Final Project Report

Jacob Kerr (jck268), Mitchell Gray (meg346), Tong Duan (td273),
Perlmutter Usernames: jck268, meg346, ttdd273w

`https://github.com/TTDDClasses/cs5220FinalProject`

1 Introduction

Our project is focused on parallelizing sparse matrix matrix multiplication on the GPU. The motivation behind this is the fact that sparse matrix multiplication or SpGEMM is used in many contexts. As we have seen in lecture, graphs can be easily represented as sparse matrices, which makes SpGEMM crucial to many graph algorithms in a high performance computing setting. Additionally, many physical systems in the world can be represented by an underlying sparse matrix. For fields such as fluid dynamics and structural modelling, SpGEMM is also widely used. Lastly, similar to dense matrix matrix multiplication, SpGEMM is also used frequently for training machine learning models. As you can see, being able to parallelize SpGEMM will enable much more efficient computations in a wide variety of fields.

1.1 Question

The main question we are tackling for this project is can we parallelize SpGEMM on the GPU in order to accelerate its performance over SpGEMM or DGEMM implementations on the CPU?

1.2 Hypothesis

Our hypothesis, based on our knowledge on GPUs and Homework 3, was that the GPU would indeed be able to accelerate the performance of sparse matrix operations. We expected GPU performance to be multiple times faster in all test case sizes. This is because matrix multiplications should be highly parallel since the computation for each entry of the matrix is independent of one another, and GPUs have significantly more threads. This means we should be able to directly parallelize the most expensive computation step.

Additionally, this is also simply parallelizing the operation on the naive implementation of SpGEMM. In the future, we could implement Gustavson's algorithm, which takes advantage of the sparse matrix format and parallelizing this algorithm for SpGEMM could potentially yield a better performance.

1.3 Results

As a quick summary of the results, what we seen is that over the naive implementation of SpGEMM on the CPU, the parallel version on the GPU is significantly faster. Intuitively, this speedup is achieved through the highly parallel nature on sparse matrices, where we are able to directly calculate each entry of the matrix independently of each other. Additionally, CSR and CSC representations also matters a lot in terms of the performance, and considering the local accesses of entries provided a surprising amount of speed up. We also discovered that our performance is bottle-necked by converting dense matrices to sparse matrices after running our algorithm.

2 Context

2.1 Member Contributions

Since this project required implementation on many different versions of SpGEMM, the implementation was evenly split up amongst each of us. Tong worked on the naive CPU and GPU implementations of SpGEMM, providing functions to convert between a dense and sparse matrix, and also a correctness check script where we are able to get timings on the code and identify the sections in the code that's slowest. Mitchell worked on creating a benchmarking suite as well as the first GPU parallelization of SpGEMM. This turns out to be rather nontrivial as we also need to copy all the arrays to the GPU as well as parallelize the computation. This also required us to adjust the correctness script for measuring the overhead for copying the arrays. Jacob worked on researching possible SpGEMM algorithms, and then attempting both small and large optimizations to our code. These were essentially completely different parallelization models.

2.2 Prior Works

Gamma: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication by Zhang, et al. (Link) creates an accelerator, Gamma, designed to optimize sparse matrix multiplication using Gustavson's Algorithm. Their approach aimed to reduce memory traffic and improve performance by addressing drawbacks of previous dataflows such as poor input/output reuse and high traffic ressulting from merging large partial out matrices. Gamma achieved a mean speedup of 2.1x over previous systems and reduced memory traffic considerably.

High-performance sparse matrix-matrix products on Intel KNL and multicore architectures by Nagasaka, et al. (Link) tries optimization strategies for Sparse Matrix multiplication specifically targeted at Intel's Xeon Phi and multicore processors. Their optimizations involved addressing memory management and thread schedule, while also using hash-table-based and heap-based algorithms. They found that performance varies with matrix characteristics, and no single algorithm dominated in all scenarios. For example, different algorithms excelled at different matrix sizes and sparsity levels.

A Systematic Survey of General Sparse Matrix-Matrix Multiplication by Gao, et al. (Link) conducts a comprehensive review and comparison of general sparse matrix-matrix multiplication methods up to 2021. The authors discuss optimization techniques, applications, compression formats, general formulations, and programming models, along with reviews of different architecture-based optimizations. In particular, this paper includes a section on GPU-based architecture optimizations. The authors state that most optimizations targeting GPU architecture aim to reduce memory traffic between global memory and on-chip memory since the overhead for data transfer is so high on the GPU. Many of these optimizations for memory access involve using shared memory as much as possible. Since global memory is expensive to access on the GPU, many authors propose storing results of intermediate operations exclusively in shared memory and only updating global memory once large chunks of computation have completed. Other optimizations made for the GPU target load balancing of thread work. Several previous works involve assigning multiple rows of A to one thread as well as grouping different non-zero elements of the input and output matrices into different blocks which are then each assigned a thread. In recent years, researchers have proposed two-level load balancing. A global load balancer assigns work to different thread blocks and local load balancers per block assign a set of rows to each thread in the block.

3 Empirical Methodology

3.1 Correctness Check

In order to make sure that the algorithm was correctly implemented, we needed to implement a correctness check for SpGEMM.

To do this, we took a very brute force approach as we do not care about efficiency in this step. We implemented functions to convert between sparse and dense matrices, and after the computation step finishes, we simply convert the resulting sparse matrix back into a dense matrix, and we use an efficient DGEMM such as BLAS to compute the correct matrix and compare for any arithmetic errors.

3.2 Serial Implementation of Sparse Matrix Multiplication

We started with a serial implementation of sparse matrix multiplication. Prior to implementing the matrix matrix multiplication, we first needed to implement the data structure to support sparse matrices. We started with a compressed sparse row format, which includes an array of row pointers, column indices, and values. There is a one to one correspondence between the column indices, and the row pointers describe where each row starts and end in relation to the other two arrays.

For the serial implementation of SpGEMM, just like a regular matrix multiplication, for an entry (i, j) in the result matrix, it takes row i from matrix A, and a column j from matrix B, and loops over all of the values in row i. However, we need all the entries in col j of B, but we can only access rows of matrix B since we are storing it in compressed row format. So, what we need to do is to loop through rows of B starting at index i and then check all the entries in that row until its column index matches with j. Upon discovering such a value, we can simply break out of the loop since all the column indices are sorted. We note that because this is a sparse matrix, we are assuming that not all rows are filled so it should be in general much faster than a dense general matrix multiply.

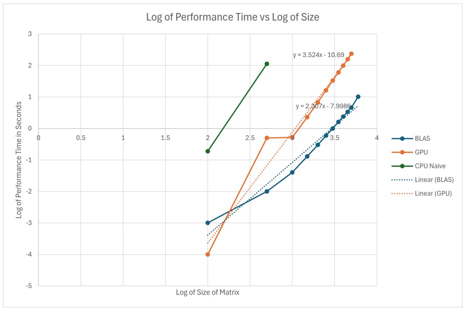

Figure 1: The log log plot of the performance for the three algorithms.

3.3 GPU Implementation of Sparse Matrix Multiplication

Now, we describe how we implemented a parallel version of the aforementioned SpGEMM on the GPU. The thread abstraction we used here is that each core of the GPU will compute for a single entry (i, j) in the original matrix. However, in order to do that, we need to copy from host to device the three arrays related to the sparse matrix to make sure it has all the information needed for the computation. This step adds a large amount of overhead to the algorithm; however, for large sparse matrices, it should be much more effective since the number of nonzeros should be a lot less than the size of the matrix which should reduce computation time.

After copying the information for each matrix to the GPU, we can then just perform the same computation step as the serial algorithm, except this time, we are computing on exactly just one entry of the matrix.

As you can see in Figure 2, the BLAS version of the code still performs the best for a regular dense matrix multiplication. However, compared to the naive implementation on the CPU for sparse matrix multiplication, parallel GPU version scales much better, where the CPU implementation already starts to time out for matrices of size 1000, indicating

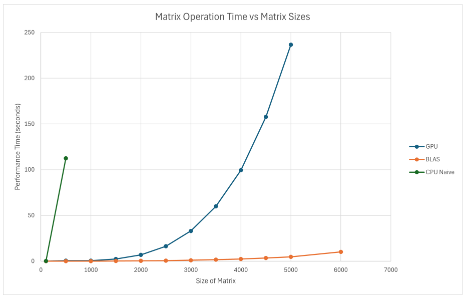

Figure 2: The performance times of the three algorithms versus the size of the matrix.

the effectiveness of the parallelization.

We've also plotted the linear scaling coefficient in Figure 1. As you can see, comparing the linear scaling coefficients, the difference is actually much less drastic. BLAS had an LSC of approximately 2.3 while the GPU parallelization had an LSC of approximately 3.5; however, this does translate to a significant difference in the actual performance times, indicating that there are probably more optimizations we can utilize. These comparisons are done on dense matrices, we will discuss the results on sparse matrices in the next section, which will be a fairer comparison for SpGEMM.

3.4 Sparsity

One very important question when it comes to SpGEMM is the sparsity of the matrix. Sparsity of the matrix is measured in terms of the number of nonzero elements over the number of total elements in the matrix. Therefore, a 95% sparse matrix will have close to all of its elements nonzero. We first implemented a sparse matrix generation by altering the dense matrix generation, where we calculated the number of elements for a given n by n matrix that needs to be set to nonzero. Then, we randomly generated indices in the range of the matrix to set to nonzero and we skip over the index if the entry is already nonzero. This way, we ensure that we have a random set of nonzero values set throughout the matrix.

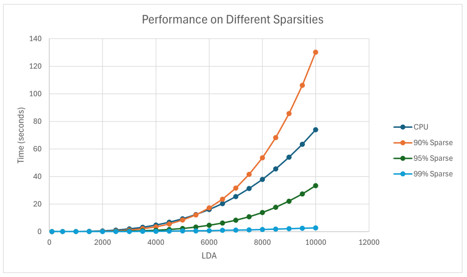

Figure 3: The performance times as we change matrix sparsity.

Again the idea is that for sparse matrices, since we are not storing all of the 0s in the matrix, we can actually avoid performing most the calculations that a dense matrix will otherwise perform, hence the performance boost. Additionally, by parallelizing, we should also be able to observe further performance boost. As seen in Figure 3, for a 95% sparse matrix, we are already seeing much better performance compared to the CPU BLAS implementation. This most likely comes from the much fewer number of nonzeros in the matrix, which decreases the overall amount of computation because less data needs to be multiplied on the GPU.

Also looking at the linear scaling coefficient for our sparsity performance, we see that the general performance is approximately 2, which is actually better than the LSC of BLAS even in the dense matrix multiplication scenario. Therefore, as you can see, for sparse matrices, the sparse representation of the matrix and parallelizing it on the GPU can help us attain very significant performance boost.

3.5 CSR vs CSC

Another thing we wanted to experiment with is whether a compressed sparse row versus a compressed sparse column format will have any impact on the performance of SpGEMM. Therefore, we went ahead and implemented CSC to determine this.

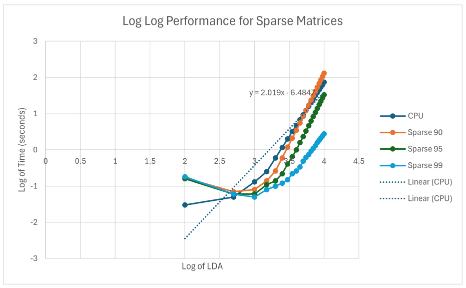

Figure 4: The log log plot of the performance for different sparsity of matrices.

Given that the matrix multiplication takes into two sparse matrices, we have an option for both the first and second matrix to choose it to be a CSR or CSC format. In other words, we can choose both of the matrices to be in CSR format, or both of the matrices to be in CSC format, or one of them to be CSR and the other to be CSC.

To determine which options are worth exploring, we thought about matrix multiplication more carefully. Since matrix multiplication takes in a row of matrix A and a column of B, storing A in CSC and B in CSR seems suboptimal, because ideally you should be accessing all the entries you will use for matrix multiplication in order. So for matrix A, you should be accessing each of the A's entries in the row order and for matrix B, you should be accessing each of B's entries in column order. Our first iteration of SpGEMM had both matrix A and B stored as CSR since CSR seems to be the most widely used format in all the previous works we've seen. However, even during our implementation, we noticed that it might be suboptimal for SpGEMM to access B's elements, because its elements are stored in a row format, while we are only trying to get all the elements in a specific column. Therefore, we decided to compare our first implementation of CSR and CSR matrix multiplication against a CSR and CSC implementation.

Just like SpGEMM for CSR and CSR matrices, we need to loop through all of the entries in the matrix, so for an entry (i, j), we need to compute its dot product by getting all of the values in row i. Then, we need to get all of the entries belonging to column j of matrix B. However, because matrix B is already stored in a sparse column format, we can directly access all the entries in that column. The only remaining thing we need to check is that the entries match up or else we will be multiplying the incorrect values together.

In describing this implementation, we notice that there's actually not too large of a difference in terms of the number of accesses, because regardless of the format, we need to check that each pair of entries we are multiplying together actually match up, and because of the nature of sparse matrices, there's a possibility that they might not match up, and there isn't a way to determine that unless we've seen all of the values in that row or column. However, there is a possibility that we see some performance boost since the accesses should be more linear and will not be jumping around to different rows of matrix during the calculation.

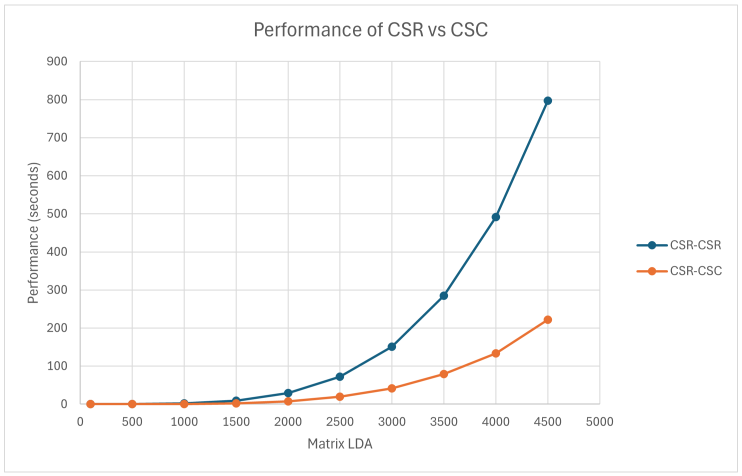

Figure 5: Performance between the CSR-CSR and CSR-CSC implementation of SpGEMM, where CSR-CSR refers to both matrices stored as compressed sparse row, and CSR-CSC refers to the first matrix being compressed row and the second matrix being compressed sparse column for 80% sparse matrix.

As you can see in Figure 5, the performance boost is actually quite drastic. The performance difference is not drastic for smaller matrix sizes. However, for larger matrix sizes, we begin to see that the performance of CSC implementation is about 4 times faster than that of CSR. This performance boost we assume probably comes from the locality provided by CSC format of the second matrix.

3.6 Performance Breakdown

One of our largest concerns with SpGEMM is the overhead of copying all the arrays into the GPU, because this is an essential step that cannot be avoided. So if this turns out to be the largest bottleneck, then it will be very difficult to actually cut down the time for the GPU. However, if we see that the performance gains outweigh the overhead, then we still have a very good reason to utilize SpGEMM parallelization on the GPU.

As you can see in Figure 6, we see a clear breakdown of where most of the inefficiencies come from. We see that the fastest segment happens during the computation, which is exactly the method we were trying to parallelize. There is some initialization overhead. However, as we can see, it seems that this is a constant cost overhead and does not vary with the size of the input, similar to the cleanup time. However, the area where we observe the largest slowdown is in the writing, and we see that as the size of the input increases, the time it takes to write the result from the GPU to CPU increases drastically. After

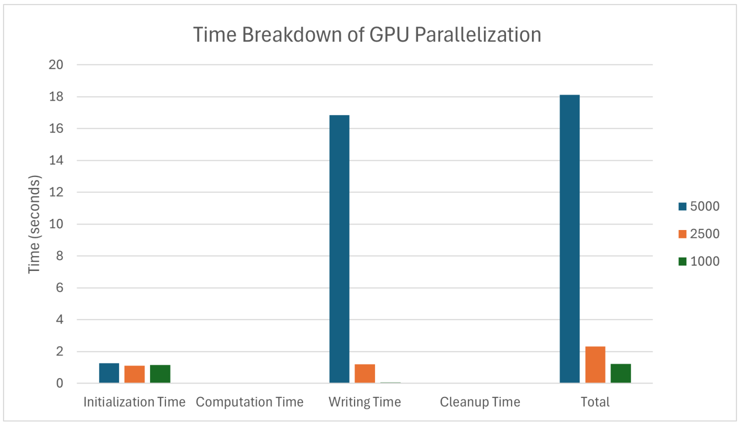

Figure 6: Time breakdown of parallel SpGEMM on the GPU, which is computed for 80% sparse matrices.

analyzing the timings further, result confirmed that converting our matrix from dense to sparse on the CPU took the vast majority of time. However, we will attempt to see whether we can improve this result or not.

3.7 Optimizing Conversion

Observing the performance breakdown, we see that the major bottleneck for our SpGEMM function comes from the conversion from a dense representation back into a sparse matrix representation. Our current implementation stores the results of SpGEMM as a dense matrix, so this bottleneck is a huge waste of computation time. If there is a way to reduce this conversion cost, this would greatly reduce the overall execution time.

We initially proposed having the SpGEMM GPU kernel generating results directly into sparse format. However, this requires the ordering of the columns and values arrays to be correct. Since threads may run in any order and the structure of the columns and values arrays must be ordered by row order, we cannot guarantee this ordering without special thread ordering. This would require complicated locking mechanisms and would ultimately reduce the algorithm to a serial process, which would defeat the purpose of parallelizing via GPU, so this proposal was rejected.

Following this, we proposed replacing the dense matrix output of the SpGEMM kernel with a slimmed down 3-tuple implementation. The 3 tuple is composed of the row, column, and value of non-zero values in the result matrix. We can then convert from this format to a sparse format. However, this conversion would require a sorting algorithm on CPU to put the values in a proper order before conversion. This sorting would be at least $O(n \log n)$, which would be greater than the $O(n)$ dense to sparse conversion currently, so this proposal was also rejected.

3.8 Scaling

We also analyze here how the problem scales as we change the number of threads per block on the GPU.

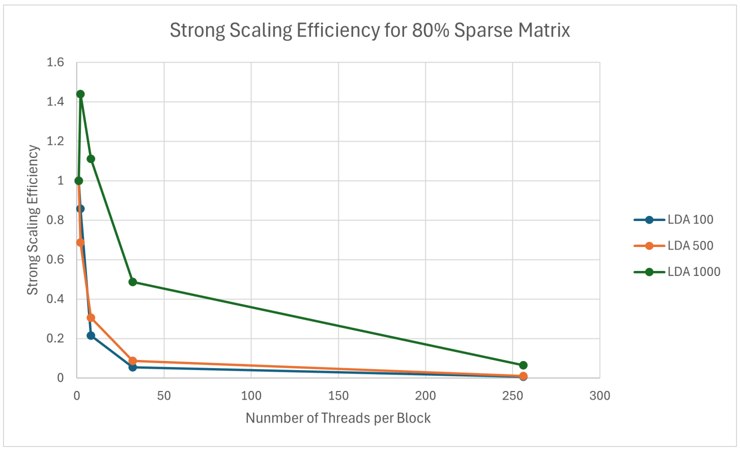

Figure 7: The strong scaling efficiency as we change the number of threads per block for different matrix sizes but for an 80% sparse matrix.

As you can see in Figure 7, we plotted the strong scaling efficiency as we change the number of threads per block from 1 to 256, we omitted 1000 here because it simply extends the graph farther out to the right. However, as you can see, we actually achieve reasonable efficiency for LDA of 1000, where the strong scaling efficiency is much more noticeable. Similar to many of the other strong scaling plots we have seen in previous homework, we observe a diminishing marginal return where the speedup we get as we increase the number of threads we use. This is especially noticeable when the number of threads is relatively small.

Interestingly, we notice that for matrix with LDA 1,000, the efficiency spikes for going from 1 thread per block to 2 threads per block, but for 8 threads and 32 threads, the amount of speed up was significantly less than going from 1 to 2.

We believe there are several reasons for this. The first reason is how we are parallelizing the actual computation. The abstraction we are using is one where each thread represents a single entry of the matrix to perform the calculations on. This means that as long as the number of cores we are using from the GPU exceeds the number of the entries in the matrix, it is able to perform the entire computation in one pass, which is why we probably will not see a computational speedup as we increase the number of threads. The other reason is that our largest bottleneck in the code is when we convert our matrix, which does not have any relation to the number of threads we are using as that step is computed directly on the CPU, which would explain the results we are seeing where there's a huge diminish in return as we increase the number of threads.

4 Conclusion

In conclusion, our project aimed to explore the parallelization of sparse matrix matrix multiplication (SpGEMM) on the GPU, with the hypothesis that this approach could significantly accelerate performance compared to CPU implementations. We implemented various versions of SpGEMM, including naive CPU and GPU implementations and attempted different optimizations and strategies.

Our results demonstrate that parallel GPU implementations of SpGEMM does offer substantial speedups over CPU implementations, especially for large and sparse matrices and in the computation step itself. The highly parallel nature of GPU computation allows for efficient computation of sparse matrices, resulting in significant performance gains. Additionally, our exploration of matrix sparsity and format revealed performance differences from sparse format, suggesting further optimizations for specific use cases, such as other matrix algorithms including matrix vector product or vector matrix product, etc..

However, our analysis also uncovered challenges and areas for improvement. Turning a dense matrix into a sparse matrix remains a significant bottleneck, particularly for large matrices. Furthermore, while our implementations achieved notable performance improvements, there is still room for further optimization and refinement, especially regarding converting our matrix from dense to sparse formatting.

Overall, our project explored the nuances of sparse matrices compared to dense matrix multiply and getting a deeper understanding of the process which hopefully allows us to explore more possibilities for high performance computing in difference domains.

4.1 Obstacles

The major obstacle we ran into was with data movement. The largest bottleneck is figuring out a way to store the result as a sparse matrix as we calculate the sparse matrix, and that turns out to be a rather complicated issue to solve, so this is definitely a point worth exploring in the future.

4.2 Unexpected Results

One unexpected result we had was that the number of threads didn't matter for performance. Since the task is embarrassingly parallel, it would make sense for the number of threads to drastically impact performance. Our explanation is that since the vast majority of our time spent isn't on computation, but instead is converting our dense matrix to sparse format, the difference in speed is currently unnoticeable. As the conversion overhead decreases, we expect the number of threads to improve performance as expected.