UPPSALA
UNIVERSITET

Accelerator-Based Programming

Final Project
Iterative Solver

Jinglin Gao

October 22, 2023

1 Sparse Matrix Storage Formates

1.1 Compressed Row Storage

CRS is a popular sparse matrix storage format on CPUs, it is a cache-friendly layout ensuring consecutive data access to the matrix elements and the column indices. The matrix is stored row-by-row in the memory. However, CRS is not friendly to GPU implementation. For GPU each thread in a wrap operates on different rows and accesses elements concurrently that are not consecutive in memory. This will limit the performance a lot since it increases the cache misses.

1.2 SELL-C-sigma

This storage format addresses the problems of CRS for GPUs, instead of storing matrix row by row it has a column-wise data layout and padding rows in each chunk to the same length. This assures a coalesced data access that is GPU-friendly. The SELL-C-σ format is an improved version of the ELLPACK format which substantially reduces the padding overhead and increases data locality between successive column computations within a wrap.

The idea of SELL-C-σ is to cut the matrix into equally sized chunks of rows with C rows per chunk, and zero-padding the rows in each chunk to match the length of the longest row within their chunk. All elements within a chunk are stored consecutively in column-major order, and all chunks are consecutive in memory. The rows of matrix N have to be padded to a multiple of C.

1.3 Benefits of SELL-C-sigma over CRS

There are several benefits of using SELL-C-σ over the CRS storage format. The most benefit is that the data are stored in column-major which allows a coalescing access pattern, this gives a huge advantage when we are using GPU to calculate matrix-vector products. For both methods each thread in a wrap works on an individual row of the matrix, however, SELL-C-σ provides GPU-friendly memory access, and it's much faster.

In the figure below there is a simplified illustration of how threads access data. As we can see in CRS the elements of a row lie right next to each other in memory and each thread accesses the first element of their row, which are not next to each other. This is not good for GPU since every thread in a wrap has to access data at the same time. For the SELL-C-σ format since the first element of each row is stored next to each other, this gives benefit for threads in a wrap to access them and decrease the cache misses.

Also, in our case, we are going to solve a linear system with the conjugate gradient method. SELL-C-σ storage format provides several benefits for it over CRS. In the CG method we need to repeatedly access rows of the matrix for dot product operations, the SELL-C-σ format has a better cache locality and hence can reduce cache misses during row accesses.

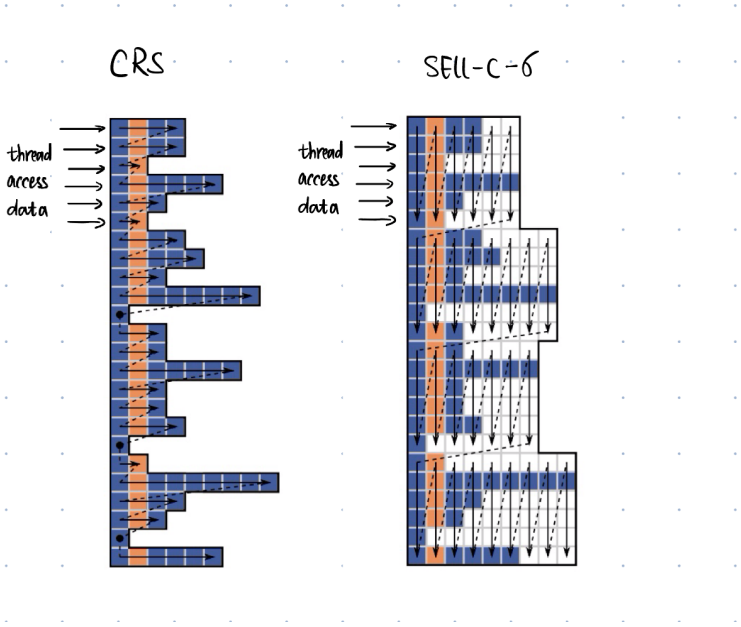
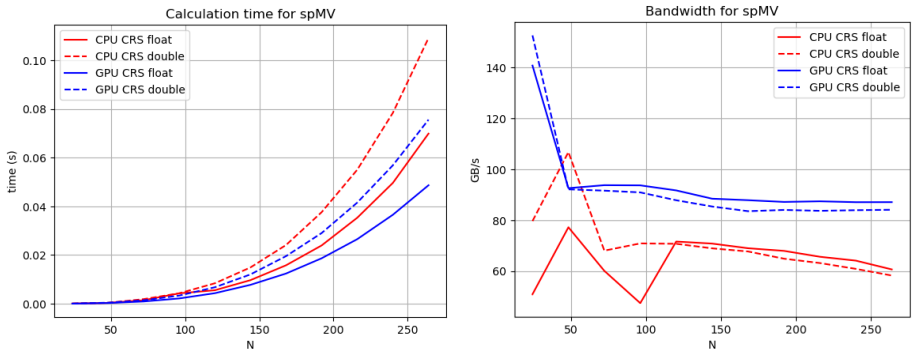


Figure 1: Data access pattern for CRS and SELL-C-σ

2 Performance of CRS on CPU and GPU

We run a single sparse matrix-vector multiplication on both the CPU node and the GPU node and compare their performance. For the CPU node, we use OpenMP to parallel the program and we set the threads to 16. The figure below shows the result of the experiment.



(a) Calculation time

(b) Memory bandwidth

Figure 2: Run a CRS sparse matrix-vector multiplication on CPU and GPU

As we can see, the CPU node for both float and double operations gives a worse performance. Also it shows in the figure that it always takes longer time for double precision operation than single precision operation no matter on the CPU node or the GPU node.

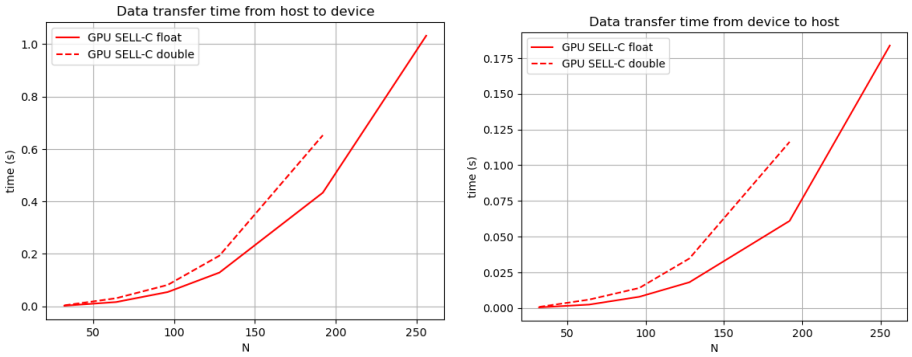
We also have the achieved memory bandwidths displayed here. Clearly the GPU node has a better bandwidth achievement than the CPU node. We can see that the maximum memory bandwidth that CRS can achieve is around $140GB/s$ on the GPU node. However, as we know from the previous assignment, the best performance our GPU can achieve is up to around $270GB/s$, apparently with the CRS method we can not achieve it. As we discussed above, the CRS is a non-coalesced method so is not preferred by GPU.

3 Performance of SELL-C-σ on GPU

I implemented this improved data storage format SELL-C-σ on the GPU node and ran experiments to see what performance this method can achieve.

In this project, C is equaled to 32, and σ is set to 1, so we assume there is no sorting. In this method, theoretically, we have to pad the number of matrix rows N to a multiple of C (chunk size) to make sure we can divide the matrix into equal chunks. But here in my implementation, I didn't come up with a padding code due to the limited time, so my code only works for matrices whose row number is a multiple of the chunk size.

For convenient reasons, I first convert the sparse matrix in CRS format to SELL-C-σ format on the host, and then transfer the data to the device and do the matrix-vector multiplication on the device, and transfer the result back to the host. The figure below shows the transfer time we need from host to device.

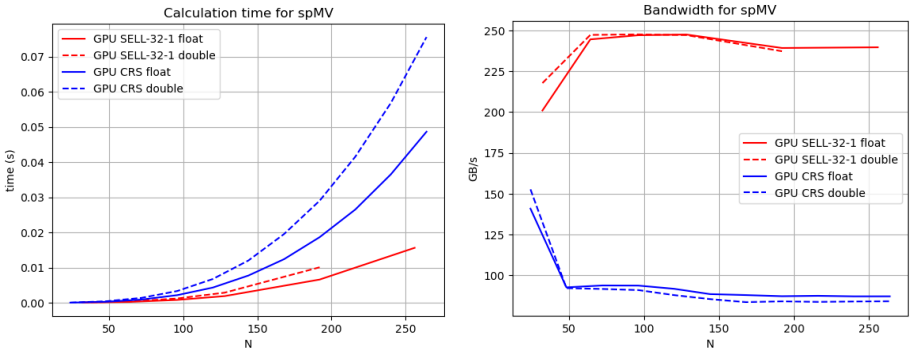


(a) Transfer time from host2device (b) Transfer time from device2host

Figure 3: Transfer time between host and device

The time we need to transfer the result vector back is significantly less than the time we need to transfer the matrix to the device. When we transfer data from the host to the device we are transferring the matrix, so we will need more time than when we transfer the result back to the host.

Same as the CRS method, we measure the performance of SELL-C-σ with the calculation time and the memory bandwidth. The figure below shows the results.



(a) Calculation time (b) Memory bandwidth

Figure 4: Performance of SELL-32-1 compare to CRS on GPU

It is obvious that SELL-32-1 needs much less computation time compared to the CRS method, this aligns with our expectation since we know that with this improved data structure, we have fewer cache misses and overhead. It is not surprising to notice that we get a huge improvement

in memory bandwidth. With SELL-32-1 we achieved around $250GB/s$ which is really close to the maximum memory bandwidth we can achieve on Tesla T4. In the implementation I set the block size to 32, so each block works exactly on one chunk. If we set a bigger block size the performance should improve again since it will need less scheduling.

The performance we achieved for the SELL-32-1 method aligns with our expectations, with this data structure we make sure that the data access pattern suits GPU well. Also, the SELL-32-1 method is more conducive to parallel processing and vectorization because it organizes data into blocks, making it easier to parallelize operations.

4 Conjugate Gradient Solver

In order to solve the linear system, we use a conjugate gradient solver to deal with the problem. During the process, we repeat matrix-vector multiplication until the error is smaller than the accuracy we can accept. Here we measure the million unknowns computed per second per iteration. The result is shown in the figure below.

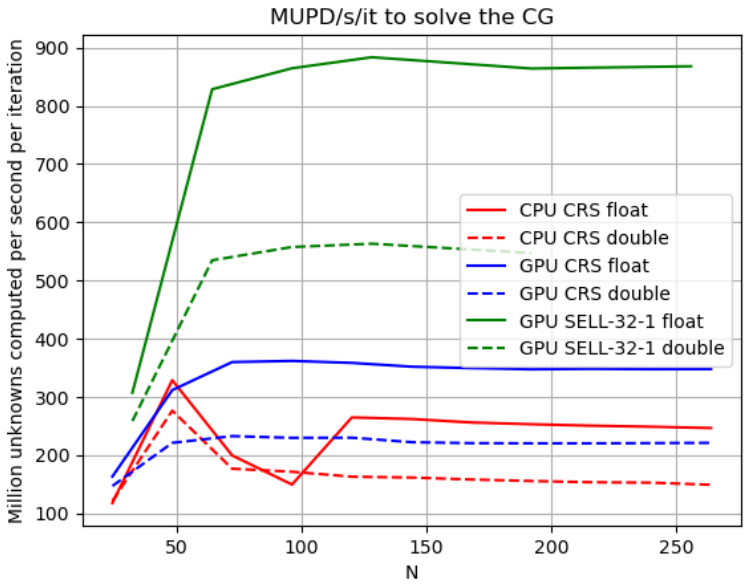


Figure 5: MUPD/s/it during the conjugate gradient solve

We can see that the SELL-32-1 implementation on GPU outperforms any CRS implementation. The maximum MUPD/s we can achieve in each iteration is around $900MUPD/s$, as we know from the previous assignment, the GPU we are using has a maximum of $8.1TFLOPs$ for single precision, so we are far away from the maximum compute capability that Tesla T4 can achieve. This indicates that in this solver we are memory bounded.

The figure below shows the iterations required to calculate the conjugate gradient solver. In the figure is clear that double precision needs much fewer iterations to achieve the final result. Also, for single precision the slover stopped after 500 iterations for N larger than around 180.

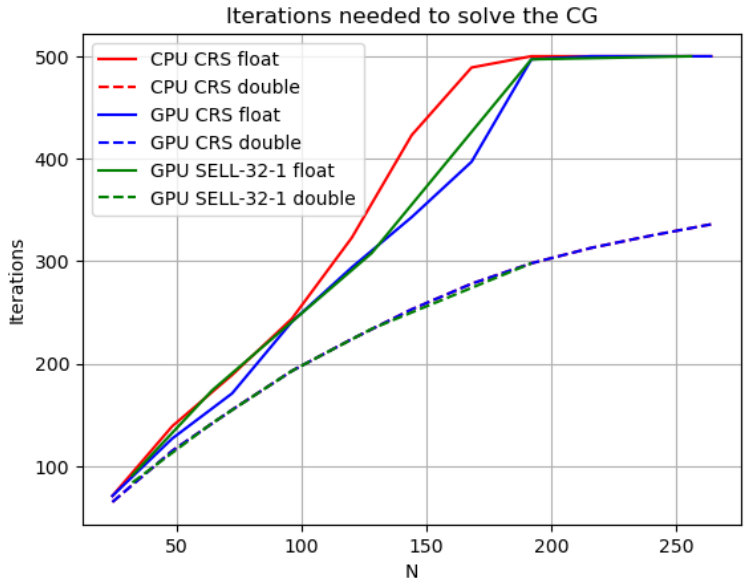


Figure 6: Iterations required for the conjugate gradient solver

We also want to know the time it takes to solve a linear system. So we measure the time spent by each method also. The result is shown below. We can see that for each method it takes around the same time for double and single precision. From the result above we know that for one matrix-vector multiplication single precision is much faster than double precision, however, with double precision we need less iterations so the total time to solve a linear system is about the same.

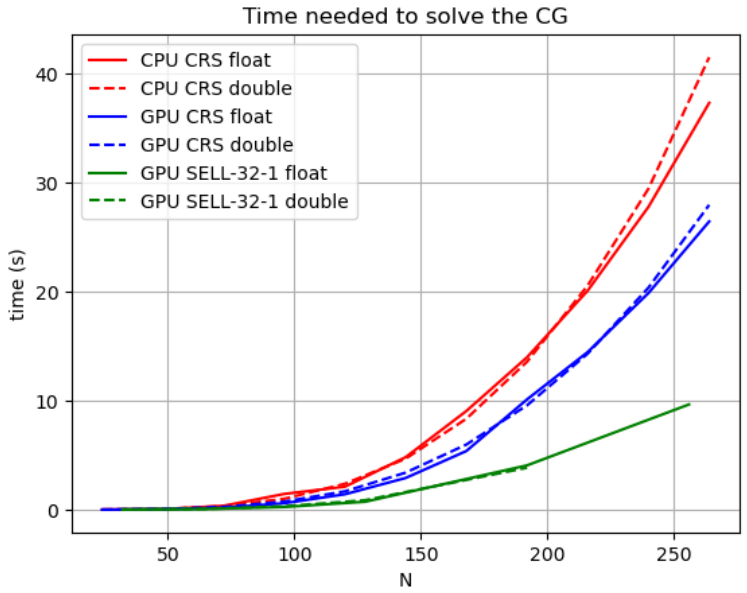


Figure 7: Time to solve the conjugate gradient

In order to know if it is worth using the GPU to solve the whole linear system problem, we need to also consider the time we used to transfer data between host and device. With the result we got from section 3, we know that we spent around $1 seconds$ to transfer the whole matrix from host to device. Then the calculation time is around $10 seconds$ (here we consider the system with $256^3 domains$). So the total time we spent to solve one linear system is still significantly less than any CRS implementations. Therefore, the SELL-C-σ method is preferred to be used for CG problems.

5 Possible limitations and further opportunities

If we want to solve a system that we have to solve many linear systems in sequence, the possible limitation should come from the memory bound. As the result shows above, we almost achieved the maximum memory bandwidth with SELL-32-1 methods but the computing throughput is far from the best, this indicates that we can not fetch enough data to let the GPU fully work. So in this case, if we have a lot of this kind of linear system our performance will be limited by the data we can fetch from memory.

There are a lot of alternative framework-based solutions to implement the SELL-C-σ format. In this course, we mainly talk about Kokkos and cuBLAS.

cuBLAS is a library that provides a GPU-accelerated implementation of the basic linear algebra subroutines. In cuBLAS there is a library called cuSPARSE that provides operations between dense/sparse matrix and dense/sparse vector. It also allows us to choose how we want to store our sparse matrix. For example, it provides also CRS format and SELL format. So instead of implementing our own method we can just use the cuSPARSE library to perform the sparse matrix.

For Kokkos, the SELL-C-σ typically stores the data in one-dimensional arrays. The corresponding arrays are also considered as one-dimensional arrays. But we can still use Kokkos to implement the whole matrix-vector product.