UPPSALA
UNIVERSITET

Accelerator-Based Programming

Assignment 1
Uppmax Environment

Jinglin Gao

October 2, 2023

1 Two Different Levels of Optimization

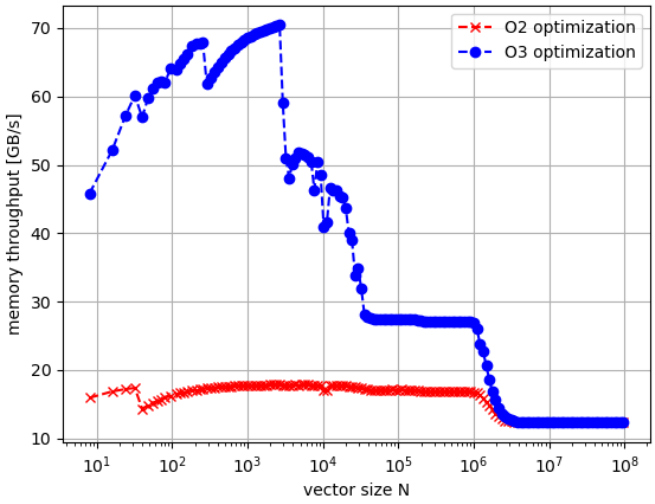

Figure 1: O2 optimization vs O3 optimization

In the result, we can see that O3 optimization gets way better performance than O2 optimization when the vector size is lower than 1e6. It is almost 3.5x faster when we use O2 optimization. This is mainly because O3 optimization looks for opportunities to do vectorization, but O2 optimization doesn't look for vectorization. So with O2 optimization is likely to generate scalar code, which means it will operate on individual data elements one at a time.

On the other hand, O3 generates vectorized code and can take better advantage of CPU SIMD instructions. When we do vectorization, we will perform operations for several data items at once. In other words, we will load a bunch of data and do the arithmetic operation on them at the same time.

Cache Information of CPU on UPPMAX		
Cache	Size	Theoretical drop size
L1 Cache	32K	2730
L2 Cache	256K	21845
L3 Cache	20480K	1747626

In the figure of O3 optimization, we can see there are three significant drops. This reflects the size of the L1 cache, L2 cache, and L3 cache. The CPU we are using in UPPMAX is Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz, the cache information is shown in the table above. In each computation, we need to access data from vectors x, y, and z, so our cache size should divided by 12 bytes to calculate when the theoretical drop should happen. In the result, we can see that the actual drop aligns with the expectation. When we need to fetch data from the L2 cache or L3 cache the throughput is significantly dropped because the latency is higher.

2 Run on Local Hardware

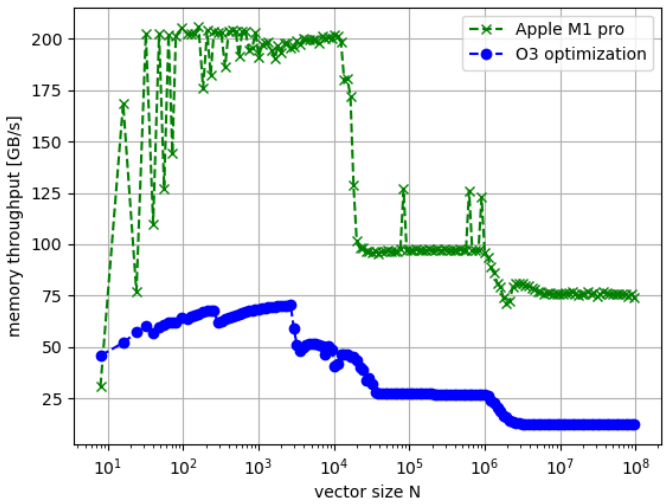

Figure 2: O3 optimization on local vs O3 optimization on Uppmax

Cache Information of Local Laptop		
Cache	Size	Theoretical drop size
L1 Cache	128K	10922
L2 Cache	24MB	2097152

Here again, I compile and run the same code on my local machine. The **-march=native** flag instructs the compiler to generate code optimized for the architecture of the host machine. It takes advantage of all the specific features and capabilities of my CPU. And it results in highly optimized code. Also here we use **-O3** flag, which enables aggressive compiler optimizations, including loop unrolling and vectorization. This makes the code finely tuned for the M1 Pro's architecture, making efficient use of its capabilities. M1 Pro offers up to $200GB/s$ of memory bandwidth, which we can see in the figure when the data size fits in the L1 cache we can achieve the maximum memory bandwidth. From the result, we can see that the performance on my local computer is approximately 2.8x that of the one run on UPPMAX.

Again the significant drop in the figure reflects the cache size of my local computer. It is obvious that I have a larger cache size in my local laptop than the core in UPPMAX has. The memory bandwidth for my local laptop is around 200 GB/s.

3 Aligned Access vs UnAligned Access

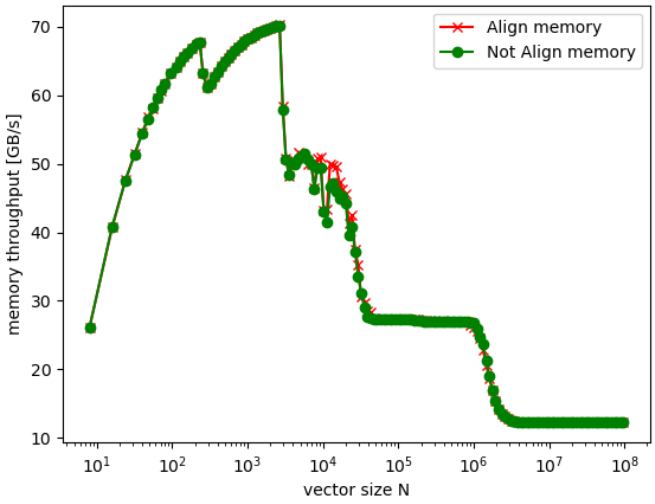

Figure 3: Align vs No Align

Here in the result, we get similar performance between aligned access and unaligned access.

In theory, SIMD load and store instruction are influenced by the memory address. Aligned memory access makes sure the pointer address in bytes is divisible by the size of SIMD vectors in bytes. While unaligned memory access the pointer address divided by SIMD vector length has a reminder. Aligned access is faster due to the organization of memory while it might increase memory usage as a penalty. However, unaligned access needs the CPU to fetch two separate parts of data and concatenate them which will end up in performance loss.

But the modern compiler makes the difference not that significant in this situation. It automatically optimizes the memory address so we got a similar result here.

Since my local laptop is Apple M1 Pro which doesn't use X86 architecture, the **vectorization.h** file won't work on my local laptop. So I didn't test the result on my own laptop.

4 CPU vs GPU

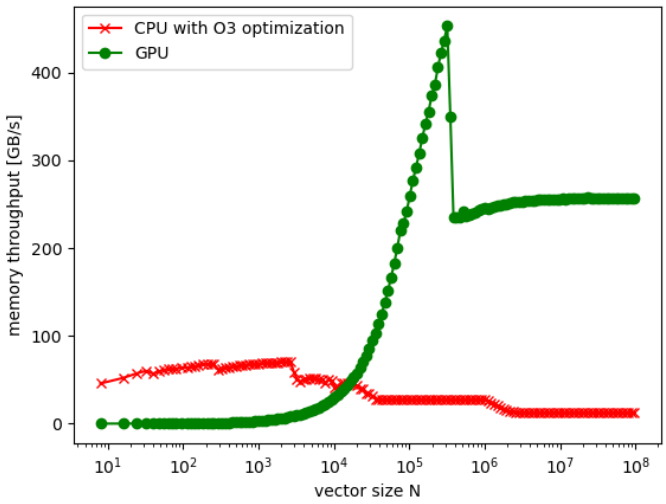

Figure 4: CPU vs GPU

The computing performance is mainly limited to the memory bound.

For small sizes, in the result, we can see obviously CPU has better performance. CPUs have larger and more complex memory hierarchies, which helps CPUs quickly access data. So for a small size of vector computations, the data can fit into caches, which gives the CPU an advantage to have better performance.

But for large sizes, GPU has way better performance than CPU. First of all when it reaches the bounds of the L2 cache, it takes more time for the CPU to fetch data from the L3 cache. But for GPU, when the vector size is big, we can make sure that we can fulfill the blocks with some job and the context switching between blocks is kind of balanced because we have more jobs in each block. This makes GPU more efficient at the task. GPU has bad performance for small sizes calculations cause there might be too much overhead and inefficient memory access patterns.

The drop in GPU performance is because it reaches its memory bound and needs to fetch data from RAM, which leads to a significant performance loss. For GPU, we do not consider the L1 cache. The GPU we are using in UPPMAX is Tesla T4 with a L2 cache of $4MB$, so we can expect that the performance drop will happen when the vector size reaches $4MB \div 12Bytes = 349525$, which aligns the drop in the graph.

5 Different Block Size

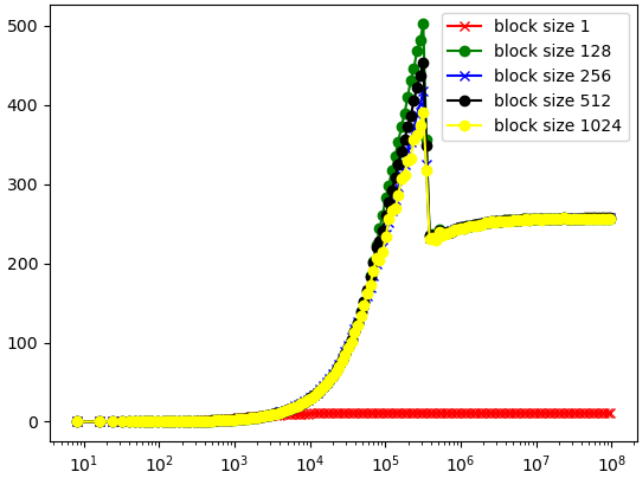

Figure 5: Different block size on GPU

The logic behind GPU programming is that we have a grid that contains a bunch of blocks and each block contains a bunch of threads. The block size is configurable, its optimum depends on the application and hardware. The block size defines how many threads are executed concurrently within a single GPU block.

Smaller block sizes result in more frequent context switches because there are more thread blocks available for execution, potentially leading to increased overhead. Context switches can be relatively expensive, particularly if they involve transferring data between global memory and the GPU's execution units.

Increasing the block size can improve performance if the GPU has enough resources to support the larger block. Also, a larger block size leads to higher occupancy which often leads to better performance.

Here in the result we can see the trade-off between getting all the blocks have work and every block context switching with each other. When different blocks communicate with each other they have to wait and synchronize it, this will lead to performance loss. The block size with 128 gives the best performance.

6 Single Precision vs Double Precision

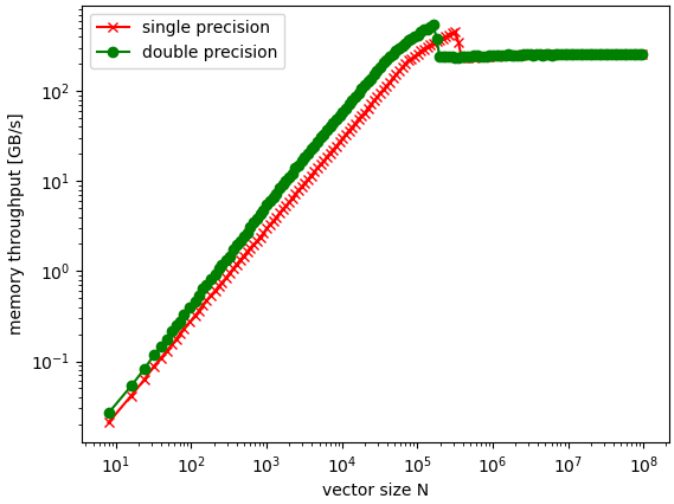

Figure 6: Compare single precision and double precision

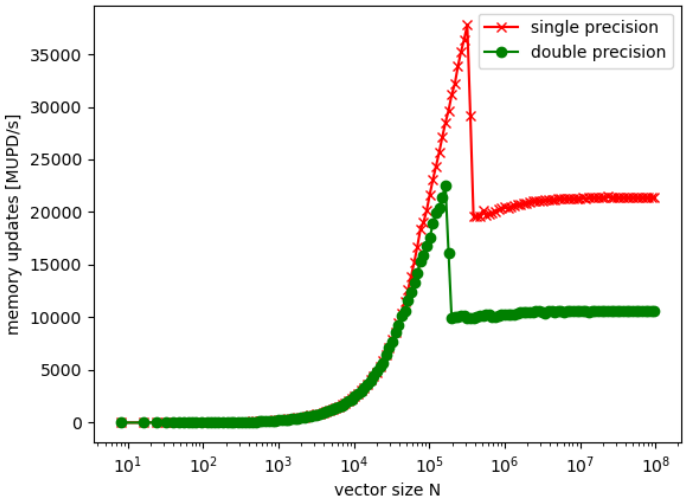

Figure 7: Compare single precision and double precision

We can see that when we port the GPU program from single-precision to double-precision, it only makes a difference for vector size between 1e4 and 1e6. Single precision gives a better throughput when the vector has a medium size.

When we computing with the vector has small size, the data can fit in cache and can be accessed fast through the program. So there is no big performance difference when the vector has small size. But as the size of vector growing, it reaches the memory bound and we have to fetch data from RAM, which leads to loss of performance.

GB/s measures memory bandwidth—the rate at which data can be read from or written to GPU memory. Transitioning to double-precision calculations may increase memory bandwidth requirements due to the larger data size of double-precision values. This can lead to a decrease in GB/s compared to single precision. In my case, for double precision the memory throughput drops at a lower size compared to single precision. This reflects the truth that larger data size reach the

memory boundary faster. But after it reaches the limit of cache size, both of them need to fetch data from RAM, so the speed is similar to each other.

The **MUPD/s** metric for double precision operations will be double times slower than that for single precision because each double precision calculation requires more processing power and memory bandwidth. Also, the GPU's double precision processing units may be fewer and slower compared to single precision units, further impacting performance.