EC527: Sparse Matrix-Matrix Multiplication

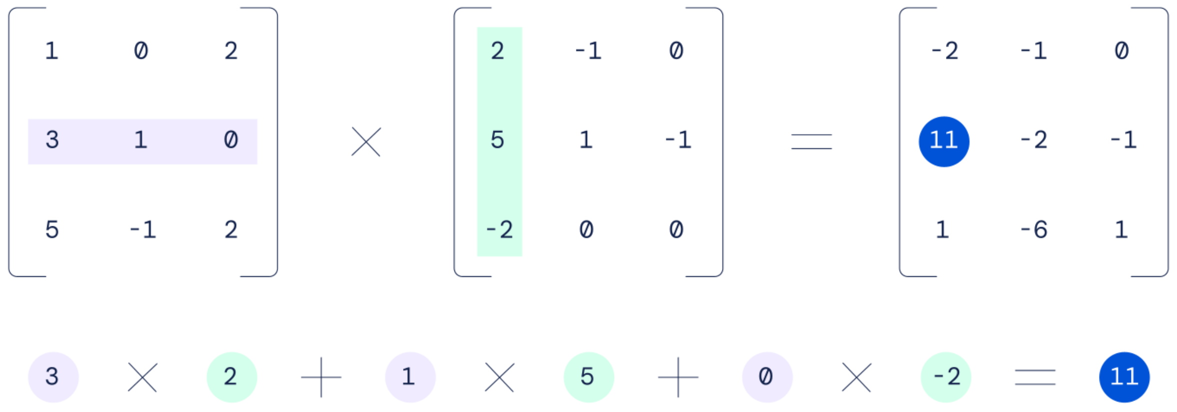
Alexander Mackanic

May 5, 2023

Contents

1 Description of the Algorithm

Matrix-matrix multiplication is an operation between two matrices that create a new matrix. The output matrix's elements are computed by calculating the dot product of each row in matrix A with each column in matrix B. The dot product is then stored in output matrix C, with the index corresponding with the row of matrix A and column of matrix B. The time complexity for the basic algorithm is O(n^3). Matrices are stored in 2D arrays with a cost of O(n^2).



Visual Representation of Matrix-Matrix Multiplication

It is a computationally intensive operation that is used in areas such as network theory, solution of linear systems of equations, transformation of coordinate systems, and many more. A sparse matrix is a type of matrix in which most of the elements are zero. There is no definition of how many elements are required to be non-zero, but a common standard is that the number of non-zero elements is roughly equal to the number of rows or columns.

$$\begin{pmatrix} 1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\ 0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\ 2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0 \end{pmatrix}$$

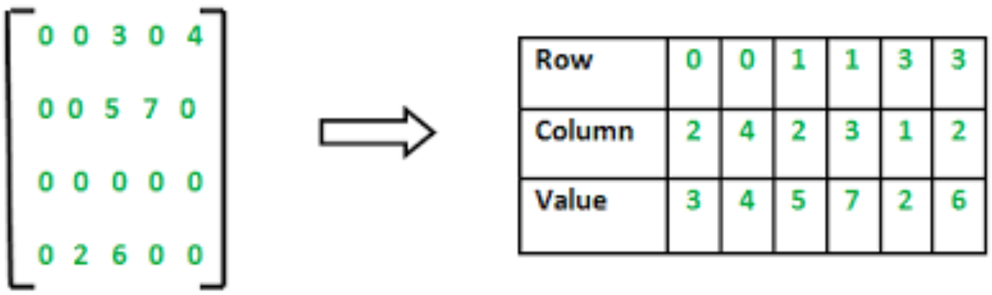
Example of a Sparse Matrix.

2 Description of the Problem

When performing matrix-matrix multiplication on sparse matrices, due to a large number of zeros, many calculations performed by the CPU are multiplication and addition of zeros, leading to many wasted computations and memory accesses. In addition, memory is wasted as most of the elements stored in the 2D arrays are zeros. This results in the need of determining a more efficient way of storing sparse matrices.

3 Description of Data Structures Used

There are several known storage formats of sparse matrices. The one I chose to store my matrices was COO or Coordinate list. COO stores non-zero elements in three subarrays, one containing row indices, another containing column indices, and finally the values. This reduces the cost of memory from the product of the number of rows and columns down to 3n, n being the number of non-zero elements in a sparse matrix. COO allows for the easy construction of sparse matrices, and substantial memory reduction.



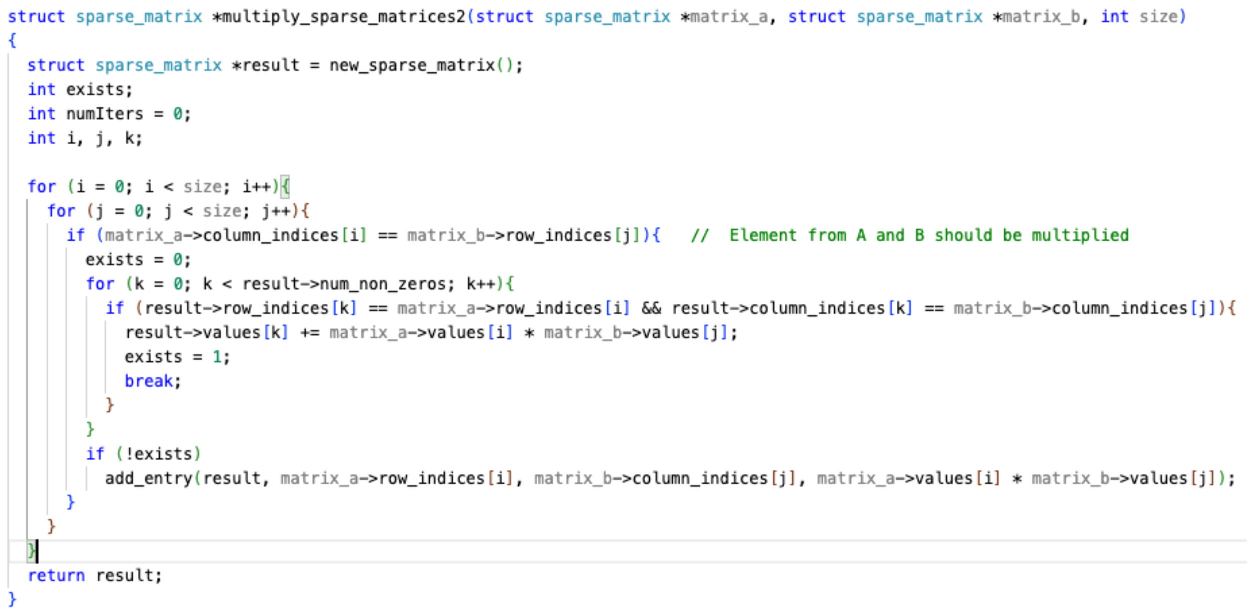
$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Visual Representation of COO format.

4 Serial Code

The base algorithm for sparse matrix-matrix multiplication is the same as traditional matrix-matrix multiplication, iterating through each element of one matrix and multiplying by each element of the other matrix if both elements pertain to the same row/column pair. However, instead of having to iterate through all elements including non-zero, with COO only the non-zero elements are iterated through and multiplied, then stored in the output matrix.



```c
struct sparse_matrix *multiply_sparse_matrices2(struct sparse_matrix *matrix_a, struct sparse_matrix *matrix_b, int size)
{
  struct sparse_matrix *result = new_sparse_matrix();
  int exists;
  int numIters = 0;
  int i, j, k;

  for (i = 0; i < size; i++){
    for (j = 0; j < size; j++){
      if (matrix_a->column_indices[i] == matrix_b->row_indices[j]){   //  Element from A and B should be multiplied
        exists = 0;
        for (k = 0; k < result->num_non_zeros; k++){
          if (result->row_indices[k] == matrix_a->row_indices[i] && result->column_indices[k] == matrix_b->column_indices[j]){
            result->values[k] += matrix_a->values[i] * matrix_b->values[j];
            exists = 1;
            break;
          }
        }
        if (!exists)
          add_entry(result, matrix_a->row_indices[i], matrix_b->column_indices[j], matrix_a->values[i] * matrix_b->values[j]);
      }
    }
  }
  return result;
}
```

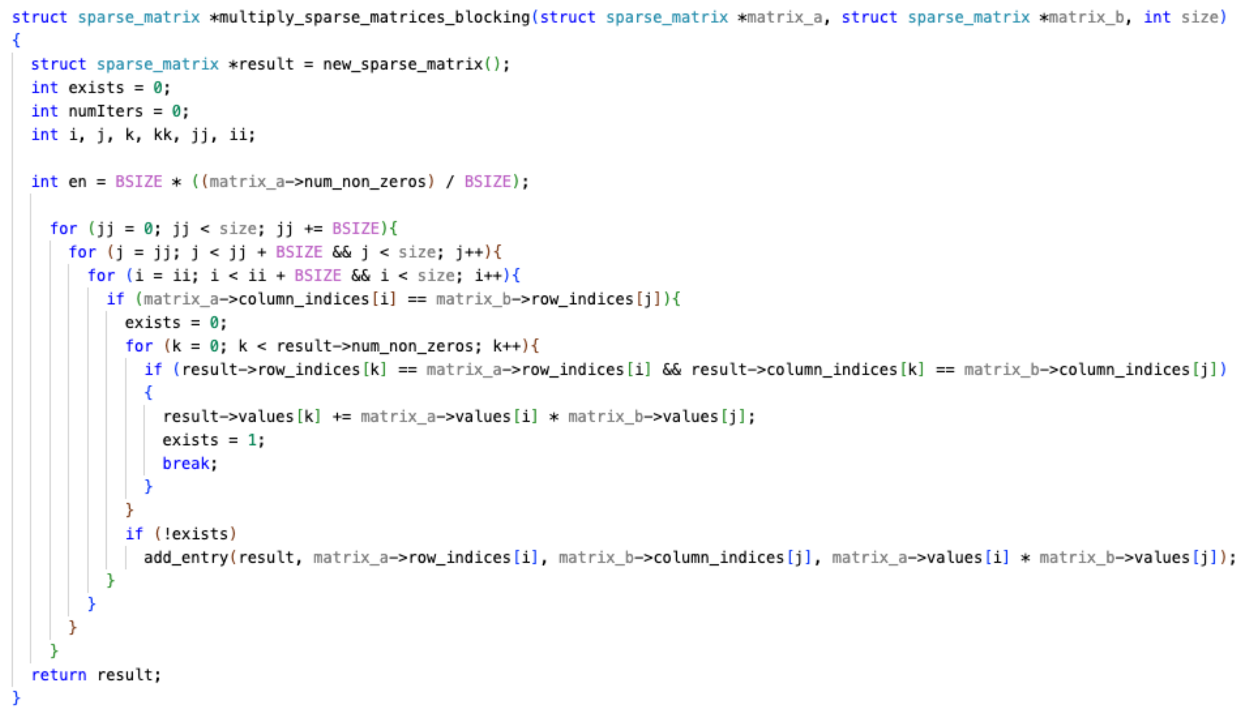
Serial Code for Sparse Matrix-Matrix Multiplication

The two outer loops iterate through all the elements of both matrix A and matrix B, and if the row and column index match, the values are multiplied and added to the result matrix. If the element exists in the result matrix the product of the two elements is then appended to the existing result element, otherwise, a new element is added to the result matrix.

Each input matrix was generated using random values between 1 – 20 with double data type, randomly being inserted throughout the matrix. Each matrix generated contained roughly around the number of rows of each matrix. Each matrix was a square matrix.

5 Optimization #1 – Blocking

Using what we learned during the semester about standard matrix-matrix multiplication, I decided to implement blocking to try and speed up my performance. I quickly came to the realization that blocking would not improve performance, because blocking takes advance of spatial locality by reducing cache misses. Since only non-zero elements are stored, all values from both matrices could fit within cache, and blocking ended up resulting in an 11% slowdown.

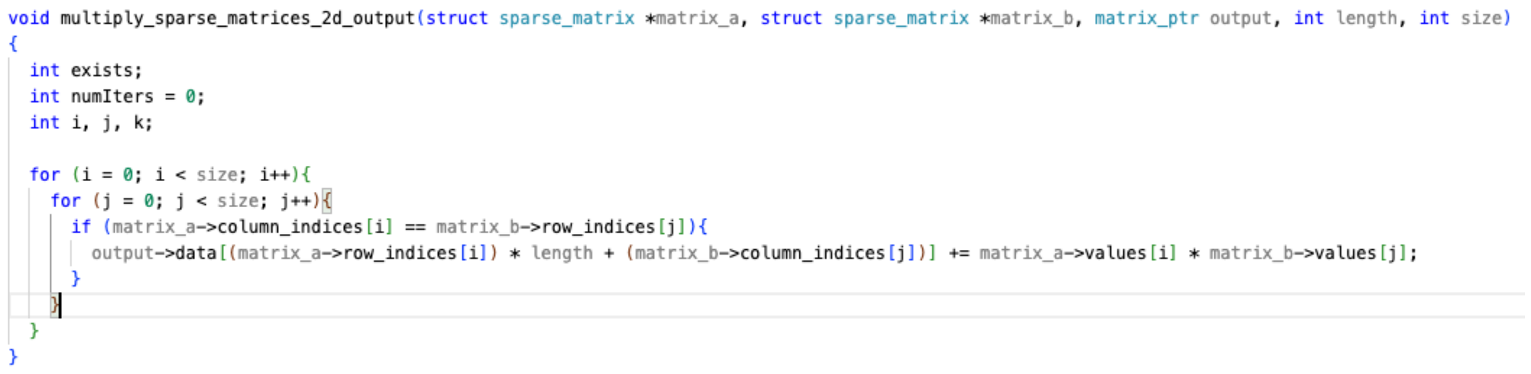


```
struct sparse_matrix *multiply_sparse_matrices_blocking(struct sparse_matrix *matrix_a, struct sparse_matrix *matrix_b, int size)
{
  struct sparse_matrix *result = new_sparse_matrix();
  int exists = 0;
  int numIters = 0;
  int i, j, k, kk, jj, ii;

  int en = BSIZE * ((matrix_a->num_non_zeros) / BSIZE);

    for (jj = 0; jj < size; jj += BSIZE){
      for (j = jj; j < jj + BSIZE && j < size; j++){
        for (i = ii; i < ii + BSIZE && i < size; i++){
          if (matrix_a->column_indices[i] == matrix_b->row_indices[j]){
            exists = 0;
            for (k = 0; k < result->num_non_zeros; k++){
              if (result->row_indices[k] == matrix_a->row_indices[i] && result->column_indices[k] == matrix_b->column_indices[j])
              {
                result->values[k] += matrix_a->values[i] * matrix_b->values[j];
                exists = 1;
                break;
              }
            }
            if (!exists)
              add_entry(result, matrix_a->row_indices[i], matrix_b->column_indices[j], matrix_a->values[i] * matrix_b->values[j]);
          }
        }
      }
    }
  return result;
}
```

Blocking Code for Sparse MMM

6 Optimization #2 – 2D Result Matrix Data Structure

Upon initial testing, it became apparent that storing the result matrix using COO format resulted in a slowdown due to the reallocation of memory in the result matrix for every new element. There was also a slowdown resulting from iterating through the entire output matrix to find/determine whether the output element already exists or not. I, therefore, decided to store the output matrix in a standard 2D array data structure to allow for constant access time to each resulting element. This resulted in a 22% speed-up compared to storing the output in a COO matrix.



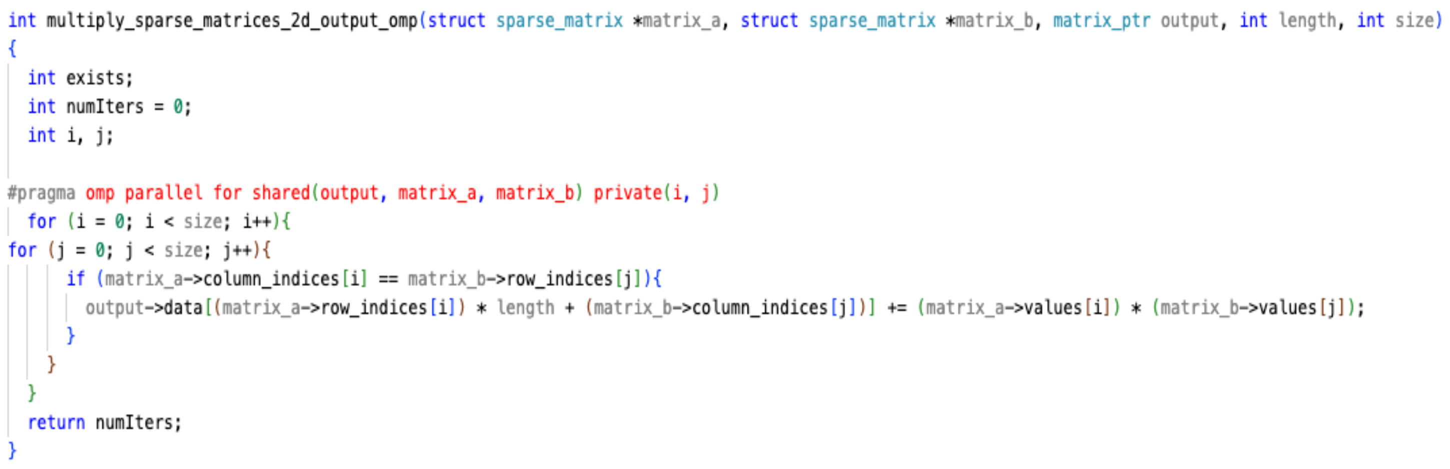
```
void multiply_sparse_matrices_2d_output(struct sparse_matrix *matrix_a, struct sparse_matrix *matrix_b, matrix_ptr output, int length, int size)
{
  int exists;
  int numIters = 0;
  int i, j, k;

  for (i = 0; i < size; i++){
    for (j = 0; j < size; j++){
      if (matrix_a->column_indices[i] == matrix_b->row_indices[j]){
        output->data[(matrix_a->row_indices[i]) * length + (matrix_b->column_indices[j])] += matrix_a->values[i] * matrix_b->values[j];
      }
    }
  }
}
```

Sparse MMM with the result stored in a 2D array.

7 Optimization #3 – OpenMP

Being aware of the incredible improvements that result from parallel processing, I had chosen to implement multithreading just as we had for MMM during the semester. I initially tried to parallelize the serial code with the COO format output, but there were errors resulting from multiple threads attempting to access the output matrix at the same time. If the inner loop that searches for the output element was parallelized, there was a dramatic slowdown due to OpenMP's overhead.

However, when parallelizing the optimization with the 2D output, there was an incredible 10.1x improvement over the serial code and a 7.1x improvement over the 2D result optimization. When using OpenMP, 8 threads were used for multithreading. Each thread was responsible for a different element of matrix A, finding elements from B that have to be multiplied, and then appending the product to the existing value of the output element.



```
int multiply_sparse_matrices_2d_output_omp(struct sparse_matrix *matrix_a, struct sparse_matrix *matrix_b, matrix_ptr output, int length, int size)
{
  int exists;
  int numIters = 0;
  int i, j;

#pragma omp parallel for shared(output, matrix_a, matrix_b) private(i, j)
  for (i = 0; i < size; i++){
for (j = 0; j < size; j++){
      if (matrix_a->column_indices[i] == matrix_b->row_indices[j]){
        output->data[(matrix_a->row_indices[i]) * length + (matrix_b->column_indices[j])] += (matrix_a->values[i]) * (matrix_b->values[j]);
      }
    }
  }
  return numIters;
}
```

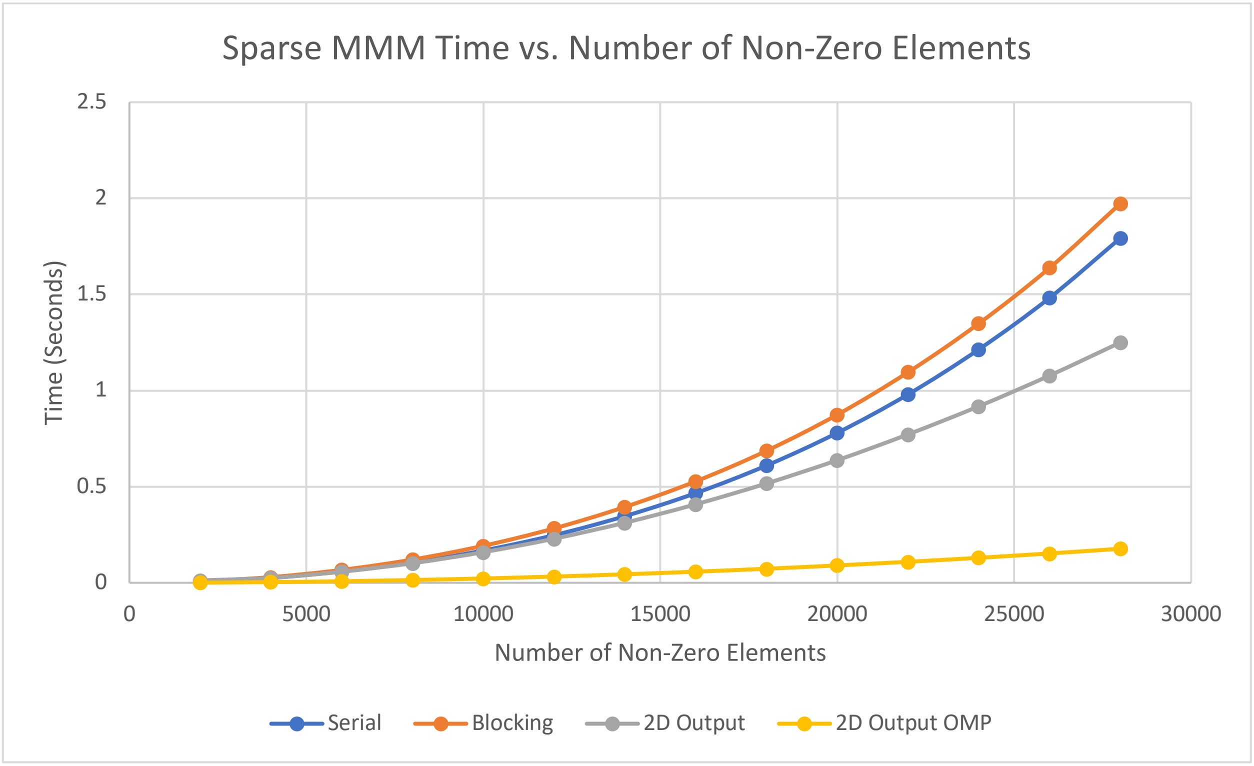
Sparse MMM using OpenMP.

8 - Results

All the outputs from each optimization were compared to standard matrix-matrix multiplication with 2D arrays to ensure correctness and validate the results of the computations. To generate the data, I created sparse matrices of size 30k x 30k, with around 30k non-zero elements. I performed MMM in increments of 2000, testing the serial code, blocking code, 2D output code, and 2D OpenMP code. Since OpenMP takes advantage of multiple threads, time had to be used to compare performance vs. number of cycles. The architecture used was the lab computers with the following specifications:

model name: Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz
cpu MHz: 1281.005
cache size: 25600 KB
CPU cores: 10

As aforementioned, the results were verified using standard 2D arrays and standard MMM. The program was compiled using the -O0 flag. Here are the overall results:



Results of different matrix sizes and optimizations

9 Further Optimizations

There are a few things that could be tested to try and further optimize sparse matrix-matrix multiplication including –

- Using different data structures other than COO format
- Vectorizing the input arrays for different parallelism
- Partitioning workload differently when using OpenMP
- Performing calculations on different architectures
- Testing sample sizes that are larger than cache
- Attempt GPU implementation

10 List of Files

- `EC527_Final_Report.pdf` – This report.
- `sparse_mmm_serial.c` – Serial reference code.
- `sparse_mmm_loop.c` - All optimization code.