Objectives

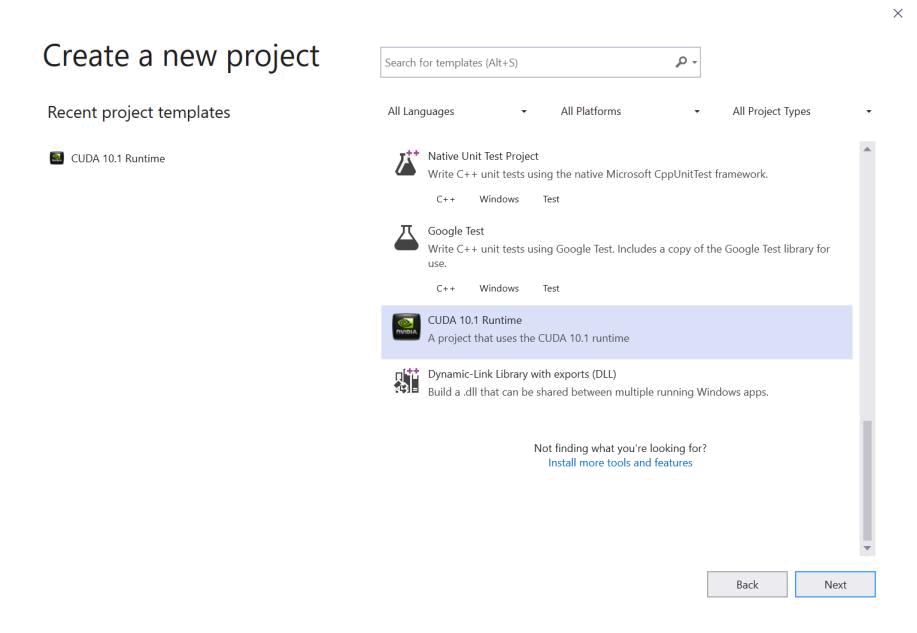
1. To introduce CUDA programming model in Visual Studio 2019
2. To write simple CUDA programs

Introduction to GPU Programming

Several programming models exist to program a GPU. Some provide better performance, while others are easier to program. There are broadly two categories: native (writing code specifically tailored for GPUs) and annotating existing code (such that the program, or parts of it, can run on either a CPU or a GPU). CUDA and OpenCL lie in the first category, while OpenACC and OpenMP lie in the second. Keep in mind that CUDA programming model can be used to program Nvidia GPUs only, while the other programing models can be used to program any GPU, e.g., AMD GPUs. CUDA is a proprietary framework created by Nvidia, while OpenCL is open source (portable). The fact that CUDA is designed for Nvidia GPUs only is one of the reasons that CUDA is the best option regarding performance. OpenMP is perhaps the easiest way of programming a GPU, providing comparable performance to OpenCL and OpenACC.

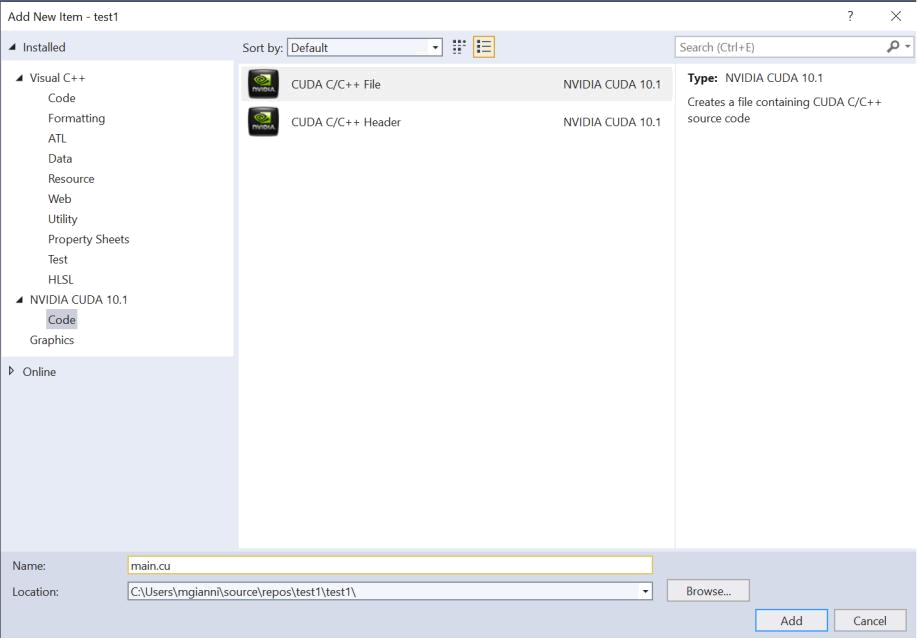
How to create a CUDA Project

1. First you need to access the SMB201 room either physically or remotely (where the Nvidia GPUs are).
2. Start Visual Studio 2019. Create a new project. Make a new CUDA 10.1 Runtime project (see Figure 1). Specify the name of your project. Then, create your project.



*Fig.1: CUDA 10.1 Runtime project*

1. The new project will contain a sample code of adding two vectors on the GPU. Delete kernel.cu from the project (if you need to develop a new program from scratch) or amend it appropriately.
2. Right click the project in the Solution Explorer and add a new item. Choose CUDA C/C++ File as the type, and pick a name (e.g. main.cu) (see Figure 2).



*Fig.2: Add a new CUDA C/C++ File*

CUDA Documentation

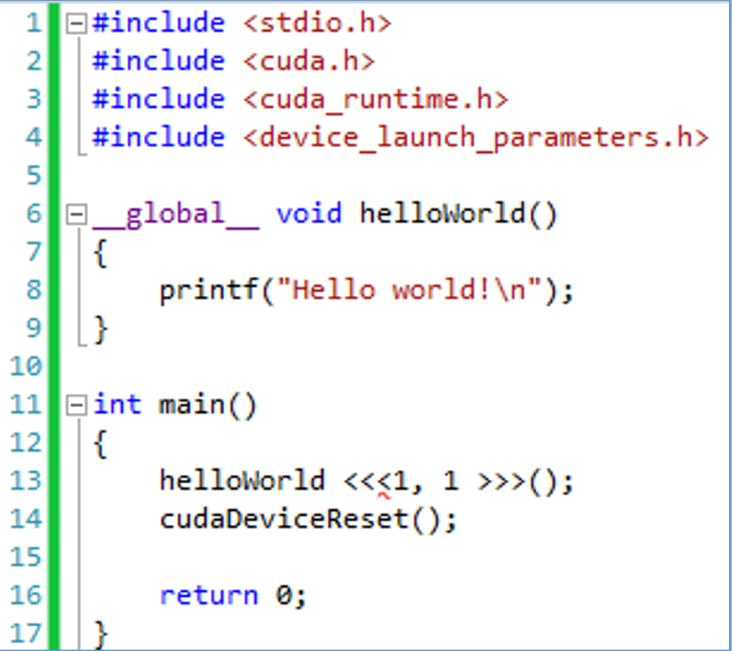
A detailed documentation is provided by CUDA in [1][2]. Use these links for any queries you might have.

First CUDA Example

Follow the steps above to create a new project in Visual Studio. The new project will contain a sample code of adding two vectors on the GPU. Delete kernel.cu from the project as we will be developing a new program from scratch. Add the following to your new ‘.cu’ source file:

* Library declarations (include those libraries in-order, using #include pragma) for cuda.h, cuda runtime.h, device launch parameters.h and stdio.h. Note: your program will compile fine even without including the CUDA libraries (nvcc automatically includes the CUDA header files), but you will get annoying red squiggles underneath any CUDA-related syntax without them all.
* A function called *helloWorld()*, which just uses printf to print a ‘Hello, world!’ message (write normal C code). Add the ‘*\_\_global\_\_*’ modifier before the function’s type; this modifier allows the function to be called from the CPU and run on the GPU.
* A normal main() C function. In main() we will write some CUDA code to launch the ‘*\_\_global\_\_ void helloWorld()*’ routine on the GPU. The syntax for calling a kernel is the same as for a normal function, but with <<<1, 1>>> between the function name and the argument brackets. The first parameter indicates the number of blocks and the second the number of threads to be used.
* Before main() ends, call the function ‘*cudaDeviceReset*’. This clears up any resources used by the current GPU device. It also causes the device to flush any output streams, so if you don’t include it you probably won’t see any text printed.

So far, the code looks like Fig.3.



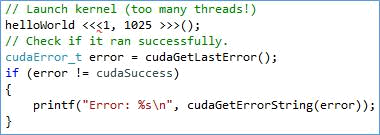
*Fig.3: Hello World Code Sample*

Compile and run the program by choosing Start Without Debugging from the DEBUG menu (or pressing Ctrl+F5). Note: if you run with debugging (F5) the console window might close immediately after the program has run. You can add a getchar() call to the end of main() to make the program wait for a key press before exiting. The output you will get will be a single ‘*Hello World!*’ message, we indicated just a single thread to run on the GPU.

CUDA extends C/C++ by allowing the programmer to define C/C++ functions, called kernels. A kernel is defined using the \_\_global\_\_ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<...>>>execution configuration syntax.

Change the number of threads (i.e. the second number in <<<1, 1>>>). The kernel will now be launched on multiple GPU cores simultaneously and you’ll see lots of hello worlds (CUDA takes care of making sure the printed output doesn’t overlap). All the threads run exactly the same code (kernel). See how high you can take the number of threads before it stops working! **The current GPU can support up to 1024 threads per block;** you can check that in this link <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications>. **So, if you specify more than 1024 threads an error will occur. In this case, you will not get an error message though**. Well-written programs use error checking to make sure nothing goes wrong and take appropriate actions when it does.

You can use the ‘cudaGetLastError’ function to get the status of the last CUDA function that was called (it returns ‘cudaSuccess’ if the last function succeeded). Passing this status into the ‘cudaGetErrorString’ function will return a nice text description of the error (see Fig.4). Let’s add some error checking to our kernel call and use 1025 threads; a message is printed now.



*Fig.4. How to get the status of the last CUDA function that was called*

To use more than 1024 threads, we must use multiple blocks of threads. Fortunately, this is easy: change the launch parameters to <<<2, 1024>>> ; now there are 2048 threads in total. You can try increasing the number of blocks to find the maximum limit, but on my PC at least a value that’s too high crashes the display driver rather than giving a nice error! We’ll see a less reckless way to figure out the maximum number of blocks later in the module. Try to change the number of blocks too, e.g., use ‘<<<2, 6>>>’. ‘<<<2, 6>>>’ will create 2 blocks of threads and each block will contain 6 threads; 12 threads in total.

Providing the number of blocks and threads as above in numbers, is not good practice. Instead, use Dim3 CUDA special data type.

***uint3/dim3***: dim3 and uint3 are CUDA defined structures (special CUDA datatype) of unsigned integers x,y,z. Dim3 can take one, two or three unsigned integer arguments:

*struct uint3 {x; y; z;};*

*struct dim3 {x; y; z;}*

***Example****:*

*dim3 blocks1D( 2 );*

*dim3 blocks1D( 2, 3 );*

*dim3 blocks1D( 2, 3, 5 );*

dim3 has 3 components .x, .y, .z each initialized to 1.

So, instead of using ‘<<<2, 6>>>’ to launch 2 blocks of 6 threads each, we can use dim3 datatype as follows

*dim3 blocks(2, 1, 1); //grid size and dimensions, it consists of blocks*

*dim3 threads(6, 1, 1); //block size and dimensions, it consists of threads*

*routine<<<blocks, threads>>>();*

Just like a block of threads consists of multiple threads, a grid consists of multiple blocks.

**Build-in predefined variables:**

*dim3 gridDim :* are built-in variables that return the “grid dimension” (i.e., the number of blocks in a grid in the x-axis, y-axis, and z-axis). Accessed using *gridDim.x, gridDim.y, gridDim.z*

*dim3 blockDim :* are built-in variables that return the “block dimension” (i.e., the number of threads in a block in the x-axis, y-axis, and z-axis). Accessed using *blockDim.x, blockDim.y, blockDim.z*

*uint3 blockIdx :* are built-in variables that return the block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code. Accessed using *blockIdx.x, blockIdx.y, blockIdx.z*

*uint3 threadIdx :* are built-in variables that return the thread ID in the x-axis, y-axis, and z-axis of the thread that is being executed by this stream processor in this particular block. Accessed using *threadIdx.x, threadIdx.y, threadIdx.z*

So, you can express your collection of blocks, and your collection of threads within a block, as a 1D array, a 2D array or a 3D array.

In the kernel code (GPU routine), you can find the current block’s position in the grid using the variables blockIdx.x (to get its index in the grid) and gridDim.x (to get the total number of blocks). Modify the printf so that it also includes information about the block position. This is done using the following command; so far, the blocks and threads are 1d and thus we need to print the .x dimension.

*printf(“\n Hello from thread %d and block %d”, threadIdx.x, blockIdx.x);*

In many cases it’s useful to know the global index of a thread. This index should be 0 for the 1st thread in the 1st block, and ‘*total\_num\_threads*’ for the last thread in the last block, where ‘*total\_num\_threads* ‘ is the total number of threads minus 1. In this case, the global index is given by

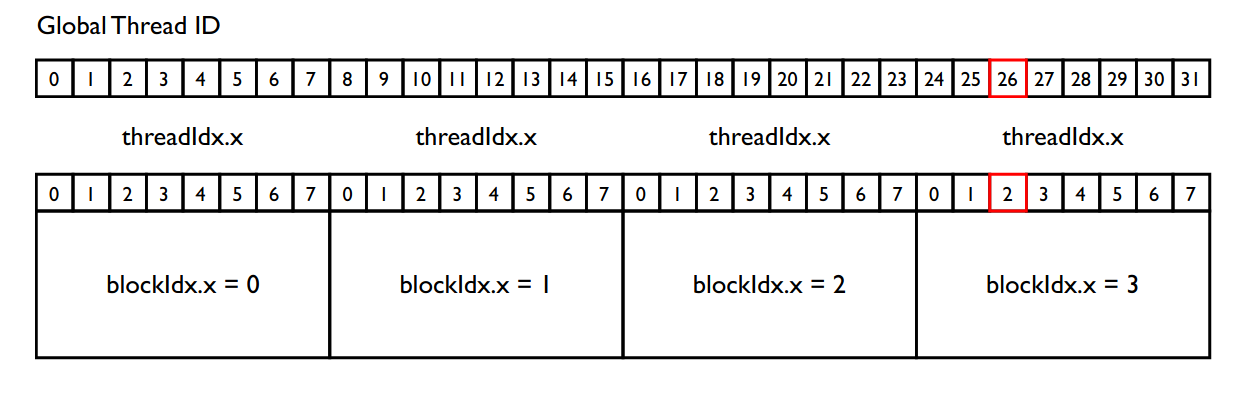
*Int global\_index = blockIdx.x \* blockDim.x + threadIdx.x;*

Assuming that the blocks and threads are defined below, the global index is shown in Fig.5.

*dim3 blocks(4, 1, 1); //grid size and dimensions, consists of blocks*

*dim3 threads(8, 1, 1); //block size and dimensions, consists of threads*

*routine<<<blocks, threads>>>();*



*Fig.5. global thread ID*

In Fig.5, the thread with ID #26 belongs to the 4th block.

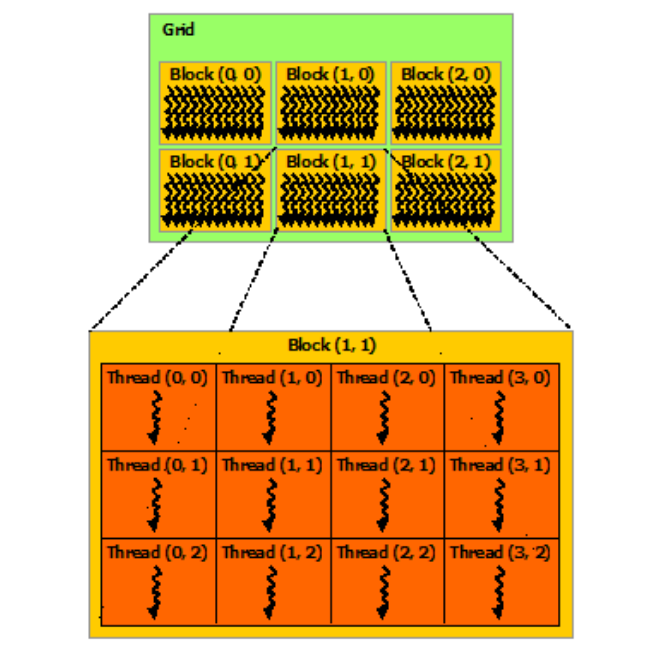
**Thread hierarchy (**the text below is taken from the CUDA documentation page [1]**)**

For convenience, threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume [1].

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.

However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated by Figure 4. The number of thread blocks in a grid is usually dictated by the size of the data being processed, which typically exceeds the number of processors in the system.



*Fig.6. Grid of thread blocks*

**2nd CUDA example – sinf()**

**Task1.** Download the ‘sinf.cu’ program. Study the program and make sure you understand what it does. This program calculates the sinf() function of an array of elements.

The arrays can be allocated either dynamically or static. In this example the arrays are allocated using malloc well-known function. The arrays are dynamically allocated into the CPU memory. These arrays cannot be accessed by the GPU. Thus, we must allocate an equal amount of GPU memory and copy the arrays from the CPU’s DDR to the GPU’s DDR.

It is good practice to name the device arrays appropriately, using the device word or the ‘d’ character.

To allocate memory on the GPU you can use the cudaMalloc function. This takes two arguments: a pointer to a pointer that should receive the memory address of the allocated memory (this will be a memory address in GPU memory), and the number of bytes to be allocated. The dynamically allocated memory must be de-allocated in the end using free() and cudaFree() functions.

Before launching the kernel, we need to copy the input data from the host array (the one stored in CPU’s DDR) into the device array (allocated on the GPU) (Fig.7). To do so, we can use the cudaMemcpy function, which takes four arguments:

* A pointer to the destination location (which can be on the host or the device).
* A pointer to the source location.
* The number of bytes to copy.
* A value indicating the direction of the transfer. This can be cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice or cudaMemcpyHostToHost. The type of the source and destination locations must correspond to the direction of the transfer e.g. for cudaMemcpyHostToDevice the source pointer should be in main memory (the host) and the desti-nation address should be in GPU memory (the device).

After the kernel has ended, the results must be stored into the CPU’s memory (Fig.9). We can use cudaMemcpy again for this, but the direction of the transfer is the opposite.

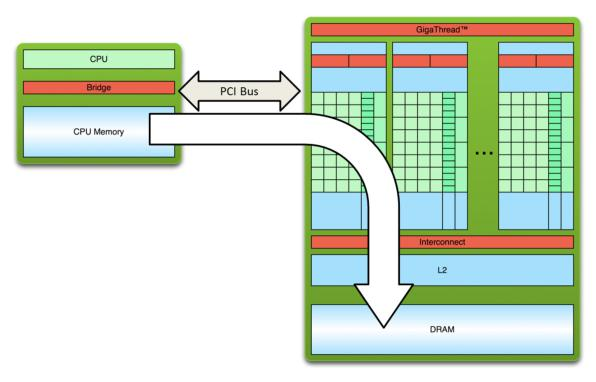
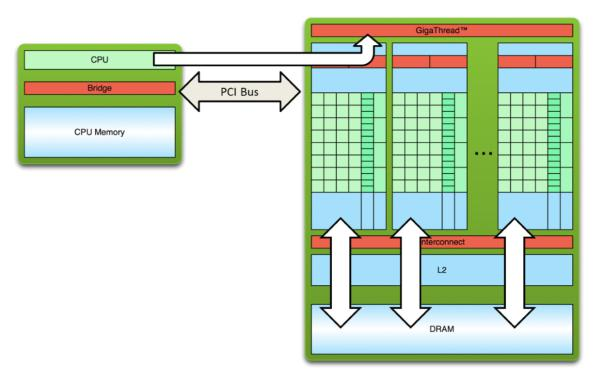
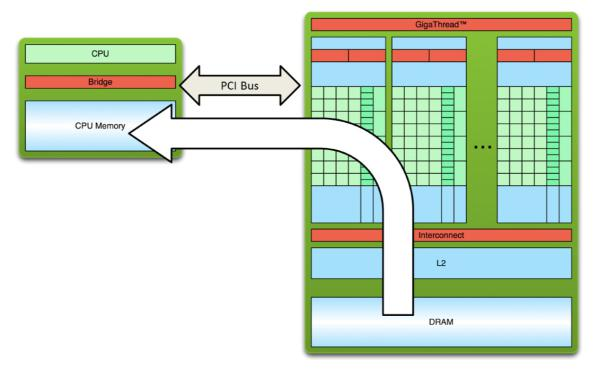


Fig.7. Copying data from the CPU’s DDR to the GPU’s DDR memory



*Fig.8. Loading and executing the kernel. The data are loaded using the GPU’s memory hierarchy*



*Fig.9. Copying data from the GPU’s DDR to the CPU’s DDR memory*

Host and device memory are distinct entities. Device pointers point to the GPU memory. These pointers may be passed to and from the host code. Moreover, they may not be dereferenced from host code. Similarly, host pointers point to CPU memory. This means that they may be passed to and from device code and that they may not be dereferenced from the device code. CUDA provides an API for dealing with the device memory (cudaMalloc(), cudaFree(), cudaMemcpy()).

Let’s study the sin\_parallel() routine. This routine is the kernel and will run on the GPU. All the GPU threads will run exactly the same code (single instruction multiple data). The routine takes as input the pointers of the two arrays in the GPU’s memory (an array pointer is the memory address of its first element). **Each thread must know what to execute and therefore the global thread index is needed**. The global thread index depends on the block’s dimension and on the grid’s dimension. In this example, they are both 1d and thus the global index (g\_id) is given by ‘*blockIdx.x \* blockDim.x + threadIdx.x*’. See Fig.5 to visualize the process.

It is important to note that in this implementation, **the number of the threads must be equal to the number of the array’s elements**. This is because each thread computes the sinf() of a single element. It is good practice to add some extra code to deal with the case where these two numbers are different. We can either use an if-condition ( *if(g\_id < N) ….*) or a modulo operation (*g\_id=g\_id % N*). Thus, if the number of the threads is larger or equal to N, the program will run correctly (*g\_id=[0, N-1]*). If the number of threads is smaller than N-1, then, some array elements will not have been computed. Make sure that (*N == num.threads x num.blocks*). Use more/less threads and see the results.

This GPU program executes very fast and thus we cannot measure its execution time. We must run it multiple times and calculate the average execution time. Keep in mind that **transferring data from/to CPU/GPU memory is a significant overhead**.

Further reading

1. CUDA Documentation, available at <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model>
2. CUDA Programming guide, available at <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications>
3. Jonathan Hui blog, CUDA Tutorial, available at <https://jhui.github.io/2017/03/06/CUDA/>