# TP09
# Introduction to CUDA

Robin Faury

03/04/20

**Abstract**

In this practical work, we will see how to convert a simple iterative process into a paralell process using CUDA language. This is a quick and simple introduction du CUDA. The aim is just to understand the syntax and how to store and access buffers in the GPU memory.

## 1 Introduction

First, generate the Visual Studio solution, compile it and run the cuda program.
Open a terminal on your working folder and use this cmake command:

```
cmake -G "Visual Studio 15 Win64" ..
```

If everything is going well, you can compile and run the cuda_lesson_helloworld executable.
Using MSVC compiler for Visual Studio IDE is not the only option. You can also use the CUDA C++ compiler via the terminal.

```
nvcc main.cu -o cuda_lesson_helloworld
```

The cuda file extention is .cu.

## 2 The host

Remember the host is the CPU part of the process. So you have to declare and fill input buffers before the GPU copy. First we have to create a buffer of float for the input buffer and another with the same size for the output result.

## 3 The device

The device is related to GPU operations. On this section, we will see how to send buffers to the VRAM and declare and run kernels.

### 3.1 The memory

Before doing the copy to the CUDA's Unified Memory from the host we must allocate the space. The function cudaMallocManaged takes as input a pointer and the data size. Remember that the bufferSize value will be spread into warps (a block of 32 threads). It should be a multiple of 32.

```
char* buffer;
cudaMallocManaged(&buffer, nbElement*sizeof(float));
```

As usual, you must delete your pointer at the end of you program. For this, use the cuda function:

```
cudaFree(buffer);
```

### 3.2 The kernel

Declaring a kernel is easy with CUDA. You just have to write your code like a C++ function and add the __global__ keyword before. This indicates the CUDA C++ compiler that your function is device code.

```
__global__
void Manderbrot(int nbElement, float *inputBuffer, float *outputBuffer)
{
        for (unsigned int i = 0u; i < nbElement; ++i)
                outputBuffer[i] = inputBuffer[i] + 1.f;
}
```

For the moment the function is designed for one thread. We can run this function using this syntaxe:

```
// We use one thread per block and one block per grid
Manderbrot<<<1, 1>>>(nbElement, buffer, outBuffer);
```

Let's increment the number of threads per block. Something like 256 may a good value for big buffers. CUDA allows us to know the index of the thread through the threadIdx.x value.

```
__global__
void Manderbrot(int nbElement, float *inputBuffer, float *outputBuffer)
{
        int index = threadIdx.x;
        for (unsigned int i = index; i < nbElement; ++i)
                outputBuffer[i] = inputBuffer[i] + 1.f;
}

// ...

// We use 256 threads per block and one block per grid
Manderbrot<<<1, 256>>>(nbElement, buffer, outBuffer);
```

In the same way we can increase the number of blocks per grid to dispatch the computation on every streaming multiprocessor.

```
__global__
void Manderbrot(int nbElement, float *inputBuffer, float *outputBuffer)
{
        int stride = blockDim.x * gridDim.x; // stride = 256 * blockPerGrid
        int index = blockIdx.x * blockDim.x + threadIdx.x;
        outputBuffer[index] = inputBuffer[index] + 1.f;
}

// ...

int blockPerGrid = (nbElement + 256 - 1) / 256;
Manderbrot<<<blockPerGrid, 256>>>(nbElement, buffer, outBuffer);
```

We can notice in this case that we run nbElement threads so we can remove the for loop. Before reading the output buffer you have to wait the kernel execution. For that you can use the cudaDeviceSyncronize function.

# 4 Mandelbrot

We can see threadIdx.x has a 'x' struct member. That means we can organize our threads per blocks (even blocks per grid) in a non linear way using the CUDA object dim3.

```
__global__
void Manderbrot(int nbElement, float *inputBuffer, float *outputBuffer)
{
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int j = blockIdx.y * blockDim.y + threadIdx.y;
        // ...
}

// ...

dim3 threadsPerBlock(16, 16);
Manderbrot<<<1, threadsPerBlock>>>(nbElement, buffer, outBuffer);
```