Hello everyone, our project is focused on parallelizing sparse matrix multiplication.

Sparse matrix multiplication or SpGemm is used in many contexts. As we’ve discussed a few times before in lecture, a sparse matrix can be considered the dual of a graph, so SpGemm is widely used in many graph algorithms. Many physical systems involve an underlying sparse matrix, such as fluid dynamics, and the same applies to machine learning context. Thus, being able to parallelize it is crucial as it will enable much more efficient computation in a wide variety of fields.

We will first discuss how we implemented SpGemm. Here’s a quick overview of compressed sparse row representation of sparse matrices, we’ve seen this before in lecture, but the idea is that we have an array of row pointers, an array of column indices, and an array of values. Column indices and values array have a one-to-one correspondence and the row pointers tell you where the row starts and ends in relation to the other two arrays.

The more interesting aspect is the actual sparse matrix multiplication. We will describe a naïve implementation of it first. Just like a regular matrix multiplication, for an entry (i,j), it takes row I from matrix A, and a column j from matrix B, loops over all of the values in that 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 do is we loop through rows of B starting at index I and then check all the entries in that row until the index matches with j. We note that because this is a sparse matrix, we are assuming that not all rows are filled so it should be faster than a dense general matrix multiply.

Now onto the actual parallelization, as a first approach, we let each core of the GPU compute the value for an entry (I, J) in the final matrix, and keep the value if and only if the value is nonzero and store it in the sparse matrix format. So we are essentially parallelizing the actual computation step.

Here’s some preliminary results, we graphed here CPU, GPU, and BLAS implementation to benchmark where our code is currently at, and we see that compared to BLAS our implementation is actually still quite slow. However, by being able to parallelize each entry, we are still able to obtain a significant amount of speed up over the naïve algorithm despite the fact that the algorithm we’re parallelizing is the same. We’ve also graphed here a log log scale plot, we see that the LSC for our GPU implementation is about a 3.5 while for BLAS it’s about a 2.3, so there’s still some optimizations we can make.

Finally, as next steps, we list here three potential optimizations. One is reducing the communication overhead when sending data from CPU to the GPU, another is utilizing more efficient and parallel sparse matrix multiplication such as Gustavson’s algorithm. Lastly, it will also be interesting to observe how operations on a compressed sparse column format will differ from our current CSR format. So that’s a quick summary of our project, thank you!