Introduction to CUDA and OpenCL

Lab 5

Kacper Kapuściak

Blind optimization

Our first exercise in laboratories was to optimize an example program by guessing. The program was adding a number to a huge vector ($n=2^{24}$) via kernel on GPU. The default example was an interesting processing grid of 1x1. I've measured the time of GPU activities ten times and calculated an average execution time: **1,884583 [s].** So this is our base for further optimization.
To contrast a tiny tiny 1x1 PG I've decided that I want to try a huge one - 500x500 and the results were instantly looking better. Average execution time: **136,53 [ms]**! That's more than 10 times faster. Currently, I don't know if that *threadsPerBlock* or *numberOfBlocks* caused such advancement. Let's try both 500x1 and 1x500 processing grids and see what will happen.

1 x 500: **133,71 [ms]**; 500 x 1 **161,03 [ms]**

Well, 500 threads per block on 1 blocks are slightly faster but the difference isn't that noticeable. Let's try something different - 20 blocks x 60 threads. **110,03 [ms]**. I've tried a few more but that was the best idea so far.

Optimizing for a particular GPU

In order to know GPU details while the code is running, we can invoke *cudaGetDeviceProperties* function that only needs to know the ID of the device. It can be also obtained in runtime with *cudaGetDevice* function. This way we can dynamically set Processing Grid size regardless of the device the code is running on. Using these functions I can now query the maximum number of threads per block (here 1024) and a number of streaming multiprocessors (here 30). Using that grid maxNumberOfThreads *x* numberOfSMs on average takes **128,83 [ms]**

Unified memory and page faults

Unified Memory is a single memory address space easily accessible both from CPU and GPU processors. A page fault is an execution fault that happens when an application accesses virtual memory that is marked nonresident by the operating system. If the access was valid, the operating system pulls the page into physical memory and updates the physical address to point there and then resumes execution. Page faults are not errors. Their occurrences are completely normal. They occur whenever there is a need to increase the amount of memory available to programs.

I've made some measurements of page fault occurrence on different scenarios:

CPU only page faults:

384

GPU only page faults:

588
Kernel execution: **29,805 [ms]**
CPU-GPU page faults:
Host->Device: **5860**; GPU: **390**; CPU: **384**;
Kernel execution: **80,489 [ms]**
GPU-CPU page faults:
Device->Host: **768**; GPU: **598**; CPU: **384**;
Kernel execution: **38,671 [ms]**

What we can see is that when CPU function was invoked first and then the kernel there were on average 5000 more page faults. Thus as we can see kernel execution was much slower than if we would execute it the other way. Kernel execution just using GPU was the fastest because the operating system didn't have to manage the memory both on the CPU and the device.

Prefetching

Easier memory management comes at a cost. Unified memory is slightly slower than bare-bone malloc. But there's a way to counter the negative impact on performance - prefetching.
To understand the impact of prefetching data I've made ten measurements for each and calculated average:

- Initialize all data structures on GPU only

Kernel execution: **39,904 [ms]** base

- Prefetch one vector to CPU

Kernel execution: **112.41 [ms]** 2.8x slower

- Prefetch two vectors to CPU

Kernel execution: **59.943 [ms]** 1.5x slower

- Prefetch all three vectors to CPU

Kernel execution: **1,3911 [ms]** 28.7x faster

Prefetching part of the vectors that calculations are made on is considerably slower than without prefetching. Prefetching all vectors makes our calculations made over 28 times faster.