ECE408 Final Report –
GPU Accelerated Ensemble Kalman Filter

Ke Liu, Tingou Liang, Mengxuan Li

1. Background

Our project is from Prof. Butala's research which combines data and physics models to characterize regions governing/affected by space weather. He uses Kalman filter, a classic linear state estimation procedure, as an optimal solution method and wrote sequential codes to build a Kalman filter. Our goal is to help reduce the running time, more generally, to write parallel codes to compute P_HT using the equation `P_HT = np.multiply(C, e @ e.T) @ H.T / (L-1)`[1]. Here, C is a symmetric, block Toeplitz matrix; e is a dense matrix; H is a sparse matrix; L is the number of the columns of e.

2. Data Structure

There're four different structures in our projects: `sparse_coo, sparse_rcs, full_r` and `toeplitz.`

- **sparse_coo**

It's the Coordinate (COO) format to store a sparse matrix and our structure includes 6 elements: N – the number of nonzero values; m – the number of rows; n – the number of columns; v – nonzero values; j – column indices; i – row indices. We use function `sparse_coo_create` to create a `sparse_coo` structure. Also, we use function `sparse_coo_free` to free a `sparse_coo`.

- **sparse_rcs**

It’s the Compressed Sparse Row (CSR) format to store a sparse matrix and it includes 6 elements: N – the number of nonzero values; m – the number of rows; n – the number of columns; v – nonzero values; j – column indices; r – row pointers. We use `sparse_rcs_create` and `sparse_rcs_free` to create and free `sparse_rcs` which are similar to the functions of `sparse_coo` except r has the size of (m+1).

- **full_r**

It’s the structure to store a full rank matrix and it only includes 3 elements: m – the number of rows; n – the number of columns; v – values (a 2D array). We use `full_r_create` and `full_r_free` functions to create and free a `full_r` structure. Since it includes a 2D array, it needs to use `malloc()/free()` m times to allocate/free memory for each row.

- **toeplitz**

It’s the structure to store a block Toeplitz matrix and it only includes 3 elements: N – the number of nonzero in the first row; n – the total number of elements in the first row; row0 – values in the first row.

A Toeplitz matrix is a matrix in which each descending diagonal from left to right is constant. It looks like the left one below, if the element at (i, j) of A is denoted $A_{i,j}$ then we have $A_{i,j} = A_{i+1,j+1} = a_{i-j}$.

$$A = \begin{bmatrix} a_0 & a_{-1} & a_{-2} & \cdots & \cdots & a_{-(n-1)} \\ a_1 & a_0 & a_{-1} & \ddots & & \vdots \\ a_2 & a_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_{-1} & a_{-2} \\ \vdots & & \ddots & a_1 & a_0 & a_{-1} \\ a_{n-1} & \cdots & \cdots & a_2 & a_1 & a_0 \end{bmatrix}$$

figure 1[2]

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{(1,1)} & \mathbf{A}_{(1,2)} & & & \cdots & \mathbf{A}_{(1,n-1)} & \mathbf{A}_{(1,n)} \\ \mathbf{A}_{(2,1)} & \mathbf{A}_{(1,1)} & \mathbf{A}_{(1,2)} & & & & \mathbf{A}_{(1,n-1)} \\ & \ddots & \ddots & \ddots & & & \vdots \\ & & \mathbf{A}_{(2,1)} & \mathbf{A}_{(1,1)} & \mathbf{A}_{(1,2)} & & \\ \vdots & & & \ddots & \ddots & \ddots & \\ \mathbf{A}_{(n-1,1)} & & & & \mathbf{A}_{(2,1)} & \mathbf{A}_{(1,1)} & \mathbf{A}_{(1,2)} \\ \mathbf{A}_{(n,1)} & \mathbf{A}_{(n-1,1)} & \cdots & & & \mathbf{A}_{(2,1)} & \mathbf{A}_{(1,1)} \end{bmatrix}.$$

figure 2[3]

A block Toeplitz matrix may look like the right one above, it contains blocks that are repeated down the diagonals of the matrix and is still a Toeplitz matrix but the individual block matrix elements, A_{ij}, are still Toeplitz matrices.

We use `toeplitz_create` and `toeplitz_free` functions to create and free a `toeplitz` structure which use `malloc()/free()` to allocate/free memory for row0, a 1D array, and assign the value of N and n.

3. Data Transformation

Since the provided data is stored in .sdb format and it's hard to use .sdb format as inputs directly, we decided to write python codes to restore the data into csv files for later use. Here, the point is the time taken by transforming data from .sdb to csv is about 1s for a matrix with size 10000 x 10000, which it is not a big part of our total running time.

- **Import data**

In provided data, there're only two kinds of formats: CSR, related to C and H; full rank, related e. We wrote `import_e` to import e, which first reads three important integers bytes-the size of the element, m-the number of rows, n-the number of columns. Also, the function `import_H` is written to import C and H which are both stored in CSR format. It reads 4 important integers: bytes – the size of the element, m – the number of rows, n – the number of columns, N – the number of nonzero values and returns `csr_matrix((v,j,r), shape=(m,n))` which will create a CSR format in python.

- **Export data**

In export data, there're four kinds of formats: CSR and COO, related to H; full rank, related to e; values in the first row, related to C. We wrote `export_e` to write the e matrix imported from `import_e` to a csv file. Also, export_rcs_1 is the function for exporting H matrix imported by `import_H` with the same method as `export_e` but writing [N,m,n] in the first row and [v,j,r] in the following three rows respectively. Another function `export_coo` is designed to transform the sparse matrix from CSR format to COO format which will be used in sequential codes. It writes [N,m,n] in the first row and [v,j,i] in the following three rows respectively. Finally, `export_sb_toe_r` is the function for C which is a block Toeplitz matrix, it writes the size of element, rank, the number of nonzero values in the first row and the total number of the first row in the first row in csv file. Then the values in the first row of the matrix are stored in the second row in csv file.

After finishing this part, these csv files are used to do the sequential or parallel calculations.

4. Overview of sequential code

We follow the rudimentary step to compute the result in sequential code. First, we import the data from the pre-processed .csv files, and then do each step of computation, which includes:

1) Transpose the full-rank matrix e and then calculate e^2 (The “transpose” step could be eliminated, but we had not thought about it at that time);
2) Compute the element-wise product of e^2 and the block Toeplitz matrix C, using C_e2 to denote the resulting product (in COO formate);
3) Transpose the COO matrix H to get H_T and compute the product of C_e2 and H_T and divide by L-1, using P_HT to denote the resulting product (in COO format);

Finally, we transform P_HT into CSR format and write it to an output .csv file.

5. Overview of parallel code

For importing and outputting the data, the parallel code is the same as sequential code. The major change in the algorithm is that we eliminated some redundant step and used CUDA kernels to reduce the time consumption by the for-loop. Here are the steps for computation in our parallel code:

1) Directly compute the square of a full-rank matrix e in a kernel;
2) Compute the element-wise product of e^2 and the block Toeplitz matrix C with the result stored in C_e2 as a full-rank matrix;
3) Directly compute P_HT in CSR format with the input H (a matrix in CSR format), C_e2 and L.

6. Functions

6.1. Functions for import data

Function name	Description
`sparse_rcs_import`	These functions take the filename of a .csv file as input and returns the pointer to an allocated structure of matrix with certain format (e.g., CSR, full-rank, etc.). They are used in both the sequential and parallel code.
`sparse_coo_import`	
`full_r_import`	
`toeplitz_import`	

6.2. Functions for sequential algorithm

Function name	Description
`matrix_T`	This function transposes a full-rank matrix.
`coo_T`	This function transposes a matrix in COO format.
`matrix_m1`	This function performs matrix multiplication of two full-rank matrices `a_matrix` and `b_matrix`. The result is put in `c_matrix`.
`matrix_find`	This function goes through the COO structure A to check if the element at (row, col) is nonzero. If so, then return true and put the value of the element in the address pointed by the input `e`; otherwise, return false.
`matrix_m3`	This function performs matrix multiplication of two matrix in COO format and returns a matrix in COO format.
`toe_full_elem`	This function performs element-wise multiplication between a block Toeplitz matrix and a full-rank matrix. Then it returns the result in COO format.
`matrix_trans`	This function transforms a matrix `A` from COO format to CSR format. The result is stored in `B`.

6.3. Functions for parallel algorithm

Below are the CUDA kernel functions used in our parallel algorithm. In practice, each CUDA kernel function should be accompanied by a wrapper function to link C code to CUDA code.

Function name	Description
`sparseVector`	This kernel performs matrix multiplication between a full-rank matrix and a matrix in CSR format. The result is stored in full-rank format.
`toe_fr_elem`	This kernel performs element-wise multiplication between a full-rank matrix and a block Toeplitz matrix. The result is stored in full-rank format.
`denseMatrixSquare`	This kernel computes the square of a full-rank matrix and stores the result in full-rank format.
`calcNonzero`	This kernel counts the number of nonzero elements in a full-rank matrix.

7. Optimization

7.1. Optimization on e @ e.T (@ stands for matrix multiplication, which is np.dot() in Python)

Before the optimization, we wrote a kernel to calculate e.T firstly, which was a waste of memory (to store the transposed matrix) and time, and then we used another kernel to do the matrix multiplication (the row of e * the col of e.T). After the optimization, we just use one kernel to do the matrix multiplication (the row of e * the row of e). The necessary changes to the code include deleting the transposition kernel and the variables related to the transposed matrix, changing the thread index of the second input matrix. The benefit of this optimization includes not only the smaller memory usage but also the consecutive access to the memory and decrease one kernel function call. The actual running time indeed decreases by about one second for matrix size of 10000 x 10000 (5% speed up).

7.2. Optimization on C_2 @ H.T (C_2 = np.multiply(C, e_2))

In this part, our optimization is based on the formula. Essentially, the formula requires us to do the multiplication of C_2 and the transposition of H (the row of C_2 @ the col of H.T). Then we decide to use the fact that A @ B = (A.T @ B.T).T, in other words, C_2 @ H.T = (H @ C_2.T).T. So we decide to do the multiplication of H and the transposition of C_2 (the row of H @ the column of C_2.T). Since the col of C_2.T is equal to the row of C_2, we don't need to do the transposition anymore and just adjust the thread index we use in the corresponding kernel (same idea as optimization 1). Also, the result we get is the col of (H @ C_2.T) and if we store it as the row of the output, then the final answer is (H @ C_2.T).T (same idea as Optimization 1). The benefits of this optimization are following: Firstly, we do not need to access a CSR matrix in column major, which is will take more time than access in row major since it cannot make good use of DRAM burst; secondly, we cancel some unnecessary steps, like calculating the transposed matrix again and again. The actual running time indeed decreases by about three second for data size of 10000 x 10000 (15% speed up).

7.3. Optimization on H @ C_2 (C_2 = np.multiply(C, e_2))

Previously, we used one-dimension block to do the calculation, which means we wrote a kernel to multiply a sparse matrix and a vector (as in the lecture notes) and used a for loop to get every row of the full rank matrix. Thus, we needed to launch kernel for multiple times. Now we use a two-dimension block and use the y-dimension to represent which row in the second matrix is being calculated, and by this way we call the kernel only once. The main change here is deleting the for loop to launch the kernel for multiple times and adding a new dimension to denote the row number of the full-rank matrix (`int col = blockIdx.y * blockDim.y + threadIdx.y`). The actual running time indeed decreases by about three second for data size of 10000 x 10000 (15% speed up).

8. Time Analysis

Our original formula in Python:

```
P_HT = np.multiply(C, e @ e.T) @ H.T / (L - 1)
```

The size of our input matrix:

Matrix Name	Size	Store method	Description
H	H_row * H_colomn = 129 * 16384	CSR	the number of non-zero element is NH = 32768
C	C_row * C_colomn = 16384 * 16384	Toeplitz (The first row)	the number if non-zero element in each row is N_row_C = 1024
e	e_row * e_colomn = 16384 * 64	Full rank	a dense matrix

Time analysis for each step of our modified parallel code:

Step	Number of floating-point operations (multiplication and addition)	Number of threads
e @ e.T	e_row * e_row * e_colomn + e_row * e_row * e_colomn	e_row * e_row
np.multiply(C, e @ e.T)	C_row * C_colomn	C_row * C_colomn
np.multiply(C, e @ e.T) @ H.T	C_colomn * NH, where NH denotes the number of nonzero elements	H_row * C_colomn
Count # of non-zero element	H_row * H_colomn	H_row

Total	O(e_row * e_row * e_colomn + C_colomn * NH)	-

As a result, the total time complexity of our algorithm is **O(e_row * e_row * e_colomn + C_colomn * NH)**, which depends on how sparse our sparse matrix H is. When applying our algorithm to the real data provided by professor Butala, the number of total valid (non-zero) multiplication or addition is about 16384*16384*64 $\approx$ **$2*10^{10}$**, which is much smaller than the value we write in our presentation, since it counts all multiplication including zeros.

9. Summary

Here is the running time of different version code on the same real data (the set of data provided by Professor Butala and its size is provided above):

Version	**Running time**	**Description**
Our sequential code	More than 1 hours	We don't make use of any special structure so the speed is OK when we tested with small dataset on the first week but it is quite slow dealing with the real data.
Sequential code from Butala	About 1 minute	This running time is provided by Professor Butala
Basic parallel code	20 seconds	Counting from importing data from .csv file and then do the calculation and export the result into a .csv file.
Python code	17 seconds	Counting from importing data from .sdb file and do the calculation.
Optimized parallel code	9 seconds	Counting from importing data from .csv file and then do the calculation and export the result into a .csv file.

(Important: Both of our basic parallel code and optimized parallel code spend about 1-2 seconds to import data from .csv file and export the result to a .csv file for the data size provided above.)

10. Reference

[1]: This expression is from Line 144 in compute_P_HT.py, provided by Prof. Butala.

[2]: https://en.wikipedia.org/wiki/Toeplitz_matrix

[3]: https://en.wikipedia.org/wiki/Block_matrix#Block_Toeplitz_matrices