	Module: **Introduction to CUDA and OpenCL**	Report title: **Report 5**		
	Instructor: **dr hab. inż. Szumlak Tomasz**			
	Name and surname: **Marcin Filipek** **Anca Dărăbuț**	Group: **11**	Date: **11.12.2019**	Grade:

1. overview

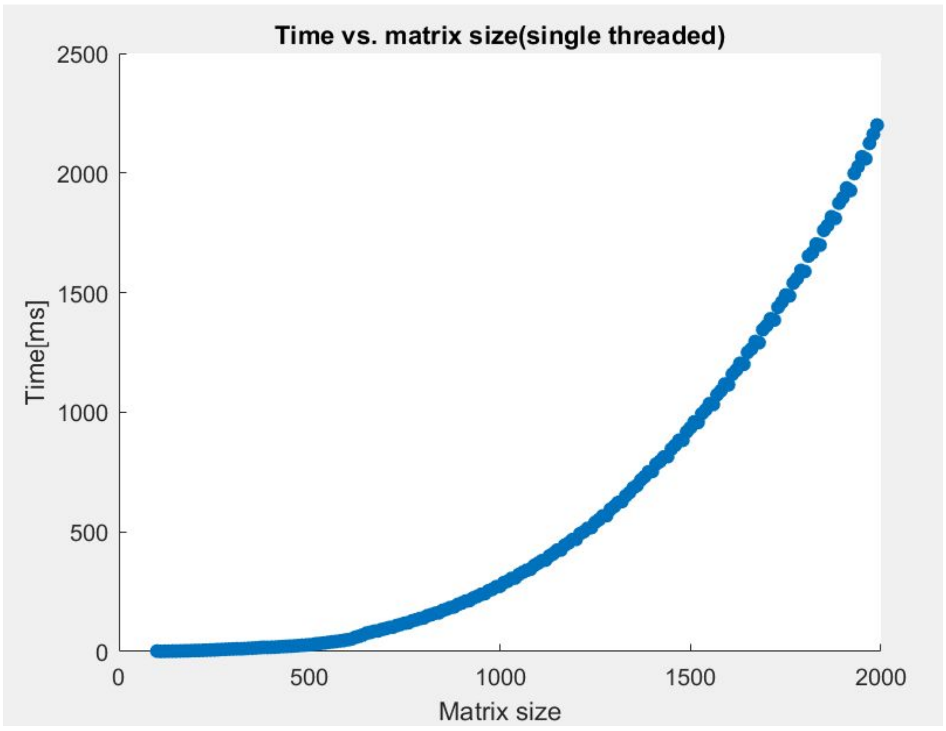
The purpose of this exercise is to create an application for the basic matrix multiplication which should be optimized to work on GPU. Starting from single threaded implementation, going through multi-threaded and shared memory implementation, to using Nvidia libraries for matrix multiplication, the time profilings were sketched.

The conditions are the same for all implementations: the size of the matrices was changed from 100 to 2000, with a step of 10.

2. base-line implementation

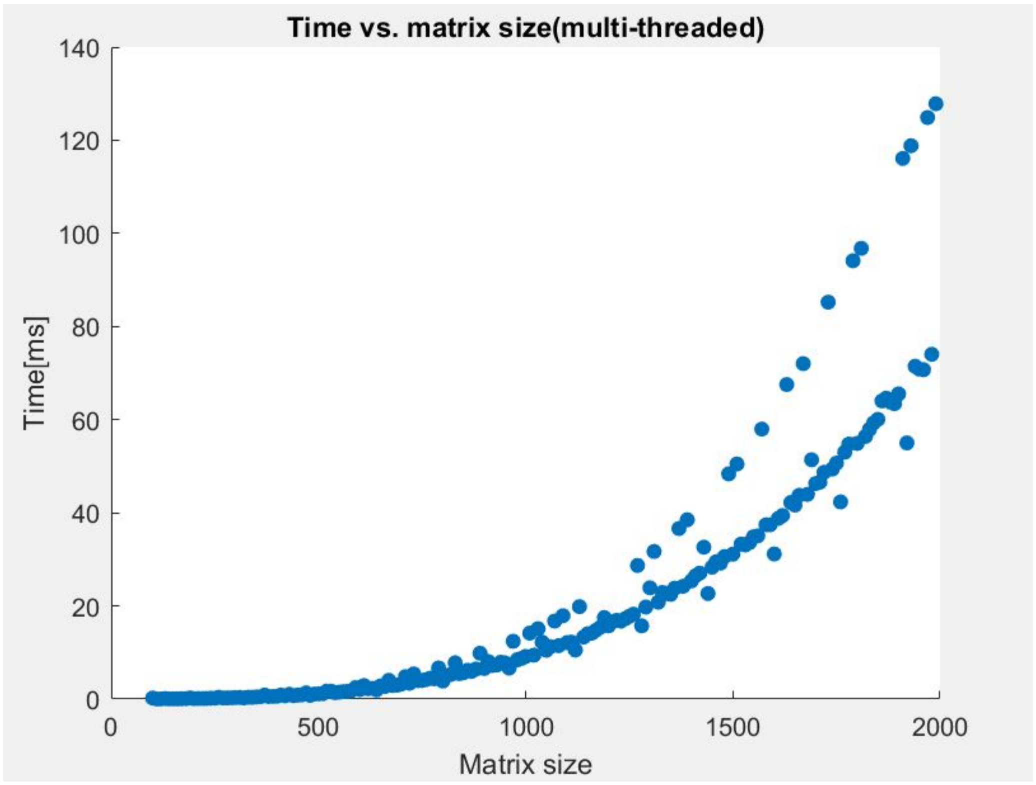
This implementation was done for two different cases: single threaded and multi-threaded.

2.1. Single threaded implementation



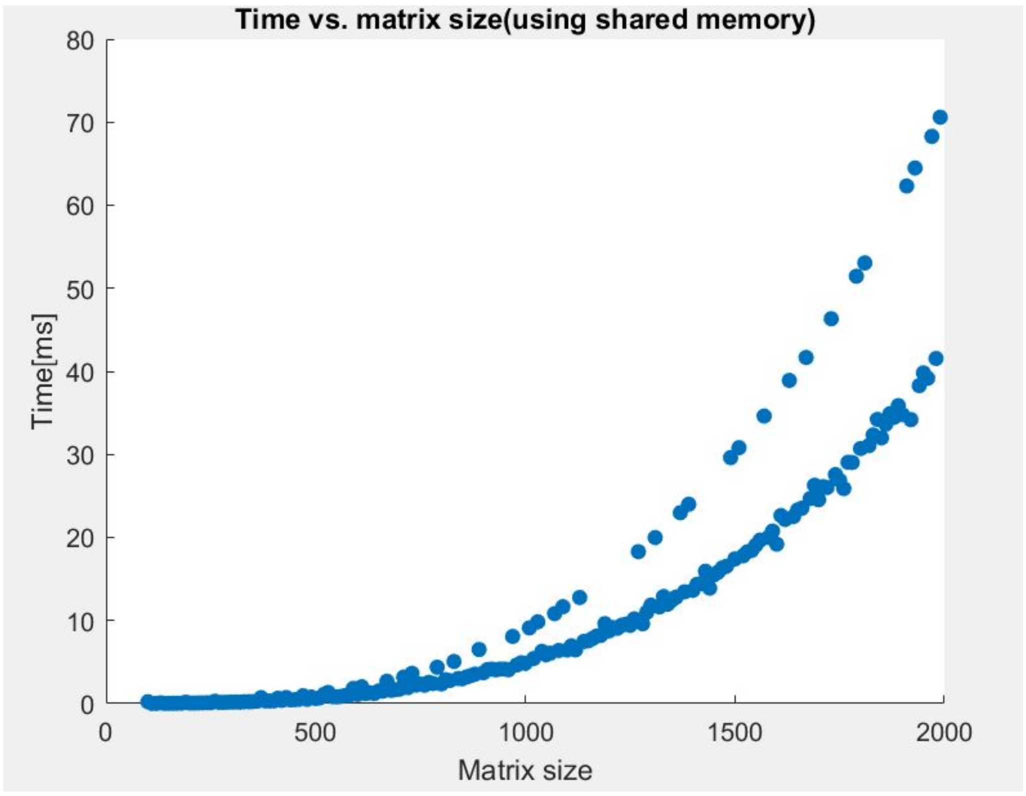
The time profiling reveals the fact that the time increases exponentially with the size of the vector. This implementation is rather slow, the time needed for the computation to be solved exceeds 2 seconds.

2.2. Multi-threaded implementation



The multithreaded implementation does not have such a smooth behaviour. For different, but close values of matrix sizes, the time may differ significantly. After all, there can be seen an increase in time spent for computing the matrix multiplication, as the matrices are greater in size.

3. shared memory implementation



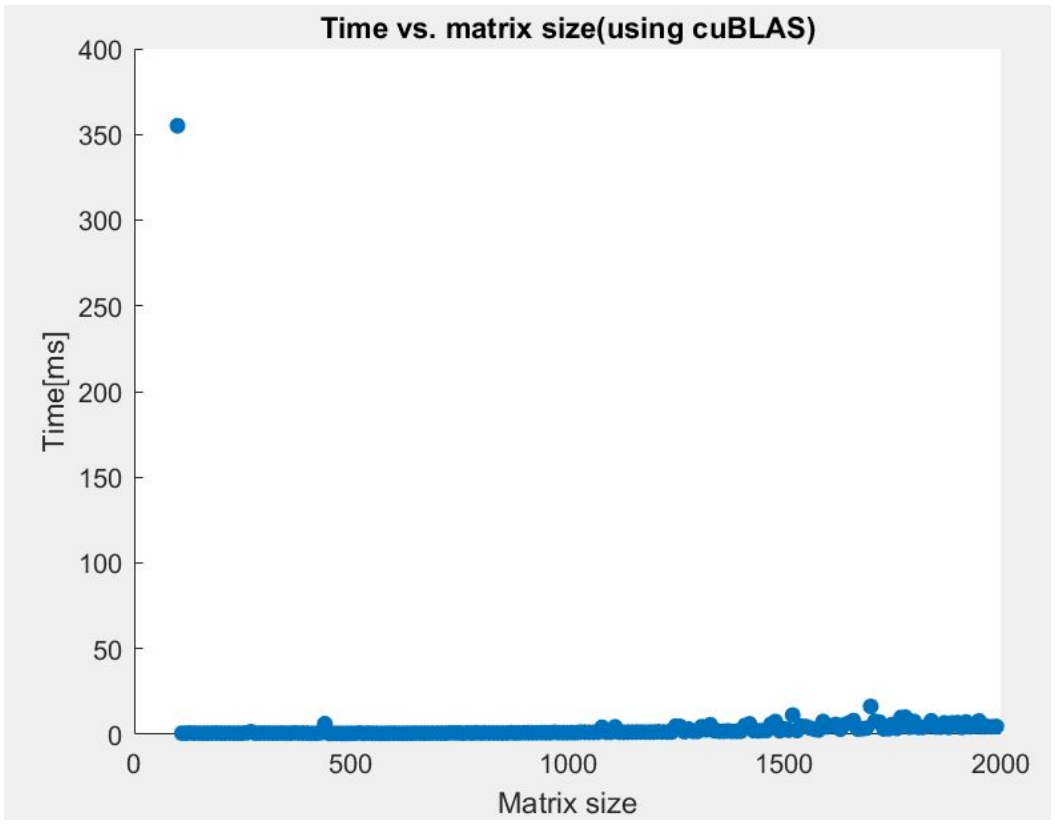
The same tendency is observed in the case when shared memory is used. Even though for some specific number of elements, more time is required, the amount of time spent performing the computation is relatively small.

4. Nvidia libraries for matrix multiplication

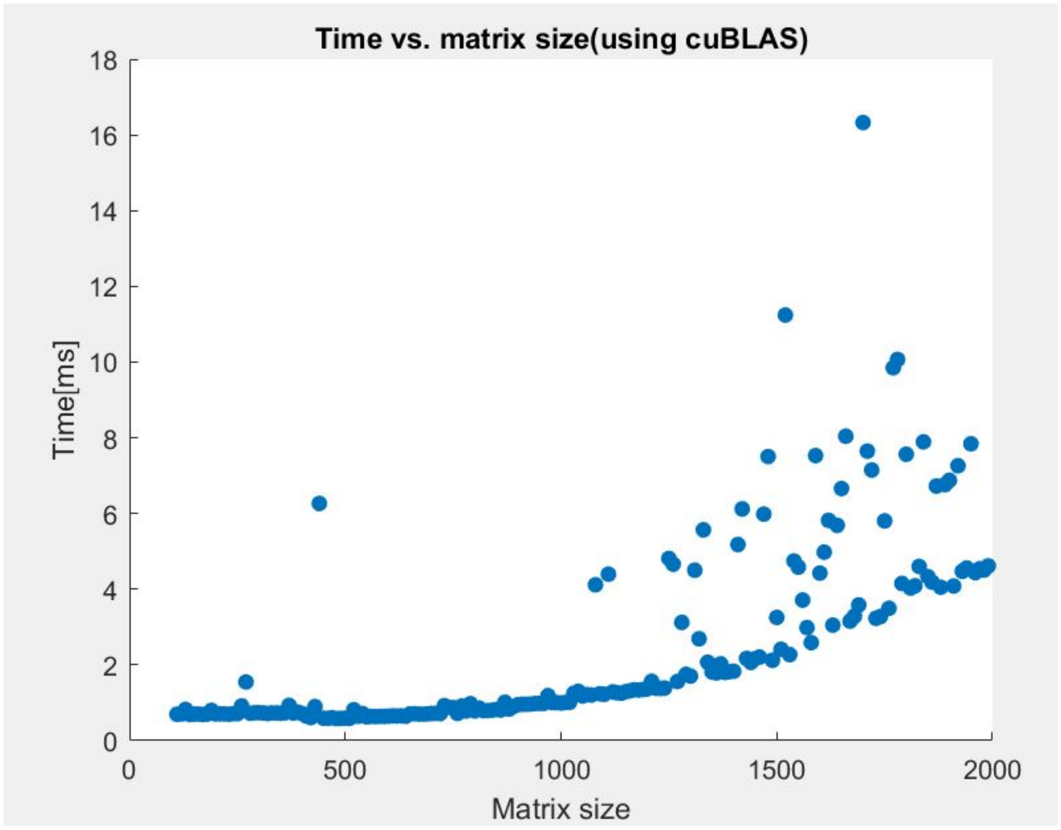
The library that was used for this part is cuBLAS library. As BLAS stands for Basic Linear Algebra Subprograms, it contains also the implementation of matrix multiplication, allowing the user to access computational resources of NVIDIA GPUs.

The raw multiplication code was simply replaced by the function **cublasSgemm_v2()**, accompanied by two functions that deal with cuBLAS handlers, i.e. **cublasCreate_v2()** and **cublasDestroy_v2().**

The time profilings for using cuBLAS functions are included below.



Due to the fact that, for the beginning, when the number of elements equals to 100, the time is significantly greater, the curve for the rest of the values, could not be appropriately sketched and it is listed below:



The behaviour of cuBLAS functions is rather peculiar, but the conclusion that the time is still enhanced when greater matrices are multiplied can be drawn.

5. summary

In order to draw a conclusion, we included in the table below the times for 2 different matrix sizes, comparing all 4 situations.

matrix size	single threaded implementation time [ms]	multi-threaded implementation time [ms]	shared memory implementation time [ms]	cuBLAS time [ms]
220	1.117248	0.031520	0.020480	0.702976
2000	2200.487549	127.842529	70.602493	4.615488

The single threaded implementation is the slowest of them all, while the multi-threaded classical one is faster than the aforementioned. However, the real battle is waged between the shared memory and cuBLAS implementation. Even though the former one seems to be faster for smaller matrices, cuBLAS functions deal better with larger matrices.