# Introduction to CUDA and OpenCL

Łukasz Gut, Grzegorz Litarowicz

Lab 03, 19.03.2020

## Introduction

In this report, we present benchmarks for different implementations of:

- Matrix addition
- Hadamard product
- Dyadic product

Furthermore, we analyze the performance between different processing grid layouts and kernel implementations and host's solutions.

All of the reasoning applies to Nvidia RTX 2060.

## Making life easier

To analyze performance of different processing grid layouts we decided to implement a very simple *kernel_dispatcher* class which deals with the implementation details, so that it is easy for us to re-run benchmarks using different layouts.

We had also provided a simple zero-overhead matrix and vector classes that can be easily parameterized using templates. They offer a very simple, yet handy interface for invoking parallel and sequential operations.

We went with templates over parameters with essential methods so that everything could be determined at compile-time, resulting in decision-making at runtime.

## Processing grid

As it was suggested in the assignment, we had implemented both 1D:1D and 2D:2D versions of the grid layout.

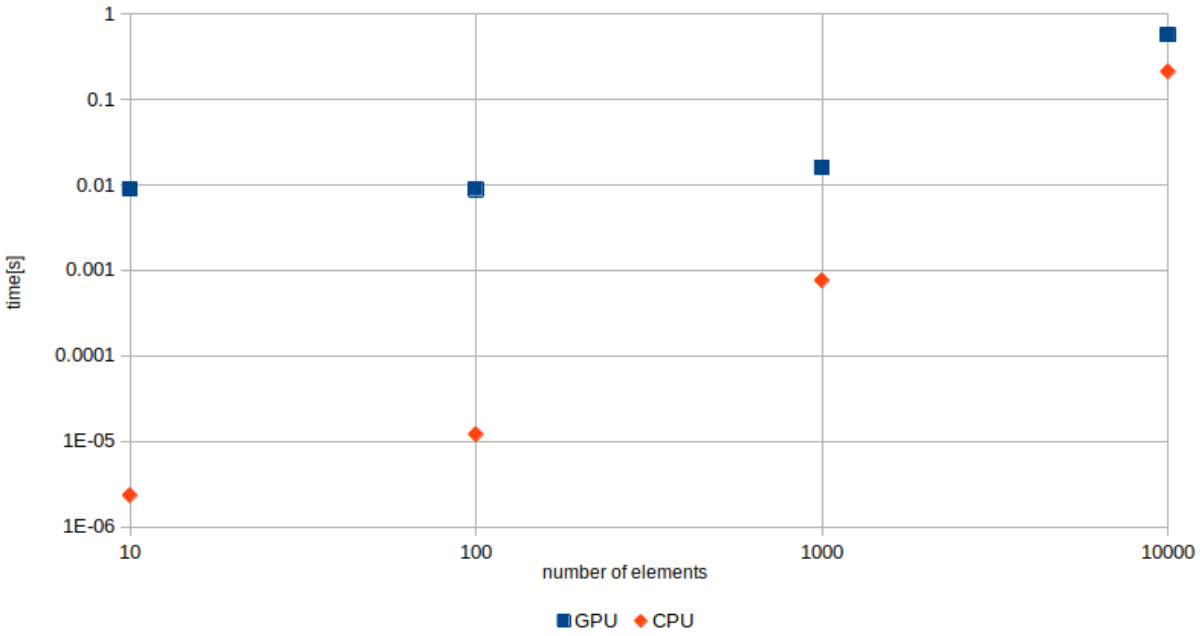
## Kernels

All kernels were implemented as function templates to easily benchmark the behavior of kernels invoked with integer/float types.

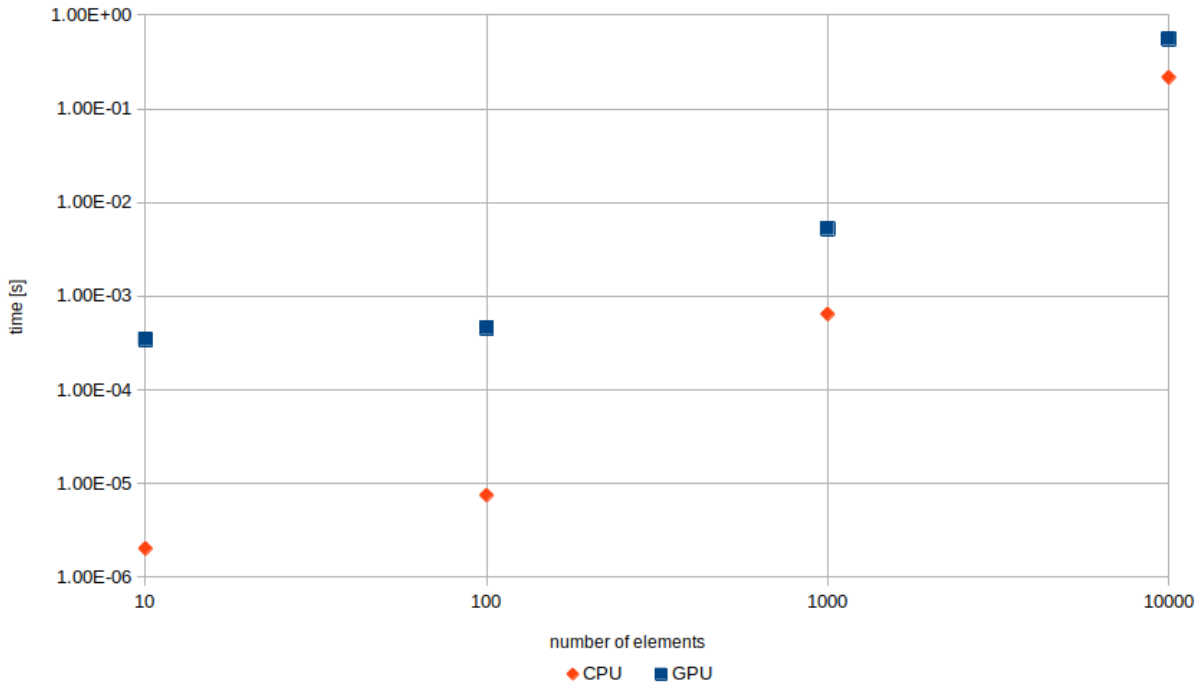
**Benchmarks**

All data points plotted below are an average from 10 consecutive runs of a given algorithm.

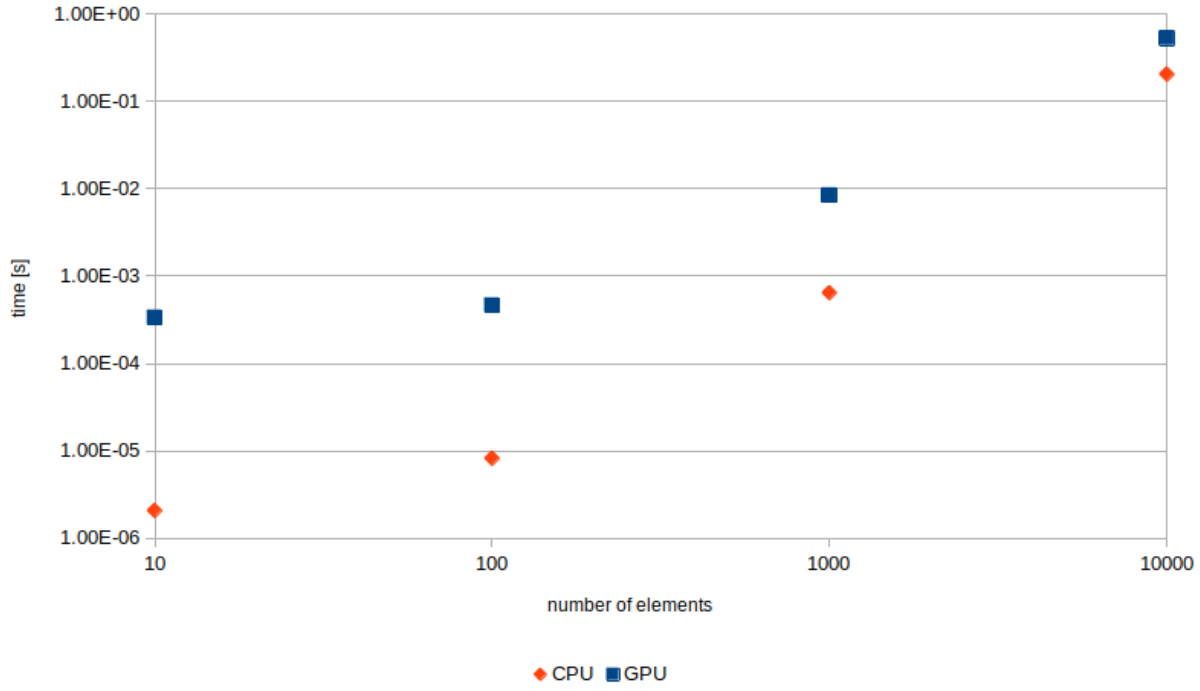
The diagram where a time of execution matrix addition is plotted against a number of elements for float type and 1D processing grid:



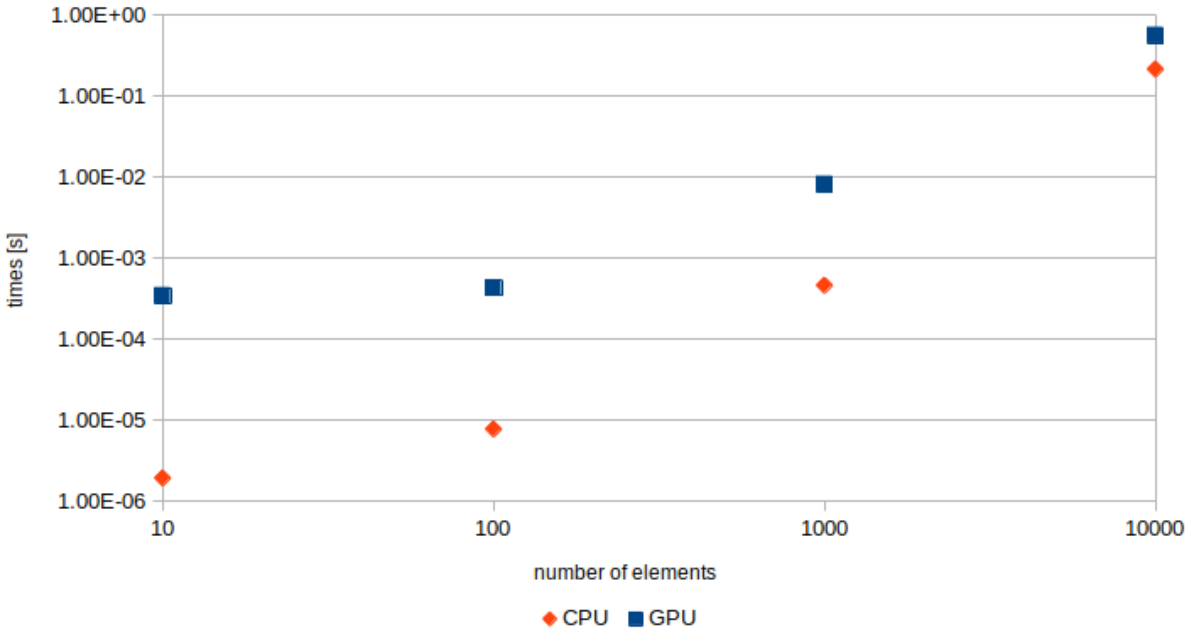
https://www.express.co.uk/news/world/1259863/hantavirus-china-latest-wuhan-coronavirus-what-is-hantavirus-pandemicThe diagram where a time of execution matrix addition is plotted against a number of elements for int type and 1D processing grid:



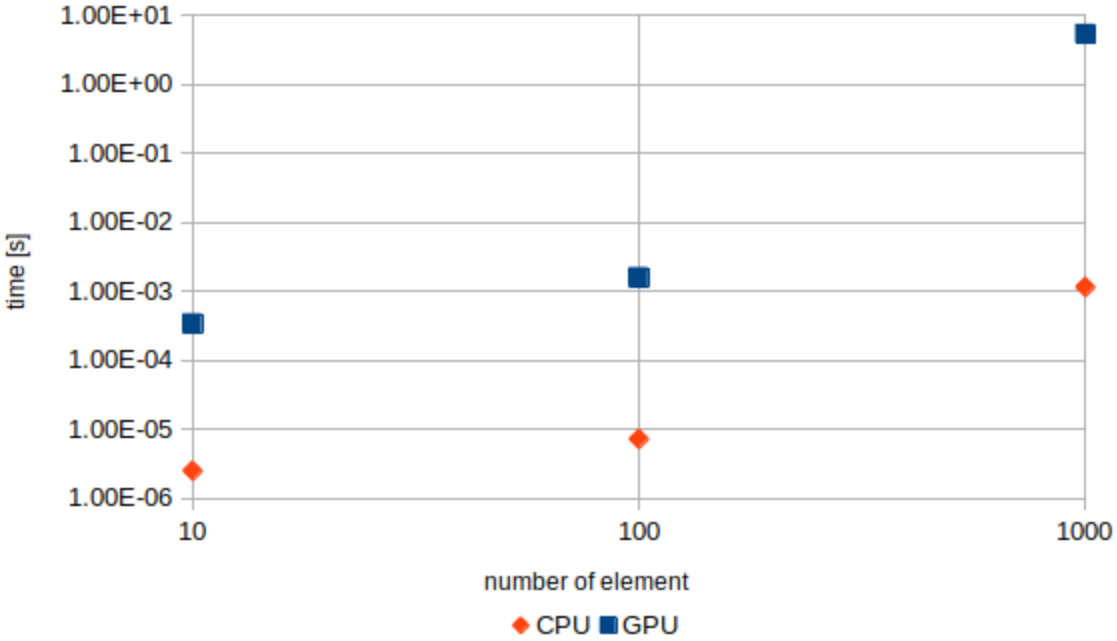

The diagram where a time of execution Hadamard product is plotted against a number of elements for float type and 1D processing grid:



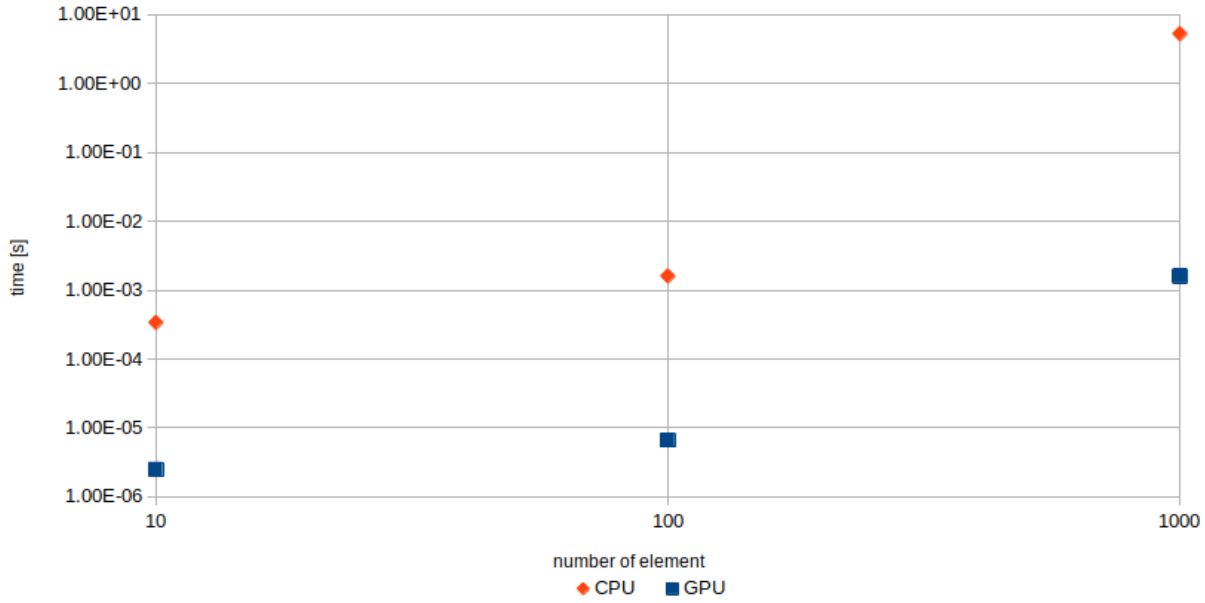
The diagram where a time of execution Hadamard product is plotted against a number of elements for int type and 1D processing grid:




The diagram where a time of execution Dyadic (tensor) product is plotted against a number of elements for float type and 1D processing grid:

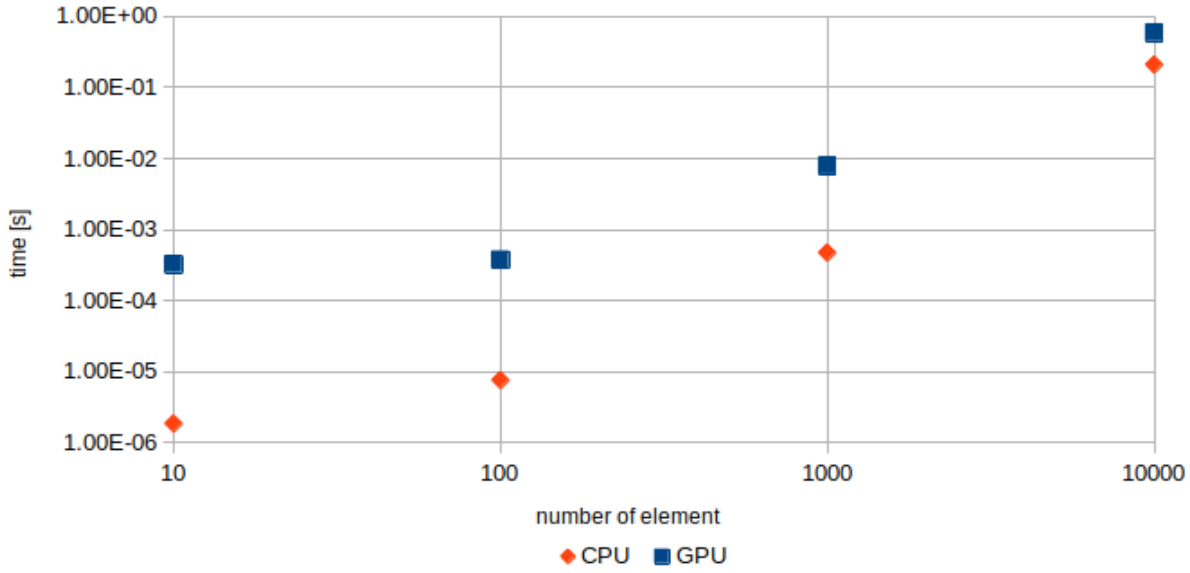


The diagram where a time of execution Dyadic (tensor) product is plotted against a number of elements for int type and 1D processing grid:

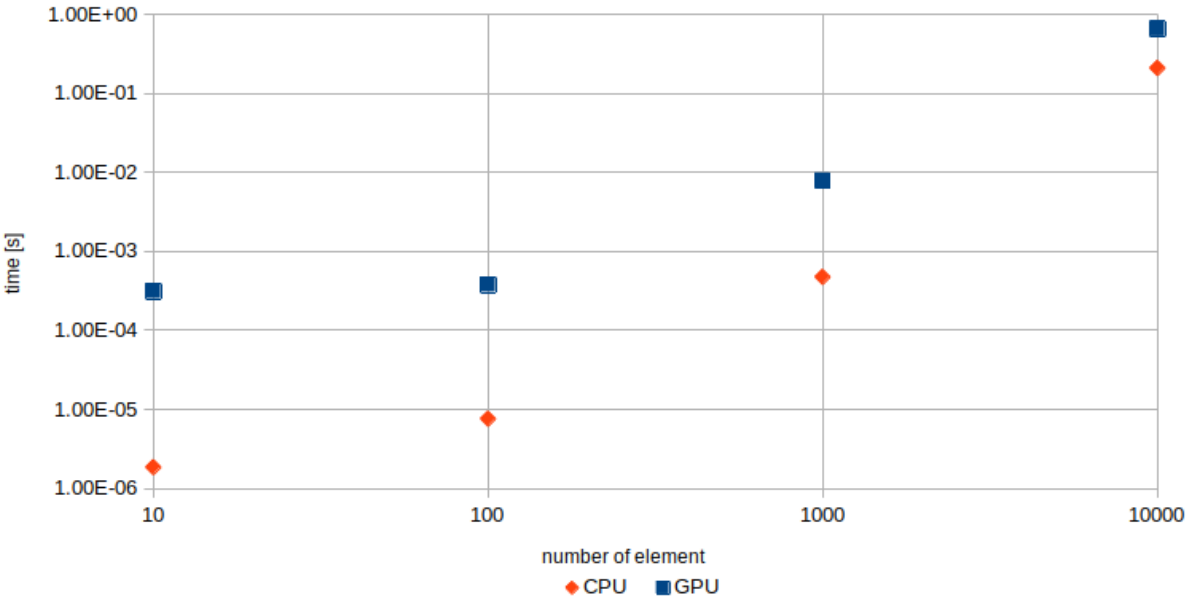



The above plots regarding Dyadic vector 1D processing grid layout were benchmarked with the maximum number of elements equal to 1000. This is because the 1D kernel implementation was so slow that it wouldn't end in any foreseeable future. This is because we did not manage to find an inverse function to calculate indices based on a single 1D entry in the kernel. However, as we will see in later benchmarks, this situation changes dramatically when a 2D approach is applied.

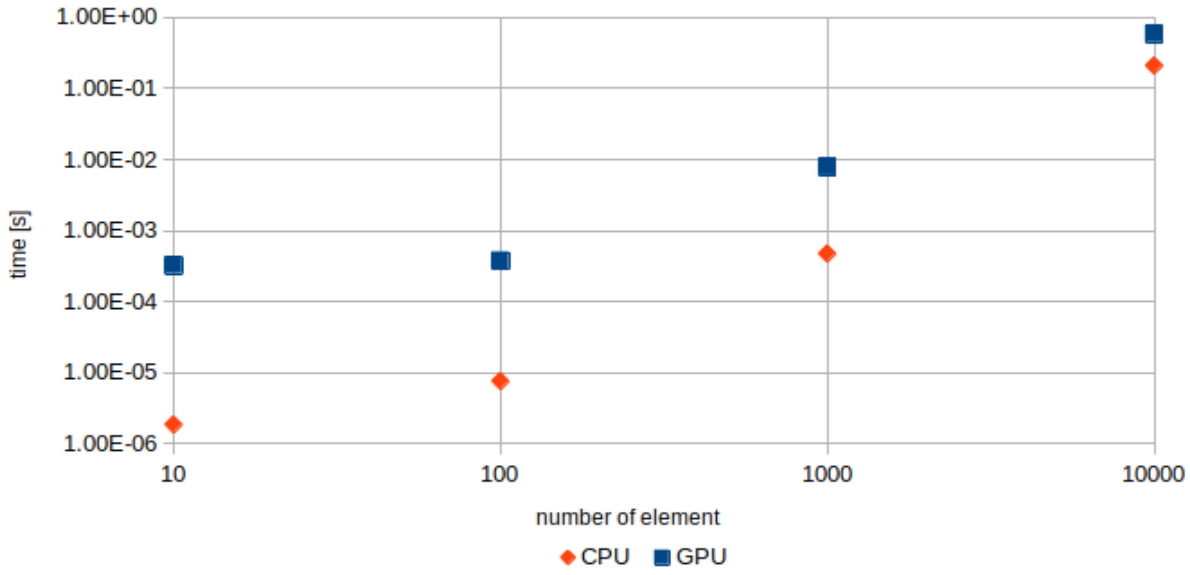
The diagram where a time of execution matrix addition is plotted against a number of elements for float type and 2D processing grid:



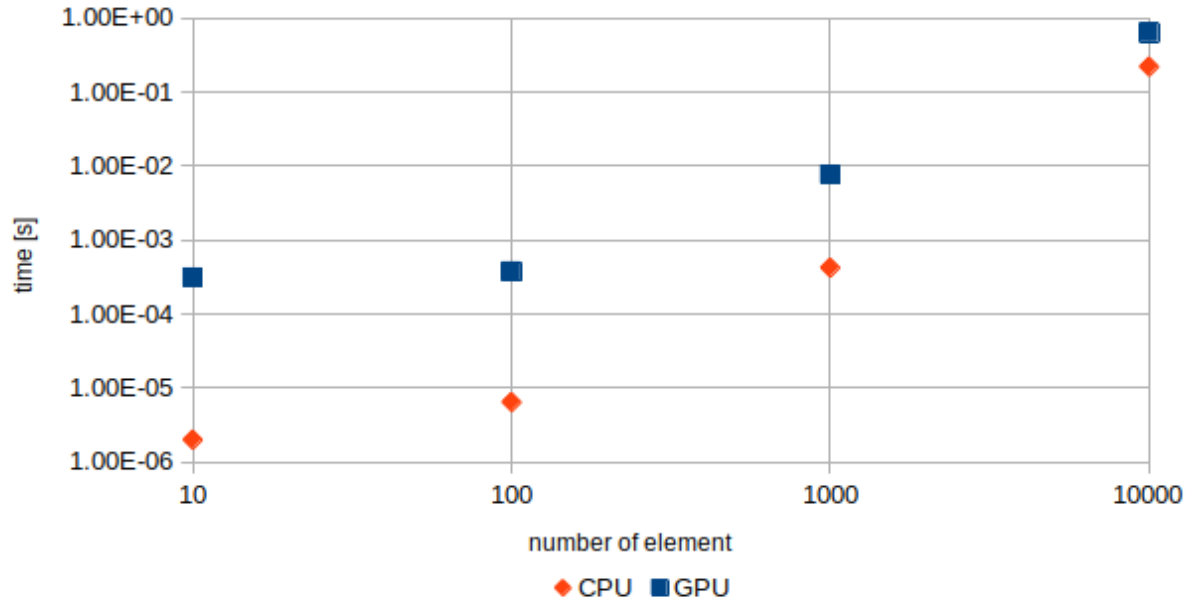

The diagram where a time of execution matrix addition is plotted against a number of elements for float type and 2D processing grid:



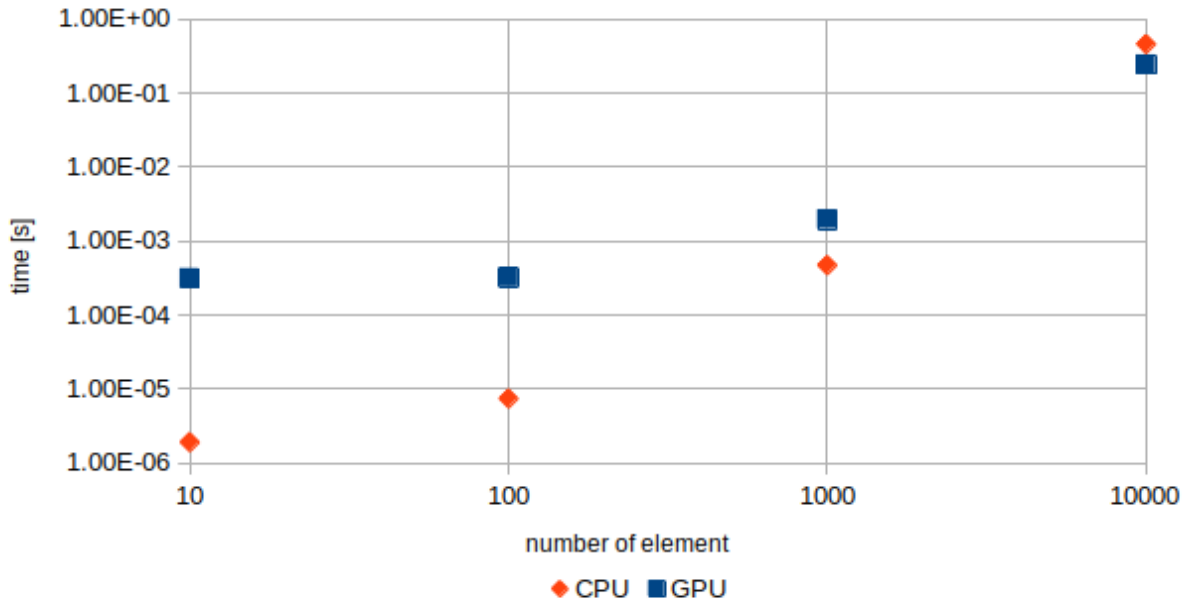
The diagram where a time of execution Hadamard product is plotted against a number of elements for float type and 2D processing grid:



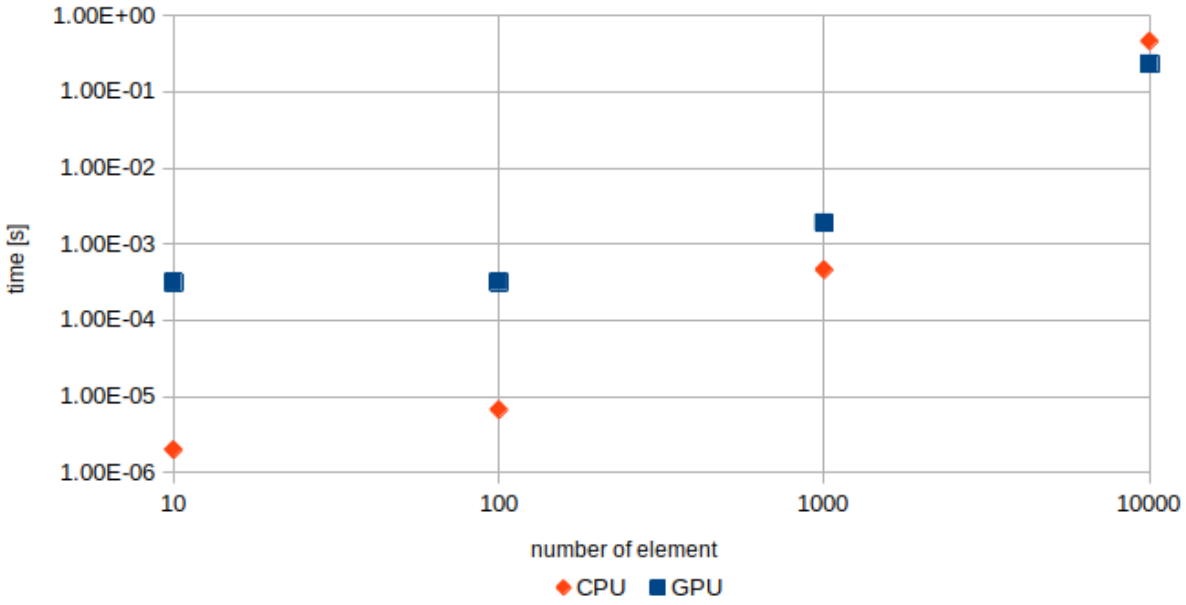

The diagram where a time of execution Hadamard product is plotted against a number of elements for int type and 2D processing grid:



The diagram where a time of execution Dyadic (tensor) product is plotted against a number of elements for float type and 2D processing grid:




The diagram where a time of execution Dyadic (tensor) product is plotted against a number of elements for int type and 2D processing grid:



## Conclusion

As we can see very simple operations on contiguous data that can be easily SIMDified by compilers are performed much faster on the host's side. We think that bad results on GPGPU are due to the time it takes to copy data from host to the device.

However, the situation dramatically changes when we want to perform some operations on a collection that requires a more advanced memory access pattern. In such a scenario when the CPU cannot easily retrieve data from cache GPGPU seems to be taking advantage. This trend can be seen when a 2D processing grid is applied to solve for the Dyadic product. This is the first time when we see that it is actually worth sending data over to the GPU, launching the kernel and sending the data back.