CUDA C: THE SIMD PARADIGM, SHARED MEMORY, AND THE DOT PRODUCT

Will Landau, Prof. Jarad Niemi

OUTLINE

- Respecting the SIMD paradigm

- Shared memory

- Implementing the dot product

Featured examples:

- `dot_product.cu`

THE SIMD PARADIGM

SIMD: Single Instruction Multiple Data

Each thread uses the same code, but applies it to different data.

Try to respect this paradigm in you code. If multiple threads access the same data, problems could arise.

EXAMPLE

Let's say we have a kernel:

```
__global__ void kernel(void){
  int x = blockIdx.x;
}
```
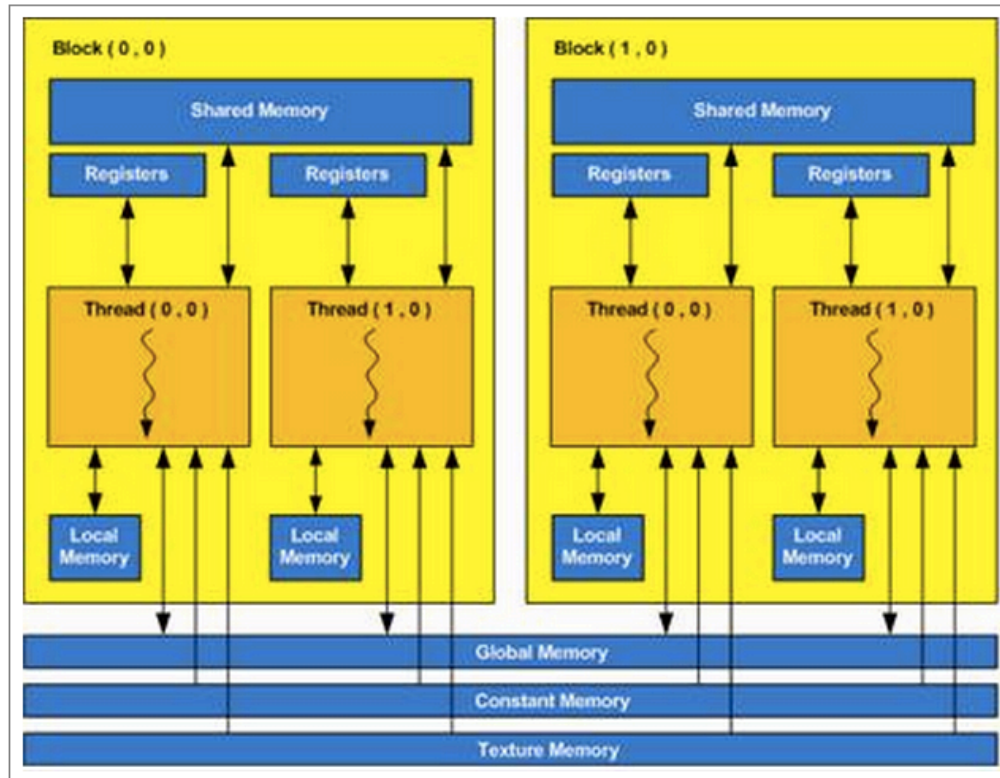
kernel<<< 6, 1 >>>( );

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

What will be the final value of x?

All the threads in the grid share the same copy of `a` in GLOBAL MEMORY.

Hence, the final value of `x` is the block ID of the thread that finishes last.

GLOBAL VS SHARED MEMORY

Why not give each block its own private copy of **x** in shared memory?

If we define:

```
__global__ void kernel(void){
  __shared__ int x = blockIdx.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<<6, 1>>();
```

Then the final values of x will ALWAYS be:

	Block 0	Block 1	Block 2	Block 3	Block 4	Block 5
Thread 0	x = 0	x = 1	x = 2	x = 3	x = 4	x = 5

NOW, WE'RE READY FOR THE DOT PRODUCT

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

First part of the code:

```c
#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] =

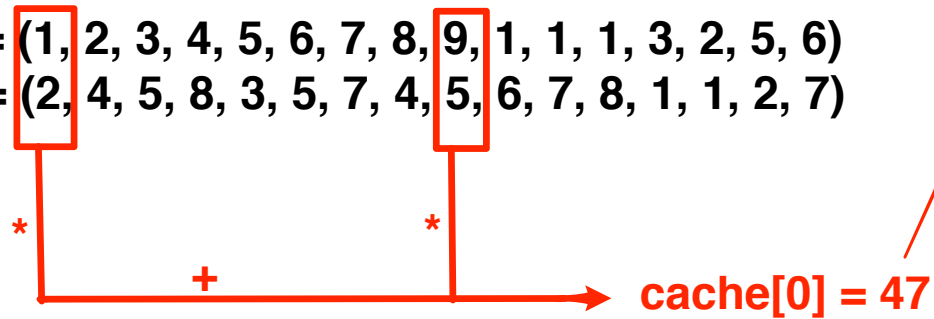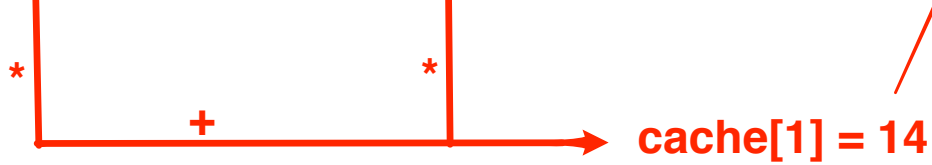dot<<2,4>>(a, b, c)

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

*   *

\+

cache[0] = 47

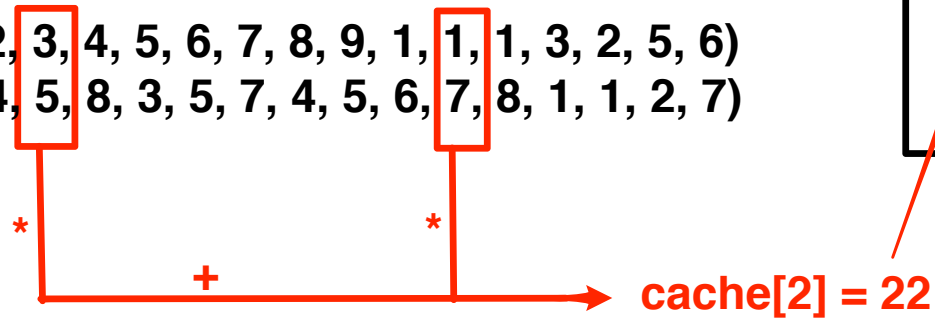| Block 0	Block 1
cache[0] = 47	cache[0] =
cache[1] =	cache[1] =
cache[2] =	cache[2] =
cache[3] =	cache[3] =

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

blockDim.x = 4
gridDim.x = 2

threadIdx.x = 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[0] =
cache[1] = 14	cache[1] =
cache[2] =	cache[2] =
cache[3] =	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)

*   *

+

cache[2] = 22

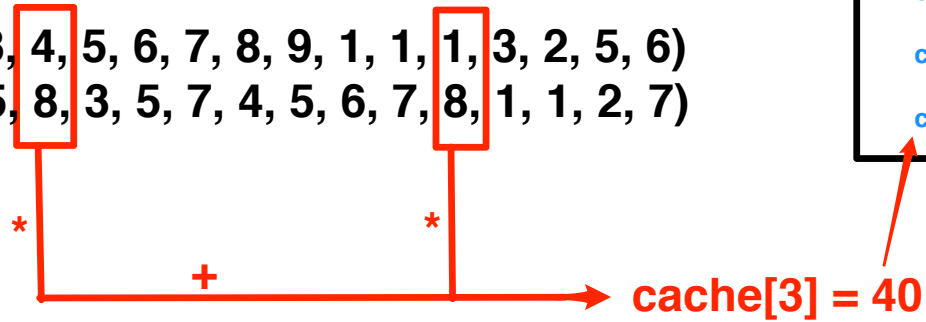| Block 0	Block 1
cache[0] = 47	cache[0] =
cache[1] = 14	cache[1] =
cache[2] = 22	cache[2] =
cache[3] =	cache[3] =

13

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)

*

*

\+

cache[3] = 40

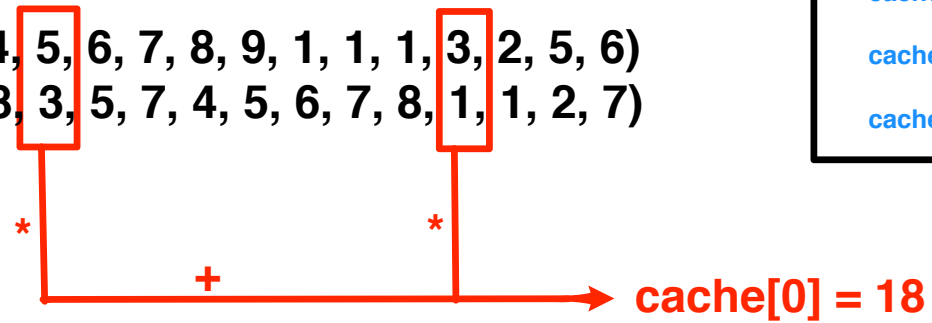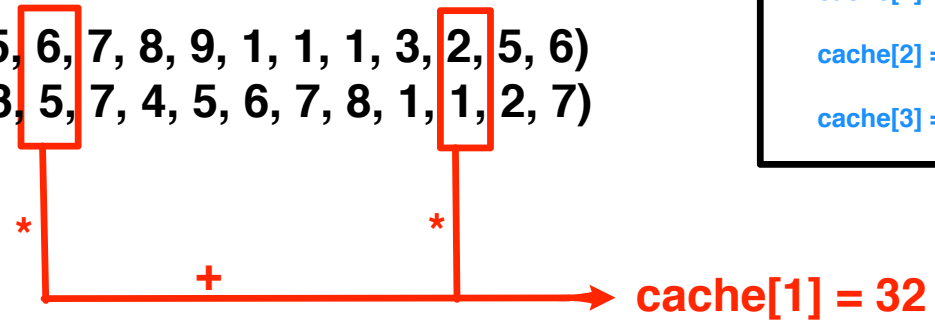| Block 0	Block 1
cache[0] = 47	cache[0] =
cache[1] = 14	cache[1] =
cache[2] = 22	cache[2] =
cache[3] = 40	cache[3] =

14

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

blockDim.x = 4
gridDim.x = 2

threadIdx.x = 0
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)

*   *

+

cache[0] = 18

Block 0	**Block 1**
cache[0] = 47	cache[0] = 18
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

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)

*   *

+

cache[1] = 32

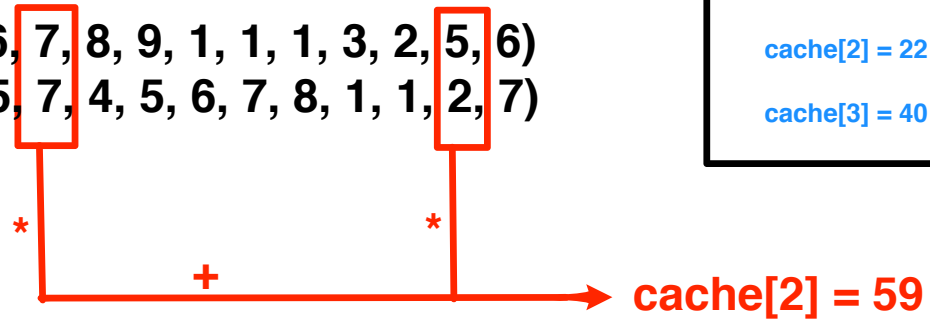| Block 0	Block 1
cache[0] = 47	cache[0] = 18
cache[1] = 14	cache[1] = 32
cache[2] = 22	cache[2] =
cache[3] = 40	cache[3] =

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)

*   *

        +   cache[2] = 59

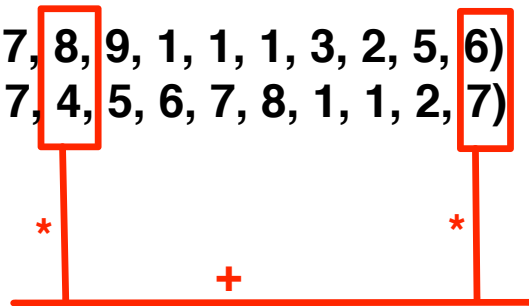| Block 0	Block 1
cache[0] = 47	cache[0] = 18
cache[1] = 14	cache[1] = 32
cache[2] = 22	cache[2] = 59
cache[3] = 40	cache[3] =

17

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)

*   *

+

→ **cache[3] = 74**

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;
}
```
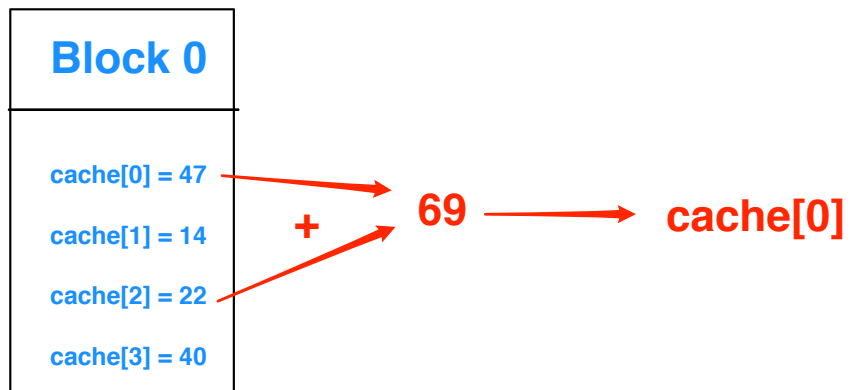
WHAT'S GOING ON

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

blockDim.x = 4
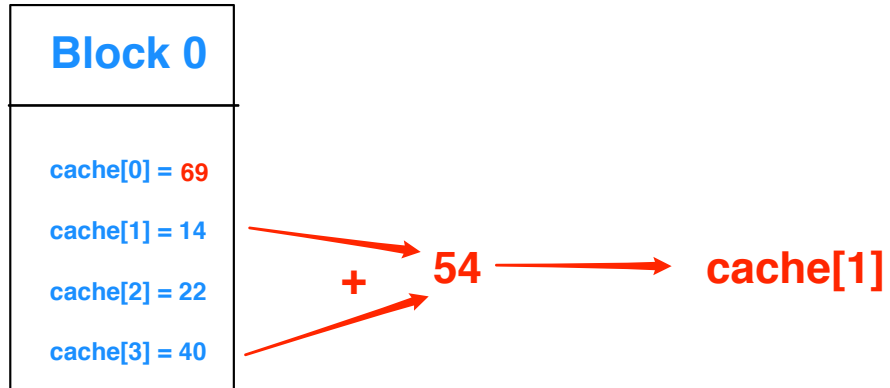gridDim.x = 2

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

Block 0

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

+ **69** → **cache[0]**

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

blockDim.x = 4
gridDim.x = 2

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

Block 0
cache[0] = 69
cache[1] = 14
cache[2] = 22
cache[3] = 40

+ **54** → **cache[1]**

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

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

__synchthreads();

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

blockDim.x = 4
gridDim.x = 2

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

Block 0

cache[0] = 69

+   **123** ⟶ **cache[0]**

cache[1] = 54

cache[2] = 22

cache[3] = 40

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

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

__synchthreads();

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

blockDim.x = 4
gridDim.x = 2

cacheIndex = threadIdx.x = 0
blockIdx.x = 0
i = 0

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

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 ) );
```

```cuda
    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 );
}
```

OUTLINE

- Respecting the SIMD paradigm

- Shared memory

- Implementing the dot product

Featured examples:

- `dot_product.cu`

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