

CUDA Tutorial

To check if your CUDA development kit has installed correctly, first goto the DOS prompt and type `nvcc --version`.

If cuda has set the appropriate path and installed the nvcc compiler you should see something similar to this:

```
C:\Users\sc-gd>nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2017 NVIDIA Corporation
Built on Fri_Sep__1_21:08:32_Central_Daylight_Time_2017
Cuda compilation tools, release 9.0, V9.0.176
```

To compile a program from the command line prompt you type:

```
nvcc program_name.cu -o program_name
```

where `program_name` is your program text file that you wrote in your favourite editor. Note that cuda source code programs have the extension `.cu`

Check all is ok by creating the following program. Call it `my_first.cu` and compile it as described above.

my_first.cu

```
#include <stdio.h>

__global__ void foo()
{
}

int main()
{
    foo<<<1,1>>>();
    printf("Hello World.  CUDA error: %s\n",
        cudaGetErrorString(cudaGetLastError()));
    return 0;
}
```

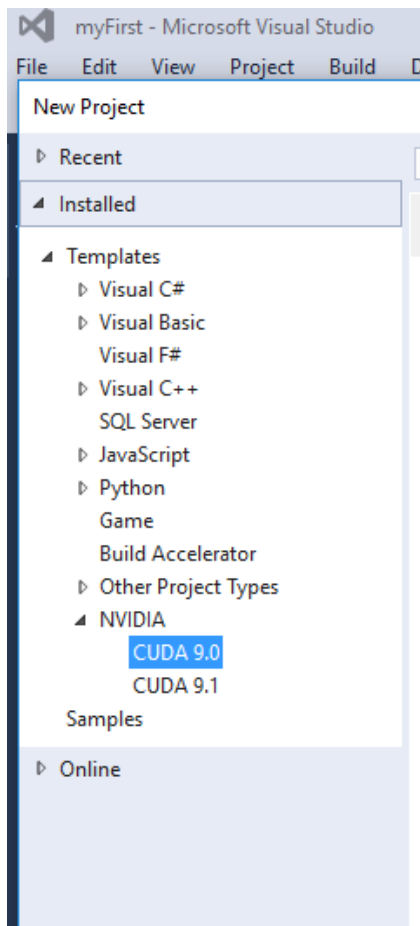
The nvcc compiler also makes use of Microsoft's C compiler. If you get an error message saying it can't find `cl.exe` then you will probably need to add the directory path that contains `cl.exe` to your system path. To do this open the **system properties** window by following these steps:

1. Open a run window from the Start Menu (type `cmd` in Start Menu search box)
2. Type `sysdm.cpl` then press enter.
3. When the system properties window appears select the Advanced tab and click on the Environment Variables button.

4. Create a new System Variable called VC140PATH and give it the directory name where **cl.exe** is installed – it's **C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\bin** on my windows 10 machine.

You can compile and execute nvcc CUDA programs using Microsoft Visual Studio 2015 and above. Here's how:-

If you haven't done so already, first install Visual Studio 2015 or above on your system then install NVidia CUDA 9.0. CUDA's installer should automatically place a Nvidia CUDA 9.0 template in all



Visual Studio applications it finds. To create a new CUDA project, select File->New->Project and select the installed NVIDIA CUDA 9.0 template then continue creating a new project in the normal way. The default project will not be empty, but a simple CUDA program that uses the GPU to add two vectors. We'll go through this in class.

If you are using an early edition of MSVS (eg 2008) you will need to add custom rules to tell Visual studio where to find all CUDA libraries, include files, etc. If you want to go down this route then please use the empty example project (for VS 2008) found in the cudaVS2008.zip file.

The difference between host memory and device memory.

As we've seen, nvcc needs the cl.exe compiler in order to successfully create CUDA programs. This is because a CUDA program has a part that runs on the host's CPU (the VS C compiler creates the executable for this) and another part that executes on the GPU (the nvcc compiler is needed here). Memory is treated separately too, the CPU cannot access GPU memory using pointers as we would normally expect, and vice versa. Instead we must manage and copy memory between host and the GPU device(s) using special CUDA specific functions. These are executed by the host CPU.

The following program illustrates how a CUDA program achieves this.

```

#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int num_elements = 16;
    int num_bytes = num_elements * sizeof(int);

    int *device_array = 0;
    int *host_array = 0;

    // malloc host memory
    host_array = (int*)malloc(num_bytes);

    // cudaMalloc device memory
    cudaMalloc((void**)&device_array, num_bytes);

    // zero out the device array with cudaMemset
    cudaMemset(device_array, 0, num_bytes);

    // copy the contents of the device array to the host
    cudaMemcpy(host_array, device_array, num_bytes,
cudaMemcpyDeviceToHost);

    // print out the result element by element
    for(int i = 0; i < num_elements; ++i)
        printf("%d ", host_array[i]);

    // use free to deallocate the host array
    free(host_array);

    // use cudaFree to deallocate the device array
    cudaFree(device_array);

    return 0;
}

```

Since gpu device memory and cpu host memory are separated then it is not difficult to accept that cpu code and gpu device code are also separate. This means that we must differentiate between code written for the host cpu and code written for the gpu device. Cuda achieves this by using the concept of `__global__` functions which tell the cuda compiler how to recognise code written for the gpu device. a `__global__` function is similar to a C program's main function in that it provides a point of entry when executing device code.

Data parallelization using `__global__` functions.

We now know how to run a `__global__` function on a gpu device but how can we run multiple copies of this function on an array of different data? Cuda achieves this by allowing one `__global__` function that you've written to be executed on N different pieces of data **at the same time**. Each instance of the `__global` function is called a kernel and all threads execute this kernel function. The obvious question then becomes how do we map **each** thread to an individual piece of data? We need a syntax that easily references the right data and that can define the size of the data set we want to work with.

The following code shows how Cuda's designers addressed this problem.

```
#include <stdlib.h>
#include <stdio.h>
__global__ void kernel(int *array)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    array[index] = 7;
}

int main(void)
{
    int num_elements = 256;
    int num_bytes = num_elements * sizeof(int);
    // pointers to host & device arrays
    int *device_array = 0;
    int *host_array = 0;
    // malloc a host array
    host_array = (int*)malloc(num_bytes);
    // cudaMalloc a device array
    cudaMalloc((void*)&device_array, num_bytes);
    int block_size = 128;
    int grid_size = num_elements / block_size;
    kernel<<<grid_size,block_size>>>>(device_array);
    // download and inspect the result on the host:
    cudaMemcpy(host_array, device_array, num_bytes, cudaMemcpyDeviceToHost);
    // print out the result element by element
    for(int i=0; i < num_elements; ++i)
    {
        printf("%d ", host_array[i]);
    }
    // deallocate memory
    free(host_array);
    cudaFree(device_array);
}
```

First notice that the `__global__` function returns nothing. This makes sense since the same function is called thousands of times in parallel but only one value can be returned to the calling function (i.e. the host CPU). Hence the designers do not allow a returned value from a `__global__` function.

Each thread needs to be mapped to a piece of data so the first thing to do is to tell the `__global__` function where to find the dataset. The array pointer is passed to the `__global__` function as a parameter. Each thread needs to be linked to a single item in that array. Cuda achieves this by maintaining some Cuda specific variables that, for each thread, contain specific information identifying that thread. All the programmer needs to do is arrange for N threads to run on an array of size N. If the number of threads launched is not the same as the number of data items to work on, then there might be some problems! Incidentally, you can pass more than one pointer variable to a `__global__` function but you still need to map threads to the right data.

By way of illustration, the previous example calculates an index by using the sum,
`index = blockIdx.x * blockDim.x + threadIdx.x`
 where `blockIdx.x` refers to this thread's block number., `blockDim.x` refers to the size of the block and `threadIdx.x` refers to the thread's number.

Block 0													Block 1												
0	1	2	3	4	5	6	7	8	..	125	126	127	0	1	2	3	4	5	6	7	8	..	125	126	127

```
block_size = 128
num_elements = 256
grid_size= 2

int grid_size = num_elements / block_size;
kernel<<<grid_size,block_size>>>(device_array);
```

Notice that we need to calculate the grid size and the block size. The array element (index) corresponding to a particular kernel is calculated as above. See that, for example, a thread whose block Id is 1 and thread ID is 3 will use data from array element index = $1 \times 128 + 3 = 131$.

Now we can see that this program copies the number 7 to all 256 elements of the array using the statement `array[index] = 7;` in the `__global__` function.

The 'x's of the variables refer to a dimension. In our table above the x dimension is horizontal and increases from left to right. We can also extend this approach to create 2 dimensional blocks by using x and y dimensions. The following code demonstrates this approach. In fact, we can think of Cuda's dim variable as a C struct similar to
`struct dim{ int x; int y; int z;}`

If not used then y and/or z are assumed to be set to 1. Later versions of CUDA allow for a 3-dimensional block that includes a `threadId.z` value - earlier versions had only two dimensions.

```

#include <stdlib.h>
#include <stdio.h>
__global__ void kernel(int *array)
{
    int index_x = blockIdx.x * blockDim.x + threadIdx.x;
    // index_x = blockIdx.x*8 +threadIdx.x
    int index_y = blockIdx.y * blockDim.y + threadIdx.y;
    // index_y = blockIdx.y*4 +threadIdx.y
    // map the two 2D indices to a single linear, 1D index
    int grid_width = gridDim.x * blockDim.x;
    // grid_width = 2*8
    int index = index_y * grid_width + index_x;
    // map the two 2D block indices to a single linear, 1D block index
    int result = blockIdx.y * gridDim.x + blockIdx.x;
    // write out the result
    array[index] = result;
}

int main(void)
{
    int num_elements_x = 16;
    int num_elements_y = 32;

    int num_bytes = num_elements_x * num_elements_y * sizeof(int);

    int *device_array = 0;
    int *host_array = 0;

    // allocate memory in either space
    host_array = (int*)malloc(num_bytes);
    cudaMalloc((void**)&device_array, num_bytes);

    // create two dimensional 8x4 thread blocks
    dim3 block_size;
    block_size.x = 8;
    block_size.y = 4;
    // configure a two dimensional grid as well
    dim3 grid_size;
    grid_size.x = num_elements_x / block_size.x;
    grid_size.y = num_elements_y / block_size.y;
    //grid_size.x = 16/8 = 2 and gridsize.y= 32/4 =8
    // grid_size & block_size are passed as arguments to the triple chevrons
    as usual
    kernel<<<grid_size,block_size>>>(device_array);

    // download and inspect the result on the host:
    cudaMemcpy(host_array, device_array, num_bytes,
        cudaMemcpyDeviceToHost);

    // print out the result element by element
    for(int row = 0; row < num_elements_y; ++row)
    {
        for(int col = 0; col < num_elements_x; ++col)
        {
            printf("%2d ", host_array[row * num_elements_x + col]);
        }
        printf("\n");
    }
    printf("\n");
    // deallocate memory

```

```

    free(host_array);
    cudaFree(device_array);
}

```

This code outputs a two dimensional array of data, each piece of data is the block number associated with the current thread. The width of the array is 16 given by the `__global__` function statement:

```
int grid_width = gridDim.x * blockDim.x; //16=2*8
```

and there are 32 rows since $\text{gridDim.y} * \text{blockDim.y} = 8 * 4$

The following command window shows the output and hence the way the blocks are arranged:

```

0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3
2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3
2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3
2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5
4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5
4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5
4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7
6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7
6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7
6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 9 9 9 9 9 9 9 9
8 8 8 8 8 8 8 8 9 9 9 9 9 9 9 9
8 8 8 8 8 8 8 8 9 9 9 9 9 9 9 9
8 8 8 8 8 8 8 8 9 9 9 9 9 9 9 9
10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11
10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11
10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11
10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11
12 12 12 12 12 12 12 12 13 13 13 13 13 13 13 13
12 12 12 12 12 12 12 12 13 13 13 13 13 13 13 13
12 12 12 12 12 12 12 12 13 13 13 13 13 13 13 13
12 12 12 12 12 12 12 12 13 13 13 13 13 13 13 13
14 14 14 14 14 14 14 14 15 15 15 15 15 15 15 15
14 14 14 14 14 14 14 14 15 15 15 15 15 15 15 15
14 14 14 14 14 14 14 14 15 15 15 15 15 15 15 15
14 14 14 14 14 14 14 14 15 15 15 15 15 15 15 15

```

It is worth modifying the code to see how changes to the block size, etc changes the way the data is organised, for example changing the size of blocks and their dimension (at the same time making the data size the same!)

Handy Hint:

If you want to output some data just once, for example you want to know the block dimension and grid dimension stored in the variables `gridDim.x` and `blockDim.x` (2 and 8 respectively in the last code example) then you could selectively overwrite your memory as follows.

```
if (blockIdx.x==0) array[index]=gridDim.x;
if (blockIdx.x==1) array[index]=blockDim.x;
```

This has the effect of making thread 0 overwrite element 0 of the array and thread 1 overwrite element 1 of the array. Try it and see. If you do this then remember that this will only work with data of the same type as that of the array.

Memory Management

So far we've learnt how to launch many threads in parallel and associate them with specific memory elements. We've seen that the kernel function is the same but the data is different (recall the SIMD architecture). We noticed that we have to specify blocks in order to achieve this but we aren't clear yet about what the purpose is of these blocks. We know that they are somehow related to the architecture of the GPU but NVidia have gone to some length to expose this architecture and develop a suitable syntax. Why? What is the point? A suitable but vague answer to this question is "to efficiently use memory." Before we explore this further let's recap what we know about memory management in CUDA.

First we know device pointers point to GPU memory and Host pointers point to CPU memory. In order for a device (GPU) to access host memory and vice versa, Cuda explicitly requires us to use special memory copy functions. This is because the following limitations apply to memory access.

Device pointers point to GPU memory

- These may be passed to and from host code but in earlier versions of CUDA they may not be de-referenced from the host code

Host pointers point to CPU memory

- They may be passed to and from device code but in earlier versions of CUDA they may not be de-referenced from the device code

The following `add()` example illustrates how we can use host and device pointers together with cuda memory copy functions to facilitate the passing and returning of parameters to/from the GPU device from/to the CPU host. Note we have already achieved this with our earlier examples but now we can go a little further by seeing how we can pass multiple parameters and access the result. The `add()` example we look at now will launch just one thread so is not very useful at all but it serves to illustrate the concepts.

```

#include <iostream>
using namespace std;

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

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 size of an int to allocate mem on CPU

    //allocate device copies of a,b,c

    cudaMalloc((void**)&dev_a,size);
    cudaMalloc((void**)&dev_b,size);
    cudaMalloc((void**)&dev_c,size);

    a= 10; b=20; //host copies of a and b initialised

    //copy input parameters 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);
    // print out the result
    cout<<"Sum is: "<<c<<endl;

    // tidy up - deallocate memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
    return 0;
}

```

Note that all accesses to dev_a, dev_b and dev_c must be achieved by using cudaMemcpy() to transfer data to/from the host. Now review the earlier code and note the differences: we used only one array in the earlier examples but three parameters in this last example; and secondly we called multiple parallel threads using a suitable block size.

How could we convert this single thread add() example to work in parallel, say 512 additions, using say 16 blocks? Let's put the input data in two arrays (does it have to be done this way?) - int a[512] and int b[512] and let the result be ultimately stored on the host in array int c[512].

Give it a go yourselves before looking at the following solution. This solution is based upon the code examples we've looked at so far.

```

#include <iostream>
using namespace std;

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

#define N 512

int main(void)
{
    int *a, *b,*c;           // host copies of a,b,c arrays
    int num_ints = N*sizeof(int);
    //lets put stuff in them

    a=(int*)malloc(num_ints);
    b=(int*)malloc(num_ints);
    c=(int*)malloc(num_ints);
    for(int i=0;i<N;i++)
    {
        *(a + i)=i;
        *(b+i)=5;
        *(c+i)=0; //init to zero
    }

    int *dev_a, *dev_b, *dev_c; //device copies of a,b,c

    //allocate device copies of a,b,c

    cudaMalloc((void**)&dev_a, num_ints);
    cudaMalloc((void**)&dev_b, num_ints);
    cudaMalloc((void**)&dev_c, num_ints);

    //copy input parameters to device
    cudaMemcpy(dev_a, a, num_ints, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, num_ints, cudaMemcpyHostToDevice);
    int block_size = 16;
    int grid_size = N / block_size;

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

    //copy device result back to host copy of c
    cudaMemcpy(c, dev_c, num_ints, cudaMemcpyDeviceToHost);
    // print out the result
    for (int i=0;i<N;i++)
        cout<<a[i] <<" + "<<b[i]<<" = "<<c[i]<<endl;
    // tidy up - deallocate memory on host then device
    cudaFree(dev_a);   cudaFree(dev_b);   cudaFree(dev_c);
    free(a);free(b); free(c);
    return 0;
}

```

We could have solved this problem in a different way but the overall approach would remain the same. We could have changed the number of threads per block, for instance we might have N blocks containing just 1 thread or N/2 blocks of 2 threads and so on. If we did decide to use N blocks of one thread our kernel call would be:

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

and the kernel code would also be slightly simpler :

```
__global__ void add(int *a, int *b, int*c)
{
/*****NO NEED FOR THIS MAPPING *****/
int index = blockIdx.x * blockDim.x + threadIdx.x;
*****/
    c[blockIdx.x]=a[blockIdx.x] +b[blockIdx.x];
}
```

Or, if we had 1 block and N threads our code would be:

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

and the kernel function would be:

```
__global__ void add(int *a, int *b, int*c)
{
/*****NO NEED FOR THIS MAPPING *****/
int index = blockIdx.x * blockDim.x + threadIdx.x;
*****/
    c[threadIdx.x]=a[threadIdx.x] +b[threadIdx.x];
}
```

This begs the question, why bother with all multiple block/thread definitions? The partial answer is it's a memory issue.

It turns out that NVidia designed CUDA with different kinds of memory. Some local to a multiprocessor (and there are lots of them!), some local to blocks of multiprocessors (a clue here) and some global. The difference between these memory types, apart from their scope, is their speed of access. Local processor memory, i.e. processor registers, are very fast to read and write to. Block memory is slower but still quite quick but can only be shared by a block of processors (think threads). Global memory can be shared by all threads but is slow. Global memory is also where data is placed/retrieved from when we copy from/to the host CPU.

So there it is, we use blocks (of threads) to share data between threads and to access memory quickly. There is another reason but let's finish this discussion first (:-)).

To demonstrate how we can use blocks of threads to our advantage let's consider another vector operation, the dot product. Recall the dot product outputs a scalar given two vectors:

$$c = \mathbf{a} \cdot \mathbf{b} \text{ where } c = a_0 b_0 + a_1 b_1 + a_2 b_2$$

We can easily compute parallel pairwise products, $a_2 * b_2$, etc. using code almost identical to the previous addition examples. But the dot product requires us to add the results of these products, in other words we need to share data between threads. How best do we achieve this?

We could try:

```

__global__ dot (int *a, int*b, int*c)
{
int temp;
temp= a[threadIdx.x]*b[threadIdx.x];  //each thread calcs. the
pairwise product
//but temp is not shared between threads, temp is private to a
thread.
}

```

But temp is not shared between threads, temp is private to a thread. To solve this problem we use blocks of threads. A block of threads shares memory called "shared memory." It is extremely fast, on chip memory (call it user-managed cache). To use this memory we declare variables with the `__shared__` CUDA keyword. We can now use shared memory to do parallel multiplication and serial addition as follows:

```

#define N 512
__global__ void dot( int*a, int*b, int*c )
{
    // Shared memory for results of multiplication
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
    // Thread 0 sums the pairwise products
    if( 0 == threadIdx.x )
    {
        Int sum = 0;
        for( int i= 0; i< N; i++ ){
            sum += temp[i];
        }
        *c = sum;
    }
}

```

Notice that all threads share the same block memory, temp[], and global memory a[], b[] and c and that we use the trick we learnt earlier to get just one specific thread - here it is thread zero- to perform the serial addition of the pair-wise products.

There is a problem here. It is hidden in the thread zero code. How can we guarantee that all the other threads have finished their multiplications before we get thread zero to sum the results? As it stands, we can't. In all probability thread zero will start summing the pairwise products before all other threads have finished calculating them. Oops! We call such problems "data hazards."

So how do we solve this problem? The answer to this is **synchronisation** of threads and it is the other reason we use blocks of threads. We can force a thread to wait until all other threads in its block have completed their task. This is just what we need to solve our problem and we use the CUDA keyword `__syncthreads()` to do this. Again note that threads are only synchronised

within a block. The code needs just one extra line in order to work correctly - the addition of a `__syncthreads()` call in the right place. The code below includes this:

```
#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
    __syncthreads(); //wait for all products to be calc'd then..

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

#include <iostream>
using namespace std;
#define N 512
int main(void)
{
    int *a, *b,*c; // host copies of a,b,c arrays
    int num_ints = N*sizeof(int);
    //lets put stuff in them

    a=(int*)malloc(num_ints);
    b=(int*)malloc(num_ints);
    c=(int*)malloc(sizeof(int)); //only one number returned from dot
    product of a.b
    for(int i=0;i<N;i++)
    {
        *(a + i)=2;
        *(b+i)=5;
    }
    *c=0; //set c to zero initially
    int *dev_a, *dev_b, *dev_c; //device copies of a,b,c

    //allocate device copies of a,b,c

    cudaMalloc((void**)&dev_a, num_ints);
    cudaMalloc((void**)&dev_b, num_ints);
    cudaMalloc((void**)&dev_c, sizeof(int));

    //copy input parameters to device
    cudaMemcpy(dev_a, a, num_ints, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, num_ints, cudaMemcpyHostToDevice);
    int block_size = N;
    int grid_size = N / block_size;
    //N threads in 1 block
```

```

    //launch dot() kernel on GPU, passing parametersgd
    dot<<<grid_size,block_size>>> (dev_a, dev_b,dev_c);
    //copy device result back to host copy of c
    cudaMemcpy(c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
    // print out the result
    cout<<" dot product of a . b is: "<<*c<<endl;
    // tidy up - deallocate memory on host then device
    cudaFree(dev_a);    cudaFree(dev_b);    cudaFree(dev_c);
    free(a);free(b); free(c);
    return 0;
}

```

Launching just one block does not utilize much of the GPU. Imagine we wanted to perform the same function but wanted to use more data and several blocks. We need to write a multiblock version of the dot product.

Each block will perform its own pair-wise product and subset of the sum. Each block will need to pool its results and add its subset of the calculation to a common location accumulating the result. The next piece of code shows how best to do this.

```

#include <iostream>
#define N 1024*2
#define THREADS_PER_BLOCK 512
__global__ void dot( int* a, int* b, int* c ) {
    __shared__ int temp[THREADS_PER_BLOCK];
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    temp[threadIdx.x] = a[index] * b[index];
    __syncthreads();
    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i= 0; i< THREADS_PER_BLOCK; i++ )
            sum += temp[i];
        // *c+=0;  /*****Danger here*****/
        atomicAdd( c , sum );
    }
}

using namespace std;

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) );
for(int i=0;i<N;i++)
{
    *(a + i)=2;
    *(b+i)=5;
}
*c=0;    //set c to zero initially

// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice);
cudaMemcpy( dev_c, c, sizeof(int), cudaMemcpyHostToDevice);
//why have we bothered with the dev_c memory copy?
// 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);

cout<<*c<<endl;

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

```

Notice the warning in the code on the commented out line `c+=sum`; and the following function call `atomicAdd(c,sum)`. What is this all about? Why have we used `atomicAdd()`? When blocks of threads want to share global memory and we want to perform a read-modify-write instruction such as `c+= sum` then we need to be sure that the complete read, modify and write process has finished before a second begins. If two threads try accessing the memory pointed to by `c` in our example then one thread can't be sure that the other thread has read and incremented the memory location. For example, thread A might read `c`'s memory location contents, say its 0, and might want to add 1 to it. The result should be 1. Thread B will access the data, read 1 and also add 1 to it to get 2 - no problems here but if thread B accesses the data before thread A has had chance to finish updating the value from 0 to 1 then thread B will read 0, thread A will update the value to be 1 and then thread B will update its copy of the data (which it has already read and is 0) to be 1 and then copy the result (1) back to the location pointed to by `c`. Instead of the final result being 2 it is erroneously set to 1. This race condition can occur when several threads are competing for the same global memory. To avoid this an atomic operation locks the memory until a complete read-modify-write has finished and `atomicAdd()` does just this.

Atomic operations are not available on "Compute Capability" 1.0 (sm_10) devices, which is what we're asking nvcc to compile for (by default). To use atomicAdd() in your code, specify at least -arch=sm_11 on the command line as follows:

```
nvcc -arch=sm_11 myprog.cu -o myprog
```

and 2.x or greater if using 64bit machines. If you are using early versions of MVS then you will need to modify the CUDA custom build rule that you will have added when you first created a VS CUDA/C++ project (right click on your project in the Solution Explorer and select "Custom Build Rules").

Now we have enough knowledge of CUDA to implement our chosen algorithms using parallel programming techniques. You will probably need more support. The CUDA by Example book is a good resource and NVidia's website programming reference is excellent.
<http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

Acknowledgements

The following tutorial is based on several sources.

NVidia's CUDA C Programming Guide.

The first part is based on Jared Hoberock's and David Tarjan's tutorial available at:
<http://code.google.com/p/stanford-cs193g-sp2010/wiki/GettingStartedWithCUDA>
and uses their code (albeit modified in places).

This course is itself based on Wen-mei Hwu & David Kirk's tutorials and lectures.

The second half uses material based on Jason Sanders' tutorial and the book CUDA by Example by Jason Sanders (again) and Jack Dongarra.

The section on CPU and GPU timing is based on Microsoft's Visual Studio reference and documentation, and Microsoft PowerShell reference material and documentation. These are available at the usual places.

The GPU timing material is extracted from several sources but based mainly on NVidia documentation available with the CUDA SDK installation.