GPU-Based Parallel Kalman Filter

Zhiyuan Lin, David Moore, Stuart Russell

University of California, Berkeley

Abstract—The Kalman Filter and its variants have been highly successful in numerous applications in technology. However, the Kalman filter is under heavy computational burden. When suffers from big data, it becomes pretty slow. On the other hand, the GPU, a processor unit with highly parallel structure, becomes more and more popular in generous purpose computing. This paper focuses on how to make Kalman filters faster on GPU while introducing flexibility between accuracy and speed. Our parallel Kalman filter can achieve nearly linear speedup in simple applications and can outperform CPU program by an order of magnitude in some real world applications.

I. INTRODUCTION

Kalman filtering [1] is a well known algorithm that has been widely used in many applications for state estimation of linear systems. It uses a series of observations over time to produce estimates of unknown states. More precise than those based on a single observation alone. However, the matrix operations involved in Kalman filter updates can be a computational burden. In order to solve this problem, we explore potential ways to parallelize the algorithm.

A natural way to make a Kalman filter faster is to parallelize it on a Graphics Processor Unit (GPU). Most existing ways to parallelize the Kalman filter simply parallelize its matrix operations. Considering the cost of transferring data between GPU and CPU, parallelizing each matrix operation might not have performance gain for some applications, especially when the matrices are not large enough to occupy all GPU cores.

This paper proposes a new way to parallelize a Kalman Filter. Inspired by Parallel Splash Belief Propagation [2], we parallelize over time instead of parallelizing matrix operations at each time step. Kalman filtering is a message passing algorithm. In many applications, it is reasonable to assume that the current state estimate relies more on recent states, and that states from long ago don’t contribute much to the current state. Based on this assumption and the abundance of GPU threads, we compute a filtered state estimate for each timestep in a separate thread, based on the most recent observations. We implement this parallel Kalman Filter and optimize the GPU implementation, focusing on an application to computing the marginal likelihood of seismic waveforms. We compare the speed and accuracy of our parallel Kalman Filter with optimized CPU implementations.

The remaining sections are organized as follows. In section 2, we introduce the necessary background of this paper, including a brief description of the Kalman Filter, and analyze existing ways of parallelizing similar algorithms. In section 3, we describe the basics of the application model we are working on. In section 4 we discuss the details of our implementation. Section 5 explores the ways to optimize our implementation on a GPU. Section 6 performs various benchmarks and evaluate the results. Section 7 makes a conclusion and give directions for future work.

II. BACKGROUND

A. Kalman Filter

The Kalman filter algorithm involves two phases: predicting and updating, corresponding to the transition model and observation model respectively. A Kalman filter takes in a series of noisy observations to estimate hidden states. In the prediction step, it predicts the estimate of the current hidden state, based on the updated estimate of the previous state. Given the noisy observation at the current time step, this predicted estimate is updated using a weighted average of the existing estimate and the observation. The Kalman gain is the weight that measures how much the new observation needs to be taken into consider. We provide details in the predict phase and update phase. We use \( k \) to demonstrate each time step, and we will use these notations defined in [Algorithm1] in the following part of this paper.

B. Approaches of paralleliztion( Related Works)

Existing approaches of parallelizing Kalman filter focus on parallelizing matrix operations. We explored parallelism in graphical model inference, and was enlightened by Parallel splash belief propagation.

1) Parallelize matrix operations: The most popular way of parallelizing a Kalman filter on GPU is to parallelize its matrix multiplication and matrix inversion in predict phase and update phase at each time step [3], [4]. But when matrix dimension is not large enough to fill all the GPU pipelines, it’s hard to achieve optimal performance. Also, data transfer between CPU and GPU can be a bottleneck, for this way of parallelizing needs to transfer data at each time steps.

2) Break data dependency: An implementation of a Kalman filter for single output system on multicore computational platforms [5] breaks the data dependencies through re-organizing calculations, by which an almost completely parallel algorithm is obtained. However, this is only suitable for a specific kind of Kalman filter applications.

3) Parallel hidden Markov model( Reduce operations) : Hidden Markov model shares many characteristics of Kalman filter, thus its parallelism may give us some insights on parallelizing Kalman filter.

CuHMM [6] is a CUDA implementation of hidden Markov model (HMM) training and classification, which is also ends up parallelizing matrix operations and has great speedup.

Another paper [7] describes algorithms for a parallel implementation of hidden Markov models with a small state space. It shows how to format the HMM algorithm using linear algebra,
Algorithm 1 Serial Kalman Filter

notations for Kalman filter:
- F - state transition matrix
- H - observation matrix
- Q - covariance matrix of process noise
- R - covariance matrix of observation noise
- $Z_k$ - observation at time step $k$
- $k$ - for each time step

function KALMAN PREDICT ($\hat{X}_{k|k}$, $P_{k|k}$)
returns predicted state estimate
variables: $\hat{X}_{k|k-1}$, predicted state estimate
          $\hat{X}_{k|k}$, updated state estimate
          $P_{k|k-1}$, predicted state estimate covariance
          $P_{k|k}$, updated state estimate covariance
inputs: $\hat{X}_{k|k}$, $P_{k|k}$

\[
\begin{align*}
\hat{X}_{k|k-1} &= F\hat{X}_{k-1|k-1} \\
P_{k|k-1} &= FP_{k-1|k-1}F^T + Q
\end{align*}
\]

function KALMAN UPDATE ($\hat{X}_{k|k}$, $P_{k|k}$)
returns updated state estimate, predicted observation estimate
variables: $\hat{X}_{k|k}$, $P_{k|k}$
          $\hat{Y}_k$, predicted observation estimate
          $S_k$, predicted observation estimate covariance
          $K_k$, optimal Kalman gain
inputs: $\hat{X}_{k|k-1}$, $P_{k|k-1}$, $Z_k$

\[
\begin{align*}
\hat{Y}_k &= H\hat{X}_{k|k-1} \\
S_k &= HP_{k|k-1}H^T + R \\
K_k &= P_{k|k-1}H^T S_k^{-1} \\
\hat{X}_{k|k} &= \hat{X}_{k|k-1} + K_k(Z_k - \hat{Y}_k) \\
P_{k|k} &= (I - K_kH)P_{k|k-1}
\end{align*}
\]

function SERIAL KALMAN FILTER ($\hat{X}_{k|k}$, $P_{k|k}$)
returns marginal likelihood (sum)
variables: $\hat{X}_{k|k}$, $\hat{X}_{k|k-1}$, $P_{k|k}$, $P_{k|k-1}$, $\hat{Y}_k$, $S_k$, likelihood, accumulated marginal likelihood
inputs: $Z$, observations
prior, the prior distribution on the initial state
$n$, total time steps (length of signal)
likelihood $\leftarrow 0$
for $k = 0$ to $n-1$ do
if $k = 0$ then
    $(\hat{X}_{k+1|k}, P_{k+1|k}) \leftarrow$ prior
else
    $(\hat{X}_{k+1|k}, P_{k+1|k}) \leftarrow$
    $\leftarrow$ KALMANPREDICT ($\hat{X}_{k|k}$, $P_{k|k}$)
end if
$(\hat{X}_{k+1|k+1}, P_{k+1|k+1}, \hat{Y}_k, S_k)$
$\leftarrow$ KALMANUPDATE ($\hat{X}_{k+1|k}$, $P_{k+1|k}$)
likelihood $\leftarrow$ likelihood
+ MULTIVARIATE_NORMAL_LOGPDF ($Z_k$, $\hat{Y}_k$, $S_k$)
end for
return likelihood
end function

which naturally lends the algorithm itself to parallelization. It uses parallel reduction [8] on one algorithm in HMM. We want to do similar work on Kalman filter, and we have tried to combine linear algebra work from two time steps into one time step, which can also be part of future work.

C. Parallel splash belief propagation

Our parallel model is based on parallel splash belief propagation [2].

Considering the natural, synchronous parallelization of belief propagation is highly inefficient. Parallel splash belief propagation focuses on parallel graphical model inference. By bounding the achievable parallel performance in chain graphical models, they develop a theoretical understanding of the parallel limitations of belief propagation. Their assumptions is, for a long chain graphical model with weak edge potentials, distant vertices are approximately independent. For a particular vertex, an accurate approximation to the belief may often be achieved by running belief propagation on a small subgraph around that vertex, which can reduce the sequential component of belief propagation to the longest path in the subgraph.

This parallelism is abstract, without respect to a particular probability model. Kalman filter is an example of a “chain” graphical model. We develop our parallel model based on this parallel splash belief propagation idea.

III. PROGRAMMING MODEL

A. GPU and CUDA

GPU-accelerated computing is the use of a Graphics Processing Units (GPU) to perform computation traditionally handled by CPU. It puts compute-intensive part of the application to the GPU, while the remainder of the code still runs on CPU. The major difference between GPU and CPU is how they process tasks. A CPU is optimized for sequential serial processing while a GPU has a massive parallel architecture consisting of thousands of smaller, more efficient cores to handle multiple tasks simultaneously.

![CUDA processing flow](Figure 1: CUDA processing flow)

CUDA [9] is a parallel computing platform and programming model invented by NVIDIA. It enables programmer to
use GPU for general purpose processing. The CUDA platform can work with programming languages such as C,C++ and Fortran, which makes GPU programming easier. We implement this parallel Kalman Filter model using Pycuda, which maps all of CUDA into Python.

B. Parallel Belief Propagation in Kalman Filter

This paper introduces a novel way to parallelize Kalman filter. We parallelize over time instead of parallelizing matrix operations at each time step. In many applications, we can assume that the current state estimate relies more on recent states, and that states from long ago don’t contribute much to the current state. Based on this assumption and considering a GPU has thousands of threads, we compute a filtered state estimate for each timestep in a separate thread, as is described in [Algorithm2]. That is, the work to compute one filtered state estimate is done by one single thread, based on the most recent observations. The number of observations required is the number of iterations of getting a result within some deviation.

We can think that each thread is dealing with a small Kalman filter that compute the marginal likelihood for one hidden state, and we have many small Kalman filters running concurrently. Since we don’t know the status of the initial state of these small Kalman Filters, each thread start out with uniform distribution with a zero mean and a covariance matrix with infinite variance. However, standard Kalman filter doesn’t work with infinite variance beliefs, so we model this initial state into a Gaussian distribution with a zero mean and a covariance matrix with user specified large variance. Once we have this initial observation-based belief for each hidden state, passing message forward is just the standard Kalman Filter prediction step.

This parallelism does introduce some redundancy. Threads are doing overlapped computation, but we haven’t found an elegant way to optimize it yet.

C. Accuracy

One important parameter in this parallel Kalman filter is the number of iterations of getting marginal likelihood result within certain accuracy. Here we name this parameter “iteration”. There is a tradeoff between accuracy and speed. Most applications will converge to true value after some iterations. We can either derive how many iterations are needed to get true value from a specific model, or get it empirically after running experiments.

Theoretical speedup = totalTimeStep / Iterations

IV. OPTIMIZATION

In order to get optimal performance of GPU, the program should expose sufficient parallelism, use memory efficiently, minimize data transfer, and avoid different execution paths within the same warp. Coalescing global memory and minimizing redundant accesses to global memory are crucial.

Algorithm 2 Parallel Kalman Filter

**function** PARALLEL KALMAN FILTER (t, Z)
**returns** the marginal likelihood at time step[t]
**input:**
- t, the thread id for each cuda thread
- Z, observations
**persistent:** F, H, Q, R
- prior, the prior distribution on the initial state
- n , iteration of filtering in each thread
**local variables** (for each thread, and k is the time step):
- Y_k, S_k, \( \hat{X}_k, k \)\( -1 \), \( \hat{X}_k, k \)\( +1 \), \( P_k, k \)\( -1 \), \( P_k, k \)\( +1 \)
- logpdf, marginal likelihood for hidden state t

for each cuda thread(t) do:

for i = 0 to n-1 do
- if i = 0 then
  \( (\hat{X}_{t-n+1|i-1}, P_{t-n+1|i-1}) \leftarrow \text{prior} \)
- else
  \( (\hat{X}_{t-n+1|i-1}, P_{t-n+1|i-1}) \leftarrow \text{KALMANPREDICT} (\hat{X}_{t-n+i|i-1}, P_{t-n+i|i-1}) \)
- end if
  \( (\hat{X}_{t-n+i+1|i-1}, P_{t-n+i+1|i-1}, \hat{Y}_k, S_k) \leftarrow \text{KALMANUPDATE} (\hat{X}_{t-n+i+1|i-1}, P_{t-n+i+1|i-1}) \)
- end for
**return** MULTIVARIATE_NORMAL_LOGPDF (Z_t, \( \hat{Y}_k, S_k \))

A. Expose Sufficient Parallelism

GPU is a parallel machine. It has lots of arithmetic pipelines. In order to exploit the abundance of pipelines, code must expose sufficient parallelism to occupy all pipelines and hide latency of pipelines. In our seismic signal model, we hope to run 15000 threads in parallel, which exposes GPU with sufficient parallelism. On the contrary, if we parallelize Kalman filter by parallelizing matrix operations, the matrices are not large enough to fill GPU threads.

B. Minimize data transfer(between host and device)

Usually, the first step to parallelize something on GPU is to identify the hotspots on serial CPU code, and parallelize them. Communication between host(CPU) and device(GPU) is expensive. Take this fact into consideration, all the computation tasks in our model are done by GPU. In this case, CPU only need to communicate with GPU when all the work have done. Besides the constant transition and observation models, host only sends observations to device, and then host gets back the marginal likelihood of each timestep after device do filtering.

C. Optimize Launch Configuration

Key of CUDA execution model is the difference between thread, threads block, grid and warp. Thread is a sequential execution unit, while threads block is a group of threads, and a grid is a collection of those thread blocks. Threads within a block can cooperate with each other, and execute on a single Streaming Multiprocessor, while thread blocks do not synchronize with each other. A thread block has several
32-thread warps. When running a CUDA program, a kernel is launched as a grid of thread blocks.

Threads don’t run simultaneously on GPU. On a multiprocessor, only a warp (a group of 32 threads) is executed physically in parallel (SIMD). GPU architecture hides latency from other warps on this multiprocessor to achieve high throughput. That is, when some warps are waiting for data, GPU executes the warp which is ready.

When configure kernel launch, thread block size should be a multiple of warp size. If one requests threads less than 32 in a block, the size will round up. We choose block size based on the experiment result. 128 or 256 threads per block usually have a good performance in our model. Then grid size is total number of thread blocks.

D. Coalescing Global Memory Access

Global memory resides on the device. Its capacity is around several GB, and is accessible by all threads, though has high latency. Operations on device are issued per warp. Threads in a warp first provide memory addresses, then hardware determines which lines of memory are needed. Scattered address patterns or patterns with large strides between threads should be avoided so that a warp can access within a contiguous region. Performance gain can be achieved by coalescing global memory access [10] within a warp, the pattern is shown in the figure.

![Global Memory Access Pattern](image)

Before we try coalescing global memory, we store most arrays in thread’s local memory. For example, a thread creates an array in kernel, then this array is stored in thread local memory. Though local memory is actually part of global memory, the so-called “global memory” is allocated outside kernel, while local memory is allocated within a single thread. Allocating some arrays from thread’s local memory to global coalesced memory achieve good performance gain, while some other arrays don’t. In order to figure out why a better way of memory accessing does not necessary have better performance, we allocate global memory not in a coalesced pattern. And this pattern turns out to be twice as slow as thread’s local memory pattern. Our understanding is, though theoretically, latency of accessing local memory is nearly the same as global memory. Registers are partitioned among threads, and the storage is local to each thread. Some of the local memory might be stored in registers so it’s faster than global memory. When changing uncoalesced memory to coalesced memory, on average, it will be twice as fast as before. The different performance between different arrays in our model can be explained as the different ways of using them in kernel code.

E. Exploit GPU Fast Memory

1) GPU Memory Hierarchy: We explore a typical NVidia GPU memory hierarchy to get an idea of how fast each memory is. Register is the fastest memory on GPU, but it’s managed by compiler, and its storage is local to each thread. Shared memory and L1 cache also has low latency (eg.1-2cycles) and high bandwidth (eg.2.5TB/s) and it is program-managed. Read-only cache and constant caches are read only and useful. L2 cache is slower than memory mentioned above, and it’s hardware-managed. Global memory is large, accessible by all threads, CPU, but has higher latency (eg.400-800 cycles) and lower bandwidth (eg.250GB/s). GPU caches are not intended for the same use as CPU caches, for the smaller size, especially per thread, and we will explore this more below.

2) Shared Memory: Shared memory is the fastest configurable gpu memory, but it’s pretty small. In our case, it can only accommodate several single value variable. We tried to put those eligible variable into shared memory, in all case, it only became 1% faster, which is the same as setting cache config to “prefer L1”. Considering using shared memory will have more complex configuration, we simply set the flag to give more cached memory to L1 cache, and let compiler to make its own decision.

3) Constant Memory: NVIDIA hardware provides 64KB of constant memory, that it treats differently than it treats standard global memory, though it resides on global memory. The constant memory is cached. If threads in a half warp want to read the same address, constant cache is as fast as reading from a register, while accessing to different addresses by threads within a half warp are serialized. In order to maximize performance, we put read-only variables in constant memory if possible, and the performance became 20% faster.
V. EVALUATION MODEL

In this section, we introduce concrete models that we have used to evaluate the parallel Kalman filter.

A. Seismic Signal Model

We model a seismic signal at a single station as a background noise process, plus arriving seismic phases. In this paper, we implement parallel Kalman filter based on this model. This probability model constructs the marginal distribution on the noisy signal as an explicit Gaussian, and compute the required likelihoods and posterior distributions using a Kalman filter. The background noise is modeled into an autoregressive process. We will briefly describe the AR process in next subsection. For more information about this model, please see this pending paper.

We have implemented the GPU-based parallel Kalman filter based on this seismic model described using Pycuda. We reduced the float operations in the Seismic signal model by customizing matrix structure and simplifying observation model.

B. Autoregressive Process

An autoregressive process [11] of order p is a stochastic process with no hidden state, in which the expected value at time t is a linear function of the values at times t-p, ..., t-1, with Gaussian noise at each step.

\[ z_t = \sum_{k=1}^{p} \phi_k z_{t-k} + \varepsilon_t, \]
\[ \varepsilon_t \sim N(0, \sigma^2). \]

C. Random Walk

No need to mention exact matrices we use (since we didn’t in above part) Random walk [12].

VI. EVALUATION

A. Setup

For parallel evaluations, we ran benchmarks on a server with a NVIDIA Quadro K4200 GPU, which has 1344 GPU cores, 4GB GPU memory with 173GB/s bandwidth and 64KB of RAM (configurable partitioning of shared memory and L1 cache), and with an Intel Xeon Processor E3-1271 v3.

For all the serial evaluations, we ran benchmarks on a computer with 2.6GHz Intel Core i5 Processor and 8GB 1600MHz memory.

B. Performance - Speed

1) Toy example - Random Walk: We first compare the performance of serial random walk (RW) and parallel RW on both 2 dimensional and 10 dimensional RW. The number of iterations is 10 here, at which the marginal likelihood converges to true value.

As shown in [figure5], the serial python implementation has approximately linear complexity, while cuda implementation is much faster and scalable. Processing a 4096 steps two dimensional RW in parallel only takes 0.05X longer than processing a 32 steps signal in parallel. For ten dimensional RW, running in 4096 threads takes 2.8X as long as running in 32 threads.

The speedup in parallel RW is great. But considering python is an interpreter language, and it spends some time on setting up stuff rather than doing float operations, sometimes the actual speedup can even exceed theoretical speedup. Despite of this, we can see great advantages when running the parallel Kalman filter.

2) Autoregressive Process: AR process is an essential part of the seismic model. We set the AR process to 10 order. In order to achieve accuracy, here we choose the number of iterations as 11 for parallel version.
We tested our GPU-accelerated Kalman Filter against a fully optimized CPU-based Kalman Filter. One trick the C++ CPU code does is that it stops computing covariance after the covariance matrix converges. If there is a missing observation, it will re-compute the covariance matrix until it converges again, losing performance. We compared the performance of parallel version and serial version in both the fully observed setting and with missing observations every 20 time steps.

As we can see in the graph, the parallel version outperforms the serial version greatly. It can achieve up to 7X speedup when some observations missed. As we expect, the parallel version achieves higher speedup when observations are not fully observed. There is a tradeoff between speed and accuracy: here we set number of iterations to 11; one can get a higher speedup by iterating less, which may have less accurate result. We will explore more about accuracy in the following part.

C. Correctness

Tradeoff between accuracy and speed is crucial in our parallel Kalman filter, while it makes this parallel Kalman filter more flexible. From the graph, we can see in the 168 dimensional seismic signal, 10-order autoregressive process, and 2 dimensional random walk, when number of iterations increases, the marginal likelihood absolute deviation decreases. The marginal likelihood converges after a few time steps. From all the experiments we have ran, we can deduce that this GPU-based parallel Kalman filter has higher performance gain when dealing with applications that can converge in less time steps.

We have already seen desirable performance gain of our parallel Kalman filter, and there are several interesting directions for future research.

1. Parallelize matrix operations when the matrix is large, and combine it with our existing parallel model. To be more specific, in a long signal, parallelize the matrix operations in time steps that have large matrices, which corresponds to arriving earthquake signal in the seismic signal, and parallelize over time when matrices are small, which corresponds to the phase that only has background noise in our application.

2. Generalize the approach to fit more applications.

3. Our parallel model does some redundant computation in each thread, which takes lots of time. We should find a way to cut the redundancy in the future.

VII. CONCLUSION AND FUTURE WORK

We have presented a new approach to parallelize Kalman filter in this paper, leveraging general purpose GPU. We parallelize it over time instead of parallelizing matrix operations. While this parallel approach is limited to applications have some observations beforehand, we have found it is powerful enough to express some Kalman filter applications.

We have evaluated our model by implementing it in Pycuda and tested it against optimized CPU code, and shown that our model has better overall performance in small dimension and long signal length applications. Also, tradeoff between correctness and performance is interesting in our algorithm.

REFERENCES