INDIAN INSTITUTE OF TECHNOLOGY, DELHI

REPORT

Assignment 4: Matrix Multiplacation on CUDA

APAR AHUJA | ENTRY NO. 2019CS10465

Course - COL380 | Prof. Subodh Kumar

April 30, 2023

Contents

Chapter 1

Summary

1.1 Approach 1: CPU Baseline

Idea: Read matrix from input file and store as vector of vector of blocks, `rowBlocks` and `colBlocks`. Sort rows/columns of `rowBlocks`/`colBlocks`. Multiply the matrices using a double for-loop that traverses `rowBlocks` and `colBlocks` and write the result vector to the output file as soon as a block is found.

Why attempted:
To establish a baseline for correctness and cpu performance without any complicated pipeline.

Results: 16599ms (read + multiply + write) with $n = 20000$, $m = 8$, $k = 100000$.

Drawbacks: No parallelization / use of GPU.

1.2 Approach 2: GPU Baseline

Idea: Read and create the full n x n matrices of A and B, transfer to GPU, multiply on GPU kernel, and return the result back to CPU. There are n/m x n/m blocks with m x m threads in each block. Write the result matrix on the file after multiplication while checking for zero-blocks. Free all the memory after run.

Why attempted: To use basic GPU arithmetic for parallelization.

Results: TIMEOUT (read + multiply + write) with $n = 20000$, $m = 8$, $k = 100000$.

Drawbacks: Memory required is high. Poor representation of spare matrices.

1.3 Approach 3: GPU Spare Representation

Idea: Read each matrix and store the blocks into a k x m x m size 1D array. Create a n/m x n/m size `non_zero_block` array to check if a particular block exists and if it exists, where does it exist inside the 1D array above, else store –1. There are n/m x n/m blocks with m x m threads in each block. Each block iterates over n/m block of it (row, x), (x, col) pairs and checks if blocks exist in A and B both, else continue. Store the result in a nxn matrix and write the result matrix on the file after multiplication while checking for zero-blocks.

Why attempted:
To improve the memory consumption to store matrices, and transfers between CPU and GPU.

Results: 37497ms (read + multiply + write) with $n = 20000$, $m = 8$, $k = 100000$.

Drawbacks: Memory usage is fine tuned, but the algorithm is poor. The code spends 28148ms inside the kernal function. Each grid block inside GPU runs a $O(n/m)$ for loop which is very slow.

1.4 Approach 4: GPU CSR Representation

Idea: To improve the algorithm use the CSR representation of spare matrices which stores the columns/rows that exist inside the matrix. Read the blocks and sort the blocks row-wise and col-wise. Now each block spends less time checking if blocks exists. Store the result in a n x n result matrix. Write the results to the file while checking for zero-blocks using a for loop.

Why attempted: Improve the time complexity of the algorithm.

Results: 13826ms (read + multiply + write) with $n = 20000$, $m = 8$, $k = 100000$.

Drawbacks: Uses less threads per block, cache utilization is poor for results matrix due to m x m block structure, checking for zero-blocks during writing to file is slow.

1.5 Final Approach: GPU Fine Tuned

Detailed approach:
1. Read the files, sort with respect to row and column indices and store the matrices in CSR format. A is stored in row-major format, B is stored in column major format. This helps in improving the cache utilization during multiplication.
2. Transfer the matrices A, B into GPU global memory and allocate memory for a result matrix.
3. Launch n/m x n/m blocks with 2 x m x m threads in each. Each block of GPU needs to multiply $O(n/m)$ blocks. Allocate two threads to multiply $O(n/2m)$ blocks each and store the results in shared memory array. Root thread will add theses two in shared memory and update the final result to the result matrix. This uses syncthreads and shared mempory for optimization.
4. The result matrix is stored block wise instead of n x n full matrix. Thus, we store n/m x n/m blocks of m x m size each in a row major format for better cache utilization.
5. A `non_zero_array` is used and marked as true whenever a block encounters a non-zero value in the sum inside the kernel call in GPU. This improves the efficiency of the write function.
6. If the sum grows more than `MAX_VAL` the loop stops and breaks on hitting the upper cap.
7. Loop unrolling is used since its known m is either 4 or 8. This improves the speed of computation, freeing the GPU of loop initialization, updates and conditional checks.

Why attempted: To reduce memory usage by not storing the intermediate result matrix.

Results: 10714ms (read + multiply + write) with $n = 20000$, $m = 8$, $k = 100000$.
This is 1.5x faster than approach 1. The speedup grows with the number of non-zero blocks.

Read Files: 327ms	**Load A, B to GPU:** 99ms
Multiply Matrices: 1757ms	**Load C to CPU:** 1206ms
Write C to File: 7322ms	**Total Time Taken:** 10714ms

Large Test Case:
Input: $n = 32768$, $m = 8$, $k = 8800000$
Output: $n = 32768$, $m = 8$, $k = 16777216$
Total Time Taken: 354740ms. Approach 1 times out.

1.6 Code Description:

1. Block: Used to store a block of a sparse matrix. The Block structure has three members: `row` and `col` are integers that store the row and column indices of the block, respectively. `data` is a vector of integers that stores the non-zero blocks.

2. readMatrix: This function reads a sparse matrix stored in a binary file. It takes the filename, the number of rows, the block size, the blocks array, a rowIndex array, and a colIndex array used to store in CSR format.

3. compareColumn/compareRow: This function compares two blocks based on their column/row index. It is used in the readMatrix function to sort the blocks in each row/column based on their column/row index.

4. writeMatrix: This function writes a block sparse matrix to a binary file. It takes as input the output file name, the number of rows, the block size, a n x n result array and a non-zero bool-list. It writes the non-zero blocks into the output file.

5. matrix_multiply_kernel: This function runs on the GPU kernel and computes the result of a block-matrix multiplication. The detailed algorithm is described above.

6. multiplyMatrixWrapper: This performs matrix multiplication of two matrices stored in binary files and writes the output to a binary file. The matrix multiplication is performed by the matrix_multiply_kernel function which is executed on the GPU. The multiplyMatrixWrapper function reads the input matrices from files and stores them in memory on the host. It then copies the matrices to the device and calls the matrix_multiply_kernel function to perform the multiplication. Finally, it copies the result matrix back to the host and writes it to a binary file.

The matrix_multiply_kernel function takes in the matrices d_a and d_b on the GPU, as well as the row and column indices of the non-zero elements in the matrices. It uses these indices to perform a sparse matrix multiplication, only computing the products of the non-zero elements. It stores the result in the matrix d_c on the GPU, as well as a boolean matrix d_non_zero which is used to indicate which blocks of the result matrix are non-zero.

The multiplyMatrixWrapper function first reads the matrix dimensions and block size from the input files. It then allocates memory on the host and device for the input and output matrices, as well as the row and column indices of the non-zero elements. It reads the input matrices from files and stores them in memory on the host. It then copies the matrices and indices to the device and calls the matrix_multiply_kernel function to perform the multiplication. Finally, it copies the result matrix and boolean matrix back to the host and writes them to binary files.

`NOTE: code files of each approach are submitted.`