Accelerating the Kalman Filter on a GPU

Min-Yu Huang
Dept. of Electrical Engineering
National Taipei Univ. of Tech.
Taipei, Taiwan
s0302151@gmail.com

Shih-Chieh Wei
Dept. of Info. Management
Tamkang University
Tamsui, Taiwan 25137
seke@mail.im.tku.edu.tw

Bormin Huang
Space Science & Engineering Ctr.
Univ. of Wisconsin-Madison
Madison, WI 53706, USA
bormin@ssec.wisc.edu

Yang-Lang Chang
Dept. of Electrical Engineering
National Taipei Univ. of Tech.
Taipei, Taiwan
ylchang@mail.ntut.edu.tw

***Abstract*—For linear dynamic systems with hidden states, the Kalman filter can estimate the system state and its error covariance considering the uncertainties in transition and observation models. In each iteration of applying the Kalman filter, the two phases of predict and update contain a total of 18 matrix operations which include addition, subtraction, multiplication and inversion. As recent graphic processor units (GPU) have shown to provide high speedup in matrix operations, we implemented a GPU accelerated Kalman filter in this work. For general reference purposes, we tested the filter on typical large-scale over-determined systems with thousands of components in states and measurements. For the various combinations of configurations in our test, the GPU accelerated filter shows a scalable speedup as either the state or the measurement dimension increases. The obtained 2 to 3 orders of magnitude speedup over its single-threaded CPU counterpart shows a promising direction of using the GPU-based Kalman filter in large-scale time-critical applications.**

Keywords** - Kalman filter, GPU, CUDA, parallel computing

I. Introduction

In 1960, the Kalman filter was introduced as a technique for state estimation of a linear dynamic system [1]. In the presence of forecast and measurement errors, the Kalman filter is capable of refining the system state with each new observation so that the new estimate will be close to the true system state. The Kalman filter has found many applications in tracking of various systems [2] for military, weather forecast [3], financial [4] and robotic navigation purposes. To implement the Kalman filter, the two phases of predict and update involve many matrix operations performed in sequence. These operations include addition, subtraction, multiplication, transposition and inversion on matrices of various sizes.

In recent years graphics processing units (GPUs) with hundreds of computing cores have become affordable for scientific computation. Currently, an NVIDIA high-end Tesla C2050 GPU card has 448 computing cores. It delivers a theoretical peak performance of over 1.03 TFLOPs in single precision. The combined features of general-purpose supercomputing, high parallelism, high memory bandwidth (144 GB/s), low cost, and compact size make a GPU-based desktop computer an appealing alternative to a massively parallel system made up of commodity CPUs. Increasing programmability of commodity GPUs allows their usage as General Purpose computation on GPUs (GPGPUs), which have recently attracted great attention for scientific applications [5,6,7].

This work explores the massively parallel capabilities of GPU in accelerating the Kalman filter computation. The NVIDIA CUDA (Compute Unified Device Architecture) platform is used for massively multi-threaded parallel computation [8,9]. Our GPU implementation of the Kalman filter is tested on an NVIDIA Tesla C2050 GPU (448 cores) delivering TFlops peak performance. Compared with a native CPU implementation, the speedup profile of computing the Kalman filter will be shown for various combinations of measurement and state sizes. The result will serve as good reference for future implementation on real large-scale applications of the Kalman filter.

The rest of this paper will be arranged as follows. Section II describes the GPU implementation of the Kalman filter. Section III shows the experimental results on GPU. Section IV summarizes the paper.

II. The Kalman Filter and Its GPU Implementation

A. THE KALMAN FILTER

The formula for implementing the discrete-time Kalman filter [1,3,4] is briefly introduced below. The Kalman filter is to estimate the true state of a process given only a sequence of noisy observations in a linear dynamic system perturbed by Gaussian noise. Assuming the dimension of the true state is n_s and the dimension of an observation is n_o, the Kalman filter is a recursive estimator which can be described by a process to evolve the state x_k at time k from its previous state x_{k-1} at time $(k-1)$:

$$x_k = F_k\, x_{k-1} + B_k\, u_k + w_k \tag{1}$$

and a mapping from the state x_k into the observation z_k:

*Send correspondence to: bormin@ssec.wisc.edu.

1521-9097/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPADS.2011.153

$$z_k = H_k x_k + v_k \quad (2)$$

where F_k is the $n_s \times n_s$ state transition model at time k, B_k the $n_s \times n_s$ control-input model, u_k the $n_s \times 1$ control vector, w_k the $n_s \times 1$ process noise, H_k the $n_o \times n_s$ observation model, and v_k the $n_o \times 1$ observation noise. Both the process noise w_k and measurement noise v_k are assumed to be Gaussian white noises with zero mean:

$$w_k \sim N(0,Q_k) \quad \text{and} \quad v_k \sim N(0,E_k), \quad (3)$$

where Q_k is the $n_s \times n_s$ covariance of process noise at time k and E_k the $n_o \times n_o$ covariance of observation noise. In addition, it is also assumed that the covariance of state estimate error at time k-1 is P_{k-1}.

Given F_k, B_k, Q_k, H_k, E_k,, the Kalman filter can give new x_k and P_k based on new z_k and old x_{k-1} and P_{k-1} in the following two phases of prediction and update [3].

1. The Prediction Phase:

$$\tilde{x}_k = F_k x_{k-1} + B_k u_k \quad (4)$$

$$\tilde{P}_k = F_k P_{k-1} F_k{}^T + Q_k \quad (5)$$

2. The Update Phase:

$$K = \tilde{P}_k H_k^T [H_k \tilde{P}_k H_k^T + E_k]^{-1} \quad (6)$$

$$x_k = \tilde{x}_k + K [z_k - H_k \tilde{x}_k] \quad (7)$$

$$P_k = [I - K H_k] \tilde{P}_k \quad (8)$$

where the $n_s \times 1$ state estimate $\tilde{x}_k$, the $n_s \times n_s$ state covariance estimate $\tilde{P}_k$, and the $n_s \times n_o$ Kalman filter gain K are the three interim variables; P_k the $n_s \times n_s$ state covariance at time k, and I the $n_s \times n_s$ identity matrix.

B. IMPLEMENTATION OF THE KALMAN FILTER

Without loss of generality, we assume that the four given model parameters F_k, Q_k, H_k, and E_k\ do not vary with time and denote them by F, Q, H, and E respectively later. Also for simplicity we consider no control-input model in eq. (1).

Table I shows the 18 matrix operations in implementing eqs. (4)-(8). Among them, A1 denote the single operation for eq. (4), B1-B4 the four operations for eq. (5), C1-C6 the six operations for eq. (6), D1-D4 the four operations for eq. (7), and E1-E3 the three operations for eq. (5).

In total there are 1+4+6+4+3=18 matrix operations needed for each iteration of the Kalman filter estimation procedure with predict and update phases. These operations include add for matrix addition, subtract for matrix subtraction, multiply for matrix multiplication, transpose for matrix transposition, and inv_spd for inversion of a semi-positive definite (SPD) matrix. For each operation, the related quantity and the matrix dimensions involved are shown in the second and third columns of Tables I. For matrix multiplication, three dimensions $m \times k \times n$ are shown which means an $m \times k$ matrix are multiplied by a $k \times n$ matrix to get an $m \times n$ matrix. For matrix inversion, the Cholesky factoring can be used for semi-positive definite matrices. It is faster than the common LU factoring or Gaussian elimination used for general square matrices. The covariance matrix is symmetric and semi-positive definite. Therefore inv_spd is used for matrix inversion.

TABLE I. 18 BASIC MATRIX OPERATIONS IN EQS.(4)-(8) FOR THE PREDICT AND UPDATE PHASES OF THE KALMAN FILTER.

Operation	Quantity	Dimensions
Predict Phase:		
A1. multiply	$\tilde{x}_k = F x_{k-1}$	$n_s \times n_s \times 1$
B1. transpose	F^T	$n_s \times n_s$
B2. multiply	$F P_{k-1}$	$n_s \times n_s \times n_s$
B3. multiply	$F P_{k-1} F^T$	$n_s \times n_s \times n_s$
B4. add	$\tilde{P}_k = F P_{k-1} F^T + Q$	$n_s \times n_s$
Update Phase:		
C1. transpose	H^T	$n_o \times n_s$
C2. multiply	$\tilde{P}_k H^T$	$n_s \times n_s$
C3. multiply	$H \tilde{P}_k H^T$	$n_o \times n_s \times n_o$
C4. add	$H \tilde{P}_k H^T + E$	$n_o \times n_o$
C5. inv_spd	$[H \tilde{P}_k H^T + E]^{-1}$	$n_o \times n_o$
C6. multiply	$K = \tilde{P}_k H^T [H \tilde{P}_k H^T + E]^{-1}$	$n_s \times n_o \times n_o$
D1. multiply	$H \tilde{x}_k$	$n_o \times n_s \times 1$
D2. subtract	$z - H \tilde{x}_k$	$n_o \times 1$
D3. multiply	$K [z - H \tilde{x}_k]$	$n_s \times n_o \times 1$
D4. add	$x_k = \tilde{x}_k + K [z - H \tilde{x}_k]$	$n_s \times 1$
E1. multiply	$K H$	$n_s \times n_o \times n_s$
E2. subtract	$I - K H$	$n_s \times n_s$
E3. multiply	$P_k = [I - K H] \tilde{P}_k$	$n_s \times n_s \times n_s$

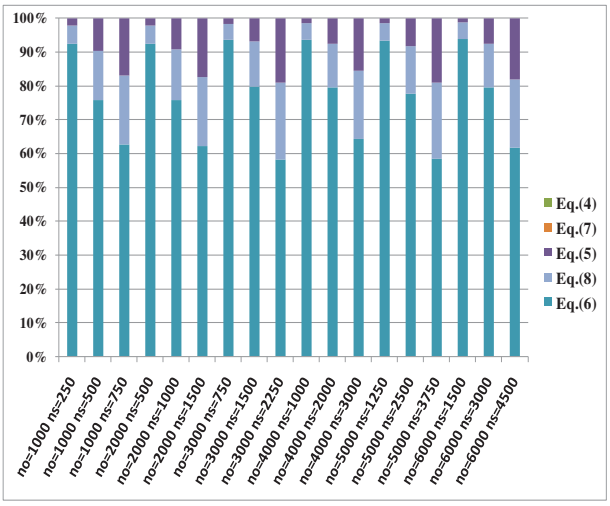

Figure 1. The relative CPU time of eqs.(4)-(8) for the Kalman filter.

For comparison with later GPU implementation, profiling of the Kalman filter on a single threaded CPU core with O3 compiler optimization is shown in Fig. 1. The observation dimension ranges from 1000 to 6000 in 1000 steps with the state dimension fixed at 1/4, 2/4, and 3/4 the observation dimension. It can be seen that eq. (6) for computing the Kalman filter gain is most time-consuming which accounts for an average of 77% of the total computation time. The next time dominant operation is eq. (8) for 13%, followed by eq. (5) for 9%. Eq. (6) mainly deals with matrix inversion while eqs. (8) and (5) mainly deal with matrix multiplication. Equations (4) and (8) take up negligible time. There is a trend which shows the time for eq. (6) will decrease as the state dimension increases.

Based on the above observation, it can be seen that matrix inversion as well as matrix multiplication will be the most and second critical operations for GPU computation. As the dimension increases, matrix inversion will eventually dominate matrix multiplication in running time. Since the GEMM (GEneral Matrix Multiplication) function in CUBLAS [10] follows the LAPACK convention which integrates matrix multiplication, addition, and transposition in one function call, we consult available open source packages which include CUDA SDK, Magma [11] and a few other sources on the network to write separate kernels for matrix multiplication, addition and transposition.

C. THE INVERSE OF A COVARIANCE MATRIX

For matrix inversion of a covariance matrix, the Cholesky factoring can be used for fast computation. Cholesky factoring can decompose a covariance matrix A as the product of a lower triangular matrix L and its transpose L^T, i.e. $A = L\,L^T$ [12]. With Cholesky factoring, the inverse X of a covariance matrix A can be solved as follows

$$I = A\,X = (L\,L^T)\,X = L\,L^T\,X = L\,(L^T\,X) = L\,Y \qquad (9)$$

where I is an identity matrix and L is a lower triangular matrix. For implementation, the three steps in sequence are as follows:

Step 1. Do Cholesky factorization: $A = L\,L^T$.

Step 2. Solve a lower triangular system for Y in $L\,Y = I$.

Step 3. Solve an upper triangular system for X in $L^T\,X = Y$.

For Cholesky factorization on a GPU, we settle on an implementation by Bouckaert [13] of a blocked factoring algorithm [14] which involves an iterative sequence of CPU inversion on diagonal blocks and GPU updates on left hand and lower blocks. For solution to a triangular system of equations, we use the TRSM (Triangle Solve Multiple) function of CUBLAS [10] which runs on GPU. We will use TRSM for solution of a lower triangular system in step 2 and for solution of an upper triangular system in step 3.

III. RESULT

To evaluate the GPU accelerated performance of the Kalman filter, we run a subset of sample points from the two dimensions of observation and model state. In particular, the dimension (*no*) of the observation y ranges from 1000 through 7000 in 1000 steps. The dimension of the model state x or *ns* ranges from 1/4, 2/4, 3/4 the dimension of the observation z. In this way we are able to show the expected results for any combination of the two dimensions in large-scale applications with n_o no less than n_s within the GPU memory limit. Without loss of generality, we use pseudo random numbers to generate the input F, Q, H, V, z, x_{k-1} and P_{k-1} for applying the Kalman filter to estimation of the new x_k and P_k. For all experiments, only results for single precision floating numbers are reported.

Our implementation and testing are based on an environment with a quad-core 2.4 GHz Intel Xeon E5620 CPU and an NVIDIA Tesla C2050 1.15 GHz GPU. The specifications of the NVIDIA Tesla C2050 GPU are shown in Table II. NVIDIA Tesla C2050 consists of 14 multiprocessors. Each multiprocessor has 32 thread processors. Each thread processor inside a multiprocessor runs synchronously. Thus, all 32 thread processors execute the same instruction at the same time. Threads are organized into three level hierarchies. The highest level is a grid, which consists of thread blocks. A thread block is a three

dimensional array of threads. A current generation of NVIDIA's GPUs group threads in groups of 32 threads called warps. A multiprocessor issues the same instruction to the all threads in a warp. When threads take divergent paths multiple passes are required to complete the warp execution. Separate multiprocessors run asynchronously. Each Tesla C2050 GPU has 3 GB device memory, whose access bandwidth is higher than CPU's access to DRAM memory.

TABLE II. SPECIFICATIONS OF THE NVIDIA TESLA C2050 GPU CARD.

Number of Streaming Processor Cores	448
Frequency of Processor Cores	1.15GHz
Total Dedicated Memory	3 GB GDDR5
Memory Speed	1.5 GHz
Memory Interface	384-bit
Memory Bandwidth	144 GB/sec

Fig. 2 shows the GPU profiling of the Kalman filter. The observation dimension ranges from 1000 to 7000 in 1000 steps with the state dimension fixed at 1/4, 2/4, and 3/4 the observation dimension. For the observation dimension n_o = 7000, there are no data available for the state dimensions ns = ¾ n_o = 5250 due to exceeding the GPU memory limit. Compared to the relative CPU time in Fig. 1, eq. (6) takes up more time while the other equations take up fewer proportions. The trend remains that the time for eq. (6) will decrease as the state dimension increases. One can expect that most of the GPU speedup in the Kalman filter computation comes from the speedup in matrix multiplication and inverse.

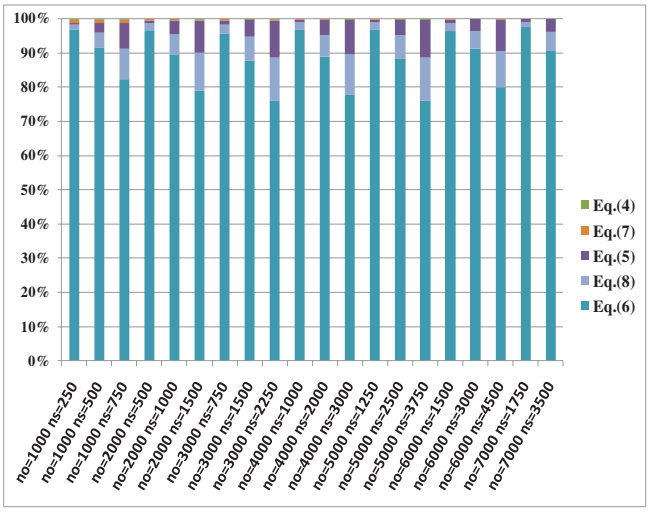

Figure 2. The relative GPU time of eqs.(4)-(8) for the Kalman filter.

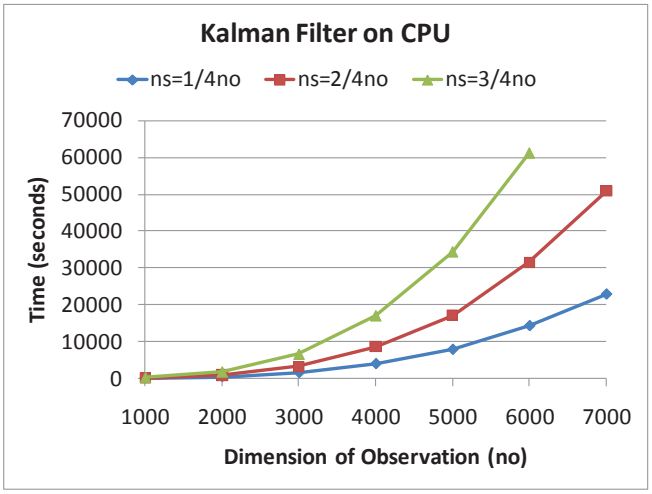

Figure 3. The CPU running time for the Kalman filter computation. The observation dimension (n_o) ranges from 1000 to 7000 in 1000 steps with the state dimension (n_s) fixed at 1/4, 2/4, 3/4 of the observation dimension.

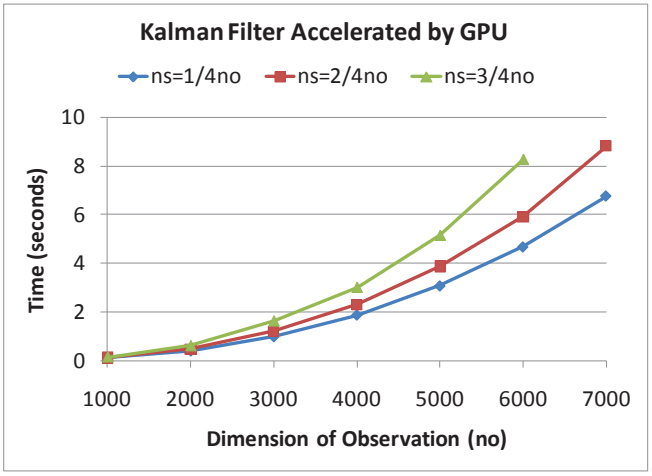

Figure 4. The GPU accelerated running time for the Kalman filter computation. The observation dimension (n_o) ranges from 1000 to 7000 in 1000 steps with the state dimension (n_s) fixed at 1/4, 2/4, 3/4 of the observation dimension.

Fig. 3 and Fig. 4 show how the computation time increases as the observation and state dimensions of the Kalman filter problem grow when running on environments with pure CPU and with aid by GPU respectively. It can be seen that on CPU the Kalman filter time is approximately linear with the state dimension. But when aided by GPU, the Kalman filter time is sublinear with the state dimension. Also on CPU the Kalman filter time has a super cubic relation with the observation dimension. But when aided by

GPU, the Kalman filter time has a relation between squared and cubic with the observation dimension.

Fig. 3 shows the GPU speedup of the Kalman filter computation with respect to a single-threaded CPU core. For the various combinations of large dimensions in test, a quite scalable speedup is seen with the observation dimension (n_o) from 1000 to 7000 in 1000 steps and the state dimension (n_s) fixed at 1/4, 2/4, 3/4 the observation dimension. In general there is a trend that the GPU speedup increases as the observation and state dimensions increase. Within the GPU memory limit, the maximum speedup of 7398x is observed with the observation dimension n_o = 6000 and the state dimension n_s = 3/4 n_o = 4500.

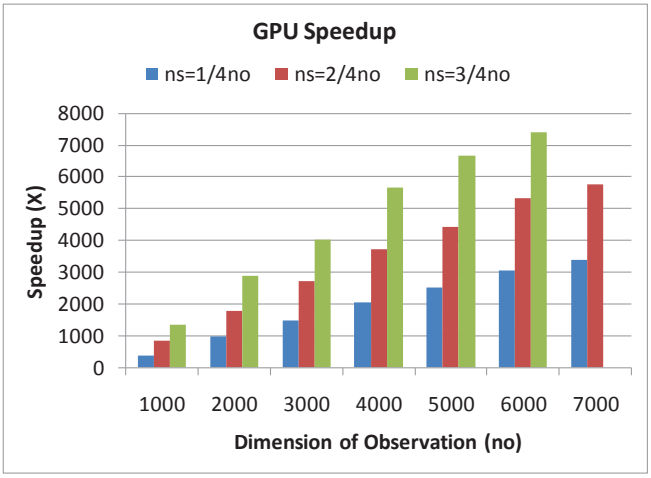

Figure 5. The GPU speedup for the Kalman filter computation. The observation dimension (n_o) ranges from 1000 to 7000 in 1000 steps with the state dimension (n_s) fixed at 1/4, 2/4, 3/4 of the observation dimension.

IV. SUMMARY

The Kalman filter is widely used in state estimation of linear systems in the presence of forecast and observation errors. The Kalman filter can compute a new estimate of the system state and its state error covariance based on a new observation, previous state and error covariance, and other given system model parameters. The Kalman filter consists of the predict phase and the update phase. There are a total of 18 matrix operations involved which include addition, subtraction, transposition, multiplication and inversion.

For single-threaded CPU, the most time consuming part of the Kalman filter is matrix inversion, followed by matrix multiplication. By GPU implementation, the matrix multiplication time of the Kalman filter is much reduced compared to the matrix inversion time. It is found that the Kalman filter time on CPU is approximately linear with the state dimension while the Kalman filter time on GPU is sublinear with the state dimension. Also the Kalman filter time on CPU has a super cubic relation with the observation dimension while the Kalman filter time on GPU has a relation between squared and cubic with the observation dimension.

For the various combinations of large dimensions conducted within the GPU memory limit, a quite scalable speedup is seen as either the observation or the state dimension increases. A maximum GPU speedup of 7398x is observed where an observation dimension of 6000 and a state dimension of 4500 are tested. The GPU speedup in this work has shown a promising direction for application of the Kalman filter in large-scale or time-critical problems.

REFERENCES

[1] R. E. Kalman, "A new approach to linear filtering and prediction problems," Journal of Basic Engineering 82 (1), 35–45, 1960.

[2] B. D. O. Anderson and J. B. Moore, Optimal Filtering, Prentice-Hall, 1979.

[3] http://en.wikipedia.org/wiki/Kalman_filter.

[4] S. Mergner, Applications of State Space Models in Finance, Unversitaetsverlag Goettingen, 2009.

[5] B. Huang, J. Mielikainen, H. Oh, and H.-L. Huang, "Development of a GPU-based high-performance radiative transfer model for the infrared atmospheric sounding interferometer (IASI)," Journal of Computational Physics, 230 (6), 2011.

[6] D. B. Kirk and W.-m. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann Publishers, 2010.

[7] J. Sanders and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley, 2011.

[8] NVIDIA, NVIDIA Cuda Programming Guide, Version 3.0, 2010.

[9] NVIDIA, NVIDIA Cuda Reference Manual, Version 3.0, 2010.

[10] NVIDIA, Cuda CUBLAS Library, Version 3.0, 2010.

[11] Magma Software, available at http://icl.cs.utk.edu/magma /software/.

[12] G. H. Golub and C.F. Van Loan, Matrix Computations, John Hopkins University Press, 1996.

[13] R. Bouckaert, Matrix inverse with Cuda and CUBLAS, available at http://www.cs.waikato.ac.nz/~remco/.

[14] H. Ltaief, S. Tomov, R. Nath, and J. Dongarra, Hybrid multicore Cholesky factorization with multiple GPU accelerators, available at http://www.netlib.org/netlib/utk/people/JackDongarra/PAPERS/hybrid-multicore-cholesky.pdf