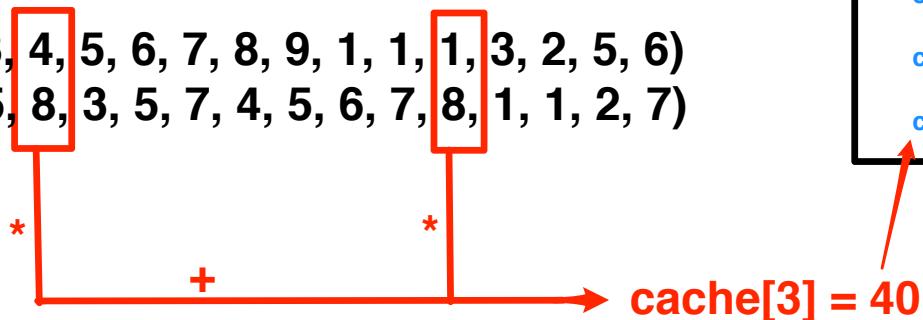
Block 0	Block 1
cache[0] = 47 cache[1] = 14 cache[2] = 22 cache[3] =	cache[0] = cache[1] = cache[2] = cache[3] =

```
dot<<2,4>>(a, b, c)
```

```
blockDim.x = 4  
gridDim.x = 2
```

threadIdx.x = 3
blockIdx.x = 0

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)

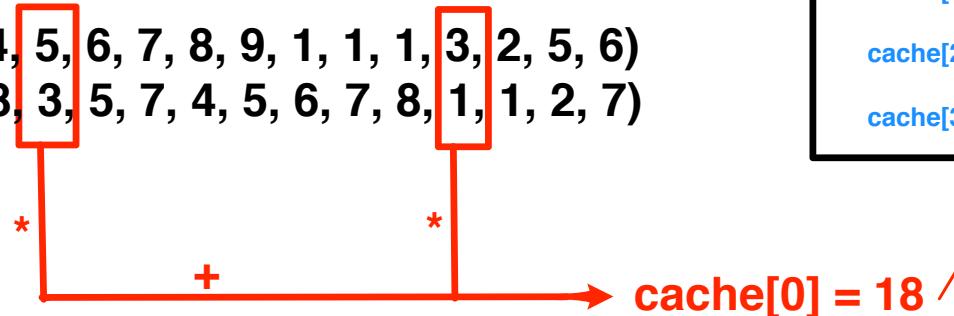


Block 0	Block 1
cache[0] = 47	cache[0] =
cache[1] = 14	cache[1] =
cache[2] = 22	cache[2] =
cache[3] = 40	cache[3] =

```
dot<<2,4>>(a, b, c)
```

```
blockDim.x = 4  
gridDim.x = 2
```

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)



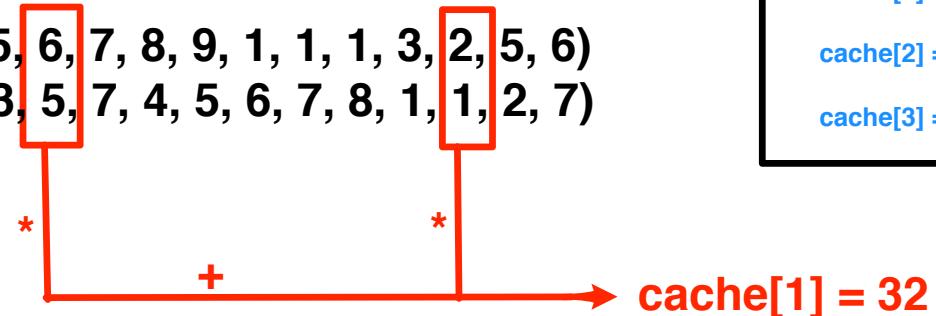
threadIdx.x = 0
blockIdx.x = 1

Block 0	Block 1
cache[0] = 47 cache[1] = 14 cache[2] = 22 cache[3] = 40	cache[0] = 18 cache[1] = cache[2] = cache[3] =

```
dot<<2,4>>(a, b, c)  
  
blockDim.x = 4  
gridDim.x = 2
```

threadIdx.x = 1
blockIdx.x = 1

```
a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)
```



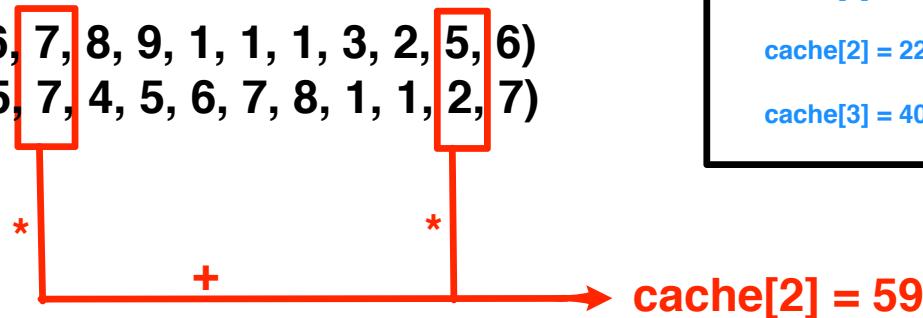
Block 0	Block 1
<pre>cache[0] = 47 cache[1] = 14 cache[2] = 22 cache[3] = 40</pre>	<pre>cache[0] = 18 cache[1] = 32 cache[2] = cache[3] =</pre>

```
dot<<2,4>>(a, b, c)
```

```
blockDim.x = 4  
gridDim.x = 2
```

threadIdx.x = 2
blockIdx.x = 1

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)



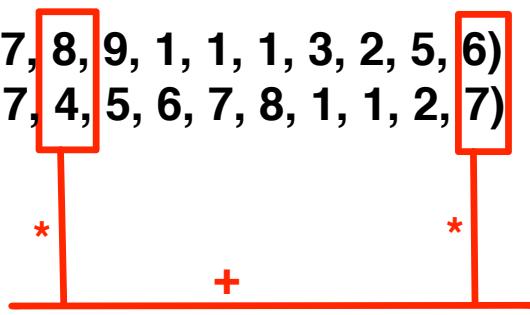
Block 0	Block 1
cache[0] = 47 cache[1] = 14 cache[2] = 22 cache[3] = 40	cache[0] = 18 cache[1] = 32 cache[2] = 59 cache[3] =

```
dot<<2,4>>(a, b, c)
```

```
blockDim.x = 4  
gridDim.x = 2
```

threadIdx.x = 3
blockIdx.x = 1

a = (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 1, 1, 3, 2, 5, 6)
b = (2, 4, 5, 8, 3, 5, 7, 4, 5, 6, 7, 8, 1, 1, 2, 7)



Block 0	Block 1
cache[0] = 47	cache[0] = 18
cache[1] = 14	cache[1] = 32
cache[2] = 22	cache[2] = 59
cache[3] = 40	cache[3] = 74

We want to make sure that `cache` is filled up for each block before we continue further.

Hence, the next line of code is:

```
// synchronize threads in this block  
__syncthreads();
```

NEXT, WE EXECUTE A PAIRWISE SUM ON cache FOR EACH BLOCK

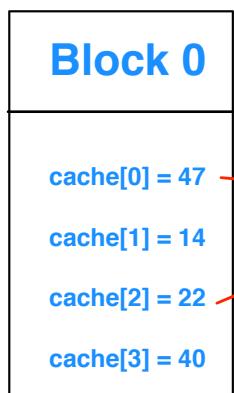
```
// for reductions, threadsPerBlock must be a power of 2
// because of the following code
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}
```

WHAT'S GOING ON

`dot<<2,4>>(a, b, c)`

`blockDim.x = 4`
`gridDim.x = 2`

cachelIndex = threadIdx.x = 0
blockIdx.x = 0
i = 2

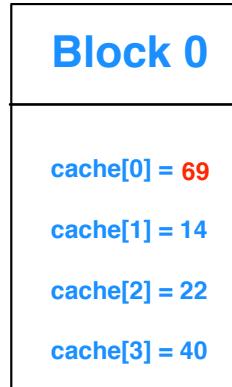


$$69 \rightarrow \text{cache}[0]$$

```
dot<<2,4>>(a, b, c)
```

```
blockDim.x = 4  
gridDim.x = 2
```

cachelIndex = threadIdx.x = 1
blockIdx.x = 0
i = 2



```
dot<<2,4>>(a, b, c)
```

```
blockDim.x = 4  
gridDim.x = 2
```

Block 0
cache[0] = 69 cache[1] = 54 cache[2] = 22 cache[3] = 40

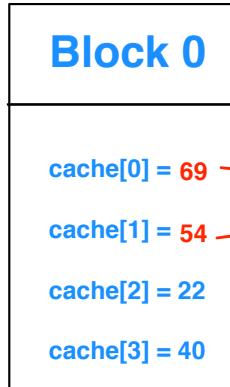
**cachelIndex = threadIdx.x = 1
blockIdx.x = 0
i = 2**

__syncthreads();

```
dot<<2,4>>(a, b, c)
```

```
blockDim.x = 4  
gridDim.x = 2
```

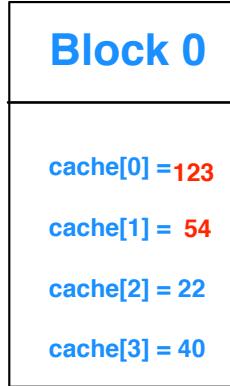
cachelIndex = threadIdx.x = 0
blockIdx.x = 0
i = 1



$$\text{cache}[0] = 69 + \text{cache}[1] = 54 = 123 \rightarrow \text{cache}[0]$$

```
dot<<2,4>>(a, b, c)
```

```
blockDim.x = 4  
gridDim.x = 2
```



cachelIndex = threadIdx.x = 0
blockIdx.x = 0
i = 1

__syncthreads();

```
dot<<2,4>>(a, b, c)
```

```
blockDim.x = 4  
gridDim.x = 2
```

Block 0
cache[0] = 123 cache[1] = 54 cache[2] = 22 cache[3] = 40

**cachelIndex = threadIdx.x = 0
blockIdx.x = 0
i = 0**

i = 0, so end the pairwise sum.

The result for block 0 is cache[0] = 123.

Similarly, the contribution of block 1 to the dot product is 183.

Next:

```
if (cacheIndex == 0)
    c [blockIdx.x] = cache [0];
}
```

So now, $c[0] = 123$ and $c[1]$ is 183.

We return c to the cpu, call it `partial_c` and then take a linear sum of the elements of `partial_c`:

```
// finish up on the CPU side  
c = 0;  
for (int i=0; i<blocksPerGrid; i++) {  
    c += partial_c[i];  
}
```

Now, c is the final answer.

COMPLETE CODE

```
#include "../common/book.h"

#define imin(a,b) (a<b?a:b)

const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
    imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );

__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
}
```

```
// set the cache values
cache[cacheIndex] = temp;

// synchronize threads in this block
__syncthreads();

// for reductions, threadsPerBlock must be a power of 2
// because of the following code
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}
```

```
    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
    }

int main( void ) {
    float *a, *b, c, *partial_c;
    float *dev_a, *dev_b, *dev_partial_c;

    // allocate memory on the CPU side
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                            N*sizeof(float) ) );

```

```
HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                           N*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
                           blocksPerGrid*sizeof(float) ) );

// fill in the host memory with data
for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i*2;
}

// copy the arrays 'a' and 'b' to the GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),
                           cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),
                           cudaMemcpyHostToDevice ) );
```

```
dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,
                                              dev_partial_c );

// copy the array 'c' back from the GPU to the CPU
HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                        blocksPerGrid*sizeof(float),
                        cudaMemcpyDeviceToHost ) );

// finish up on the CPU side
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}

#define sum_squares(x) (x*(x+1)*(2*x+1)/6)
```

```
printf( "Does GPU value %.6g = %.6g?\n", c,
        2 * sum_squares( (float)(N - 1) ) );
```

// free memory on the GPU side

```
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_partial_c );
```

// free memory on the CPU side

```
free( a );
free( b );
free( partial_c );
```

```
}
```

LECTURE SERIES MATERIALS

These lecture slides, a tentative syllabus for the whole lecture series, and code are available at:

<https://github.com/wlandau/gpu>.

After logging into you home directory on impact1, type:

```
git clone https://github.com/wlandau/gpu
```

into the command line to download all the course materials.

REFERENCES

David B. Kirk and Wen-mei W. Hwu. “Programming Massively Parallel Processors: a Hands-on Approach.” Morgan Kaufman, 2010.

J. Sanders and E. Kandrot. *CUDA by Example*. Addison-Wesley, 2010.

Michael Romero and Rodrigo Urra. ”CUDA Programming.” Rochester Institute of Technology.
http://cuda.ce.rit.edu/cuda_overview/cuda_overview.html