

Exercise 1

This code demonstrates how to write and launch a simple CUDA kernel

```
#include <stdio.h>

__global__ void helloWorld(float f)
{
    printf("Hello thread %d, f=%f\n", threadIdx.x, f);
}

int main()
{
    helloWorld<<<1, 10>>>(1.2345f);
    cudaDeviceReset();
    return 0;
}
```

Compile with NVCC

```
nvcc helloWorld.cu -o helloWorld -arch=sm_20
```

The `-arch` flag with the `sm_20`, which stands for Streaming Multiprocessor and the number indicates the features supported by the architecture.

A GPU is built around an array of Streaming Multiprocessors (SMs) (see Chapter 4 in [CUDA Programming Guide](#) for more details). A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

In order to allow for architectural evolution, NVIDIA GPUs are released in different generations. New generations introduce major improvements in functionality and/or chip architecture, while GPU models within the same generation show minor configuration differences that „moderately“ affect functionality, performance, or both. ~ from [NVIDIA NVCC Manual](#)

You should get this output in the terminal:

```
Hello thread 0, f=1.234500
Hello thread 1, f=1.234500
Hello thread 2, f=1.234500
Hello thread 3, f=1.234500
Hello thread 4, f=1.234500
Hello thread 5, f=1.234500
Hello thread 6, f=1.234500
Hello thread 7, f=1.234500
Hello thread 8, f=1.234500
```

```
Hello thread 9, f=1.234500
```

In this particular case, you are launching only one block with 10 threads. You can try to play around with the kernel invocation parameters and try to change them to different values. If you for example call the `helloWorld` kernel like this

```
helloWorld<<<2, 5>>>(1.2345f);
```

then you will launch the same number of threads but only 5 threads per each of the two blocks as opposed to 10 threads per single block.

If you also modify the *printf* call in the kernel to be something like this

```
printf("Hello block %i running thread %i, f=%f\n", blockIdx.x, threadIdx.x, f);
```

then you should get this output:

```
Hello block 0 running thread 0, f=1.234500
Hello block 0 running thread 1, f=1.234500
Hello block 0 running thread 2, f=1.234500
Hello block 0 running thread 3, f=1.234500
Hello block 0 running thread 4, f=1.234500
Hello block 1 running thread 0, f=1.234500
Hello block 1 running thread 1, f=1.234500
Hello block 1 running thread 2, f=1.234500
Hello block 1 running thread 3, f=1.234500
Hello block 1 running thread 4, f=1.234500
```

You might have noticed that the thread ID is only unique within its block. More often than not, you will be writing programs that will launch tens, hundreds or even thousands of blocks and therefore you will need a way to get the unique thread ID. The way to do it in the case of one dimensional grid of blocks it with one dimensional threads is as follows:

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
```

If you add that code at the beginning of the kernel and then substitute the *threadIdx.x* in the *printf* call with *idx*, you should get the following output where each thread has a unique index.

```
Hello block 0 running thread 0, f=1.234500
Hello block 0 running thread 1, f=1.234500
Hello block 0 running thread 2, f=1.234500
Hello block 0 running thread 3, f=1.234500
Hello block 0 running thread 4, f=1.234500
Hello block 1 running thread 5, f=1.234500
Hello block 1 running thread 6, f=1.234500
Hello block 1 running thread 7, f=1.234500
Hello block 1 running thread 8, f=1.234500
```

```
Hello block 1 running thread 9, f=1.234500
```

Sometimes you might need to run multi-dimensional grid of blocks and/or multi-dimensional thread blocks. The way to do this is by using *dim3*, which is an integer vector type based on *uint3* that is used to specify dimensions. When defining a variable of type *dim3*, any component left unspecified is initialized to 1.

Try writing the following code before the kernel call:

```
dim3 grid(2, 2, 1);  
dim3 block(2, 2, 1);
```

Now you can pass the *grid* and *block* variables directly as the kernel launch parameters.

```
helloWorld<<<grid, block>>>(1.2345f);
```

Since you are now launching 2D grid of blocks each with 2D threads, you need to change the calculation of the unique thread and block index. See this code

```
int blockId = blockIdx.x + blockIdx.y * gridDim.x;  
int threadId = blockId * blockDim.y * blockDim.x + threadIdx.x + threadIdx.y *  
blockDim.x;
```

try to understand it and then update the *printf* function to the following:

```
printf("Hello block %i (x %i, y %i) running thread %i (x %i, y %i), f=%f\n",  
blockId, blockIdx.x, blockIdx.y, threadId, threadIdx.x, threadIdx.y, f);
```

When you now launch the kernel you should see something similar to this:

```
Hello block 0 (x 0, y 0) running thread 0 (x 0, y 0), f=1.234500  
Hello block 0 (x 0, y 0) running thread 1 (x 1, y 0), f=1.234500  
Hello block 0 (x 0, y 0) running thread 2 (x 0, y 1), f=1.234500  
Hello block 0 (x 0, y 0) running thread 3 (x 1, y 1), f=1.234500  
Hello block 1 (x 1, y 0) running thread 4 (x 0, y 0), f=1.234500  
Hello block 1 (x 1, y 0) running thread 5 (x 1, y 0), f=1.234500  
Hello block 1 (x 1, y 0) running thread 6 (x 0, y 1), f=1.234500  
Hello block 1 (x 1, y 0) running thread 7 (x 1, y 1), f=1.234500  
Hello block 3 (x 1, y 1) running thread 12 (x 0, y 0), f=1.234500  
Hello block 3 (x 1, y 1) running thread 13 (x 1, y 0), f=1.234500  
Hello block 3 (x 1, y 1) running thread 14 (x 0, y 1), f=1.234500  
Hello block 3 (x 1, y 1) running thread 15 (x 1, y 1), f=1.234500  
Hello block 2 (x 0, y 1) running thread 8 (x 0, y 0), f=1.234500  
Hello block 2 (x 0, y 1) running thread 9 (x 1, y 0), f=1.234500  
Hello block 2 (x 0, y 1) running thread 10 (x 0, y 1), f=1.234500  
Hello block 2 (x 0, y 1) running thread 11 (x 1, y 1), f=1.234500
```

Here we have the unique block and thread IDs together with the x,y block and thread indices.

If you get bored try [this tutorial](#) from Stanford.

Just for the reference, this is the final version of the helloWorld source code:

```
#include <stdio.h>

__global__ void helloWorld(float f)
{
    int blockId = blockIdx.x + blockIdx.y * gridDim.x;
    int threadId = blockId * blockDim.y * blockDim.x + threadIdx.x + threadIdx.y *
blockDim.x;

    printf("Hello block %i (x %i, y %i) running thread %i (x %i, y %i), f=%f\n",
blockId, blockIdx.x, blockIdx.y, threadId, threadIdx.x, threadIdx.y, f);
}

int main()
{
    dim3 grid(2, 2, 1);
    dim3 block(2, 2, 1);

    helloWorld<<<grid, block>>>(1.2345f);
    cudaDeviceReset();

    return 0;
}
```