# 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 others 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 a 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 parallel... My logic is demolished by this article that I've found; someone did an experiment about GPU efficiency. He runs 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.". His experiment is summed up as seen in chart 1.

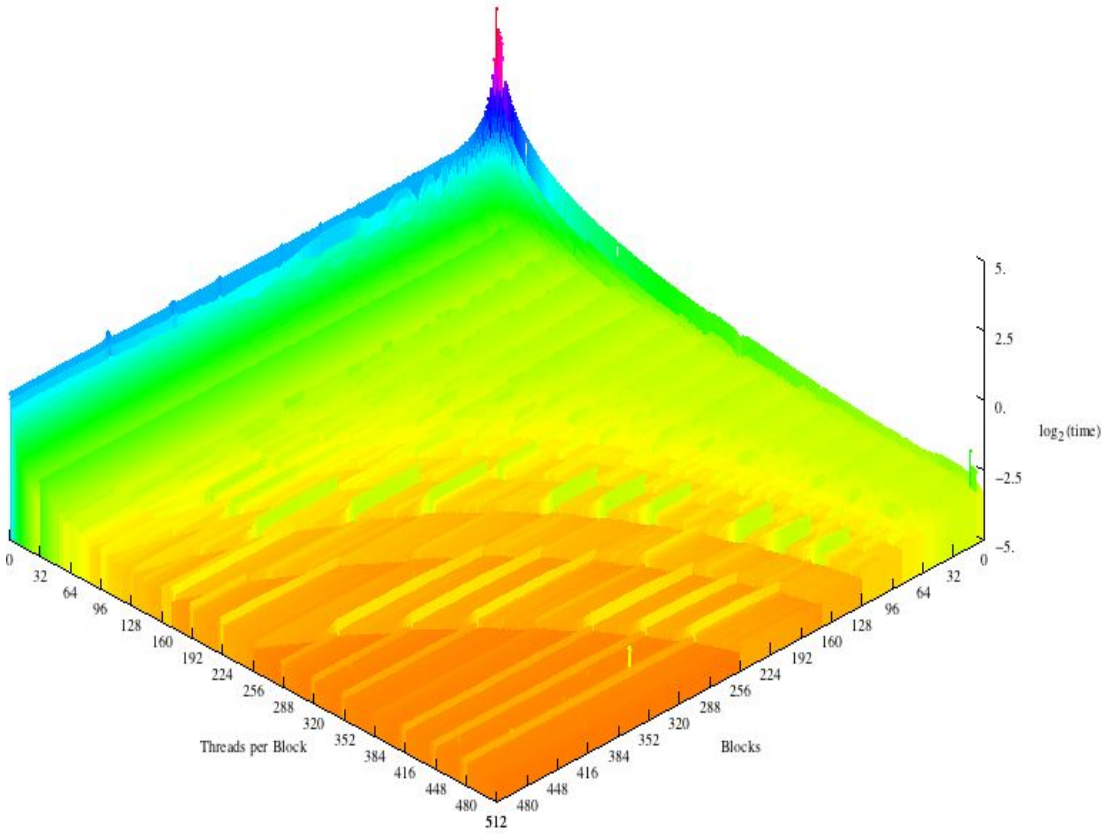


*Chart 1.* The dependency of a number of threads per block, number of blocks, and time. Source: http://selkie.macalester.edu/csinparallel/modules/CUDAArchitecture/build/html/2-Findings/Findings.html

The z-axis is the log2(time) which means the lower 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 the fact the memory is used not only for structure allocation but our program needs to run and the card itself etc. so probably the memory accessible 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 theoretical number of elements that we could possibly allocate. To have some staring point I've tried running our example program with both 500000000 and 515915776 elements and only the former work. 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 the number of elements by arbitrarily taken the 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 the 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 algorithm 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 a single-core CPU. At first, I've used the default processing grid provided in the exercise that is calculated dynamically.

```
blocksPerGrid=(numberOfElements+threadsPerBlock-1)/threadsPerBlock
```

threadsPerBlock equals **64** and numberOfElements changes every ten iterations.

The results are quite interesting:

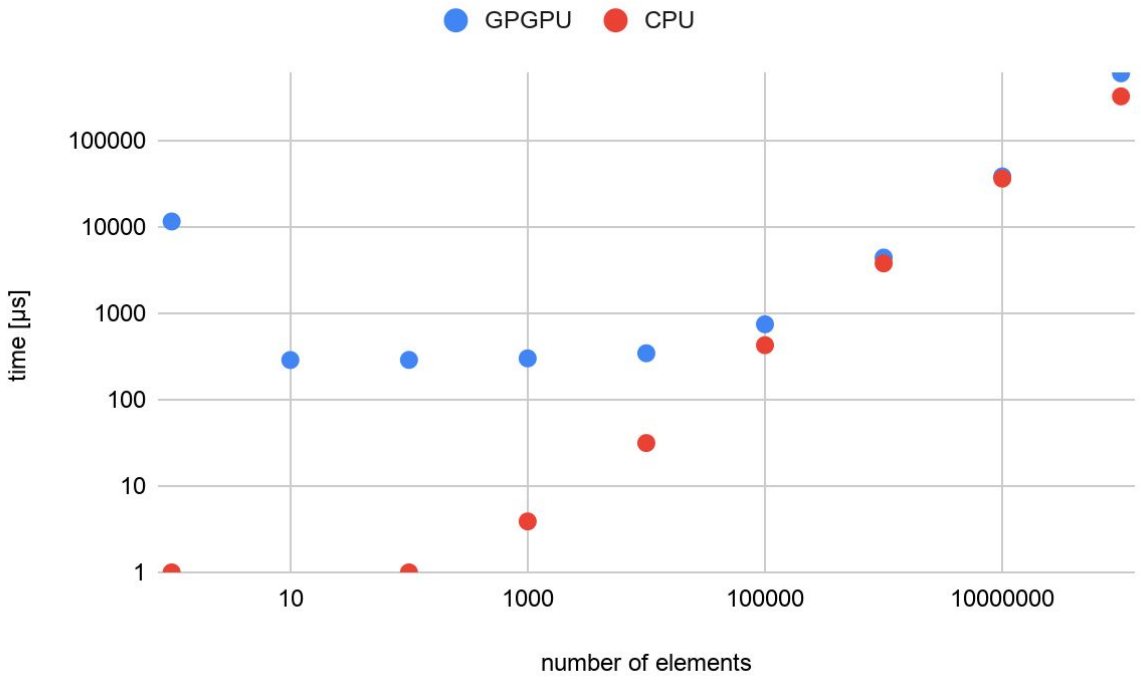


*Chart 2.* The dependency of the number of elements and time for vector addition on CPU and GPGPU for 64 threads per block and a variable number of blocks.

Whan can be immediately seen is a spike in performance for the first measurement. There is a thing in GPGPU that the first calculation is always very slow so I've removed the error from the next charts. are slower than CPU because fetching data to GPGPU takes a considerable amount of time. Let's try running this experiment for different processing grid.

In the next attempt, I've tried to use hardcoded PG with 512 threads and 32 blocks.

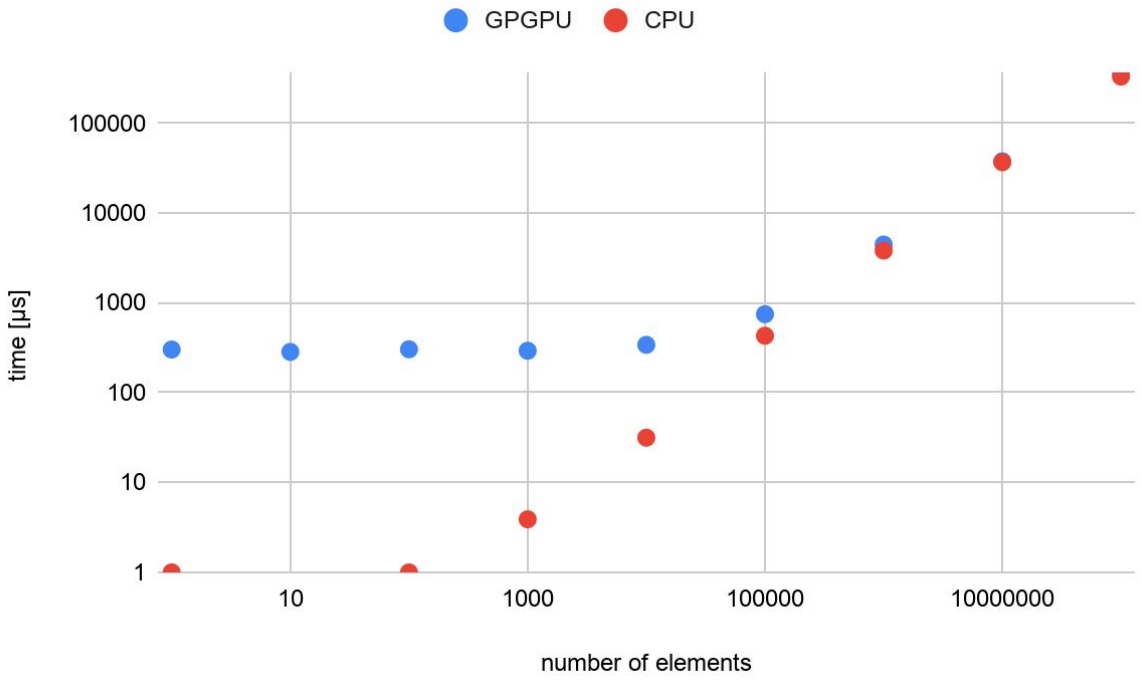


*Chart 3.* The dependency of the number of elements and time for vector addition on CPU and GPGPU for a grid 512 threads per block and 32 blocks.

The difference between Chart 2 and Chart 3 cannot be seen on the chart but measurements on 512 x 32 for $1 * 10^8$ elements differ by a lot: $611773\ [\mu s] - 360268,3\ [\mu s] = 251504,7\ [\mu s]$ on average. That's 1.7x faster for the biggest measurement in terms of the number of elements.

For the last try, I've come up with a 128x128 processing grid.

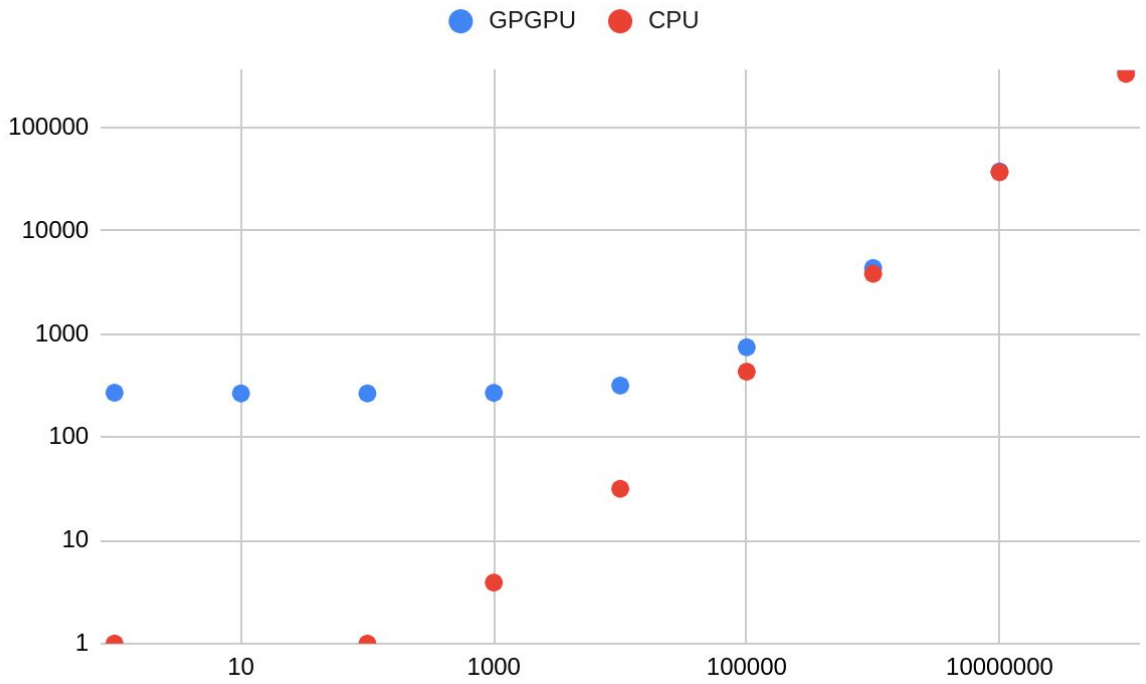


*Chart 4.* The dependency of the number of elements and time for vector addition on CPU and GPGPU for a grid 128 threads per block and 128 blocks.

And this time I've seen no improvement with execution time in any way. I should probably see for improvements in other parts of code but that will be discussed on many future laboratories :)

### Conclusion

For small vectors making calculations on GPU doesn't make much sense. It's way faster on CPU because copying data from a host to device and back is a very resource-heavy process. For this simple problem - vector addition - CPU is a clear winner. 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 sets of data. I see a lot of room for improvement in memory allocating and fetching.

Kacper Kapuściak