Introduction to CUDA and OpenCL
Lab 2 report

12.03.2020
Piotr Litwin
Paweł Skalny

Introduction

During lab we were testing parallel vector adding, the performance and the processing grid's impact. We started by analysing last lab's device query to find out the number of streaming multiprocessors and available CUDA cores for each of them. Nextly, our task was to find the maximum size of input vectors. We knew the size is maximal when a bit larger vector caused a crash. With that knowlage we could check the execution time and compare them with single-threaded calculations.

Results

The first check – number of streaming multiprocessot and CUDA cores:

Device	Multiprocessors	CUDA Cores
GeForce RTX 2060	30	64
GeForce GTX 780 Ti	15	192

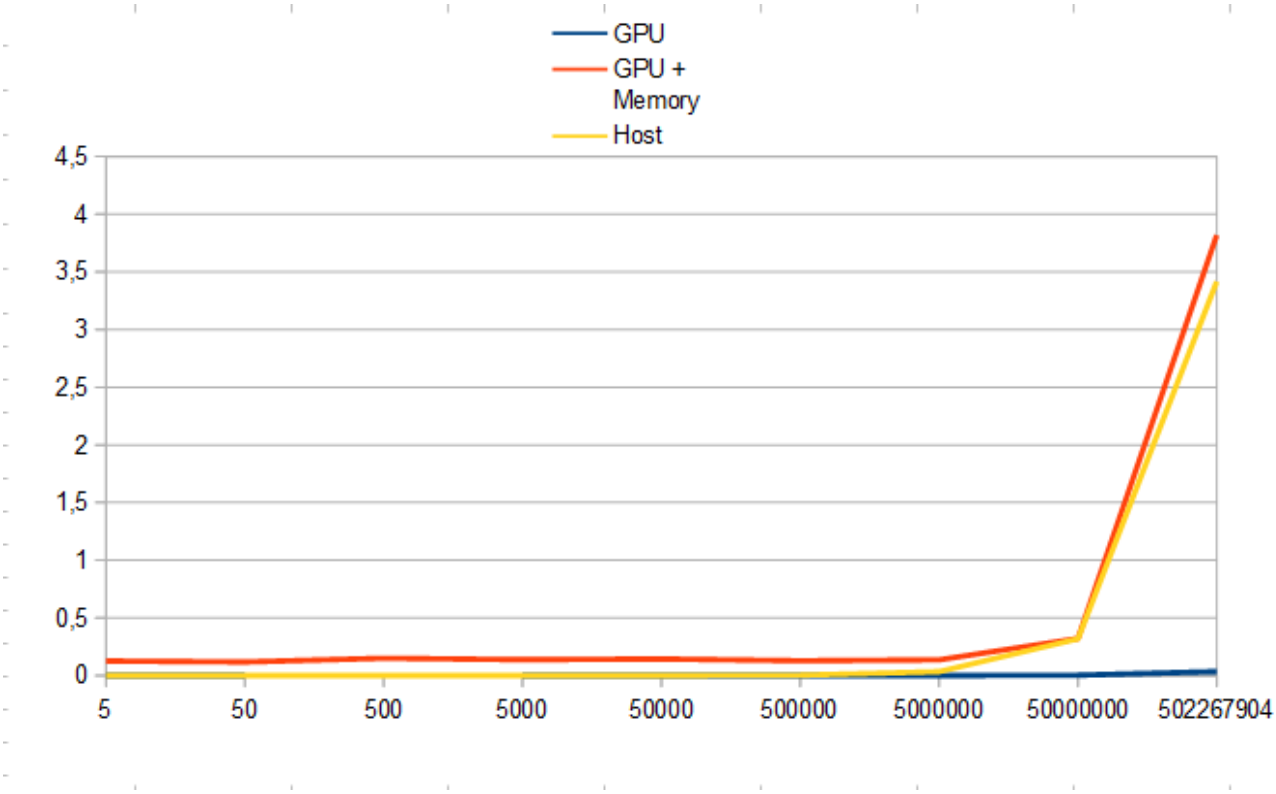
Table 1. Number of streaming multiprocessot and CUDA cores for available devices

Then with many approaches we finally found exact size of input vectors: **502267904**. The programm has crashed because there were not enought memory for device vectors. We had three vectors of float type (size * 3 * 4 bytes) what gives about 5.6GB of allocated space.

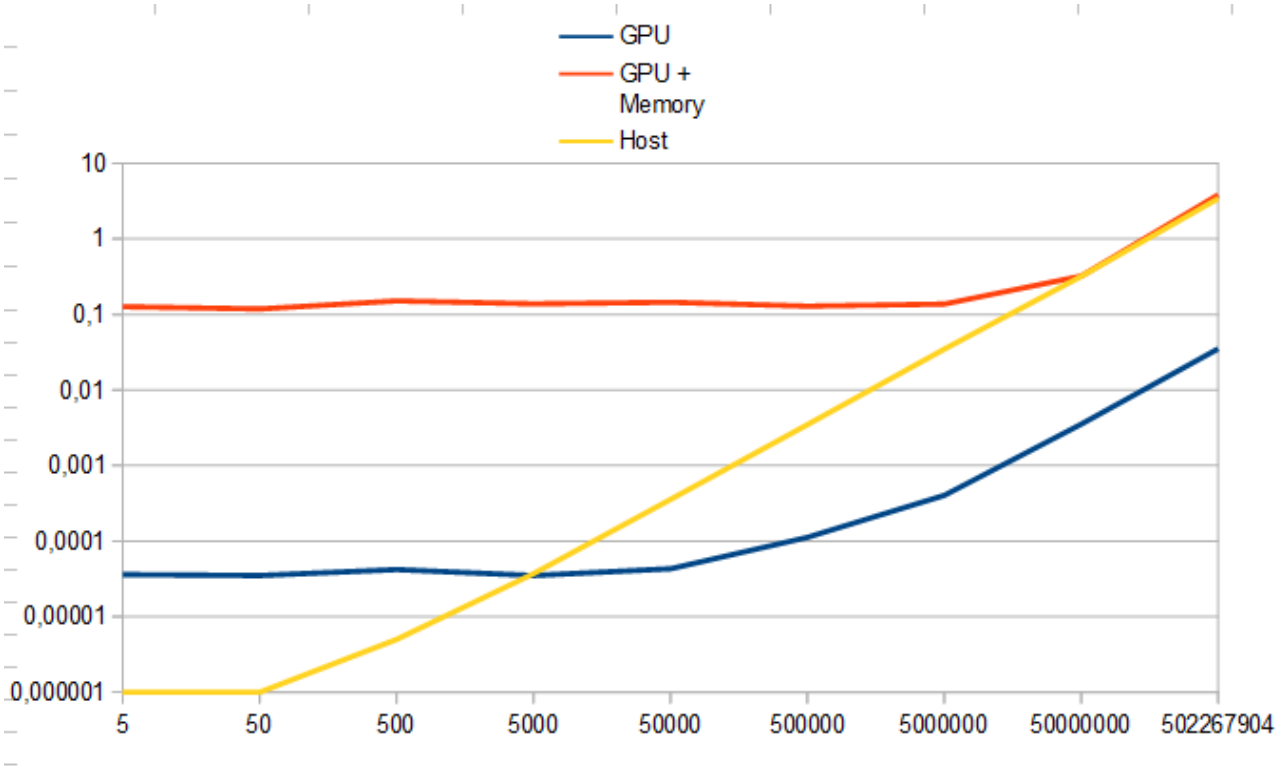
We measured time of computing for host and device side while adding vectors. We also count a device memory allocation (and all needed operations), because it is essential for move whole computing on the device side. Our checkpoints started at vectors with 5 elements and increased logarithmically, until reach the maximum size.

Vector's size	GPU Computing	GPU + Memory transfer	Host (CPU)
502267904	0,035076 s	3,818792 s	3,415964 s
50000000	0,003530 s	0,321970 s	0,319433 s
5000000	0,000403 s	0,136985 s	0,034667 s
500000	0,000112 s	0,128731 s	0,003471 s
50000	0,000043 s	0,144904 s	0,000356 s
5000	0,000035 s	0,138277 s	0,000037 s
500	0,000043 s	0,150945 s	0,000005 s
50	0,000035 s	0,118204 s	0,000001 s
5	0,000036 s	0,126805 s	0,000001 s

Table 2. Time of computing depends on vector's size - comparision.



Pic. 1. Graph from Table 1.



Pic. 2. Graph from Table 1. in logarithmic scale

At the end, we've checked provided example of thread indexing within processing grid to have a look at implementation.

Conclusions

Firstly, we can guess, that threads in blocks of grid should match amount of CUDA Cores and we should use the least bloks we need to solve the problem. It will avoid running kernel on unused threads (when we already obtain the result). Fortunly, default grid's setup runs 256 threads, which is multiple of 64 and works realy well.
While performance test, we observed kernel function is pretty fast. The linear time dependency on size starts between 50 and 500 milion elements. It is the point where we begin to lose properties of parallel computing. On the other hand, allocating GPU memory takes comparatively long time (part of it is constant). Overall, for simply functions it may be slower then host side computing considering memory handling.