UPPSALA
UNIVERSITET

Assignment 1:
Using the CPU and the GPU

Accelerator-based Programming

Oskar Tegby

September 2022

1 Introduction

This assignment studies the computational bandwidths that scalar and vectorized implementations have on the CPU, as well as that of a parallelized implementation on a GPU using CUDA. The bandwidth is tested on the simple vector update. Namely

```
for (int i = 0; i < N; ++i)
   z[i] = a * x[i] + y[i];
```

which is completely vectorizable and parallelizable. In this report, we discuss the effects that the cache hierarchy and graphics memory have in this setting, since these vector operations are bandwidth-bound. All tests were run on the Snowy node of the UPPMAX cluster, as well as locally on an Intel i3-7100U.

2 Tasks

2.1 Task 1

The code generates two vectors, x and y, with random floating point numbers uniformly distributed between zero and one. It then performs the linear algebra operation $z = a \cdot x + y$ elementwise using a real scalar a and the said vectors. It repeats this operation twenty times using one for loop that, and one for the element-wise operations on the vectors. The former lets us average out any noise in the bandwidth measurements so that it does not affect our results noticably. Thus, we assume that there is no computational noise here.

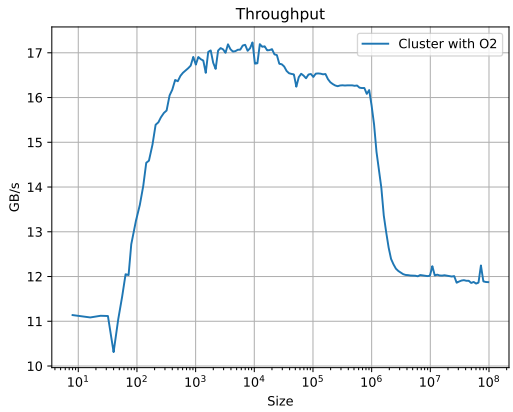
2.2 Task 2

The compilation flag O2 compiles the code without vectorization, and the 03 flag with it. This is visible since the peak throughput is about 17 Gb/s with the scalar O2 flag, whereas it is about 70 Gb/s with vectorizing O3 flag, as is seen in Figure 1.

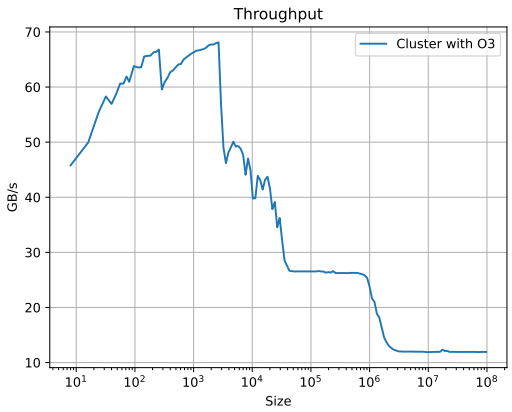
Assuming that the compiler vectorizes with 128 bit instructions, which each fits four floating point values (since a float is four bytes, and a byte is eight bits), this means that we get almost ideal speedup. Otherwise, if we are instead using 256-bit or 512-bit instructions, then what we observe is a poor speedup.

The reason why we would observe this near ideal increase in throughput is because the code is perfectly vectorizable, which means that we do not miss any out on any improvement because of poor memory alignment or poor memory accesses. If anything, then there should be some slowdown because of packing the scalar values into vectors, but this seems to be negligible here. The assembly code could be studied to figure this out for certain, but this was not done here (admittedly, increasing unnecessary doubt).

As seen in Figure 1b, we have distinct drops in throughput. The reason for which is that we first exhaust the L1 cache, then the L2 cache, and, lastly, the LLC cache, which introduces computational latency. We see these three distinct drops in Figure 1b. That decreases the throughput since we have to wait longer to fetch data since data not fitting in cache results in capacity misses. A fundamental fact of microelectronics is that the latency grows with the size of the memory. Thus, this effect gets worse as the caches fill up. However, this is something which we do not clearly observe here.



(a) Cluster with O2.



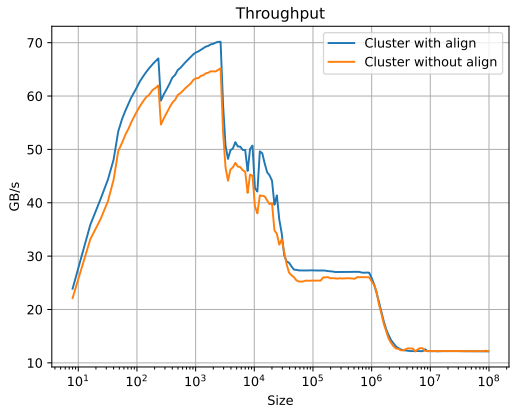
(b) Cluster with O3.

Figure 1: The cluster running the code compiled with the O2 and O3 compilation flags, respectively.

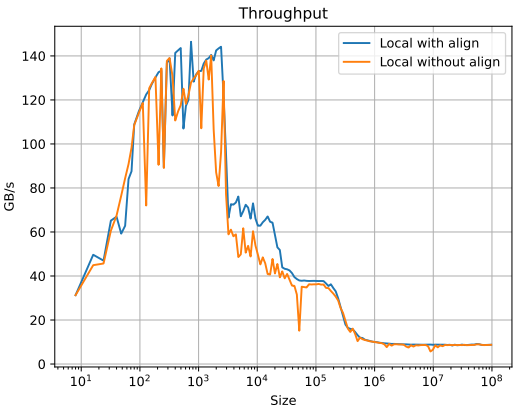
2.3 Task 3

Running the code with `-march=native` on our local machine, we get about 110 Gb/s throughput, which is a great increase in throughput when compare to the cluster. This is likely because of differences in computer architecture since the clockspeeds only differ by 200 MHz. For instance, some architectures lower the frequency when running SIMD instructions because their pipelines use a lot of energy.

As seen in Figure 2 , the effect of using the `align` runtime flag almost seems negligible both on the cluster and locally, which is counterintuitive given that the code is compiled with the O2 flag. The reason for which is that it does not enable any vectorization, and, thus, adding it manually should result in about the same speedup that the O3 flag gives the code. This means that the results should be strikingly similar, but they clearly are not. The most likely explanation is that the flag was not used correctly.



(a) Cluster with and without align.



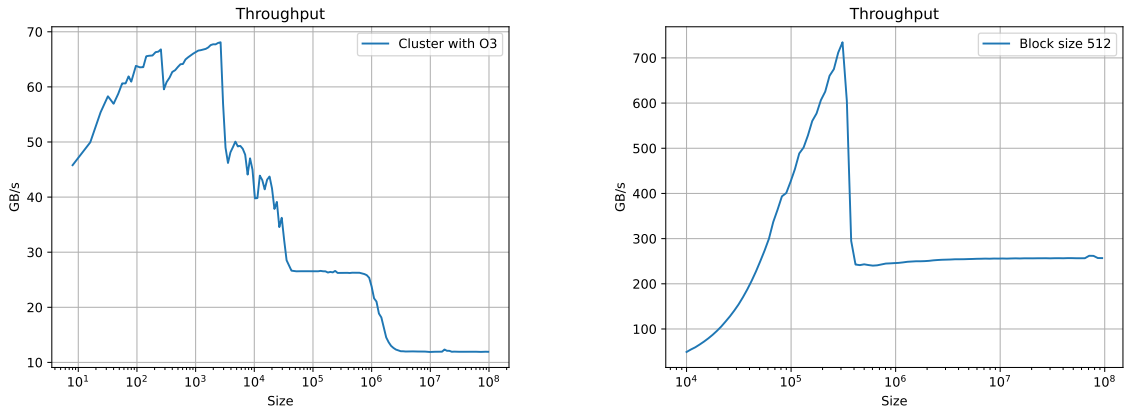
(b) Local with and without align.

Figure 2: The cluster and local compiled with O2, and run with `-align 0` and `-align 1`, respectively.

However, one result contradicting that in Figure 2 is the fact that the local computation without align has a throughput which varies much more compared to the aligned version of the code, which supports the idea that alignment actually took place. Presumably, there is some way to investigate this closer.

2.4 Task 5

As seen Figure 3b, the performance is initially small and then peaks just before it drops significantly. The reason for which is that there initially is too little data to efficiently use most of the threads of the graphics card, and later on there is too much data in order to fit in the L2 cache, which means that we have to transfer the data from the main graphics memory instead. That severely limits the performance, as we can see by it flat-lining just below the main memory bandwidth, which is 320 Gb/s.



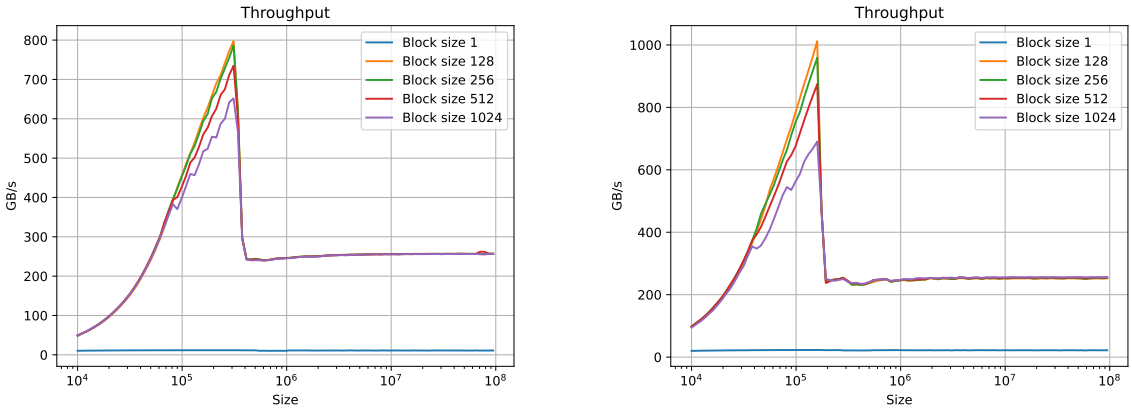
(a) Cluster with O3.

(b) CUDA with block size 512.

Figure 3: Comparison between SIMD and CUDA.

2.5 Task 6

As seen in Figure 4a, a block size of one is not large enough since we cannot use the graphics card as intended with that little work per computational unit. Graphics cards have thousands of streaming multiprocessors (SM) which are split into blocks (who are, in turn, split into grids). The block size sets the number of elements each block works on, so we need to set it high enough to keep each SM of the block busy. This elementarily explains the behavior that we see in Figure 4.



(a) CUDA with floating point precision, GB/s.

(b) CUDA with double point precision, GB/s.

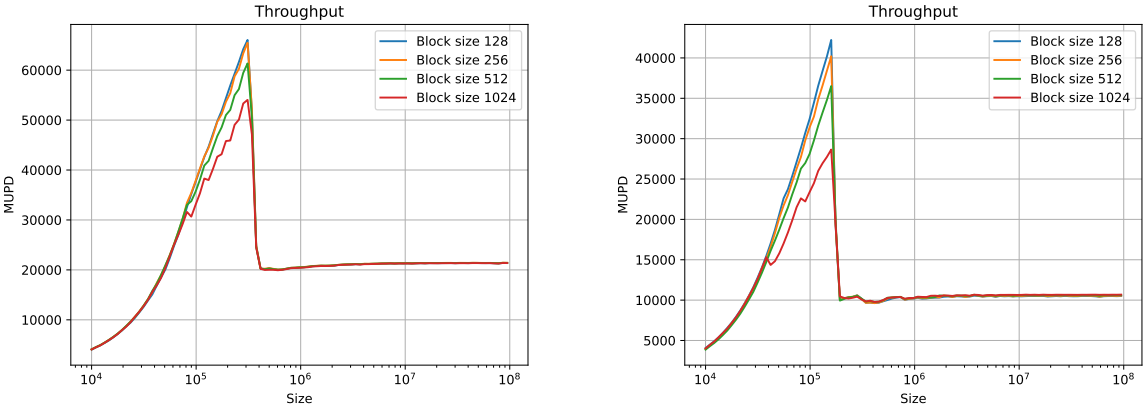
Figure 4: CUDA using block sizes 1, 128, 256, 512, and 1024 with `single` and `double` precision.

When we increase the block size, we get greater throughput until we hit the block size limit of the graphics card, which is 1024 on the graphics cards on Snowy. The reason, aside from asignning enough work to each block, is that each block has a shared memory which is faster than accessing the rest of the memory hierarchy. Thus, working within that memory is faster than working across different blocks. Hence, we should use the maximum block size in order to use as much of that memory as possible.

2.6 Task 7

Doubling the data size increases the throughput, but it decreases the number of updates per second since we are able to transfer fewer numbers when we double the precision, as is seen by comparing the height of the peaks in Figure 5a and 5b. That is, we can send more data with double precision, but not the double that we need in order to compute with double precision. Hence, it is clear that we are memory-bound in this setting (as opposed to being compute-bound).

The Roofline model states that we either are in one or the other bound, which makes sense as we either lack the ability to compute the data that we get, or that we do not get the data fast enough to begin with. Here, it is easy to perform operations, so the need for bandwidth is the only thing that can limit us. That is not usually the case. Note that we are measuring the throughput in gigabyte per second and million updates per second in Figure 4 and 5, respectively.



(a) CUDA with floating point precision, MUPD.

(b) CUDA with double point precision, MUPD.

Figure 5: CUDA using block sizes 1, 128, 256, 512, and 1024 with `single` and `double` precision.