UPPSALA
UNIVERSITET

Accelerator-Based Programming

Assignment 4
KOKKOS

Jinglin Gao

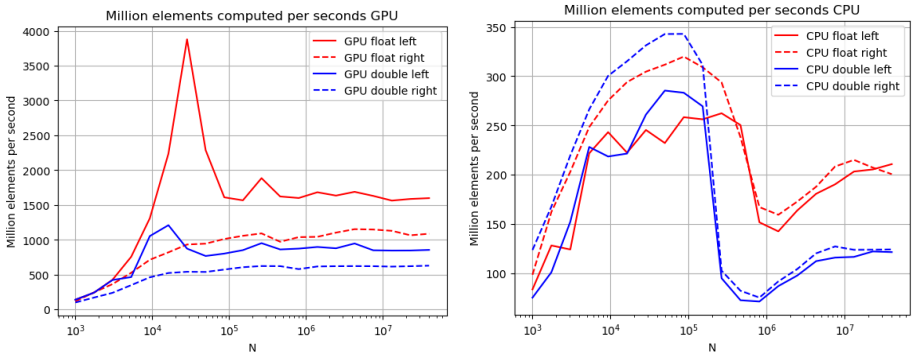
October 11, 2023

1 Performance Comparison of Float and Double Numbers

In the first task, the calculation was performed in both float and double precision to compare the different performances they achieved for both CPU and GPU. The range of elements is between 1000 and $4 * 10^7$. We measure the performance by calculating the number of million elements computed per second, memory bandwidth, and floating point operations.

The figure below shows the results of the million elements computed per second on the CPU and GPU. The experiments are performed for both layout right and layout left. When we change the layout type we expect different results from the CPU and GPU since they have different memory access patterns. For the CPU the memory access pattern is caching, and the threads are independent. We would expect **LayoutRight** will have better performance on CPU.

On the other hand, the GPU is coalescing so all threads in a group (or wrap) must finish their loads before any thread can move on. In this case, we would expect **LayoutLeft** to have better performance on GPU.

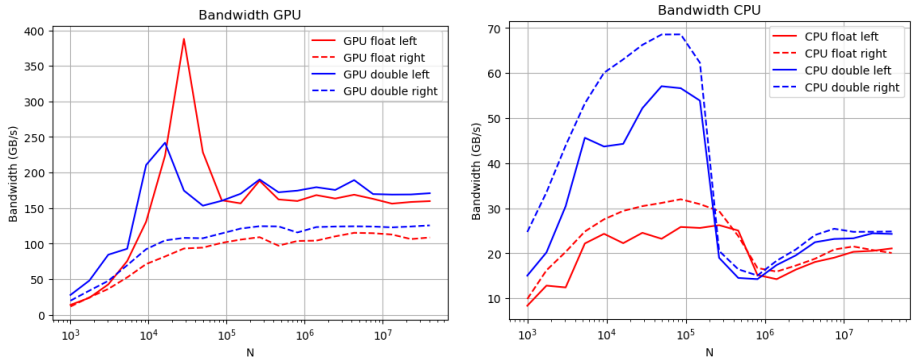


(a) Mecps on GPU (b) Mecps on CPU

Figure 1: Million elements computed per second

In the result, we can see that for the GPU we do get better performance when we use **column-majored** memory style. Basically for **LayoutLeft** the performance is about 1.6× faster than **LayoutRight**. Also when we are using single precision the overall performance is around 2× faster than when we are using double precision. This is aligned with our expectations because double precision is double sized than single precision. It will take a longer time to fetch data from memory.

For the CPU, for both single and double precision we get better performance for **row-majored** memory style. This is aligned with our expectations. But in my result, for the small size of elements, the double precision has better performance than the single precision. This is not what we expected from the result. Theoretically, float precision should have almost 2× better performance than double precision. One possible reason for this result might be our architecture is optimized for double-precision operations.



(a) Memory Bandwidth on GPU (b) Memory Bandwidth on CPU

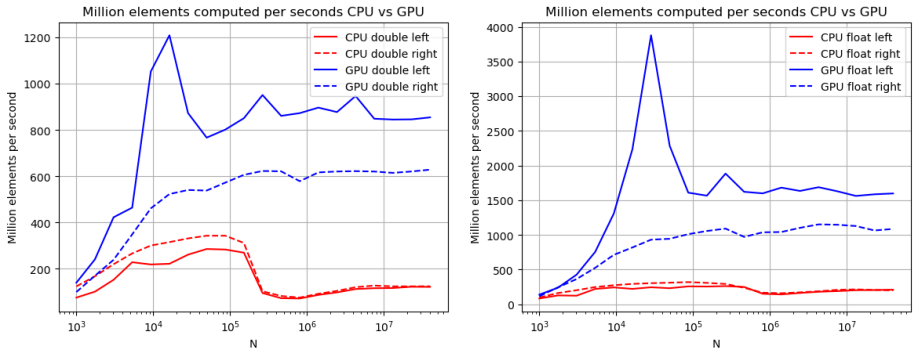
Figure 2: Memory Bandwidth

The figure above shows the results of the memory bandwidth for both CPU and GPU. We can see that for the GPU left layout with double and float precision the memory bandwidth is settled at around $160GB/s$, and for the right layout with double and float precision, the memory bandwidth is settled at around $125GB/s$. From the previous assignments we already know we are using Tesla T4 and its L2 cache is 4MB. So we would expect there is a drop at around the elements number is 40000 when we are using single precision and 20000 for double precision.

Cache Information of CPU on UPPMAX			
Cache	Size	Theoretical drop size (float)	Theoretical drop size (double)
L1 Cache	32K	320	160
L2 Cache	256K	2560	1256
L3 Cache	20480K	204800	102400

The chart above shows the information about the cache size on the CPU. For the CPU we get the highest memory bandwidth for double precision with the right layout. We can observe a significant drop at `N` around 10^5 this is because we are running out of the L3 cache and start fetching data from RAM.

We get similar results of floating point operations (GFlop/s) with a million elements computed per second since the formula we need to compute the floating point operations is just simply multiply 84 (the number of operations we need to perform for one element) by the million elements computed per second.



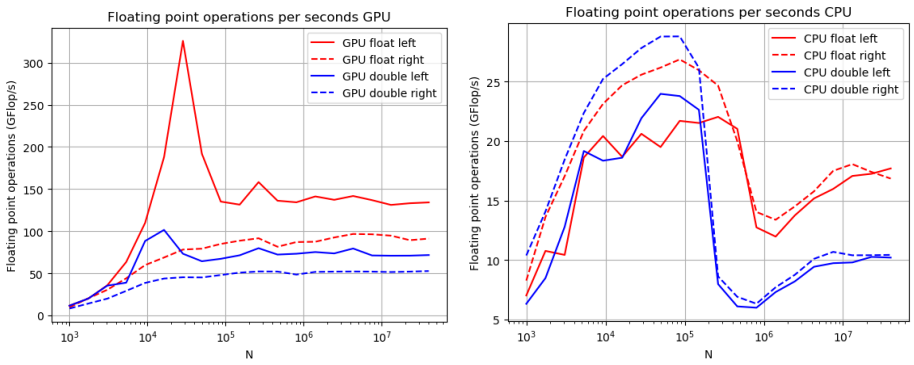
(a) Mecps Compare (float) (b) Mecps Compare (double)

Figure 3: Million elements computed per seconds CPU vs GPU

If we compare our results of CPU and GPU we can see clearly that GPU has better performance than CPU for both single and double precision.

2 Predict the Achievable Performance

Let's first take a look at the achieved performance we get for both GPU and CPU. This figure is similar to the one for the million elements computed per second. If we look at this result we will notice that the limiting resource is memory bandwidth cause the GFlops does not increase after it achieves its "maximum".



(a) Floating point operations on GPU

(b) Floating point operations on CPU

Figure 4: Floating point operations

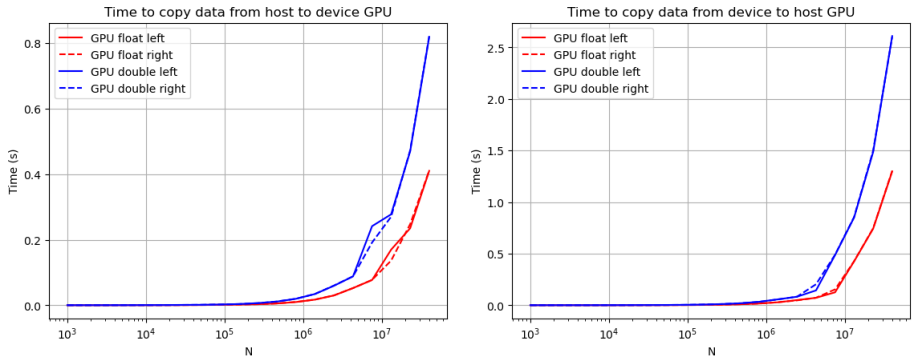
Here we just talk about the performance we achieved by using left layout. From the datasheet, we know that for single precision we can expect to achieve $8.1TFLOPS$, but in our implementation, we can at most achieve around $300GFlops$. This clearly shows that we are not reaching the limit of the compute-bound. Also from the datasheet, we know that the maximum memory bandwidth is around $300GB/s$, but for our implementation, our memory bandwidth settles at around $160GB/s$, we also know from the previous assignment that even for the simple implementation we won't achieve the maximum $300GB/s$ so here the result we got is align with the expectation and for the whole problem is memory bound.

One of the reasons that we are not getting better performance is probably we have not only multiplication and addition but also division and subtraction. The GPU is optimized for multiplication and additions but division and subtraction are slow operations. So that might be the reason we can not reach a better performance.

In conclusion, we didn't achieve the best performance for either memory bandwidth or computational capabilities. This is probably because in this algorithm when we compute each element we need both a lot of memory access and a lot of computing resources. So when we finish fetching data from the cache the GPU is just busy with computing and writing back so we cannot achieve the best performance.

3 Transfer time between host and device

Since we are using GPU to compute we need to transfer the Jacobi matrix before the computing starts and transfer the result back after the computation ends. We want to measure the time we need to transfer between host and device and compare it with the actual computing time to see if is worth it to use GPU.



(a) Transfer time: host to device

(b) Transfer time: device to host

Figure 5: Transfer time between host and device

From the result, we can see that we need more time to transfer the result matrix back to the host. In more detail, we need 3× more to finish transferring back data to the host than transferring data to the device. This is expected because the resulting matrix has a bigger size than the Jacobi matrix. But in theory, it should just be 1.78× more to transfer data back obviously in my implementation we are spending more time on transferring.

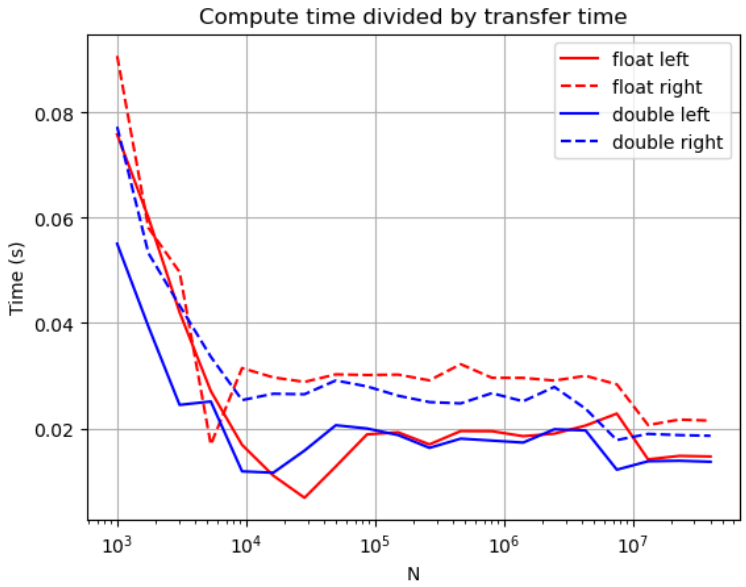


Figure 6: Computation time divided by transfer time

Then I plot the computation time divided by the total transfer time. We can see in the plot that the transfer time is significantly larger than the time we use to do the actual computing. So if we want to know if we actually get better performance on GPU we also need to compare the total time we used to perform a GPU computation and the time we need to perform CPU computation.

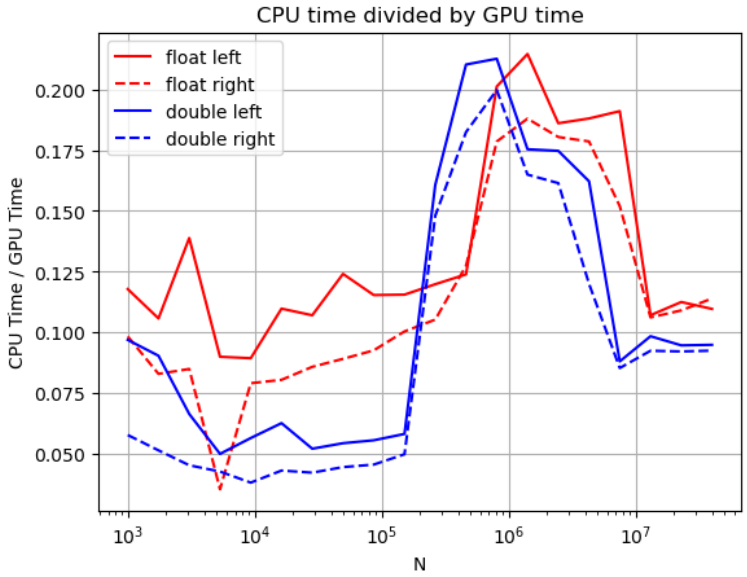


Figure 7: CPU Computation time divided by GPU Computation time

From this figure, we can see that we are actually using more time to compute the elements when we are using GPU. This means that this algorithm is not worth using GPU since we are spending too much time on transferring data back and forth.

4 Full Finite Element Pipeline

For the full finite element pipeline, if we want to know which step should be run on GPU and which step should be run on CPU we would want to know if the computation time is that significantly small so that we want to take time to transfer data between host and device. From the last question we already know that for this algorithm it isn't worth transferring data back and forth because the time we spent on transferring is significantly larger than the computing time. And the CPU has the ability to perform the computation. So in the whole pipeline maybe the only step worth using GPU is **the linear system is solved with an iterative solver** cause this step just needs us to transfer data one time from host to device and the computational load is kind of heavy.

For the other steps, we need to access memory a lot to either store data or fetch data and also the computational load is not heavy enough so we don't want to use GPU since it won't give us much better performance than CPU if we also consider the time to transfer data back and forth.