Introduction to CUDA and OpenCL

Lab 4

Kacper Kapuściak

Introduction

In the fourth laboratories, I've focused on using and understanding unified memory. I've also developed a simple wrapper for easier error handling and implemented a naive matrix multiplication algorithm. Later on, I've made measurements that checks how much does unified memory and prefetching data to device affects performance. The results are quite interesting.

All operations were executed on Nvidia RTX 2060 graphics card.

Unified Memory

But what exactly is unified memory? According to Nvidia Developer Blog - "Unified Memory is a single memory address space accessible from any processor in a system." and it is a really good definition. It greatly simplifies CUDA development and makes the code easier to maintain. Using it is as simple as calling the ‘’cudaMallocManaged" function and then freeing the memory with "cudaFree". It allocates and frees the memory that is accessible from both GPU and CPU! We no longer have to manually allocate memory with "malloc" and then copy the memory back and forth from GPU. The code is both simpler and easier to use.

Grid-Stride Technique

When we operate with large data sets most of the time number of elements is far greater than the number of threads that we have. A way to handle this "more data than threads" problem is to use grid-stride loops The technique helps to improve performance when the size of the data set exceeds hardware capability. It is based on a for-loop within the kernel that map indices to "jump over" every iteration to another place in an array using grid dimensions to calculate the next "jump" - hence the name.

Measurements

I did measurements for matrices starting from 8x8 and then doubled that number i.e. next matrix was 16x16 till a maximum of 1024x1024 matrix. For each case, I've made 10 measurements and then took the average time of the execution. I've used **nvprof** to measure kernel execution time. For each test, I've drawn a chart of number of elements and time of execution with the logarithmic scale applied to both axes.

At first, I've come up with a really bad Processing Grid - 2 blocks per grid x 2 threads per block - in order to make improvements over time. I am going to use this example as a base and compare other results to this measurement.

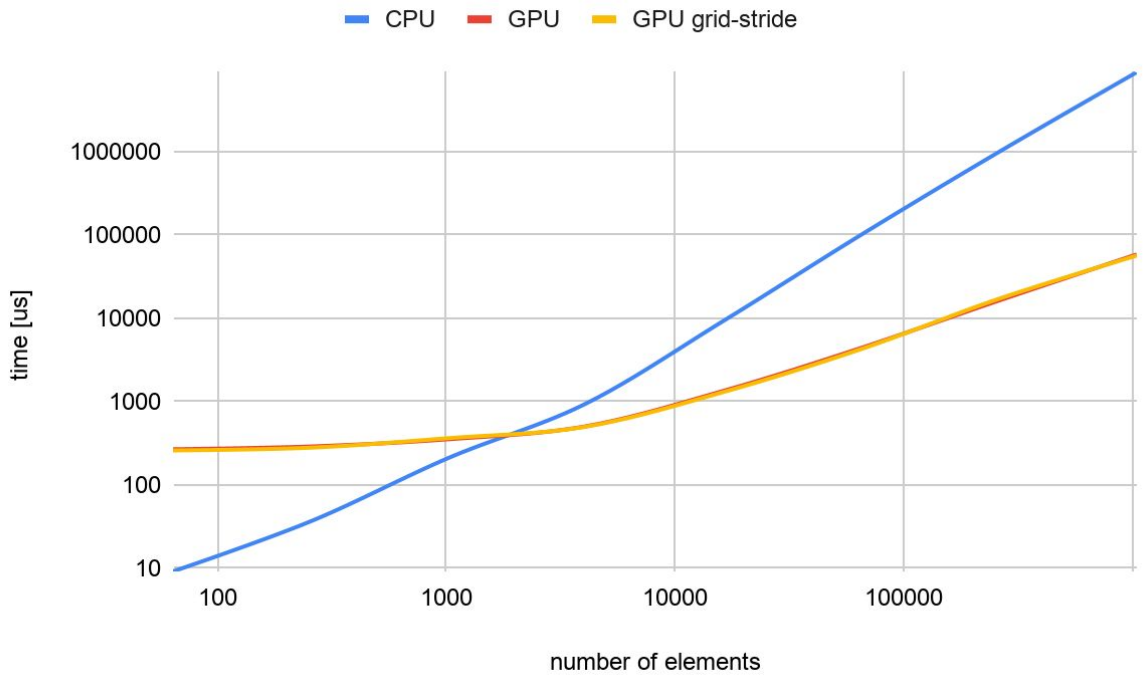

Chart 1. *Matrix multiplication on CPU, GPU using 2x2 PG and GPU using 2x2 grid-stride technique - native malloc*

As we can see, starting from 32x32 matrix calculations are being made faster on GPU than on CPU. This trend will occur in every other measurement in this report. On average of all calculations on CPU took **1.26 s** for using bare-bone malloc. GPU took **18.6 ms** thanks to very fast calculations on bigger matrices. Calculations using the grid-stride technique were slightly faster **15.57 ms**. That's a 19% increase in speed.

Let's make the same calculations on 2x2 PG using Unified Memory:

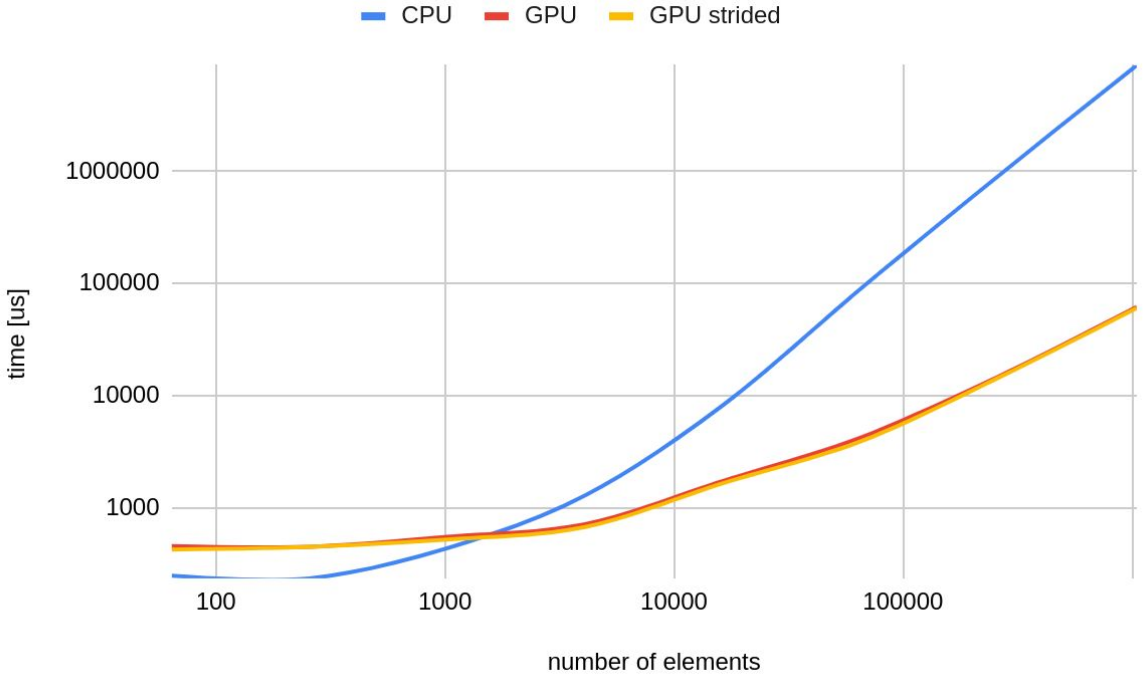

Chart 2. *Matrix multiplication on CPU, GPU using 2x2 PG, and GPU using a 2x2 grid-stride technique - unified memory.*

Using unified memory calculations were generally slower both on CPU and GPU but I've seen slight improvement between grid-stride and classic kernel: CPU took **1.29 s**. GPU without grid-stride **16.95 ms**. GPU using grid-stride: **14.8 ms**. Again between CPU and GPU difference is huge. Grid-stride gives a slight improvement of 14.5%.

Let's try with different Processing Grid - 16 blocks per grid X 16 threads per block but I'm going to interlace the measurements. Let's go back to native malloc implementation:

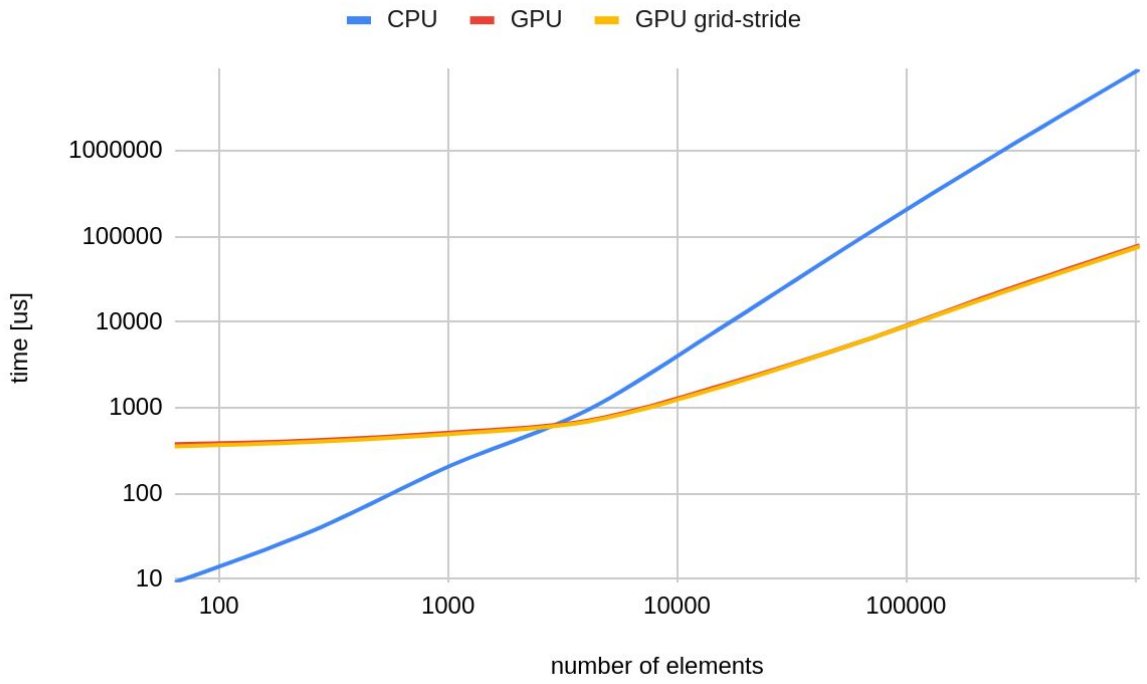

Chart 3. *Matrix multiplication on CPU, GPU using 16x16 PG and GPU using 16x16 grid-stride technique - native malloc*

As we can see Chart 3 doesn't look much different than Chart 1. But we can see the difference in numbers. Using 16x16 grid is considerably faster than the 2x2 one. GPU: **10.27 ms** that's over 8 ms difference. For grid-stride, it's **10.12 ms**. Between grid-stride and without stide we have 1.5% difference. The big improvement came from a different Processing Grid layout. Here in terms of milliseconds use, od grid-stride technique isn't very noticeable but we should always consider even the smallest improvements.

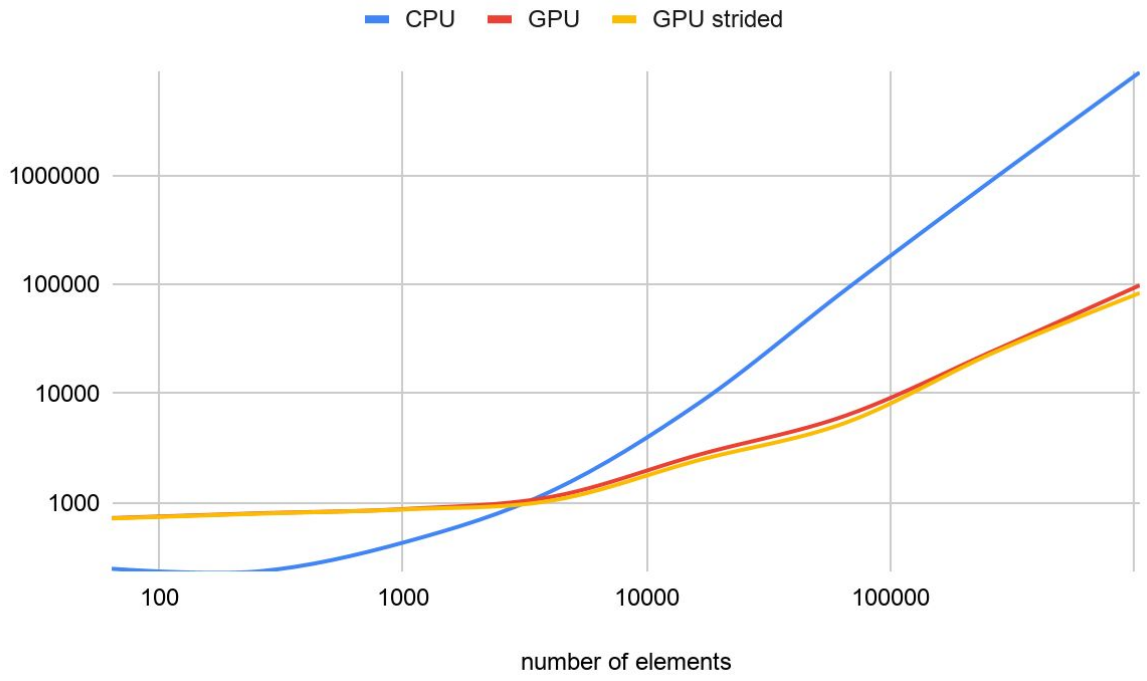

Chart 4. *Matrix multiplication on CPU, GPU using 16x16 PG and GPU using 16x16 grid-stride technique - unified memory*

Same story as in previous examples: change of the Processing Grid is the most noticeable part and the difference can be is in the numbers. The average GPU execution time for all calculations is **12.31 ms** and with grid-stride, it's **11.92 ms**. It is slower than native malloc but not by far in this example. Only 1.8 ms.

At last, we shall compare the best Processing Grid I've come up with. 32 (wrap size) * 30 (number of SMs) blocks per grid X 128 threads per block. It is the most efficient PG layout of these examples.

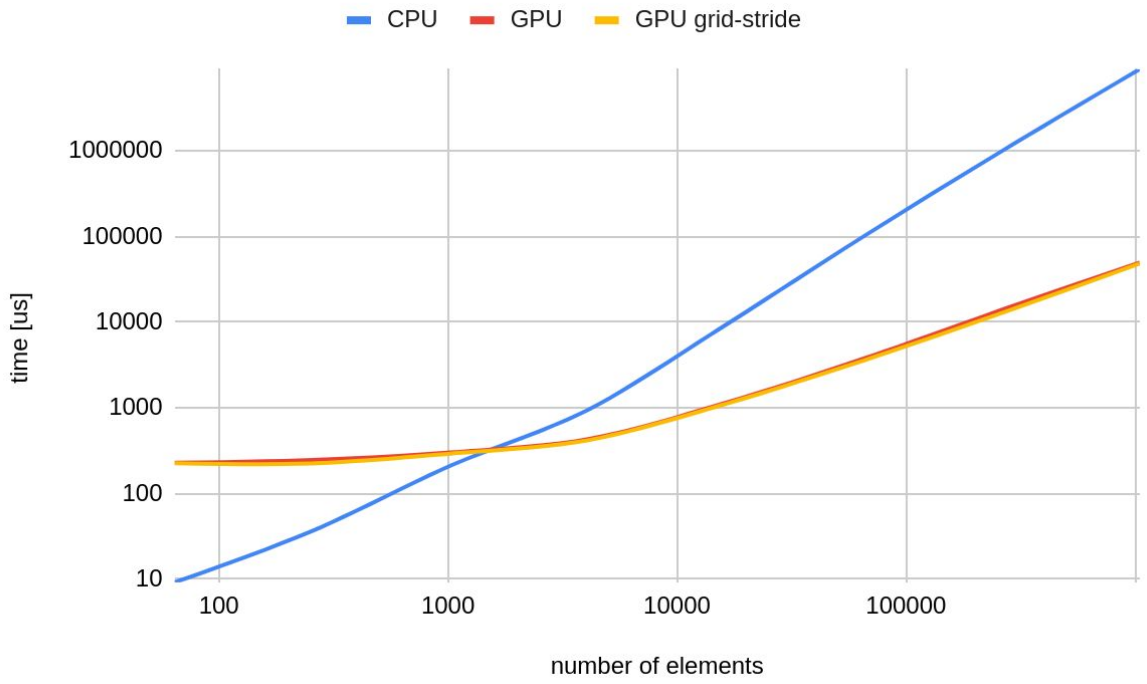

Chart 5. *Matrix multiplication on CPU, GPU using 32*30x128 PG and GPU using 32*30x128 grid-stride technique - native malloc*

With the native malloc approach finally, we've gone below 10 ms threshold. For GPU: **8.74 ms** and grid-strided GPU: **8.39 ms**. Between Grid-stride 30*30x128 PG and non-grid-stride 2x2 PG is more than 10 ms difference. In percentage, we can say: The best solution I've come up with is 2.2 times faster than the worst one (excluding CPU).

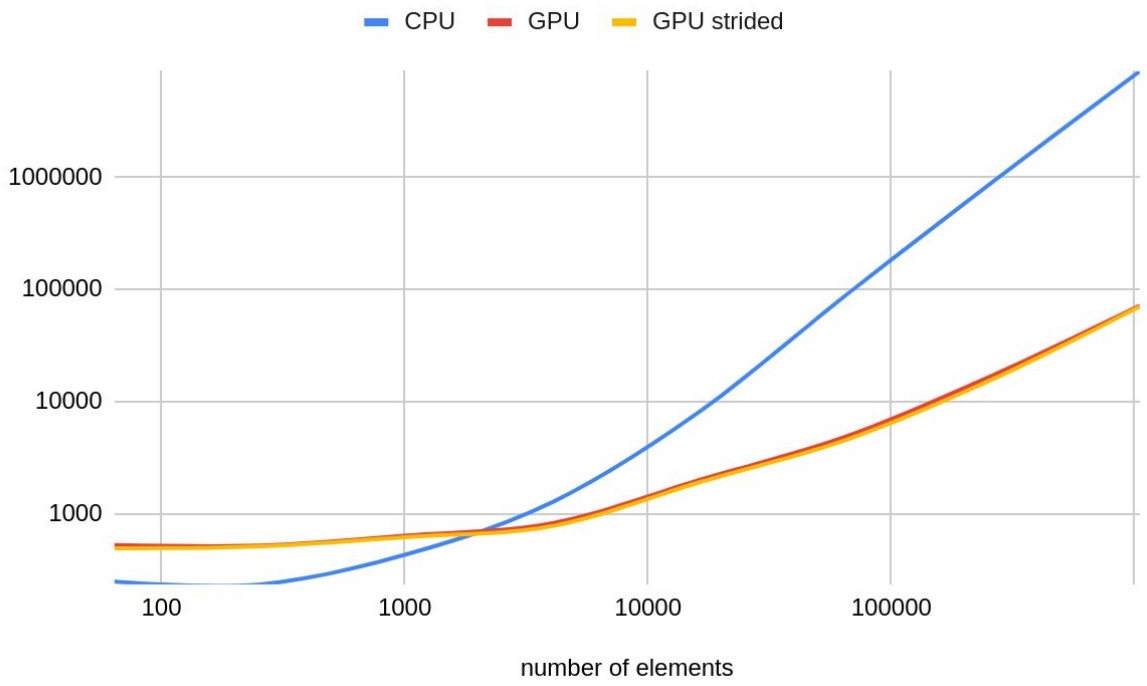

Chart 6. *Matrix multiplication on CPU, GPU using 32*30x128 PG and GPU using 32*30x128 grid-stride technique - unified memory*

As per usual using unified memory is slightly slower than native malloc and it doesn't depend on the Processing Grid at all. GPU: **10.61 ms**. GPU using the grid-stride technique: **10.32 ms**. Grid-stride in this example gave us only a 2.8% improvement.

Conclusion

My first thought is that the matrix multiplication on GPU is highly effective starting from relatively small matrices. Starting from 1024 elements in a matrix that is 32x32 it was faster to do calculations on GPU.

As we can see for unified memory not only we have some additional overhead for CPU but all measurements are slower. But that was what I was expecting. Higher memory abstraction from hardware always comes at a cost. In average unified memory was up to XXX slower than bare bone malloc.

Another thing that we can see is that optimizing the Processing Grid layout should be our top 1 priority with CUDA because it easily makes huge differences straightaway.