CS483 Milestone 1 Document

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

1. Sequential Code Evaluation

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
```

2. Math Part (Potential) Improvement

As the problem described, the computation involves $\{ [C \cdot (e \mathbin{@} e^T)) \mathbin{@} H^T \} / (L - 1)$ 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 .

[Updated] From Prof. Butala, the current application only needs C to be a symmetric toeplitz matrix.

2.1. [General C]

First, note that if C is removed from { [C · (e @ e^T)) @ H^T } / (L – 1) and temporary leave out L-1 since it's just a scalar, we can apply the associative property of the matrix multiplication on (e @ e^T) @ H^T to get e @ (e^T @ H^T), which can save a lot of computation since L, M<<N. Now consider the computation involves C = [C_1, …, C_N], where Ci is the i-th row of C, and let e = [e_1, …, e_n], where ei is the i-th row of e. In our computation, we can first calculate Di = ((e^T ⊙Ci) @ H^T), where e^T ⊙Ci here means multiplying each row of e^T by a row vector Ci like e^T *Ci in python, for i from 1 to N, and then calculate fi = ei @ Di for i from 1 to N before concatenate them to get f = [f1,...,f_N]. The computation job is equivalent to one NxN matrix dot product, one NxL matrix and LxM matrix multiplications and N times of LxN matrix and NxM matrix multiplications.

2.2. [Extremely Sparse C] (deferred from this project)

Second, suppose C is extremely sparse. Note that 90% sparse may not be good enough. If C is extremely sparse, e.g., only O(N) elements are non-zero, the computation can be reduced accordingly.

2.3. [Sufficient Sparse C] (deferred from this project)

For applications that approximation is acceptable, another way is to do eigenvector matrix decomposition, given a sufficient sparse C. Suppose C is approximately decomposed to $Q\Lambda Q^{-1}$, where Λ is the diagonal matrix, and Q is nxL matrix from the L most significant eigenvectors of C. Then, we can tweak the computation to have d = Q · e and essentially mainly compute d @ (d^T @ H^T). However, in this case, we need to survey further about eigenvalue decomposition for matrices in parallel computation.

2.4. C is a block Toeplitz matrix

In this case, C only takes O(N) memory, and in parallel programming, we can save global memory access. Essentially each Ci from the general case (2.1) can be from the same O(N) memory, similar to the advantage of the extremely sparsity. Nevertheless, when N is large, more parallel programming methods will need to discuss.

3. Initial discussion on parallel programming implementation

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.

An update of this document
https://docs.google.com/document/d/1ZUH5jbEhxNwV5M9TZpUCagT3fq6v1j50mUe9fCsvcdE/edit?usp=sharing

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