Sparse Matrix-Sparse Matrix Multiplication

Alex Berdkan

Suleiman Jaber

Mohamad Hussein Younes

**Problem Description**

*Describe the computation that you are planning to parallelize in this project. Your description should tackle the following questions:*

* *What is the problem that the computation solves?*
* *Why is the computation important and where is it used in practice?*
* *How does the computation work? What procedure does it follow to solve the problem?*

*Use illustrations where needed. Make sure to cite any references that you use.*

This description should be submitted with Milestone #1.

The computation we are planning to parallelize is Sparse Matrix-Sparse Matrix Multiplication, in this Project we are doing it for the situation when the second matrix is small. This process generates their product C = A x B from two sparse matrices A and B where only the non-zero values are kept and handled to conserve space and computational time (by not processing the 0 values).

Recommendation systems, graph algorithms, scientific computing, and machine learning pipelines all find significant applications for sparse matrix-sparse matrix multiplication. Scalability of the applications on large datasets depends on its performance.

The SpMSpM method computes, using sparse matrices A and B, the product C = A × B. Operating just on the non-zero values utilizing compressed techniques for efficiency, it does not save all elements including zeros.

**Artifact**

*Create a private GitHub repository where you will include the code that you will develop throughout the project. The repository should include a README for how to compile and run the code. Make sure to share the repository with the following GitHub usernames: ielhajj, rana-dbouk. Also include the link to the repository below.*

This link should be included with Milestone #1.

https://github.com/Suleiman50/SPMM-GPU-Project

**Sequential Implementation**

*Describe your sequential implementation of the computation.*

A description of this implementation should be submitted with Milestone #1. The code for the implementation should also be pushed to the GitHub repository in a file called kernelCPU0.cu.

Sequential implementation (kernelCPU0.cu) of Sparse Matrix-Sparse Matrix Multiplication iterates over each row in matrix A using CSR representation. For each of the non-zero values, A(i,k), it takes the corresponding row in matrix B and gets the product with each B(k,j). The result is stored in output matrix C in COO format. Each (i,j) pair is tracked using a marker array to ensure it has not been computed before, so values are inserted or accumulated correctly. The implementation fills the output matrix C dynamically each step.

**Basic Parallel Implementation**

*Explain how you parallelize the computation on the GPU initially. Specify how the work is divided across thread blocks as well as threads within each thread block. You do not need to describe any optimizations in this section. You only need to describe the initial basic parallelization strategy that you implement. Use illustrations if needed. Make sure to cite any references that you use.*

A brief description of the implementation idea should be submitted with Milestone #1.

A detailed description of the implementation should be submitted with Milestone #2. The code for the implementation should also be pushed to the GitHub repository in a file called kernel0.cu.

Brief:

The strategy is to parallelize Sparse Matrix-Sparse Matrix Multiplication operation by assigning every row of matrix A to one CUDA thread. The thread will do the same thing as its serial counterpart except in parallel, and since there is considerable independence between rows, this thread-per-row strategy is appropriate in this case.

Detailed:

Each CUDA thread is responsible for processing one row of matrix A in kernel0.cu implementation. The values of entries in the row that are non-zero are loaded in CSR form and each thread computes corresponding entries of matrix B products. pos\_out is an atomic counter used to reserve slots in resulting matrix COO and thread-local array of markers avoids duplicates. Device memory allocations to hold temporary values and synchronization post kernel execution is ensured. The method has higher throughput when compared to the serial technique due to row-level parallelism.

NOTE: comments are also present in the code to explain it.

**First Optimization**

*Describe the first optimization that you apply to your implementation. Explain what bottleneck or other source of inefficiency this optimization alleviates and why you think it should be effective.*

A brief description of the optimization idea should be submitted with Milestone #2.

A detailed description of the implementation of the optimization should be submitted with Milestone #3. The code for the implementation should also be pushed to the GitHub repository in a file called kernel1.cu.

For our first optimization of parallel sparse matrix–sparse matrix multiplication (SpMSpM), we optimized the kernel by introducing dynamic occupancy tuning. Maximizing multiprocessor occupancy is a key step in optimizing GPU usage as it allows for better latency hiding and utilization, thus improving parallel efficiency and reducing execution time.

We replaced fixed grid and block size definitions with a dynamic occupancy-based approach; with cudaOccupancyMaxPotentialBlockSize() being the key function here. This tweak allows the kernel to dynamically adapt to different devices, thus avoiding the underutilization of GPU cores.

**Second Optimization**

*Describe the second optimization that you apply to your implementation. Explain what bottleneck or other source of inefficiency this optimization alleviates and why you think it should be effective.*

A brief description of the optimization idea should be submitted with Milestone #3.

A detailed description of the implementation of the optimization should be submitted with Milestone #4. The code for the implementation should also be pushed to the GitHub repository in a file called kernel2.cu.

In our second optimization for the sparse matrix–sparse matrix multiplication (SpMSpM) kernel, we addressed the overhead associated with per-thread marker array initialization. Previously, each thread cleared its own slice of the marker array using a loop. This introduced redundant work and uncoalesced memory writes, impacting performance especially for large matrices with many columns.

We eliminated the in-kernel clearing of the marker array and replaced it with a single bulk memory set using cudaMemset before kernel launch. The marker array is now pre-initialized to -1 in global memory:

CUDA\_ERROR\_CHECK(cudaMemset(d\_marker, 0xFF, numRows1 \* numCols2 \* sizeof(int)));

By doing this, we avoid the loop:

#pragma unroll

for (unsigned int j = 0; j < colsB; ++j)

m[j] = -1;

thus reducing instruction count and global memory traffic.

**Third Optimization**

*Describe the third optimization that you apply to your implementation. Explain what bottleneck or other source of inefficiency this optimization alleviates and why you think it should be effective.*

A brief description of the optimization idea should be submitted with Milestone #4.

A detailed description of the implementation of the optimization should be submitted with Milestone #5. The code for the implementation should also be pushed to the GitHub repository in a file called kernel3.cu.

**Brief:**

The third optimization improves memory efficiency and reduces global memory traffic by using **shared memory** instead of global memory for the marker array. We split the computation into two phases: a **symbolic phase** to count unique non-zero positions per row, and a **numeric phase** to actually compute and store the results. This approach allows preallocating the exact memory needed and avoids the overhead of dynamic writes with atomic operations in global memory.

**Detailed:** In the third optimization, the SpMSpM kernel is redesigned using a **two-phase computation strategy** with **per-thread shared-memory markers** to eliminate global marker usage:

1. **Symbolic Phase – spmspm\_gpu3\_symbolic**:  
   Each thread processes a row i of matrix A and determines how many unique (i,j) output positions will be generated when multiplied with matrix B.

-A shared memory slice of size colsB is allocated for each thread.

-The thread iterates over non-zeros in A’s row and then over corresponding B entries to mark columns j that contribute to the result.

-These counts are stored in rowCounts[i].

1. **Prefix Sum (Scan)**:  
   On the host, the rowCounts array is scanned to produce rowPtrsC, which determines where each thread will write its output in the final COO arrays. This guarantees conflict-free output index assignment without atomics.
2. **Numeric Phase – spmspm\_gpu3\_numeric**:  
   With precomputed output positions, each thread again processes row i, reusing the shared-memory marker slice:

-It computes values as A(i,k) \* B(k,j) and writes results directly to the preassigned position in COO arrays.

-If the marker shows a column j hasn't been written yet, it writes a new entry.

-If it's already seen, it simply accumulates to the previously written value.

1. **Result Handling**:  
   After the numeric phase, the total number of non-zero entries is retrieved from the last value in rowPtrsC and written back to the COOMatrix struct.

**Evaluation**

*Evaluate your implementations in comparison with each other. You only need to report the kernel time for the parallel implementation (do not worry about copy time). You should run the code on multiple input sizes/datasets and include a table and a plot that show how the execution time and speedup vary for different input sizes/datasets. Describe the trend that you observe in the plot. Make sure to specify the details for the system you are running on, including the type of CPU and GPU you are using and the amount of memory available in the system. Your evaluation should be done using the university cluster to allow us to replicate them.*

The evaluation of the sequential implementation should be submitted with Milestone #1.

CPU time: 5.86 ms

The evaluation of the basic parallel implementation should be submitted with Milestone #2.

Kernel time: 0.94 ms

The evaluation of the first optimization should be submitted by Milestone #3.

Kernel time: 0.893 ms

The evaluation of the second optimization should be submitted by Milestone #4.

Kernel time: 0.716 ms

The evaluation of the third optimization should be submitted by Milestone #5.

Kernel time: 0.462 ms

**References**

*Include a list of references that you cite in your report.*