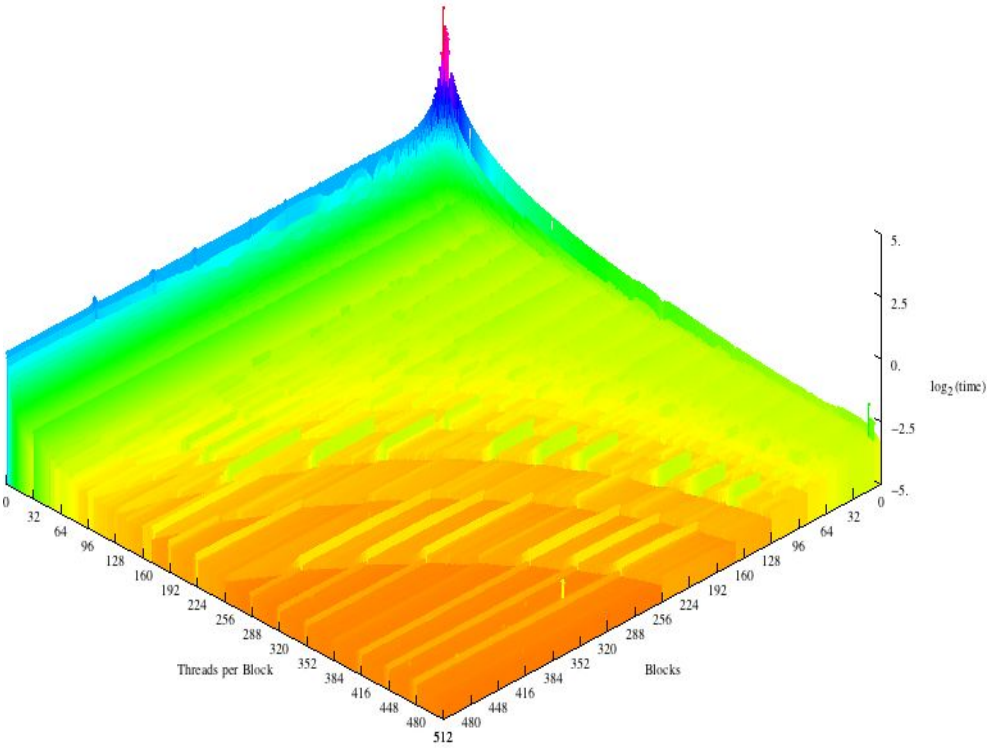
Introduction to CUDA and OpenCL

Lab 2 Kacper Kapuściak

Processing grid

The first thing that we were asked during the laboratories was to get to know our hardware so I've run Nvidia supplied utility called "device query". It provides information about the graphics card in a nice, clean way. Graphics card that we are using - Nvidia GeForce RTX 2060 - has **30** multiprocessors with **64** CUDA cores each and that gives us 1920 CUDA cores in total. But that wasn't the only information the program outputted. It also told that the maximum number of threads per multiprocessor and per block is **1024** and that the total amount of global memory is 5904 MBytes (**6190989312 bytes**). We'll need that information later on.

Each streaming multiprocessor can process one block at the same time. Blocks can be as big as 1024 threads. In order to avoid wasting resources (that is cores waiting for other to finish), we'd like our grid to be multiplies of 30. Each multiprocessor has 64 cores so I think it would the best to create grid with rows of 30 blocks and every block with 64 threads each. I'm not sure if cores and threads are mapped one to one or cores can spawn many many threads and process them in paralel... My logic is demolished by [this article](#) that I've found; someone did an experiment about GPU efficiency. He run a script for every grid size between 1 and 512 blocks and every number of threads per block between 1 and 512 which produced 262,144 data points. He stated that: "When in doubt use more threads not less, creating threads is inexpensive.". Graph from his experiment:



The the z axis is the log2(time) that

means the lower the better. By his logic the more threads the better.

Maximum number of elements

In the first section we've learned that the total amount of memory is 6190989312 bytes. But I guess it's impossible to allocate that much memory due to fact the memory is used not only for stucture allocation but our programme needs to run and the card itself etc. so probably the memory accesible for the end-user is smaller. One float number takes 4 bytes of memory and we are allocating 3 float vectors. Dividing 6190989312 by 3 and by 4 we receive **515915776**. That's the theoritical number of elements that we could possibly allocate. To have some staring point I've tried running our example programme with both 500000000 and 515915776 elements and only the former worked. So I know that the maximum number of elements is somewhere between these two numbers. In our example code I've commented out every line that isn't connected to memory allocation (i.e. kernel execution) and wrapped it with while loop. I implemented some kind of "divide and conquer" algorithm to find out the maximum number of elements:

1. Start from 500000000 and increase number of elements by arbitrarily taken step of 2000000 elements
2. Increase number by that step until allocating memory is unsuccessful
3. Go step back (decrease number by that too big step) and divide that step by half. Continue incrementing with halved step.
4. Stop incrementing when step is smaller than 2

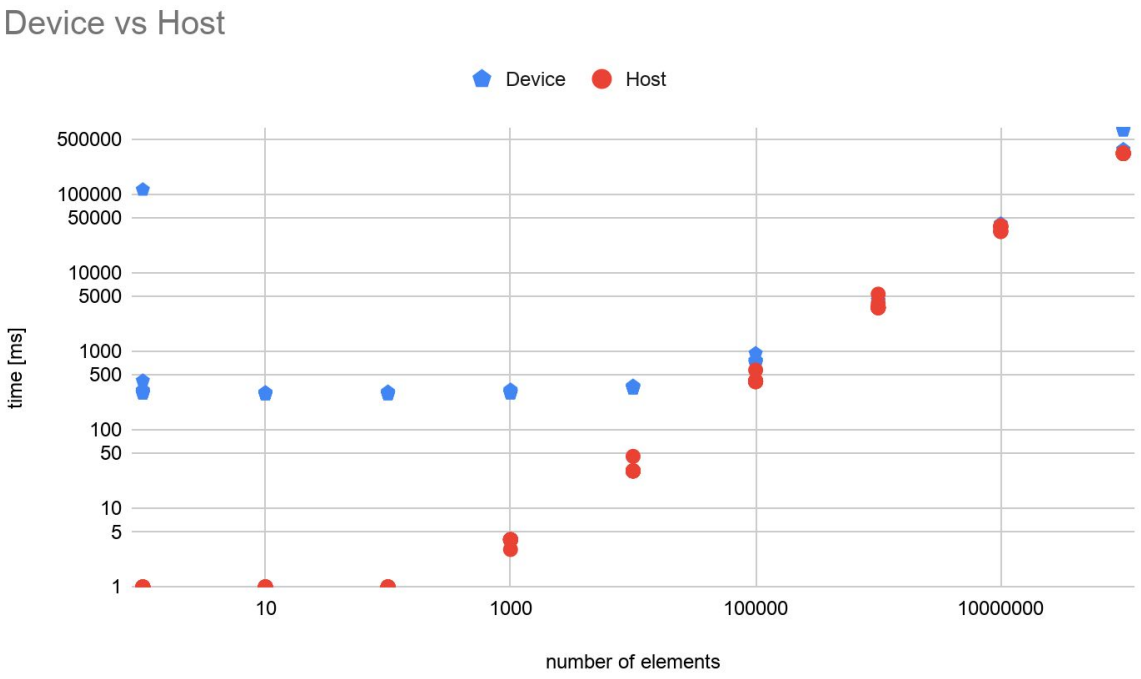
Last few iterations:

```
numElements: 502267944 | step: 61 | i: 21
Number too big! Going step back and decreasing it by half.
numElements: 502267913 | step: 30 | i: 22
Number too big! Going step back and decreasing it by half.
numElements: 502267898 | step: 15 | i: 23
numElements: 502267913 | step: 15 | i: 24
Number too big! Going step back and decreasing it by half.
numElements: 502267905 | step: 7 | i: 25
Number too big! Going step back and decreasing it by half.
Done
```

I had to double check that manually but it was indeed true that the biggest number of elements that I could allocate was: **502267904.** My naive algorythm with only 25 (26) iterations found the maximum number of elements.

Measurements

In this part of the labs we had to make some measurements and comparisons between parallel computing and computing on single-core CPU. The results are very interesting:



For small numbers making calculations on GPU is completely useless. It's way faster on CPU because copying data from host to device and back is very resource heavy. For this simple problem - vector addition - CPU is a clear winner for me. Also there is a weird GPU property that the first computing is a few times slower than it is normally. Fetching data to and from GPU takes ~300ms so it's not worth computation saving for small computations.

Kacper Kapuściak