ECE 408 final report

Team members: Ching-Hua Yu, Hanchen Ye, Hanxiao Lu, Mihir Rajpal

Resources

We did not make use of a private GPU to run our project on it. Instead we use RAI to help us build and debug the project. The resources we utilize in this project are listed below:

1. A working C version of Kalman filter
2. A working python version of Kalman filter
3. A python script that generates random inputs of arbitrary size
4. Libmdb_matrix library, GNU Scientific Library (GSL), libconfuse, Fastest Fourier Transform in the West (FFTW), especially with packages:
   libgsl-dev
   libgsl-bin
   libgsl23
   libgslcblas0
   fftw2
   libopenblas-dev
   liblapacke-dev
   libfftw3-dev
   Exuberant-ctags
5. a working version of Python 3 with packages:
   Python3-numpy
   Python3-scipy
   Python3-sh

Application

The problem we are solving is to use cuda to accelerate the linear algebra computation part on the Toeplitz matrix, which is a sparse and structured matrix. Specifically, we are using cuda to write down a version similar to python version with formula :

$$\{[C \cdot (e @ e \hat{} T)] @ H \hat{} T \} / (L - 1)$$

Where C is a symmetric, N×N, block Toeplitz matrix, e is an N×L matrix, and H is an M×N matrix. N is large (10k, 100k, or larger), while M and L are small (10s). @ is matrix multiplication and T is transpose.

Python has a built-in library package numpy that supports the same computation. But our goal is to write our own code and use parallel programming to defeat this built in computation in terms of runtime and performance.

Background

We studied and tested the given sequential code of the targeted algorithm on the RAI system.This sequential code runs a C version and a Python version of the algorithm and compares the results of these two implementations. The testing results the RAI system returned to us are shown as below, which indicates that the sequential code is correctly executed and produces the expected outputs.

```
loading e from /tmp/e
N=100  L=10

loading H from /tmp/H
m=20  n=100  N=2000

loading C from /tmp/C
rank=1
n_phy=[ 50 ]
n    =[ 100 ]
N_phy=[ 50 ]
N    =[ 100 ]
P_HT_c == P_HT? True
```

As the problem described, the computation involves { [C · (e @ e^T)) @ H ^T} / (L – 1) where C is a symmetric, N×N, block Toeplitz matrix, e is an N×L matrix, and H is an M×N matrix. N is large (10k, 100k, or larger), while M and L are small (10s). Because the sparsity of the matrix C may not be that optimistic (detailed later), we will start by discussing the case when C is not a general matrix, an extremely sparse matrix, a sufficiently sparse matrix, and finally when C is a block toeplitz matrix . From Prof. Butala, the current application only needs C to be a symmetric toeplitz matrix.

We are going to leverage the fact that M is small, so splitting H ^ T into vectors of length N instead of utilizing the sparsity is more advantageous since the potential sparsity also introduces lack of regularity, which can be a significant burden according to the lectures on sparse matrix multiplication. This technique uses the formal definition

of matrix multiplication I learned in Math 415 to make things faster. I am also going to leverage the fact that C is a block Toeplitz matrix.The steps in the next paragraph are repeated for each unit of rows equivalent to the block width B starting with the first B rows.

First, we check if N is small enough that we can run everything in one kernel or we have to split it up. If we have to split it up, we split the row into several segments so that the following algorithm will work and then call a separate kernel solely for reducing the results obtained into the final result. Next, we linearize blocks of C by copying the last row of it and then appending the last B - 1 elements of the first row of the block to our copy of the last row of the block. From this, with the appropriate indexing, we can reproduce any row of that block of C by simply computing the appropriate index offset. The reason this is possible is that C is a block Toeplitz matrix, which is a special type of matrix where the blocks have the property where all elements on the same diagonal are equal. Therefore, the first row is simply the last B elements of this 1D array, the second row is obtained by shifting the start index one unit to the left, the third row is obtained by shifting the start index two units to the left, and so on until we reach the last row, which is the first B elements of the 1D array. Also, we store the first B rows or e ^ T in constant memory since computing an element with row index i and column index j of e times e ^ T is equivalent to taking the inner product of the i th and j th rows of the matrix e ^ T , and the access patterns for memory improve if we read a single row as opposed to a single column given matrices are stored in row-major order and we use this data in every block since we are computing the first few rows of the resulting matrix. Then, we copy H and e ^ T into the GPU if we haven't already done so or different memory from the memory copied is necessary and then launch a kernel (called K1 for convenience). Each block in K1 will copy the necessary rows of e ^ T that are not in constant memory into shared memory and then compute the C dot (e @ e ^ T) for that block and then launch a separate kernel (called K2 for convenience). Each block in K2 will pick a row of H (columns/vectors in H ^ T) and multiply the corresponding scalars by the corresponding elements of the block and then perform a reduction to get the corresponding output for the block, which then a fixed set of blocks in K1 will perform another reduction on to get the corresponding B rows of the output, which is then left on the GPU until we are done processing the entire matrix.

Finally, we copy the entire resulting matrix back to the CPU if we haven't already done so and we have our result. The runtime of this algorithm, from a theoretical perspective and assuming no serialization (for now) is $\Theta(N+MN+LN + \log_2(B) + \log_2(N/B)) = \Theta(N(M+L)) \approx \Theta(N)$ since M <<N and L << N. This is a reasonable runtime since M and L are usually set by the user and the user would understand that increasing these values would increase the runtime as well as the fact that this computation is

otherwise being done in linear time. This algorithm, however, requires compute capability 3.5+, so we humbly request to use hardware that is a little more modern.

Implementation

In milestone2, we implement a simple version of CUDA code for the targeted algorithm. In this CUDA implementation, the sparsity of matrix H and the special structure of the Toeplitz matrix C is not considered, thus all the matrices are stored and computed as dense matrices. We implement 2 CUDA kernels, compute_P_kernel and compute_P_HT_kernel, for P = C · (e @ e ^ T) and P_HT = (P @ H ^ T) / (L – 1), respectively. We implement a host function for loading data into the GPU memory, calling CUDA kernels, and writing back results. The execution results of RAI are shown as below, when N=100, L=10, and M=20.

```
0. Problem size: N=100, L=10, M=20

1. Load data into CPU memory.
Loading /tmp/e into CPU memory...
Size of /tmp/e is: 1000
Loading /tmp/H into CPU memory...
Size of /tmp/H is: 2000
Loading /tmp/C into CPU memory...
Size of /tmp/C is: 10000
2. Allocate GPU memory.
3. Write data into GPU memory.
Latency: 0.045000ms
4. Call GPU cuda kernel.
Latency: 0.042000ms
5. Read results from GPU memory.
Latency: 0.026000ms
6. Save results to file.
Saving /tmp/P_HT into file...
Size of /tmp/P_HT is: 2000
7. De-allocate CPU and GPU memory.
P_HT_cuda == P_HT? True
```

Optimization

Optimization 1

For the first optimization, we improve the memory storage efficiency of matrix C, leveraging the special structure of the Toeplitz matrix. In the simple implementation, we

consider C as a normal dense matrix. In this optimization, we are using the first row and the last row to represent the whole matrix C, which largely reduces the memory budget of storing matrix C from N*N to 2*N. Basically, in the python host code, we compress matrix C in this way:

```
C_compress = np.concatenate((C[-1], C[0, 1:], np.zeros(1)))
```

While in the CUDA kernel, we index the value in matrix C with the following code:

```
C[N - 1 + P_col - P_row]
```

where P_col and P_row is the coordinate of the elements in matrix C.

Optimization 2

For the second optimization, we fuse the compute_P and compute_P_HT kernels in the simple implementation to one compute_P_HT kernel, for avoiding to store the N*N dense matrix P in GPU memory. Previously, we first calculate all the elements in matrix P and store them in the GPU memory, then call compute_P_HT kernel to multiply the matrix P and matrix H. After fusing these two kernels into one kernel, we use a shared P_tile array to store a tile of matrix P in the shared memory of GPU:

```
__shared__ double P_tile[TILE_SIZE][TILE_SIZE];
```

This optimization avoid us to store the large N*N matrix P in GPU memory. Meanwhile, this shared P_tile array also opens some opportunities for us to reuse the matrix P between different threads in the same thread block.

Optimization 3

For the third optimization, we take use of the sparsity of matrix H to reduce the total operations of the algorithm. Given matrix H is a sparse matrix of which the density is typically lower than 10%, we choose to use CSR format to store and transfer matrix H. We first compress the matrix H into CSR format in the host python code, and pass non-zero values of matrix H to the CUDA kernel. On one hand, this optimization will help us to reduce the memory budget of storing matrix H, on the other hand, the high sparsity of H allows us to perform less operations while generating the same results. In the CUDA code, we fetch the non-zero values in matrix H with the following code:

```
int H_indptr_start = H_indptr[P_HT_col];
int H_indptr_end = H_indptr[P_HT_col + 1];

… …

for (int t = H_indptr_start; t < H_indptr_end; t ++) {
    int H_tile_col = H_indices[t] - n;
    if (H_tile_col < TILE_SIZE) {
        P_HT_value += P_tile[tile_row][H_tile_col] * H[t];
        H_indptr_start += 1;
    } else {
        break;
    }
}
```

Results

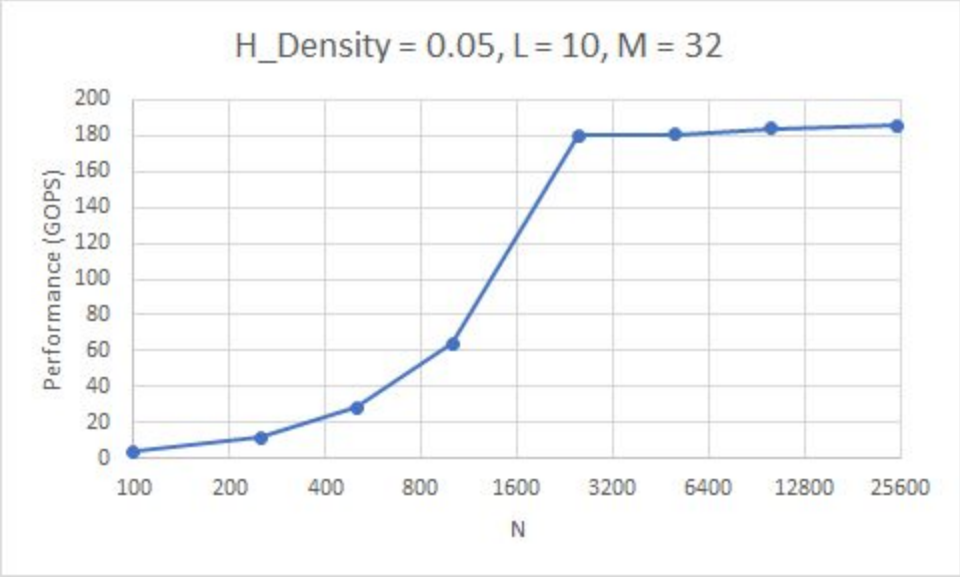
We evaluate our implemented CUDA kernel on RAI remote system, to evaluate the performance of our kernel under different size of N and different sparse density of H, we perform two sets of experiments. The results are shown in the following tables.

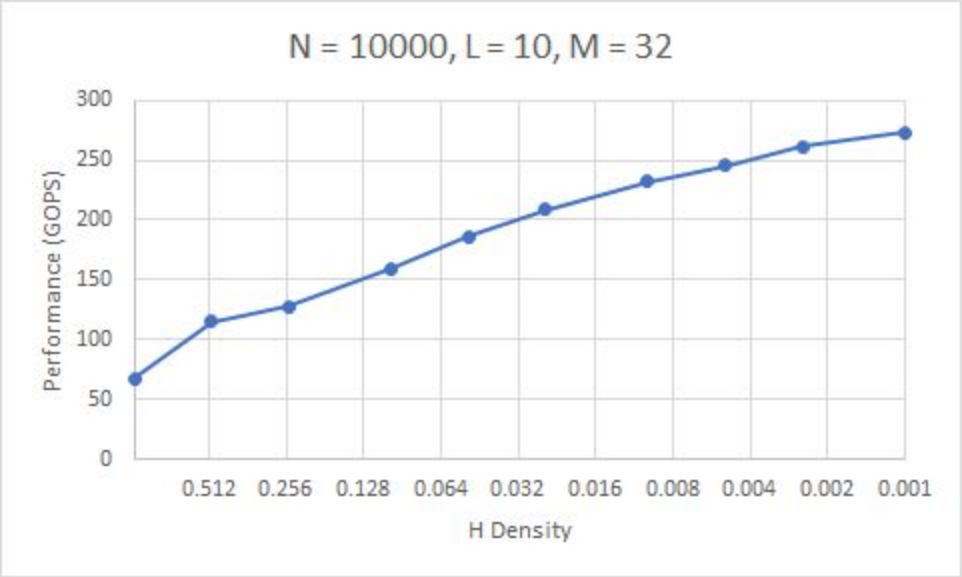
H_Density	N	Prob. Size	Lat. (ms)	Perf. (GOPs)
0.05	100	0.000126	0.034	3.705882353
0.05	250	0.0007875	0.068	11.58088235
0.05	500	0.00315	0.112	28.125
0.05	1000	0.0126	0.196	64.28571429
0.05	2500	0.07875	0.438	179.7945205
0.05	5000	0.315	1.746	180.4123711
0.05	10000	1.26	6.861	183.6466987
0.05	25000	7.875	42.372	185.8538658

H_Density	N	Prob. Size	Lat. (ms)	Perf. (GOPs)
1	10000	4.3	63.703	67.50074565
0.5	10000	2.7	23.519	114.8007994
0.25	10000	1.9	14.91	127.4312542
0.1	10000	1.42	8.889	159.7480031
0.05	10000	1.26	6.776	185.9504132
0.025	10000	1.18	5.655	208.6648983
0.01	10000	1.132	4.874	232.2527698
0.005	10000	1.116	4.545	245.5445545
0.0025	10000	1.108	4.235	261.6292798
0.001	10000	1.1032	4.044	272.7992087

In the first experiment, we keep H_Density = 0.05, M = 32, and L = 10, and scale the size of N from 100 to 10,000. We can observe that the performance is increasing with the scale up of N, which indicates that our implementation has high scalability and is able to deliver high performance for large samples. In the second experiment, we keep N = 10,000, M = 32, and L = 10, and scale the sparse density of matrix H from 1 to 0.001. Our CUDA kernel successfully achieves a significantly lower latency and higher performance under low density of H, which is the typical case in real-world applications.

We also visualize the performance of our kernel in the following figure, which clearly shows the performance curve of our CUDA kernel under different configurations of the size of N and the density of H.





Here, we did not perform control experiments for demonstrating the impact of our optimization, because these optimizations not only improve the performance of the kernel, but most importantly, they ensure the kernel is runnable under the case of large input samples. If the special structure of matrix C and the sparsity of matrix H is not considered, the GPU memory limitation will prevent us from performing the experiments above.

Conclusion

In terms of better performance of our code, we could use parallel programming to speed up the code execution rather than existing library packages. We don't have to use numpy to calculate the matrix multiplication when we are not satisfied with the runtime. Instead, writing our cuda code could easily speed up the whole process with decent effort.

When we design the kernel function, we should carefully evaluate the complexity of building kernels with different dimensions. Specifically, 2-D dimension or 3-D dimension is sometimes complicated to build. Our group firstly found out the way to build up 2-D and 3-D dimension kernels. But this fails because of some unknown bugs. Therefore we turn to the 1-D dimension kernel and finally we make it success without any hitch.

The CUDA kernel code can be found under the /topic-1-enkf/src directory. And executing `rai -p .` under the /topic-1-enkf directory allows you to test our kernel on the RAI system. For the 2-D case of the algorithm, we largely reuse the same code as the 1-D case, where only the decompress of matrix C is changed. Some bugs make the

results uncorrect. However, in the case of 2-D, our kernel is able to deliver the same performance as the 1-D case.

Reference

We didn’t reference any conference or journal papers for this project.

Appendix

Project: Kalman-filter
http://lumetta.web.engr.illinois.edu/408-S20/projects/Kalman-filter.pdf

Public gitlab (sequential codes etc)
https://gitlab.engr.illinois.edu/ece408_sp20/ece408-final-project/

Team gitlab
https://gitlab.engr.illinois.edu/ece408_sp20/group_30