Report

Lab 2

24.03.2020

by Mateusz Bałuch & Weronika Miszczak

1. The processing grid optimization

The first thing we are going to talk about is something called “processing grid”. When we are calling kernel function, we must specify dimensions of this grid. It consists of the number of blocks and threads per block (in some more complicated cases this may have 2 or 3 dimensions). In our first example we have this numbers calculated depending on the amount of data to process. Sometimes this way of describing grid is good enough, but if we have access to information about streaming multiprocessors and the amount of cuda cores which each of SM have, we can use it to reach the maximum performance of our GPU.

Example code to gather this information we learnt from another example from cuda samples:

```
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);
int mp_count = deviceProp.multiProcessorCount;
int cuda_cores_per_mp = _ConvertSMVer2Cores(deviceProp.major,
                                              deviceProp.minor);
```

And now we have information about the number of streaming multiprocessors in “mp_count” variable, and the amount of cuda cores in each multiprocessor in “cuda_cores_per_mp” variable.

So we can create a grid with mp_count blocks and each block would provide cuda_cores_per_mp threads to execute our kernel function.

2. The cuda limitations

Our next task was to found the limitations in provided code that adds two vectors using cuda cores. So we are increasing the numElements variable to crash the program. Using the divide and conquer method we found that program is still working with 5.02 * 10^8 numbers in each array and crashes with 5.03 * 10^8 elements. Crash message was “Failed to allocate device vector C (error code out of memory)!”.

We are using float numbers in each array, so each number are stored using 4 bytes in memory. We have 3 arrays with 5.02* 10^8 elements. To calculate the size of used memory we must multiply these 3 numbers:

size = 4 * 5.02 * 10^8 * 4byte

what gives us a result of about 7,48GB of memory.

3. Understanding processing grid layout

The last thing we have done was looking at example code that print out variables describing current thread position. The program is creating 1D grid with 2 blocks. Each block is also 1D structure and have 3 threads. We can observe that cuda runs each thread in parallel, because order of print outs isn’t always in order.

We want to access our data in consistent way, so we need to identify which part of provided array belongs to current thread. This can be done using variables provided by cuda (“threadIdx”, “blockIdx”, “blockDim”, “gridDim”).

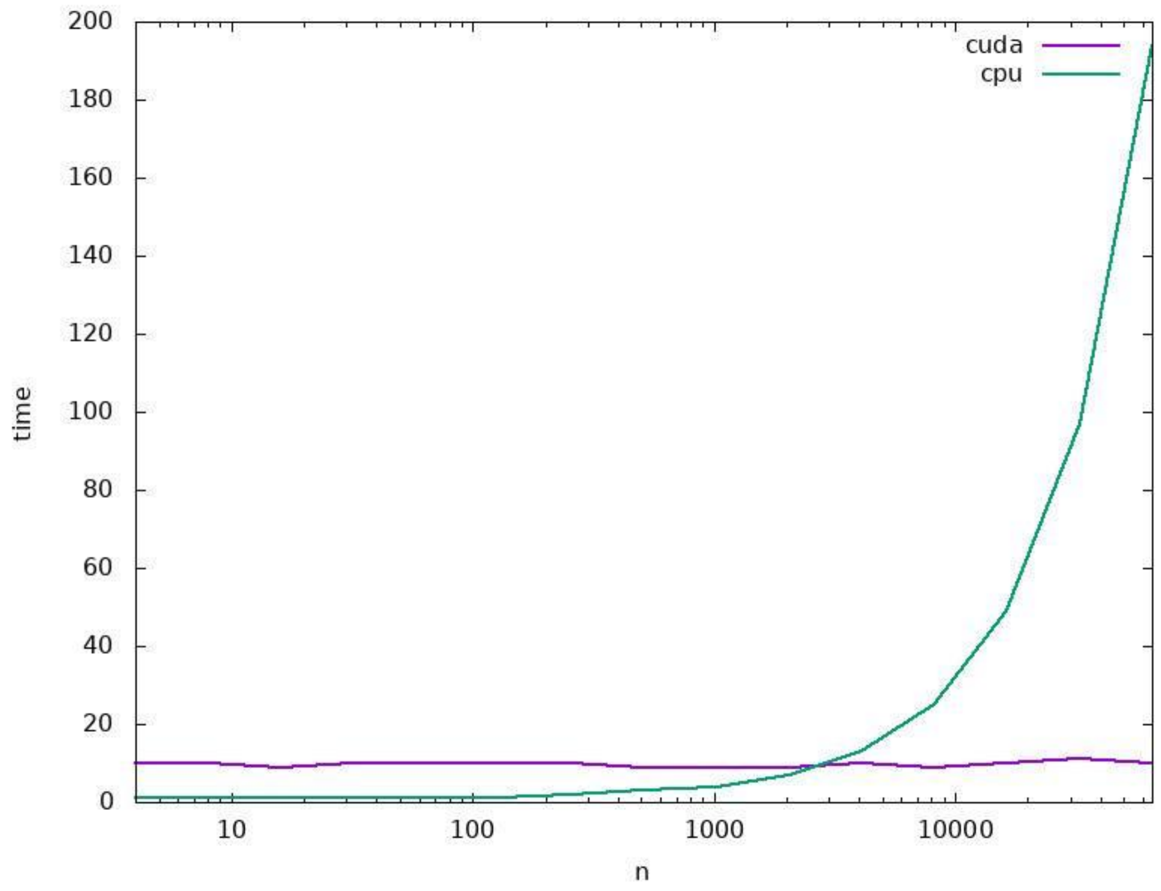
For 1D example the equation is often very simple:

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

But generally, it depends on processing grid structure, sometimes grid is 2D or 3D but block is 1D or vice versa. So, there is no rule that dimensionality of grid must equals dimensionality of a block.

4. A few words about time

Next thing we changed in code was adding loop to make some tests and measure time of calculations on GPU and same on CPU. Using this data, we can make a plot describing time comparison in a vector addition between these two different ways of processing data.



The x axis on plot are sizes of vectors given to calculations. Y axis is the time of operation (in microseconds). From this point we can draw conclusions that cuda cores can process a lot of data in small amount of time. But on the other hand, this operation costs about 10ms to prepare the GPU to work. So cuda cores are faster only if we are adding vectors larger than ~5000 elements. In a case of smaller vectors CPU should be faster. We should always do similar tests to our algorithm to make sure which device would be faster in our case.