Introduction to CUDA and OpenCL

Łukasz Gut, Grzegorz Litarowicz

Lab 02, 12.03.2020

Introduction

In this report, we present ideas for optimizing processing grid definition, discover and analyze the maximum number of elements that we can send to the GPGPU. Furthermore, we analyze the execution time of parallel vector addition and sequential vector addition, both on host and device.

All of the reasoning applies to Nvidia RTX 2060.

Processing Grid definition optimization

To optimize the processing grid definition we have to maximize the streaming multiprocessor warp occupancy.

Nvidia RTX 2060 offers us 30 Streaming Multiprocessors, which means that we would like to have $n * 30$ blocks per grid, where $n \geq 1$. It would be best to have just 30 blocks, but that is not possible because there is an upper limit of threads per block. In our case, it is 1024.

We had also discovered that Nvidia provides a CUDA Occupancy Calculator in the form of an Excel spreadsheet which comes in handy when it comes to optimizing the number of threads per block and blocks per grid.

Elements number analysis

The maximum number of elements we were able to allocate is 502 267 904.

The total amount of global memory on our GPU: 6190989312 bytes. One float takes 4 bytes and we allocate 3 vectors of them the theoretical limit would be:

Number of elements = 6190989312 / 12 = 515915776

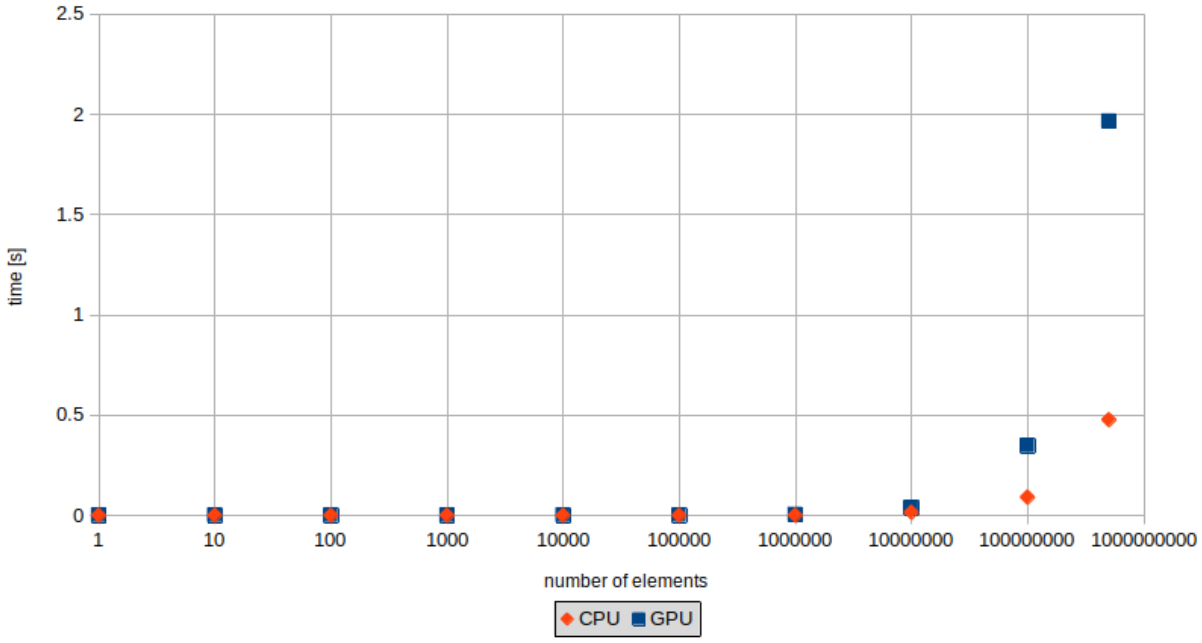
So if we would live in an ideal world that would be a number that we would be able to allocate but in reality, this number crushes the program. That's why we calculated this number in the following steps:

1. Close allocating, copying and freeing memory in for loop
2. Set i start value on 515915776
3. Try to execute allocating a number of elements = i
4. As long as the allocation is unsuccessful we decrease the number about 100 000
5. When an allocation is successful we start to increase the number about value 10 times smaller than previously, and we do that until allocation is unsuccessful
6. Then we start to decrease the number about value 10 times smaller than the value we were increasing.
7. We continue this algorithm until we start to increase our value by 1.

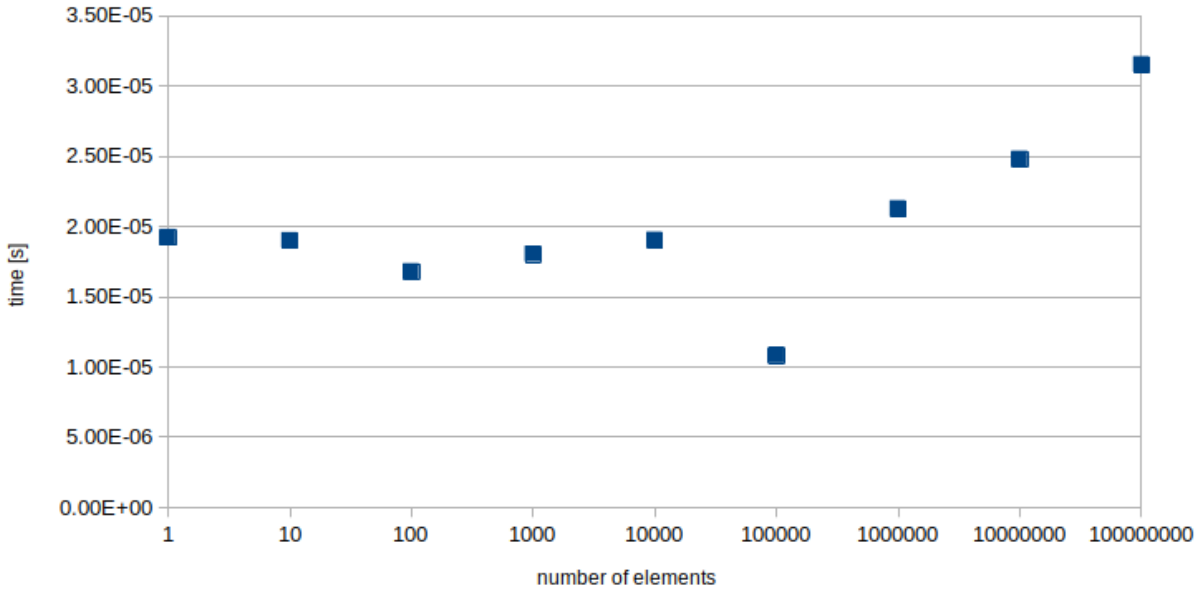
Benchmarking

To accurately measure the time we decided to use std::chrono from C++11's standard library. For measuring purposes, we implemented a simple timer class which allows us to calculate execution time at chosen granularity.

The diagram where a time of allocation, copying of data to and adding vectors is plotted against a number of elements in vector for the algorithm execution on GPU and CPU:



The diagram where a time of execution is plotted against a number of elements in vector :



Conclusion

In this report, we described an idea for optimizing the definition of Processing Grid as well as analyzed the maximum number of elements a device is able to allocate. We also did some benchmarks both on the host's and device's side.

We conclude that it is not worth doing simple vector addition calculations on the GPGPU because the cost of moving data between host and device is very large. What is more, modern CPUs come with SIMD support and can perform simple floating-point math very efficiently. The fact that we are dealing with a contiguous array is also to the CPU's advantage because it can maximally utilize the L1 cache.

We also found it surprising that the *first* data copying to the device takes much more time than the consecutive copies.