Report 1

Introduction to CUDA and OpenCL

Parallelism exposed by the CUDA Programming Model, addVec sample.

Dariusz Biela

Karol Sawicki

1. Introduction

To start learning about CUDA Programming model we had to prepare some C/C++ code. We choose to start with the addVec CUDA sample code, which allowed us to observe how to prepare simple function that could be executed on the GPU. Our observation was that we have to prepare two data structures, one to be held by CPU in RAM and one to be held by GPU in its own VRAM. At the beginning we have to allocate memory, on the CPU we could do that with well known C functions like malloc, but GPU needed its specific cudaMalloc variation. After allocation we used function named cudaMemcpy to copy vectors data to the GPU. Before execution we had to prepare processing grid, which is some kind of structure that allows GPU to effectively do calculations. At the first run we decided to run code without any changes to the grid.

2. Calculations, time and limitations

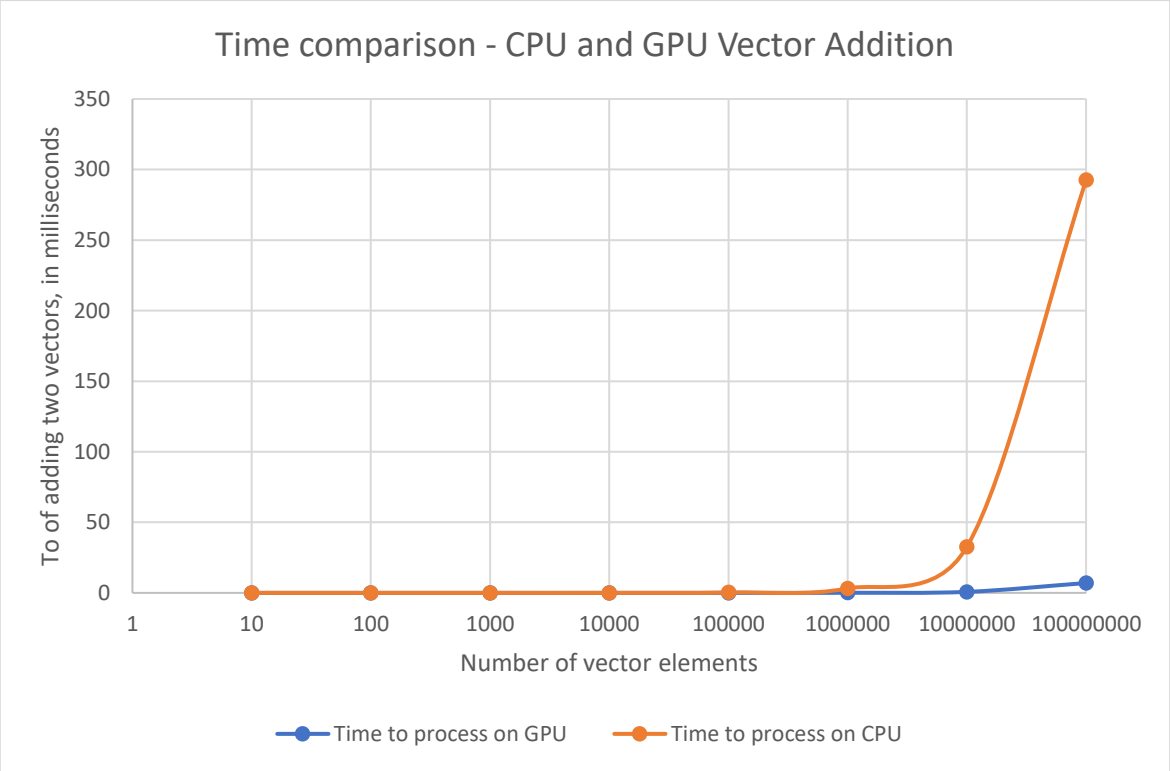
Our first task was to learn how big vectors we can add. To do this we decided to prepare a program loop that each time was trying to allocate ten times bigger vectors. The biggest vectors we could allocate had 100000000 elements. The limitation was clearly the GPU memory. It is really important, because GPU's VRAM is always the same, we cannot replace it, like the CPU's RAM. In that case we should know how to prepare data that must be sent to GPU.

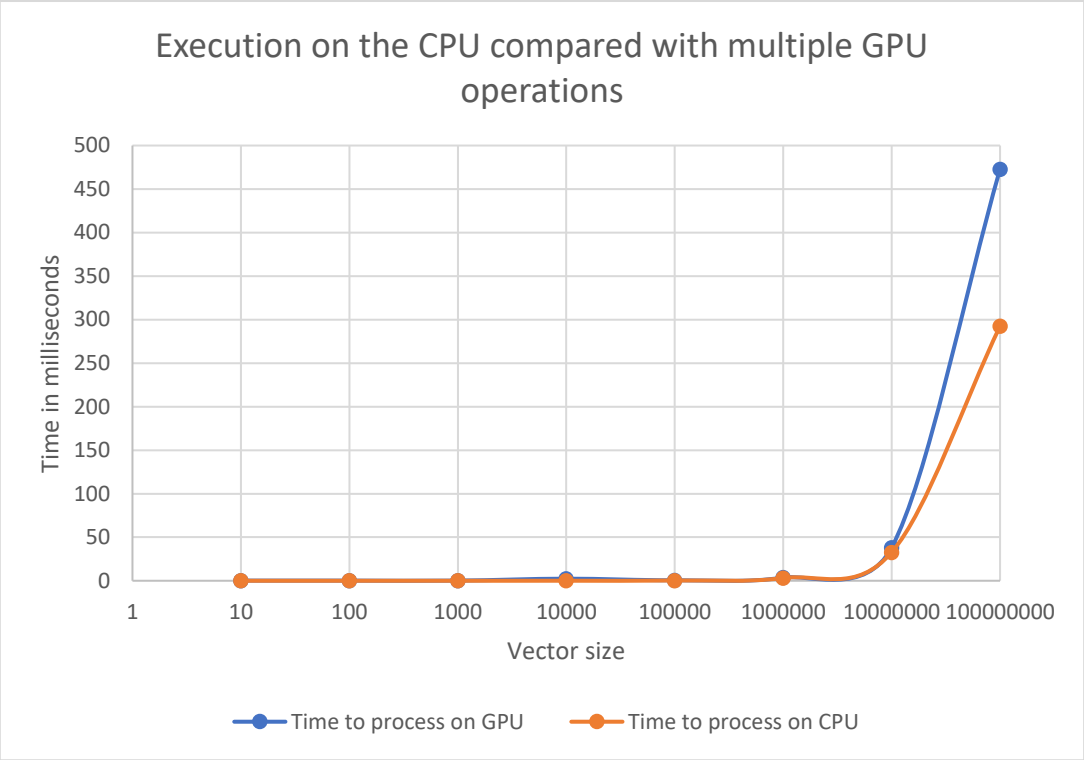
In the main loop we put the second one, which was measuring addition time, for each vector size we prepared a .txt file with execution time. After that we prepared another vector adding function, that could be executed on the CPU directly to compare GPU and CPU execution time. At the end we measured combined time of allocating device memory, execution and copying the data to the GPU and from the GPU. This measure contain some other operations too, like checking errors and displaying error messages, if they exist.

We tried to prepare a more suitable processing grid too, but the effects were a bit surprising – the best execution times we got for the default grid.

Time	Number of Blocks
314,883698	1
141,84845	2
71,257149	4
36,183777	8
18,823072	16
9,82192	32
5,240672	64
5,218208	128
5,206272	256
5,385312	512
5,684896	1024

At the end we calculated average value for each vector size and prepared two graphs.





3. Conclusions

Our first graph illustrates only the execution time, but we have to remember that GPU execution is not the only operation. Although execution itself may be really fast, copying the data to the device to execute code and to the host to get results may be too expensive. In that case we should remember that GPU is not good for small data portions. In the other hand – second graph illustrates that allocating GPU memory and copying the data is really painful for the performance. The measured time contains some other operations too – like checking errors, this is important too, because it rises values and because of it it's hard to say which method is the best for vector addition. In the case of raw performance – GPU is ideal for adding big vectors.

Default processing grid may be good for really simple operations, like vector addition, but a little bit more complicated can gain from preparing a more suitable grids. Changing the number of threads per block may help a little bit too.

We should know, that working with CUDA requires knowledge about some CUDA specific rules. Good example may be that every function prepared for GPU has to be void – it cannot return any value.

Every GPU function needs an additional mark: _global_ for the functions called from CPU and GPU or _device_ for functions that can be called only from GPU code. We should remember that those functions always are executed on the GPU. An additional key word is telling us only from which part of code we can call them.