Introduction to CUDA and OpenCL

Łukasz Gut, Grzegorz Litarowicz

Lab 04, 26.03.2020

Introduction

In this report, we measure the performance of naive matrix multiplication implementation, both on host and device. We also analyze how using Unified Memory affects performance on both host and device. Furthermore, we will see if *nvprof* can give us more insight into what is happening on the GPGPU.

All of the reasoning applies to Nvidia RTX 2060.

Error handling

During these labs, we improved on error handling a little bit. We had added calls to *checkCudaErrors()*, which is a simple wrapper for error handling provided by CUDA framework. It gives more insight into runtime errors that may occur.

We didn't feel like reinventing the wheel. :)

Unified Memory

Unified Memory is a single memory address space accessible from any processor in a system. This means that it can be accessed from multiple CPUs and multiple GPUs. This hardware/software technology allows applications to allocate data that can be read or written from code running on either CPUs or GPUs. It is very handy because it reduces the amount of boilerplate code and uses heuristics to optimize locality.

Using it is as simple as calling *cudaMallocManaged()* function instead of *malloc()*. Freeing the memory is also done using *cudaFree()* instead of *free()*.

nvprof

Nvprof is a command-line profiler that Nvidia provides as a part of their CUDA SDK. It will become deprecated soon and will be replaced by Nsight Compute, which is a fully-featured interactive kernel profiler. However, the nvprof is still very easy to use and a handy tool.

It provides information about min/max/average:

- Kernel execution time
- Memcpy time
- cudaMallocManaged
- cudaDeviceSynchronize
- cudaFree
- cuDeviceGetAttribute
- cudaLaunchKernel
- cuDeviceTotalMem
- cuDeviceGetName
- cudaGetLastError
- cuDeviceGetPCIBusId
- cuDeviceGet
- cuDeviceGetCount
- cuDeviceGetUuid

Benchmarking

The diagram where the time of execution of matrix multiplication is plotted against a number of elements in a row for int and allocation with malloc:

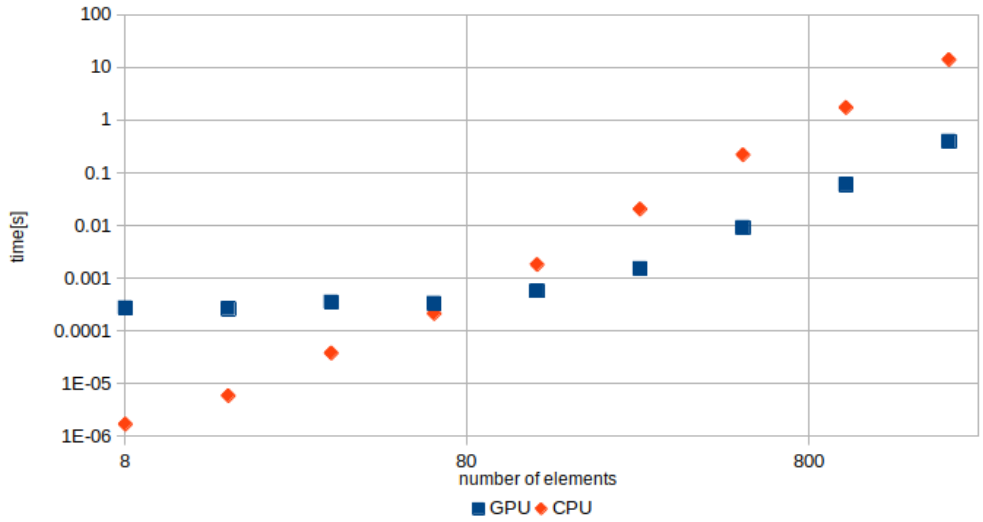

Chart 1

The diagram where the time of execution of matrix multiplication is plotted against a number of elements in a row for double and allocation with malloc:

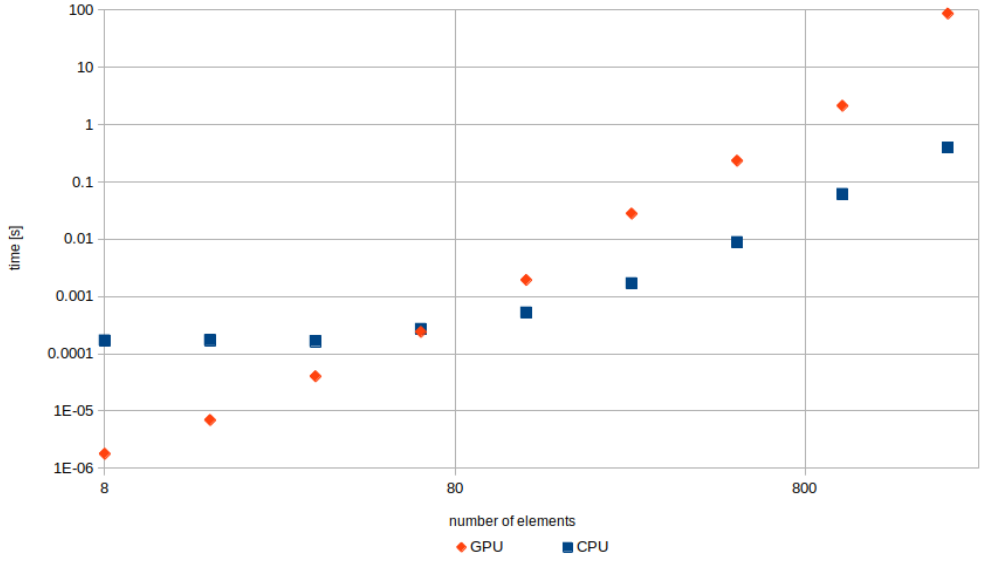

Chart 2

The diagram where the time of execution of matrix multiplication is plotted against a number of elements in a row for double using unified memory and cudaMalloc:

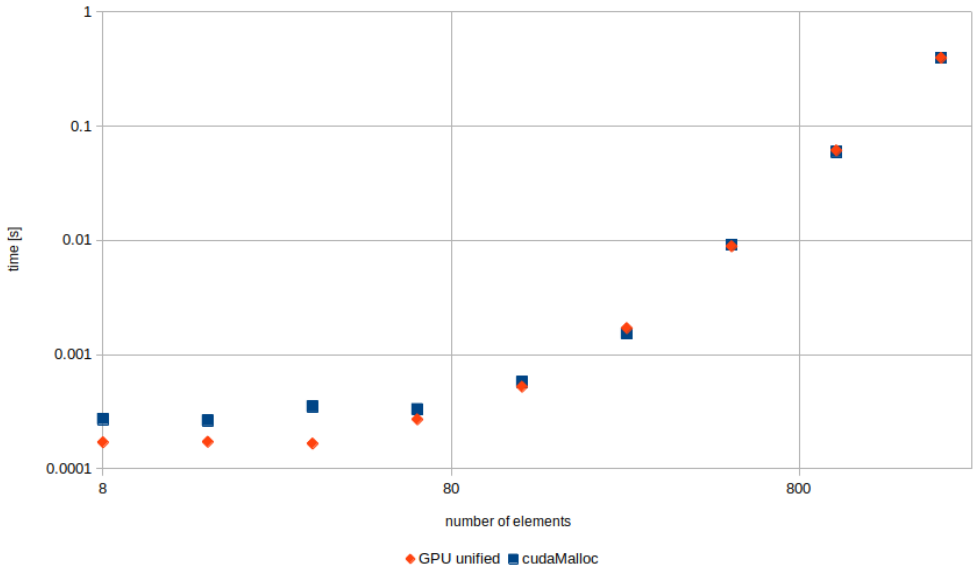

Chart 3

The diagram where the time of execution of matrix multiplication is plotted against a number of elements in a row for int using unified memory and cudaMalloc:

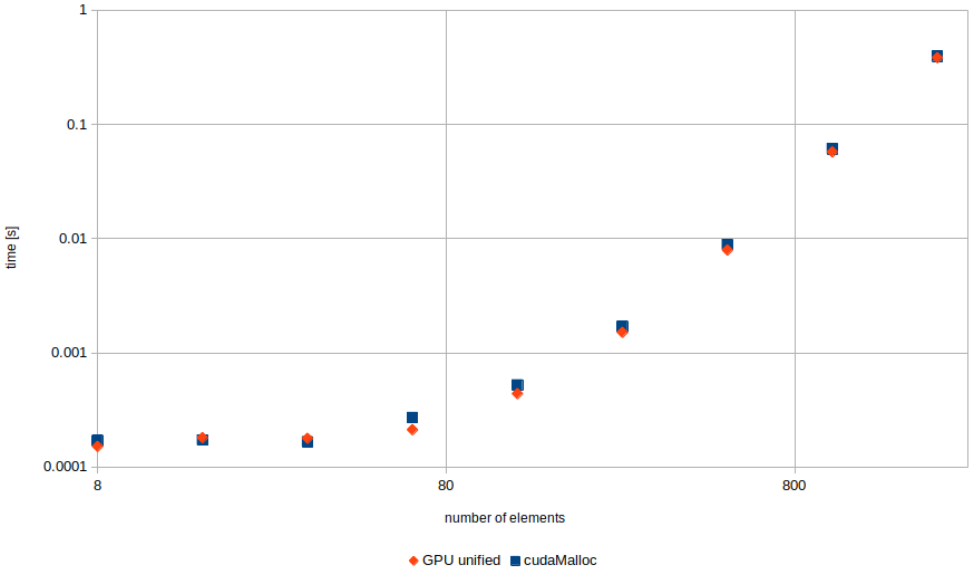

Chart 4

The diagram where the time of execution of matrix multiplication is plotted against a number of elements in a row for double using unified memory and malloc:

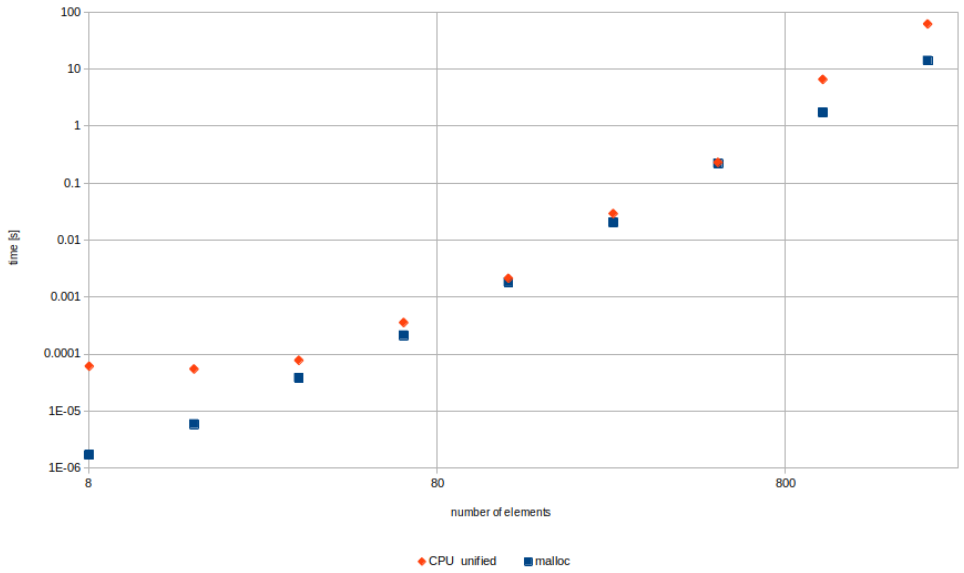

Chart 5

The diagram where the time of execution of matrix multiplication is plotted against a number of elements in a row for int using unified memory and malloc:

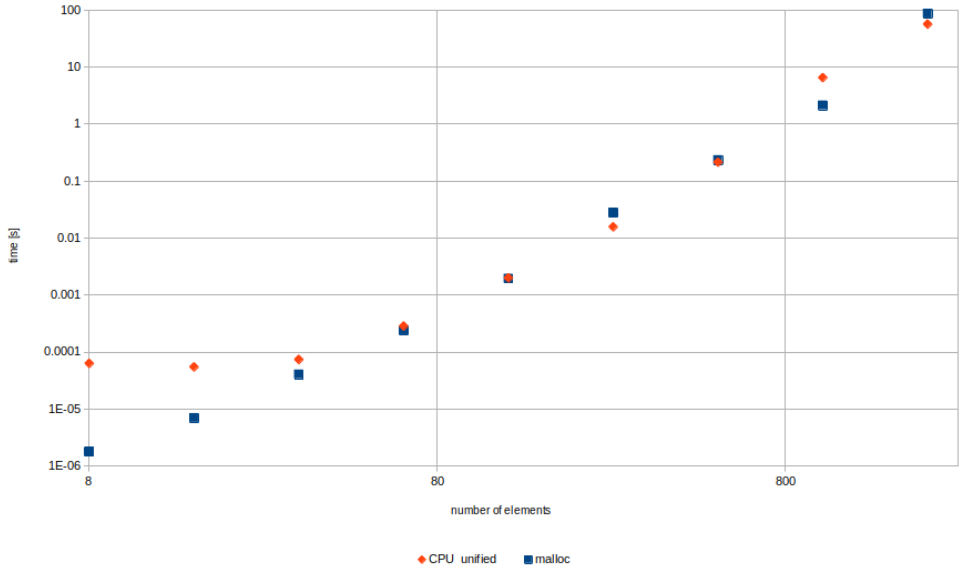

Chart 6

Conclusion

We can see that matrix multiplication is an operation that can highly utilize GPUs. Starting from matrix size 64x64 it is preferable to do multiplication on the device (chart 1 - 2) .

We can also observe that using *double* type is much more expensive on the CPU than using *int* type. It took about a minute to compute matrix multiplication on the CPU using *doubles* and only about 10 seconds to compute the same operation using *ints*. In the case of GPU, it seems like the difference in types is negligible. This favors the GPU in scientific and numerical computations because in such scenarios we almost always want to work with *doubles* because of numeric errors. (chart 1 vs 2).

Moreover, a very interesting thing happens when we switch from *cudaMalloc()* to Unified Memory. When the size of the matrix is small the performance using GPU is vastly improved (chart 3 - 4), but only slightly reduced in case of CPU (chart 5 - 6). On large data sets we start to see that using native solutions becomes important again. However, the big win here is that this method of allocating memory simplifies the code significantly because we don't have to deal with memory copying all the time, which is very nice.