


AGH University of Science and Technology		
Introduction to CUDA and OpenCL.		Year: 2019/2020
Exercise no: 1	Subject: .	
Name and surname: Natalia Pluta Przemysław Grabowski		Team no: 02
Date of laboratory: 07.10.2019 r	Date of submission: 14.10.2019 r	Note:

INTRODUCTION

The most important purpose of our laboratory class was to familiarize ourselves with first parallel code with CUDA C. The program made us aware of how powerful this technology could be in terms of the massively parallel structure of many problems.

TASK 1.

Description of query.

First of all, we had to check out how much memory and how many CUDA-capable devices are on our system. Luckily, there is a very easy way to gain this kind of information. In order to do that we had to find in CUDA samples code called deviceQuery. This program contains functions which allow ourselves to find out answers on our previous questions. After compiling, we were able to finally run our program and as a result we received different details about CUDA devices.

```

Device 0: "GeForce GTX 1060 6GB"
  CUDA Driver Version / Runtime Version      10.1 / 10.0
  CUDA Capability Major/Minor version number: 6.1
  Total amount of global memory:              6076 MBytes (6371475456 bytes)
  (10) Multiprocessors, (128) CUDA Cores/MP: 1280 CUDA Cores
  GPU Max Clock rate:                        1759 MHz (1.76 GHz)
  Memory Clock rate:                         4004 Mhz
  Memory Bus Width:                          192-bit
  L2 Cache Size:                             1572864 bytes
  Maximum Texture Dimension Size (x,y,z)      1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers 1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(32768, 32768), 2048 layers
  Total amount of constant memory:             65536 bytes
  Total amount of shared memory per block:     49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                  32
  Maximum number of threads per multiprocessor: 2048
  Maximum number of threads per block:         1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size (x,y,z):   (2147483647, 65535, 65535)
  Maximum memory pitch:                       2147483647 bytes
  Texture alignment:                          512 bytes
  Concurrent copy and kernel execution:        Yes with 2 copy engine(s)
  Run time limit on kernels:                   No
  Integrated GPU sharing Host Memory:          No
  Support host page-locked memory mapping:     Yes
  Alignment requirement for Surfaces:          Yes
  Device has ECC support:                     Disabled
  Device supports Unified Addressing (UVA):     Yes
  Device supports Compute Preemption:          Yes
  Supports Cooperative Kernel Launch:          Yes
  Supports MultiDevice Co-op Kernel Launch:    Yes
  Device PCI Domain ID / Bus ID / location ID: 0 / 3 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.1, CUDA Runtime Version = 10.0, NumDevs = 1
Result = PASS

```

The list contains data that possibly could be required to optimize our code in terms of parallel programming using GPU. For instance, we have to keep in mind total amount of global memory because once when we exceed this number a program will not work properly (we won't get any outcome out of it). Others important specifications are maximum number of threads per multiprocessor or per block, due to the fact that threads are basics element of the data to be processed. Well, we always want to find the fastest configuration of our program but depending on purpose of it and demand for resources, sometimes we will need as many as possible multiprocessors at other times more required will be cooperation between kernel and the CPU.

TASK 2.

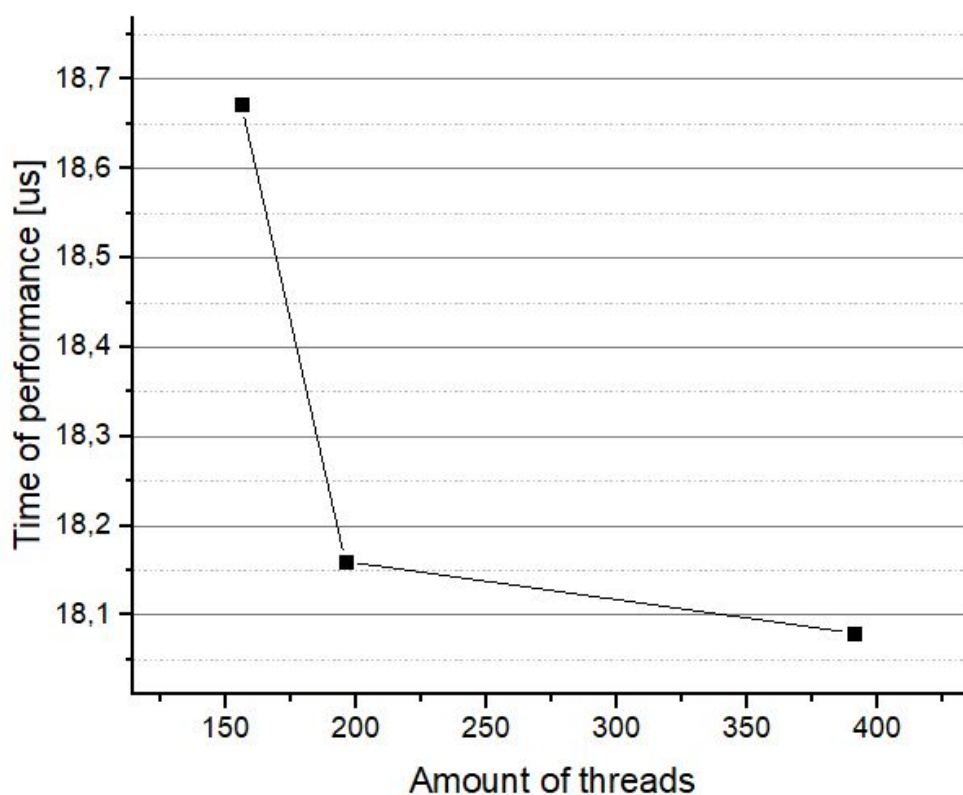
Vector addition using CUDA architecture.

The second program that we were introduced to during lab classes was called vectorAdd and could be found in CUDA samples as well. The whole point of this code is basically adding one vector to another and storing the result in third vector. However in this program we can see that add() is a device function (run on GPU). It is very important to remember to allocate three arrays on the device using cudaMalloc() function, after that we finally can to copy the input data from the host to the device. We also need to clean after ourselves and to to this we use the cudaFree() function.

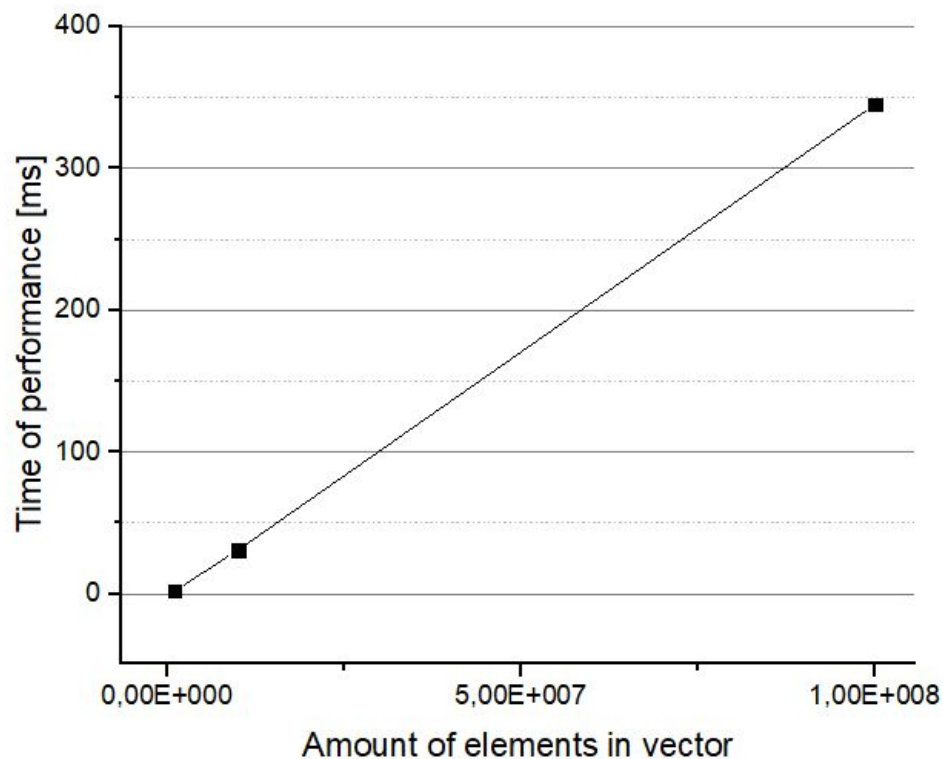
Graphs with time performance.

Table of different parameters in function for gpu to perform the vector addition with the time of GPU activities [CUDA memcpy DtoH]

Type	Integer	Integer	Integer	Integer	Float	Float	Double	Double	Double
Elements in added vector	5*10e4	5*10e4	5*10e4	10e7	10e5	5*10e4	10e5	10e6	10e7
Blocks	391	196	156	390625	196	3907	3907	39063	390625
Threads	128	256	321	256	256	256	256	256	256
Time	18.080us	18.160us	18.672us	351.84ms	1.9172ms	17.952us	1.9352ms	30.943ms	345.36ms



Graph 1. Dependence of time for calculating from amount of threads for 50.000-elements vector to add.



Graph 2. Dependence of time for calculating from amount of elements in vector for double type of data.

- ❖ The nonlinear descent in graph 1. shows that there is no simple link between amount of calculated elements and the size of grid used in processing. That size should probably be taken by many trials. But in this particular case the number of repeats was not enough.
- ❖ On the graph 2. it is clearly seen that number of elements to be calculated straightly corresponds to time that must be taken by processor.
- ❖ To properly make a well working CUDA program we have to experimentally find a size of grid with the shortest time performance.