UPPSALA
UNIVERSITET

Assignment 4: Kokkos

Accelerator-based Programming

Oskar Tegby

October 2022

1 Introduction

This assignment studies the computational bandwidths that a computational kernel have on the GPU when implemented using Kokkos. The bandwidth is tested using finite element matrices arising from a 3D discretization of the Laplacian on a mesh of simplices. Notably, the computations using these matrices are completely parallelizable. In this report, we discuss the performance that we achieve using various settings in Kokkos. All tests were run on the Snowy node of the UPPMAX cluster.

2 Tasks

2.1 Task 1 and 2

In Figure 1, we observe the measure million elements computed per seconds (MELEM/s) computed using the layout which is appropriate for the corresponding device. That is, for CPUs we want `LayoutRight` (row-major) to get contigous memory loads, and for GPUs we want `LayoutLeft` (column-major) in order to coalesce the memory accesses.

Given that application is fully parallelizable and lacks divergent computations, the computations on the GPU should be performed much faster than on the CPU, which has much fewer ALUs running in parallel. However, somewhat surprisingly, the GPU performs better than the CPU by a factor 10 in our measurements. This observation strongly suggests that the memory bandwidth is limiting the performance.

In order to examine the veracity of that belief, we study Figure 2a and Figure 2b. There, we observe that the peak throughput is about 210 GB/s for the GPU, and 50 GB/s for the CPU. However, here, it is crucial to recall that the ratio between the data transfer and the compute dictates the performance on the graphics card. That is, we may have a higher throughput on the GPU than the CPU, but if we only perform a few floating-point operations on each data, then that will not matter.

2.2 Task 3

The theoretical throughput of this GPU is 320 GB/s, and we are loading 9 floats for the J matrix since it is 3×3 and 16 floats for the A matrix since it is 4×4. Thus, we are loading 25 floats, each of which are 4 bytes, for every of the N matrices.

Thus, we load the data from the CPU's RAM to the RAM of the GPU in

$$\frac{N \cdot 25 \cdot 4}{260} = 2.56N$$

nanoseconds over the PCIe interface. Furthermore, the computations on the GPU take

$$\frac{N \cdot 68}{8} \approx 12N$$

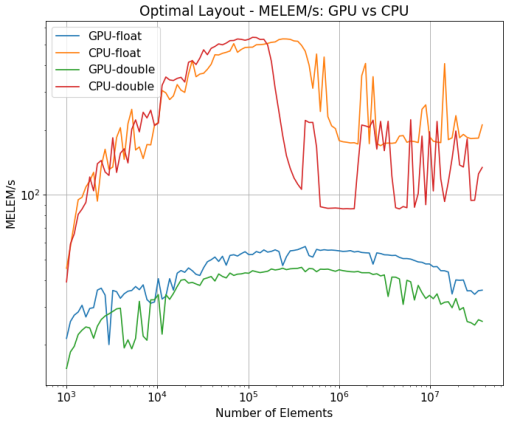
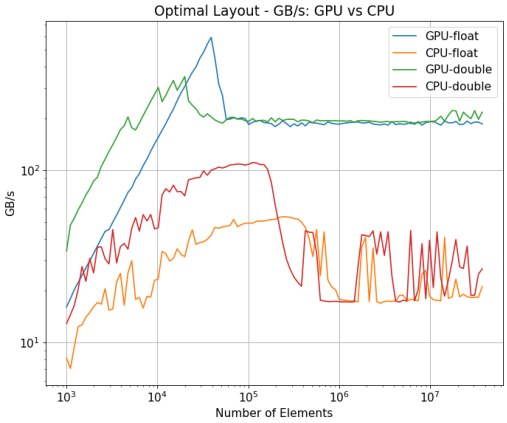
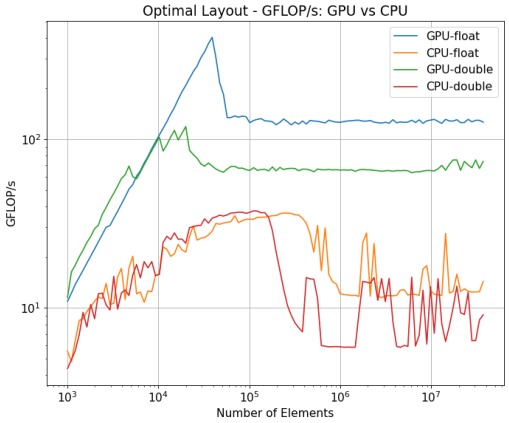


Figure 1: Optimal layouts on the CPU and GPU, respectively, using `single` and `double` precision.



(a) The throughput of the computation.



(b) The FLOPS of the computation.

Figure 2: The throughput and FLOP/s achieved during the computations.

picoseconds since the computational power of the GPU is 8 TFLOP/s and we are performing 68 FLOP per iteration. Clearly, the limit lies in the interconnect since it is a factor of a thousand higher than that of the computation time.

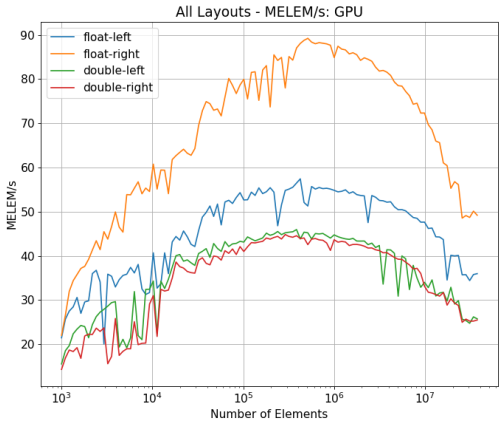
2.3 Task 4

In order to study the differences in data layouts, we study Figure 3a and Figure 3b. The theory tells us that `LayoutLeft` should be better for the GPU since it naturally coalesces the memory accesses, and that `LayoutRight` should be better for the CPU since it instead gives contiguous memory accesses.

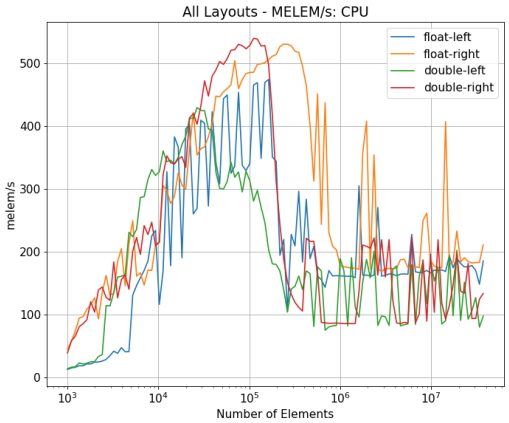
Again, surprisingly, we observe in Figure 3a that the opposite results are observed for the GPU compared to our expectations. A possible answer could be that the memory accesses somehow are misaligned because of the misaligning memory accesses when the CPU is sending the data to the GPU. However, that is not what is supposed to happen, so we are left unable to explain this behavior.

In Figure 3b, we instead observe the expected behavior, that `LayoutRight` indeed results in a higher throughput. In both cases, we, of course, observe that floating-point precision trumps double-point precision since we divide the size of the data being sent in half. We also observe that ratio somewhat

in Figure 3a where the memory hierarchy is less pronounced, but less so in Figure 3b. There, we instead observe that the spikes resulting from cache accesses occur at different problem sizes instead. That is, the graphs are clearly shifted rightwards instead of upwards for the CPU.



(a) The GPU with LayoutLeft and LayoutRight.



(b) The CPU with LayoutLeft and LayoutRight.

Figure 3: The MELEM/s on the GPU and CPU with `single` and `double` precision, respectively.

2.4 Task 5

In order to study the ratio between the compute and the transfer, we plot the transfer time between the CPU and the GPU, and the execution time of the kernel itself when the data is on the GPU. We find these results in Figure 4. There, we observe that the transfers back and forth dominate the execution time.

Here, using the notation **l** for `LayoutLeft` and **r** for `LayoutRight`, as well as **f** for `float` and **d** for `double`, we observe that the transfer of floating-point precision data with `LayoutLeft` is the fastest one, and that double precision with `LayoutRight` is among the slower ones. That is to be expected as we now have discussed at length.

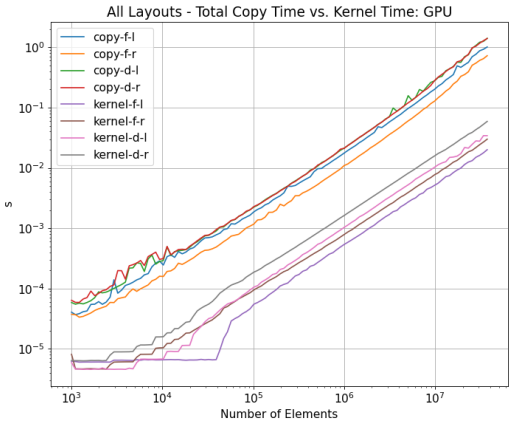


Figure 4: The time it takes to transfer the data and execute the kernel, respectively.

Notably, the former contradicts the earlier results showing that the usage of floating-point data gave a lower throughput, which supports the notion that there is some other factor at play in that scenario.

2.5 Task 6

The issue here clearly lies in the ratio between memory transfer and the amount of computations performed on the data being sent back and forth. This means that we should not off-load computations to the GPU unless there is a significant amount of computations being done on each element.

Here, this motivates keeping the computation of the Jacobians on the CPU since it would be absurd to sense so little data to compute them on the GPU. The reason is that we would not need the vertices and connectivity information after we have computed the Jacobians, so sending it that data will be useless for the rest of the computation. Furthermore, the vertices are likely to affect more than one value in the Jacobian, so that having a more advanced pipeline which better handles divergent computation and unorderely data movement yields a greater impact on performance. However, that will be insignificant here when compared to the transfer time of the data itself.

As observed in Figure 1, the CPU beat the GPU in this setting, so it is unreasonable that this part of the pipeline should be off-loaded to the GPU. However, when computing sparse matrices, we do not need to send nearly as much data due to the sparsity. Thus, it is plausible that they will be computed much faster on the GPU if they are large enough, given the usage of data structures for sparse matrices.

Lastly, solving a linear system with an iterative solver is a typical application for the GPU since we need to perform many operations on the same data, often containing several parallel steps such as in conjugate gradient (as used in FGMRES). Thus, it should be off-loaded on the GPU if enough iterations are needed to solve the linear system of equations.