Aleksander Skraastad

Assignment 2

TDT4200

Programming with CUDA

Autumn 2015

Norwegian University of Science and Technology

Table of contents

Chapter 1

Theory

1.1 Problem 1 - Architecture and programming models

1.1.1 A - Architectural differences

NVidia Maxwell

The Maxwell architecture's workhose is the streaming multiprocessor, or shader unit. This in turn consists of several stream processors. The cuda cores are homogenous multiprocessors, consisting of many low-power cores, each with their own registers.

ARM big.LITTLE

This is a heterogenous architecture focusing on coupling slow low-power cores with powerful high-power cores. While both being connected to the same memory, swapping between cores on the fly becomes possible to meet performance demand changes.

The cores are arranged in two clusters, namely the high and low performance clusters. During swap, using a cache coherent interconnect, relevant data are transferred from one cluster to the other through the L2 cache.

This model allows heterogenous multiprocessing, where high performance tasks can be run on the "big" cores, where low performance tasks can be allocated to the "LITTLE" cores.

Vilje

Vilje is a distributed memory supercomputer system with 2 Xeon eight-core processors per node. Each "machine" exposes 32GB of memory, with 2 NUMA Nodes per machine, each with 16GB memory. Vilje is a typical MIMD supercomputer, able to run a multitude of difference tasks with different data spread over it's many cores.

Typical modern CPU

Typical modern CPUs are multicore processors with homogenous cores, like intels i-series desktop cpu's. They also support features like SMT (simultaneous multithreading, which Intel calls HyperThreading).

1.1.2 B - NVidia's SIMT

SIMT, which is short for Single Instruction Multiple Threads, is a parallel execution model that simulates multithreading on SIMD processors. The apparent execution of much more tasks than the amount of processors available is achieved by having multiple work items in lock-step. This is analogous to SIMD "lanes".

SIMT is used in the NVidia Maxwell architecture, where stream processors execute the same instruction on cores in groups of 32 in lock-step. An execution on a group of 32 cores is called a warp. By having many active threads, processor utilization is maximized, as they can quickly switch to a new thread when a thread becomes inactive while waiting for data.

1.1.3 C - Classification using Flynn's taxonomy

NVidias Maxwell architecture fits the SIMD and also SIMT models. This is because the way that the same instructions are run on multiple cores, but with different data. The SIMT addition is how the cores seem to perform more tasks than there are available cores, due to overflooding the processors with active threads, and executing in lock-step, reducing wait-time for new data.

ARM big.LITTLE i believe fits the MIMD description, as it supports heterogenous multiprocessing, where all cores are active and running at the same time. Running different tasks on different cores (with different data) based on their computational needs.

Vilje fits directly into the MIMD description, as do most of the top500 supercomputers today.

A typical modern cpu fits into the MIMD description, as a modern processor exposes multiple cores that can execute different instructions on different data in parallel.

1.2 Problem 2 - CUDA GPGPUs

1.2.1 A - Threads, Grids, Blocks

The term *Thread* represents the fine grain unit of parallelism in CUDA. It is the concurrent code and state executed on the processor. Threads are executed physically in parallel in groups called warps.

A block constitutes a group of threads that are executed together and constitutes the basic unit of resource management.

A grid is a group of blocks that have to be completed before the next phase of the program can start on the GPU.

In short, grids map to GPUs, blocks map to stream multiprocessors, threads map to individual stream processors, and warps are groups of 32 threads that execute simultaniously.

1.2.2 B - Algorithm

This seemlingly easy task revealed that its been a while since i did any math.

Here is my proposed solution, given the H values for CPU and GPU:

$$3500n * log_2(n) = 35n * log_2(n) + \frac{7 * n}{r}$$

$$100 * log_2(n) = log_2(n) + \frac{1}{5r}$$

$$100 * log_2(n) - log_2(n) = \frac{1}{5r}$$

$$99 * log_2(n) = \frac{1}{5r}$$

$$log_2(n) = \frac{1}{495r}$$

$$n = 2^{\frac{1}{495r}}$$

The $\frac{7n}{r}$ is the time needed to transfer the input data to the GPU ($\frac{2n}{r}$) and output data back from the GPU ($\frac{5n}{r}$), given the bandwidth of r data per time unit.

1.2.3 C - kernel1() and kernel2()

In this code example, kernel2() will run faster, as kernel1() will suffer from intra-warp divergence. When threads inside a warp disagree on execution path, the different paths are executed serially until all threads agree again and they converge and continue with the next instruction.

Kernel2 avoids this because it uses block index, and the provided dimensions are powers of 2 equal to or greater than the warp size (32).

1.2.4 D - CUDA Terms and their usages

Warps

As previously explained, a warp is a group of threads (32) all executing the same instruction in lock-step. Hence, to maximize utilization with regard to warps, we want to avoid conditions like the one presented in $kernel1()$, which have data-dependant branch conditions.

It can, however be difficult in real-world situations to avoid warp divergence altogether. But research has been performed that show significant performance boosts when performing pre-optimizations that group the branching threads into warps of their own, eliminating the intra-warp divergence.

Hence, all threads in a warp should agree on every instruction, to maximize stream processor utilization.

Occupancy

Occupancy is a measure of how efficiently a kernel is able to supply enough parallel work to the available stream processors. It is defined to be the number of active warps over the maximum number of warps. Since resources are allocated on a per-block basis, using too much resources per thread may limit the occupancy. Maximizing the global memory bandwidth requires us to have enough transactions in flight to hide latency. This can be achieved by either increasing occupancy or ILP.

The main limiting factors are register usage, shared memory usage and block size.

NVidia provides the CUDA Occupancy Calculator and Visual Profiler to aid the programmer in investigating memory bandwidth / occupancy.

Memory Coalescing

Memory coalescing can have different meanings depending on the context. In a CUDA program, memory coalescing means memory accesses that are consecutive from each thread. Instead of having multiple small memory accesses, they can be grouped into fewer larger accesses, thus increasing throughput.

Example of thread-consecutive memory accesses which are coalesced:

```
Thread 0: 0,1,2
Thread 1: 3,4,5
Thread 2: 6,7,8
Thread 3: 9,a,b
```

Example of non-coalesced access:

```
Thread 0: 0,4,8
Thread 1: 1,5,9
Thread 2: 2,6,a
Thread 3: 3,7,b
```

The latter is non-consecutive and drastically reduce memory throughput, as access is not coalesced into a single access.

Local Memory

Local memory space resides in device memory. Therefore it has low bandwidth and high latency as global memory.

A difference however is the organization of words, which are 32-bit and accessed consecutively by consecutive thread IDs. Memory accesses to local memory are therefore fully coalesced as long as all threads in the warp access the same relative address.

Local memory is used for example if $registerspilling$ occurs, and the kernel uses more registers than is available. Large structures or arrays that consume too much register space would also be placed into local memory by the compiler.

All in all, minimizing use of local memory, and in turn maximizing use of shered memory will increase the performance of the program by eliminating low bandwitdh-high latency accesses.

Shared Memory

Shared memory reside on-chip This gives low latency. Additionally, shared memory is divided into equally sized modules called banks, which allows high bandwidth under optimal access patterns. This however requires that memory accesses are distributed among the available banks, if the programmer wants to achieve maximum bandwidth.

If two accesses fall into the same bank, there is a bank conflict and the accesses have to be serialized, thus greatly reducing performance. Citing the CUDA Programming guide:

"The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause n-way bank conflicts."

Knowing the mapping between memory addresses and banks is therefore very important in order to maximize performance.

Shared memory is accessible from all threads in a thread block.

Chapter 2

Code

2.1 Transfer time results

Runtime results on $its-015-04$:

```
its-015-04:~/tdt4200/ex02/lenna$ ./gpu_version
H->D: 0.160896ms
Kernel: 0.055680ms
D->H: 0.176384ms
Tot transfer: 0.337280ms
```

Several runs were made but all yielded similar results with relatively small difference in runtime.

2.2 Optimizations of transfer

One can increase the performance of the program by utilizing page-locked host memory as described in the CUDA Programming Guide. If the dataset is small enough to not exceed the limitations of PL-host memory.