CS483 Milestone 2 Document

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

1. Code Results

For this milestone, 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 $P = C \cdot (e \mathbin{@} e^T)$ and $P\_HT = (P \mathbin{@} H^T) / (L - 1)$, respectively. We implement a host function for loading data into the GPU memory, calling CUDA kernels, and writing back results. The CUDA kernels and host function can be found in directory `topic-1-enkf/src/`. The execution results of RAI are shown as below.

```
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
```

2. Potential Points of Improvement for Final Checkpoint

For this code, we decided to follow the wise advice of Professor Erickson in his book *Algorithms* and start by implementing code that will work without worrying too much about time and memory usage. This algorithm should run faster than the sequential code since it does parallelize the computation using two simple matrix multiplication kernels. As a result, control divergence is not an issue, but memory coalescing, reuse, overall memory usage, and memory bandwidth to and from the GPU are issues that could hinder our performance as N gets larger and larger. As a result, we are going to need to modify our approach so that our memory usage is more realistic even as N gets larger than the size of the CPU's RAM (ideally). The first step we would need to take to do this is to determine the size of the blocks of the C matrix in order to know what size blocks we should break the computation into, at

least initially. For this, a naive solution would be to check every possible block size by testing the appropriate borders for equality given that the slides recently released to us indicate that the C matrix can be divided into identically-sized blocks. While this has some level of potential parallelism associated with it, it might be made more efficient by using a different approach, which is a potential avenue of exploration after all other potential optimizations have been made. Then, once this block size is known, we can leverage this to split the computation up into more manageable chunks and try other approaches to the multiplications itself within each block to take advantage of structure of the matrices as well as using shared and constant memory to reduce internal memory latency within each block. We can also take advantage of the sparsity of the matrices to reduce the memory bandwidth to and from the GPU and to and from Global Memory, which could be constraining factors if we are not cognizant of these limitations. These approaches will be attempted by the final project due date and we may consider including different approaches in our final result depending on the data size and block size if these optimizations make smaller computations slower while making larger computations possible.

An update of this document
https://docs.google.com/document/d/1lju6xf8X0x3y-CWX0RUFIaaU_oJbJmcNVAj6UFz7z9s/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