Introduction to CUDA and OpenCL

Łukasz Gut, Grzegorz Litarowicz

Lab 05, 02.04.2020

Introduction

In this report, we discuss and analyze memory accesses using unified memory on CPU, GPU, CPU & GPU, GPU & CPU. We also take a step back and analyze the vectorAdd example provided by Nvidia with regards to page faults and try to improve on data initialization. Lastly, we will take a look at the prefetching technique.

Processing Grid

We first started by optimizing the Processing Grid layout. It turns out, that optimal one, when using stride technique, is 256 blocks per grid and 64 threads per block. We chose these numbers taking into consideration hardware capabilities. The important numbers are:

- Streaming multiprocessors: 30
- Cuda cores per streaming multiprocessor: 64
- Warp size: 32

The idea was to map blocks of 64 threads (twice the warp size) in one to one relationship to Cuda cores so that the execution of a single instruction would take a single cycle. The intuition turned out to be good.

Memory Access

To analyze memory accesses we used code samples that were provided and slightly adapted them. The sources can be found in the *rtx-unified-memory* folder.

First, let's see what happens when we call *cudaMallocManaged()*. The sequence of operations is as follows:

1. Allocate new pages on the GPU
2. Unmap old pages on the CPU
3. Copy data from the CPU to the GPU
4. Map new pages on the GPU
5. Free old CPU pages

As we can see, on finish the memory pages are mapped on the GPU side. This means that accessing them will be very fast. Unfortunately, the CPU is not in an ideal situation here, meaning that when we want to access the pages from the CPU, they will have to be mapped.

From the above, we can conclude that accessing the memory from the GPU first should be faster, than accessing it from the CPU first. Let's see the benchmarks.

Memory accesses	Time [s]
CPU	2.27529 s
GPU	0.813474 s
CPU-GPU	3.78206 s
GPU-CPU	2.98422 s

And this is in fact what we observe when playing around with the samples. The most interesting bit is the CPU-GPU vs. GPU-CPU. There is almost 1 second of difference, but operations done on the memory do not differ! What is more, when we first access the memory on the CPU and then want to use it on the GPU it isn't mapped immediately. Instead, when the kernel tries to access the memory that does not exist in the page table it will map it. This extends the execution time of each kernel call significantly. So all of this is the page's fault... :)

This conclusion is backed by Nvidia's dev blog on Page Migration mechanism for Pascal and Volta architectures. We didn't find any evidence that this mechanism was changed in the Turing architecture.

Optimizing vector initialization

Based on the previous analysis we decided to adapt the vector addition sample so that the vector would be initialized on the GPU instead. Let's see the benchmarks:

Memory initialization	Time [s]
CPU	0.428363
GPU	0.159875

As we can see, there is almost a 3x speedup when we initialize the data on the GPU.

Prefetching

Asynchronous memory prefetching lets us copy memory that we will use from a different place in the background of executing our calculations. In the given example, we added checks for a and b vectors to simulate their usage on CPU.

1) Prefetching before GPU initialization makes the code run faster

Using only vector c on the CPU	Time [s]
No Prefetch	0.146651
Prefetch of vector c	0.14432

2) Prefetching before usage on CPU a, b, c

Using vectors c, b and a on the CPU	Time [s]
No Prefetch	0.330079
Prefetch all of them	0.286975

3) Conclusion - always prefetch data when you know you will need it!

Conclusion

We conclude that memory accesses when dealing with high-performance computations are of utmost importance. You have to be wise and know when and in what order you want to access the memory on both host and device.

Using Unified Memory model is very handy but it is not a silver bullet. Using it efficiently is not so easy because it nicely abstracts all of the memory synchronization details, which can potentially result in a very hard to find the bottleneck.

Fortunately, there exists a memory prefetching mechanism, which used correctly, can speed up the execution time significantly! Asynchronous prefetching is your best friend!