Introduction to CUDA

This lab introduces you to CUDA computing with a few small exercises. We have deliberately avoided larger amounts of code for this lab.

1 Trying out CUDA

To get started, we will use the `simple.cu` example introduced during the lecture, the simplest CUDA example. All it does is to assign every element in an array with its index.

1.1 Setup

The NVIDIA graphic cards we are going to use are located on dedicated virtual environment. They are accessible only through `ssh`. Each student is assigned a specific VM. Please ask for the number of you VM.

To connect to the virtual machine, you mush use

```
ssh -K login@vmgpuxxx.ensimag.fr
```

where `login` is your current login and `xxx` is the server number given to you.

You need to modify the path variable in order for the compiler to be found

```
export PATH=$PATH:/usr/local/cuda/bin
```

1.2 Compile and run simple.cu

You can compile it with this simple command-line :

```
nvcc simple.cu -o simple
```

For lab purposes, this command line works nicely. For completion, we may also use the more elaborate syntax :

```
nvcc simple.cu -lcudart -o simple
```

or

```
nvcc simple.cu -L /usr/local/cuda/lib64 -lcudart -o simple
```

The program is executed simply with

```
./simple
```

(Feel free to build a makefile.)

If it works properly, it should output the numbers 0 to 15 as floating-point numbers.

As it stands, the amount of computation is ridiculously small. We will soon address problems where performance is meaningful to measure. But first, let's modify the code to get used to the format.

- How many cores will `simple.cu` use, max, as written ?
- How many streaming multiprocessors will `simple.cu` use, max, as written ?

1.3 Modifying `simple.cu`

Allocate an array of data and use as input. Calculate the square root of every element. Inspect the output to verify it is correct.
To upload data to the GP-GPU, you must use `cudaMemcpy()` call with `cudaMemcpyHostToDevice`.
Note : If you increase the data size here, please check out the comments in section 2.

- the calculated square root identical to what the CPU calculates ? Should we assume that this is always the case ?

2 Performance and block size

Now let us work on a larger dataset and make more computations. We will make fairly trivial item-by-item computations but experiment with the number of blocks and threads.

2.1 Array computation from C++ to CUDA

Here is a simple `C++` program that takes two arrays (matrices) and adds them component-wise. The program is available in `matrix_cpu.cpp`
It can be compiled with

```
g++ matrix_cpu.cpp -o matrix_cpu
```

and run with

```
./matrix_cpu
```

Write a CUDA program that performs the same thing, in parallel ! Start with a grid size of $(1, 1)$ and a block size of (N, N). Then try a bigger grid, with more blocks.
We must calculate an index from the thread and block numbers. That can look like this (in 1D) :

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

- How do you calculate the index in the array, using 2-dimensional blocks ?

2.2 Larger data set and timing with CUDA Events

In order to measure execution time for your kernels you should use CUDA Events. A CUDA event variable "myEvent" is declared as

```
cudaEvent_t myEvent;
```

It must be initialized with

```
cudaEventCreate(&myEvent);
```

You insert it in the CUDA stream with

```
cudaEventRecord(myEvent, 0);
```

The 0 is the stream number, where 0 is the default stream. To make sure an event have finished (received its value), call

```
cudaEventSynchronize(myEvent);
```

Important ! You must use `cudaEventSynchronize` before taking time measurements, or you don't know if the computation has ended !

Finally, you get the time between two events with

```
cudaEventElapsedTime(&theTime, myEvent, laterEvent);
```

where `theTime` is a float.

For timing CPU code, you can use the following code :

```
#include <chrono>
using namespace std::chrono;

auto start = high_resolution_clock::now();
```

Note that in CUDA, large arrays are best allocated in C++ style : `float *c = new float[N*N];`

- Vary N. You should be able to run at least 1024x1024 items.
- Vary the block size (and consequently the grid size).
- Write a kernel for initialization too.
- What happens if you use too many threads per block ?

Use this statement to check errors at runtime :

```
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess)
std::cout«cudaGetErrorString(err)«std::endl;
```

- Is this related to the last question ? Why ?

Note 1

You can get misleading output if you don't clear your buffers. Old data from earlier computations will remain in the memory.

Note 2

Also note that timings can be made with or without data transfers. In a serious computations, the data transfers certainly are important, but for our small examples, it is more relevant to take the time without data transfers, or both with and without.

> **Note 3**
> When you increase the data size, there are (at least) two problems that you will run into :
> The maximum number of threads per block is limited, typically 1024 on current GPUs. If you use more, your computation will be incomplete or unpredictable. The limitations of the platform you work on can (and should in a serious application) be inspected by a "device query", using `cudaGetDeviceProperties()`. This includes number of threads (again, 512), number of blocks in a grid (high !) and amount of shared memory (typically 16k).

1. At what data size is the GPU faster than the CPU ?
2. What block size seems like a good choice ? Compared to what ?
3. Write down your data size, block size and timing data for the best GPU performance you can get.

2.3 Coalescing

In this part, we will make a simple experiment to inspect the impact of memory coalescing, or the lack of it.

You calculated a formula for accessing the array items based on thread and block indices. Probably, a change in X resulted in one step in the array. That will make data aligned and you get good coalescing. Swap the X and Y calculations so a change in Y results in one step in the array. Measure the computation time.

> **Note 4**
> The results have varied a lot in this exercise. However, lack of difference can be caused by erroneous computation, so make sure that the output is correct.

- How much performance did you lose by making data accesses non-coalesced ?

3 Mandelbrot

This exercise aims at writing a parallel algorithm to generates a visual representation of the Mandelbrot set. The algorithm we use in this part to compute such a set computes each pixel independently. i

Since this would allow to process all pixels in one parallel step, it is called an embarassingly parallel algorithm. However, a processor has typically much less computing units (cores) available than pixels. Thus, the entire picture to be computed must be shared among all cores and each core can sequentially compute its part, while other cores also compute theirs at the same time.

Although the algorithm is embarassingly parallel, one must take great care when distributing the work, as bad work partitioning can produce parts harder to compute than others. If all cores receive one part and one part is significantly longer to compute, then one core will takes more time to run, delaying the completion of the overall algorithm and leaving other unused. In contrast, if all cores work for the same amount of time on their part, then available cores are exploited optimally and the runtime of the algorithm is further reduced.

We provide a sequential code for computing the mandelbrot set.

You can compile the mandelbrot program using the following command

```
g++ mandelbrot.cpp readppm.cpp -o mandelbrot
```

- Put in a timer in the code. Experiment with the iteration depth. Also try with double precision. (Note : On some CUDA installations you may need to specify `-arch=sm_30` or higher on the command-line.)

In this exercise, you may need one more CUDA event call : `cudaEventDestroy()`, to clean up after a CUDA event and avoid memory leaks.

- What were the main changes in order to make the Mandelbrot run in CUDA ?
- How many blocks and threads did you use ?
- When you use the Complex class, what modifier did you have to use on the methods ?
- What performance did you get ? How does that compare to the CPU solution ?
- What performance did you get with float vs double precision ?
- Is load balancing an issue here ? Discuss.