Report

Communication between host and device
Comparing efficiency of GPU computing for different sizes of data and grid configurations

Paweł Lipiór, Michał Żoczek

October 22, 2019

1 Communication between host and device

1.1 Obtaining information about device

Before writing any code utilizing CUDA it is necessary to obtain information about device that is available for use. This can be done using one of the CUDA API functions - `cudaGetDeviceProperties`. Its usage is fairly simple: function reads all the data involving chosen device and writes it to a `cudaDeviceProp`, structure defined as a part of CUDA library. The structure contains variables for all device properties including number of CUDA cores, available memory of different types, clock rates, etc.
Another way to access this data is to use `cuDeviceGetAttribute` function, which returns only specific property's value.

1.2 Analyzing CUDA program structure using VectorAdd example

In most programs making use of CUDA platform we can distinguish four important steps:

1. Initializing data:
   Allocating memory on host, preparing data. In our example it consist of preparing three arrays representing vectors and filling them with randomized numbers.

2. Transferring data to device:
   Allocating memory on device and copying data for computation. While host memory allocation use standard C function `malloc`, reserving memory on device is accomplished using function from CUDA library - `cudaMalloc`. Here we make three vectors of the same length as in first step and copy our data there using `cudaMemcpy` function.

3. GPU computing:
   Invoking kernel function with specified grid configuration and copying result to host memory. Massive parallel computing is done in this step. In VectorAdd program the kernel is responsible for adding two vectors.

4. Freeing memory:
   Deallocating memory. Similar to allocation it is done using two different functions - `free` for host and `cudaFree` for device.

2 Comparing efficiency of GPU computing for different sizes of data and grid configurations

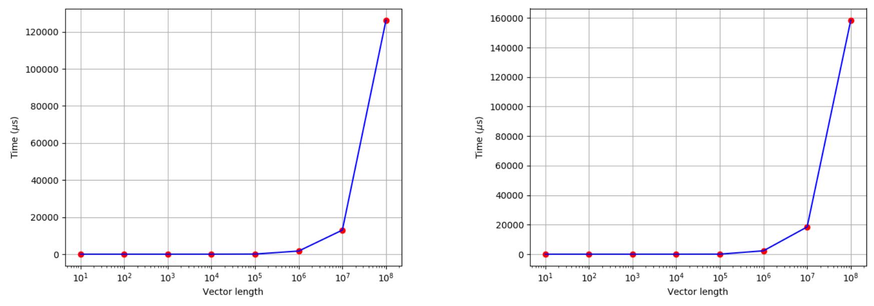
2.1 Performance of vectorAdd program as a function of data size

Operating on big data structures could be a challenge for traditional computing utilities. With CUDA technology it can be done faster and more efficiently. We decided to check how size of data structure affects program execution time using a NVidia's sample program `vectorAdd`. Thanks to tool called `nvprof` it was possible to check execution time of consecutive steps performed by the program: copying data from host to device with `cudaMemcpy`, `vectorAdd` kernel execution and copying data back to host. Program was ran five times to get average times of each operation. Results of those tests are shown in table 1.

avg vectorAdd [us]	**avg memcpy HtoD [us]**	**avg memcpy DtoH [us]**	**Vector elements**	**Blocks**	**Threads**
3.1104	1.8112	0.9600	10	1	256
3.0656	1.8624	0.9524	100	1	256
2.6178	1.9888	1.0664	1000	4	256
3.7352	8.6656	3.7696	10000	40	256
18.5728	72.3720	31.2760	100000	391	256
164.2500	1705.4000	2297.2200	1000000	3907	256
1618.1000	13011.6000	18475.8000	10000000	39063	256
16141.4000	126316.0000	158588.0000	100000000	390625	256

Table 1: Comparison of functions' execution time depending on data size.

Based on collected results semi-log plots were created (Fig. 1-3).



(a) Copying data from host do device. (b) Copying data from device to host.

Figure 1: Execution time of functions for transferring data between host and device.

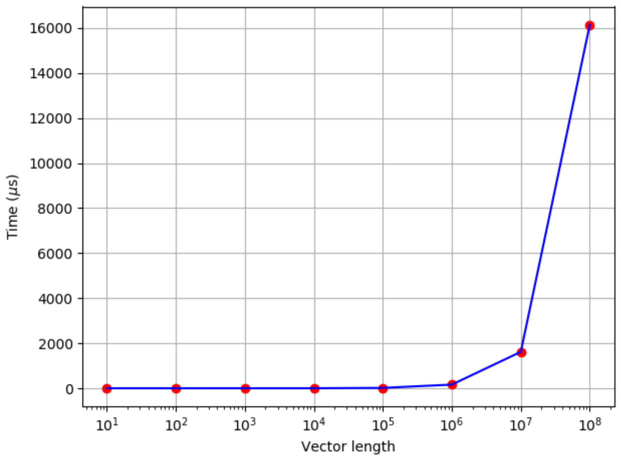


Figure 2: Kernel execution time.

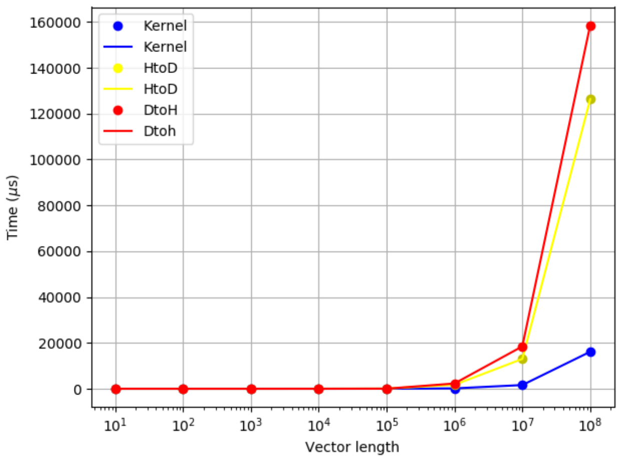


Figure 3: Comparison of execution time between CUDA functions.

Fig. 1 shows execution time of functions for data transfer. As the plot's shape is similar to that of a logarithmic function we can conclude that execution time is proportional to data size. Fig. 2 shows kernel execution time and judging from its shape we can draw the same conclusion as in case of `cudaMemcpy` functions. Fig. 3 is the most interesting, as it shows comparison between all three functions. We can see that for quite small vector size the execution time is basically the same. But the larger the size of data structure the greater difference between functions. The execution time of `memcpy` rise faster than that of a kernel. Thus we can conclude that the bottleneck of GPU computing is not the cores' performance but mostly transferring data between host and device.

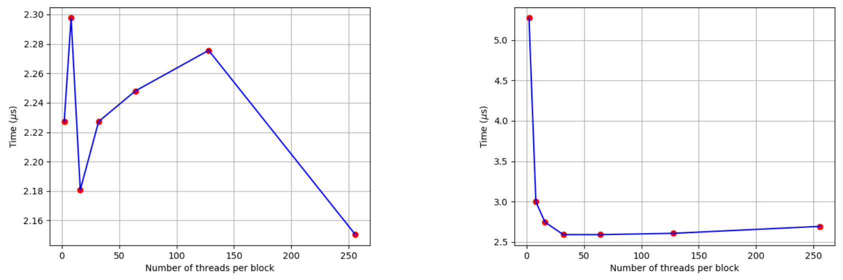
2.2 Grid configuration's impact on performance

Another important aspect of working with CUDA technology is grid configuration. It is important to use optimal settings, otherwise it will negatively impact performance. To find best configuration we manipulated two grid parameters: `blocksPerGrid` and `threadsPerBlock`. Again, measurements were done using `vectorAdd`'s functions. In addition `checkIndex` kernel was added to visually check grid configuration. As in previous tests `nvprof` was used to measure execution time of `vectorAdd` kernel and `cudaMemcpy` functions. Data size was set to fixed 1000 elements. Test results were recorded in Table 1.

avg vectorAdd [us]	**avg memcpy HtoD [us]**	**avg memcpy DtoH [us]**	**Threads**
5.2736	2.2272	1.4096	2
2.9984	2.2976	1.1136	8
2.7456	2.1808	1.1840	16
2.5920	2.2272	1.1504	32
2.5920	2.2480	1.2000	64
2.6084	2.2756	1.1440	128
2.6944	2.1504	1.1600	256

Table 2: Comparison of functions' execution time depending on number of threads per block.

Based on collected results plots were created (Fig. 4-6).



(a) Copying data from host do device. (b) Copying data from device to host.

Figure 4: Execution time of functions for transferring data between host and device.

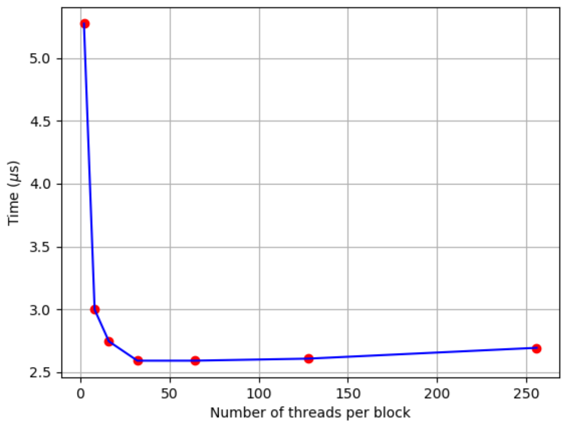


Figure 5: Kernel execution time.

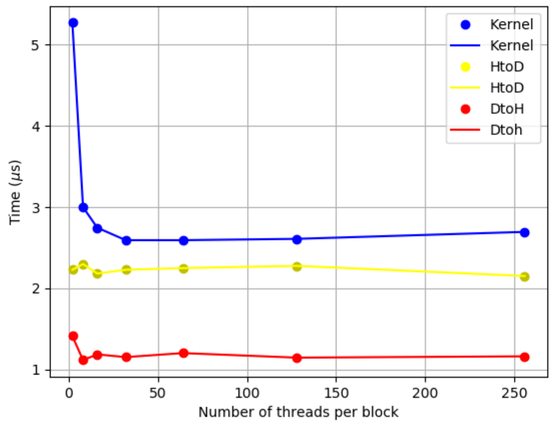


Figure 6: Comparison of execution time between CUDA functions.

From those charts it is possible to come to conclusion that choosing the most optimal number of threads is by no means an easy task. Not only every function behaves differently with the same parameters, but also simple increase in number of threads does not always produce better results (as seen in, for example, kernel function where after getting to optimal number of 32 every increase in threads causes loss in performance). Fig. 6 shows also that for every function optimal number of threads is different. In this case, as kernel has longest execution time it is probably best to focus on its performance. Unfortunately to achieve the best results it is necessary to analyze every problem separately, as there is no universal solution for maximizing performance.