FALL 2012 - ECEN/CSCI - 5593

ADVANCED COMPUTER ARCHITECTURE

Prof. Dan Connors

Final Project Report

Topic: Implementation of Kalman Filter on GPU using CUDA

Team Members:

Anitha Ganesha

Nivedita Swaminathan

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Motivation:

In 1960, R.E. Kalman published his famous paper describing a recursive solution to the discrete-data linear filtering problem. Since that time, due in large part to advances in digital computing; the Kalman filter has been the subject of extensive research and application, particularly in the area of autonomous or assisted navigation[1]. The Kalman filter has numerous applications in technology. They are used in tracking objects (e.g., missiles, faces, heads, hands), in fitting Bezier patches to (noisy, moving) point data. A common application is for guidance, navigation and control of vehicles, particularly aircraft and spacecraft[2]. High-precision vehicle navigation technique uses visual and inertial- based measurements that are fed into a unique Kalman filter based algorithm for pose estimation (position and orientation).

Furthermore, the Kalman filter is a widely applied concept in time series analysis used in fields such as signal processing and econometrics. Many computer vision applications like stabilizing depth measurements, Feature tracking, cluster tracking, laser scanner, and stereo-cameras for depth and velocity measurements also use kalman filter. The properties of kalman filter along with the simplicity of the derived equations make it valuable in the analysis of signals. The Kalman Filter and the Kalman Smoother have been extensively used in biomedical signal processing. Kalman filter is also used in seismology in analyzing in-situ seismic data.

All the above mentioned kalman filter applications involve matrix operations. With the increase in the size of the matrix increases the complexity of the overall computations. The present day GPUs have a lot of computational power within them, as they have more number of processing cores which can process the data in parallel when compared to normal CPUs with sequentially processing capability. This ability of the GPUs can be used for dealing with the complexities of the matrix operations involved in the kalman filter.

1.2 **Problem Statement**

The kalman Filter operates in two phases namely predict and update phase which runs over several iterations before making the final prediction. In every iteration the two phases update and predict has overall 18 matrix operations including Matrix addition, Subtraction, Inverse and Transpose. Our goal is to explore the options of parallelizing various matrix operations. The GPU implementation of the kalman filter will be done using CUDA programming language. Amongst the five operations listed, Matrix multiplication and Inverse are the ones which are highly computationally intensive. Therefore, we have used shared memory to fasten the Matrix multiplication and have used Cholesky algorithm which has proven to be best for the Matrix inverse case. We will implement the kalman filter on CPU and compare the results obtained with GPU. Here, mainly we focus on comparing the execution time on both CPU and GPU and analyze the speedup obtained in case of GPU for varying input size.

2 **EXTENDED MOTIVATION**

2.1 GPGPU

General-purpose computing on graphics processing units (General-purpose graphics processing unit, **GPGPU**) is the utilization of a graphics processing unit(GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU) [3]. Any GPU providing a functionally complete set of operations performed on arbitrary bits can compute any computable value.

GPU computing offers unprecedented application performance by offloading compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU [4]. From a user's perspective, applications simply run significantly faster [4].

2.2 CUDA Parallel Architecture

CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU) [4].

The CUDA architecture consists of 100s of processor cores that operate together to crunch through the data set in the application [5].

• Each SIMD core is represented by a group of threads [5].

• The whole GPU with its many SIMD cores is represented by a number of thread groups [5].

In CUDA a thread group is called a block. All threads of a thread group will be executed on the same SIMD core. Therefore a thread group should at least contain a thread for each execution unit in the SIMD core. Since interleaving is used to hide latency we also need to provide enough threads for effective interleaving. Thread groups with fewer threads will waste calculation performance since execution units will be idle. In practice even more threads are needed in a thread group to effectively hide latency. In the CUDA programming model the programmer writes code for a single thread. Whenever this code is executed on a GPU, the programmer needs to set up the number of threads and thread groups. Therefore algorithms targeting the GPU need to be build with this two level parallelism in mind [5].

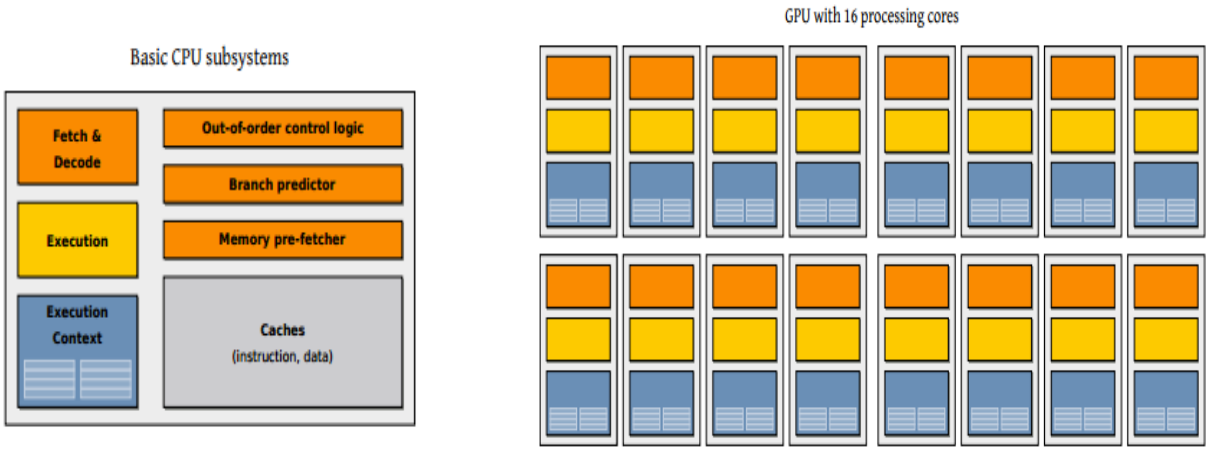

Fig1: CPU and GPU architecture

The advantages of using CUDA over general-purpose computation on GPGPUs using graphics APIs:

- Using CUDA enables the code to read from arbitrary addresses in memory.
- CUDA exposes a fast **shared memory** region (up to 48KB per Multi-Processor) that can be shared amongst threads which can be used as a user-

managed cache, enabling higher bandwidth than is possible using texture lookups [6].

- It offers faster copying and read backs to and from the GPU.
- Integer and bitwise operations are fully supported, including the integer texture lookups [6].

3 RELATED WORKS

The introduction to kalman filter discusses about the implementation details of the kalman filter, where in they give a complete explanation about the various stages involved in the kalman filter [1].

Min-Yu Hang and others have published a paper on kalman filters implementation on GPU [8]. Here they study the performance of GPU and its speed up under different input conditions. They have identified the operations within the kalman filter which slows down the overall execution time and have suggested mechanisms to improve the speed when implementing on GPU. In their evaluation, they try to push the GPU to its maximum limits and try to compare the speed up achieved with respect to CPU computation.

4 EXPLANATION

4.1 Kalman Filter

The Kalman filter is essentially a set of mathematical equations that implement a predictor-corrector type estimator that is optimal in the sense that it minimizes the estimated error covariance—when some presumed conditions are met [1].

The Kalman filter addresses the general problem of trying to estimate the state ‘x’ of a discrete-time controlled process that is governed by the linear stochastic difference equation[1]:

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

with a measurement ‘Z’ that is

$$z_k = H_k\, x_k + v_k \qquad (2)$$

The random variable and represent the process and measurement noise (respectively). They are assumed to be independent (of each other), white, and with normal probability distributions[1].

$$w_k \sim N(0,Qk) \text{ and } v_k \sim N(0,Ek) \qquad (3)$$

In practice, the process noise covariance and measurement noise covariance matrices might change with each time step or measurement, however here we assume they are constant[1].

Assuming the dimension of the true state is *ns* and the dimension of an observation is *no*, the *ns*ns* matrix 'F' in the difference equation (1) relates the state at the previous time step *k-1* to the state at the current step *k*, in the absence of either a driving function or process noise. Note that in practice 'F' might change with each time step, but here we assume it is constant. The *ns*ns* matrix 'B' relates the optional control input 'u' to the state 'x'. The *no*ns* matrix 'H' in the measurement equation (2) relates the state to the measurement z_k. In practice 'H' might change with each time step or measurement, but here we assume it is constant[1].

The Kalman filter estimates a process by using a form of feedback control: the filter estimates the process state at some time and then obtains feedback in the form of (noisy) measurements. As such, the equations for the Kalman filter fall into two groups***: time update equations and measurement update equations.*** The time update equations are responsible for projecting forward (in time) the current state and error covariance estimates to obtain the a priori estimates for the next time step. The measurement update equations are responsible for the feedback i.e. for incorporating a new measurement into the *priori* estimate to obtain an improved *posteriori* estimate[1].

The time update equations can also be thought of as ***predictor phase*** equations, while the measurement update equations can be thought of as ***update phase*** equations. Indeed the final estimation algorithm resembles that of a predictor-corrector algorithm for solving numerical problems[1].

1. Predict Phase

$$x_k^- = F_k\, x_{k-1} + B_k\, u_k \quad (4)$$

$$P_k^- = F_k\, P_{k-1}\, F_k^T + Q_K \quad (5)$$

2. Update Phase

$$K_k = P_k^- H^T [\, H\, P_k^- H^T + E_k]^{-1} \quad (6)$$

$$x_k = x_k^- + K_k\, [z_k - H x_k^-] \quad (7)$$

$$P_k = (I - K_k H) P_k^- \quad (8)$$

The time update equations (4) & (5) project the state and covariance estimates forward from time step *k-1* to step *k*. 'F' and 'B' are from equation (1), while 'Q' is from equation (3).

The first task during the measurement update is to compute the Kalman gain, K_k. The next step is to actually measure the process to obtain z_k, and then to generate an a *posteriori* state estimate by incorporating the measurement as in equation (7). The final step is to obtain an a *posteriori* error covariance estimate via equation (8) [1].

After each time and measurement update pair, the process is repeated with the previous *posteriori* estimates used to project or predict the new *priori* estimates. This recursive nature is one of the very appealing features of the Kalman filter—it makes practical implementations much more feasible [1].

4.2 Implementation

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).

In total there are 18 matrix operations needed for each iteration of the Kalman filter estimation procedure with predict and update phases. These operations include matrix addition, matrix subtraction, matrix multiplication, matrix transposition, and inversion of a semi-positive definite (SPD) matrix.

The following table gives the 18 matrix operations involved in the kalman filters' predict and update phases.

Matrix Operation	Equation	Size
Multiply	$x_k^- = F\, x_{k-1}$	(ns*ns)x(ns*1)
Transpose	F^T	(ns*ns)
Multiply	$P_{k-1}\, F^T$	(ns*ns)x(ns*ns)
Multiply	$F\, P_{k-1}\, F^T$	(ns*ns)x(ns*ns)
Add	$P_k^- = F\, P_{k-1}\, F^T + Q$	(ns*ns)+(ns*ns)
Transpose	H^T	(no*ns)
Multiply	$P_k^- H^T$	(ns*ns)x(ns*no)
Multiply	$H\, P_k^- H^T$	(no*ns)x(ns*no)
Add	$H\, P_k^- H^T + E$	(no*no)+ (no*no)
Inverse	$[\, H\, P_k^- H^T + E]^{-1}$	(no*no)
Multiply	$P_k^- H^T\, [\, H\, P_k^- H^T + E]^{-1}$	(ns*no)x(no*no)
Multiply	Hx_k	(no*ns)x(ns*1)
Subtract	$z_k - Hx_k^-$	(no*1)- (no*1)
Multiply	$K_k\, [z_k - Hx_k^-]$	(ns*no)x(no*1)
Add	$x_k^- + K_k\, [z_k - Hx_k^-]$	(ns*1)+ (ns*1)
Multiply	$K_k H$	(ns*no)x(no*ns)
Subtract	$I - K_k H$	(ns*ns)-(ns*ns)
Multiply	$(I - K_k H) P_k^-$	(ns*ns)x(ns*ns)

Table 1 : Different Matrix operations in Kalman Filter

4.3 CPU Implementation

The kalman filter was implemented in the CPU. All the five equations in the kalman filter were implemented for one iteration. Here the input sizes of the state estimate (ns) and the measurement (no) matrix were varied and their execution times are tabulated. Here all the inputs and the constants considered are generated using rand function and the inverse operation is done using Cholesky decomposition for semi-positive definite matrix.

4.4 GPU Implementation

Here we have implemented the kalman filter equations in the GPU using the CUDA programming language. We have written kernel codes for each of the five matrix operations namely addition, subtraction, transpose, multiplication and inversion. These kernel codes are in turn used for the 18matrix operations in the

kalman filter. The kernel codes for the matrix operations are in the matrix_kerenl.cu (file included)

Matrix Multiplication:

The tiled implementation is used for matrix multiplication. This is one way to reduce the number of accesses to global memory to have the threads load portions of matrices A and B into shared memory, where we can access them much more quickly. This is done to increase the computation to memory ratio in the GPU execution. We have decomposed matrices A and B into non-overlapping submatrices of size BLOCK_SIZExBLOCK_SIZE. If we load the left-most of those submatrices of matrix A into shared memory, and the top-most of those submatrices of matrix B into shared memory, then we can compute the first BLOCK_SIZE products and add them together just by reading the shared memory. But here is the benefit: as long as we have those submatrices in shared memory, every thread in our thread block (computing the BLOCK_SIZExBLOCK_SIZE submatrix of C) can compute that portion of their sum as well from the same data in shared memory. When each thread has computed this sum, we can load the next BLOCK_SIZExBLOCK_SIZE submatrices from A and B, and continue adding the term-by-term products to our value in C. And after all of the submatrices have been processed, we will have computed our entries in C. The kernel code for this portion of our program is shown below [7].

Matrix Inversion:

For matrix inversion of a covariance matrix, the Cholesky factoring is used for fast computation. The approach to calculate the co-variance matrix is as follows.

The covariance matrix P is decomposed to lower triangular matrix L and its transpose L^T i.e., $P = L\,L^T$ [10].

The inverse is calculated as shown below.

$I = AX = (LL^T)X = L(L^TX) = LY.$

For Cholesky factorization on a GPU, we settle on an implementation by Bouckaert of a blocked factoring algorithm which involves an iterative sequence of CPU inversion on diagonal blocks and GPU updates on left hand and lower blocks using CUBLAS (inverse.cpp-file included). For solution to a triangular system of equations, we use the TRSM (Triangle Solve Multiple) function of CUBLAS [9] which runs on GPU.

5 METHODOLOGY

The kalman filter is executed for one iteration done with state(x) and measurement matrices (z) of different sizes. The state estimate 'x' matrix's size (ns*ns) is varied in the range of 1000 to 3000. The size of the measurement matrix z, no is chosen such that ns is $1/4^{th}$, $2/4^{th}$ or $3/4^{th}$ of no.

The input vectors of the kalman filter, i.e., *Q, H, V, z, x, P* are generated using a pseudo random sequence. The inverse of the co variance matrix can be applied on a positive definite matrix. Hence the co-variance matrix is initialized to a positive definite matrix.

6 EVALUATIONS

Our evaluations are based upon the execution time taken by the GPU and comparing it with the CPU execution time. While implementing in GPU the memory transfer time is also calculated and is used when comparing with the CPU time.

The computations are carried out on a quad core Intel(R) Xeon(R) E5420 2.5 GHz CPU and Tesla C2075 Graphics Card. The table below shows the specifications of this card.

Number of Streaming Processor Cores	448 cuda cores
Frequency of Processor Cores	1.15GHz
Total Dedicated Memory	5 GB
Memory Speed	1.566 GHz
Memory Interface	384-bit
Memory Bandwidth	144 GB/sec

Table 2: Specifications of Tesla C2075

The card consists of 14 multiprocessors which in turn has 32 cuda processors.

Each cuda processor inside a multiprocessor runs synchronously.

The execution threads are organized into grids that consist of thread blocks.

The thread blocks consist of a large no of threads that are grouped into warps of 32 threads. The threads belonging to a warp execute the same instruction.

State Matrix(ns)	Measurement Matrix(no)	CPU execution time(ms)	GPU execution Time(ms)	Memory Transfer Time(ms)	Overall Execution Time(ms)	Overall Speed Up
1000	250	17017.1289	12.195	2.457	14.652	1161
1000	500	22084.2421	21.33	2.736	24.181	917
1000	750	29617.6914	34.021999	3.253	37.274	794
2000	500	189564.578	21.445	7.21	28.54	6613
2000	1000	255140.906	57.743	8.527	66.27	3850
2000	1500	345406.937	120.288	9.91	138.48	2654
3000	750	774001.437	34.023998	14.894	48.917	15822
3000	1500	960695.437	128.57	22.59299	151.16	6355
3000	2250	1123567.89	157.54	31.65	189.19	5939

Table 3 : Evaluation results of kalman filter execution on CPU and GPU

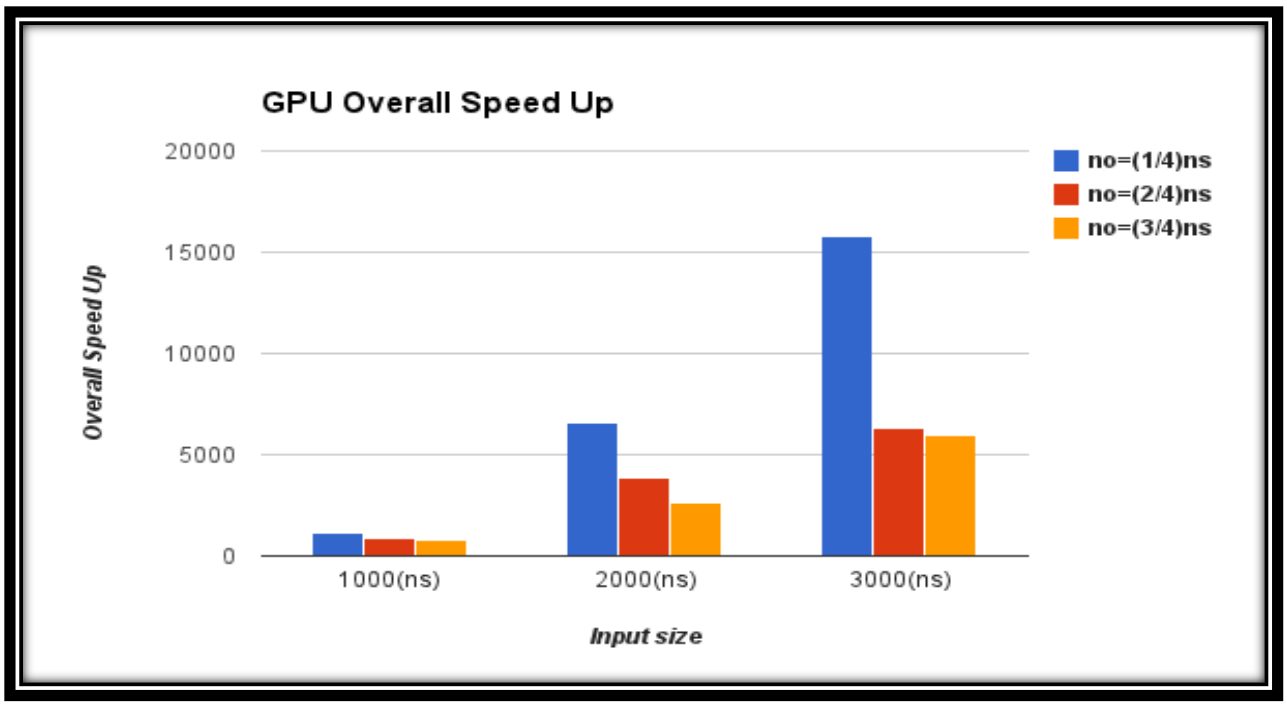

Figure 1: The GPU Execution time versus CPU Execution time

Figure 1 shows the speed up achieved by the GPU for different state dimension input size. It can be seen that for a fixed ratio of ns and no the speed up increases with increase in ns. Also, for a fixed ns value, maximum speed up is achieved in case of observation dimension (no) being 1/4th of ns when compared to other two cases.

Figure 2 below shows the memory transfer overhead that is observed in each case of GPU execution. It can be seen that the overhead is within acceptable limits and it always remain lower than the GPU computation time. Hence, the overall execution time of GPU remains much lower when compared to CPU execution time.

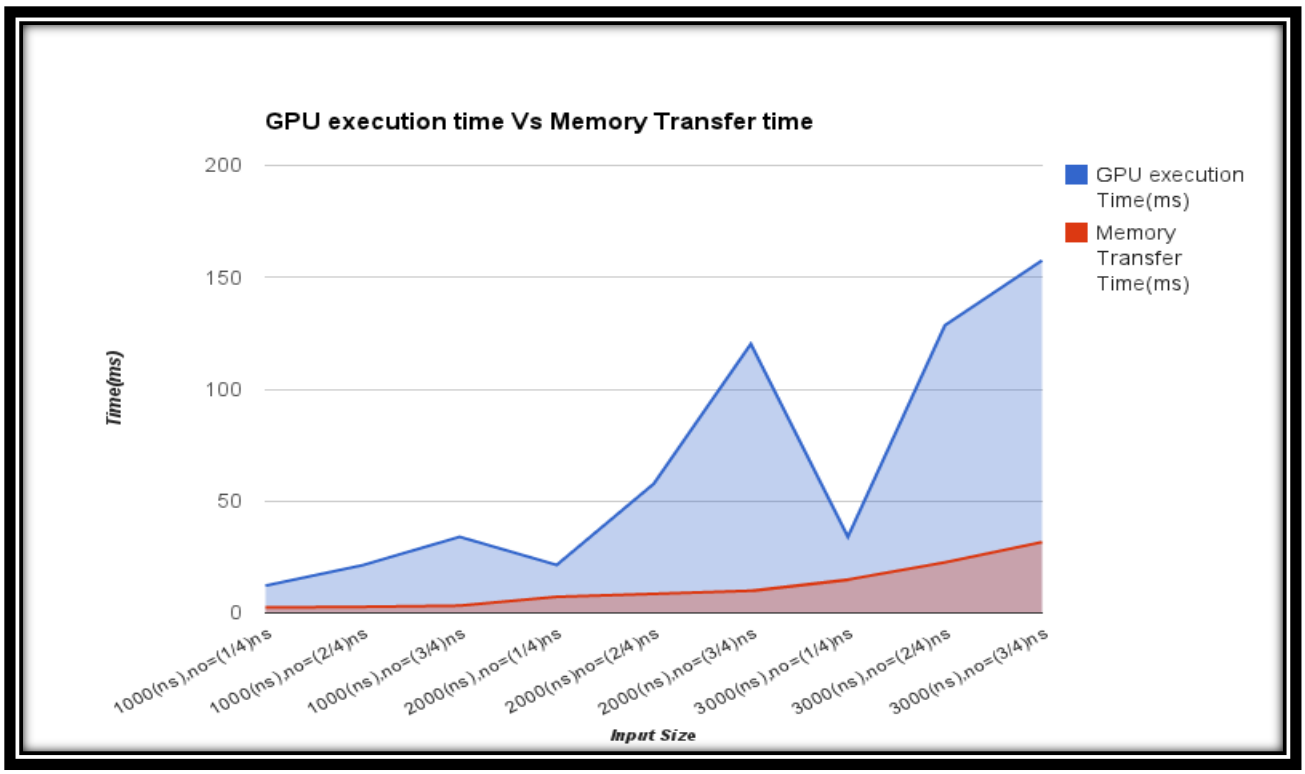

Figure 2: GPU computation time versus the Memory transfer time

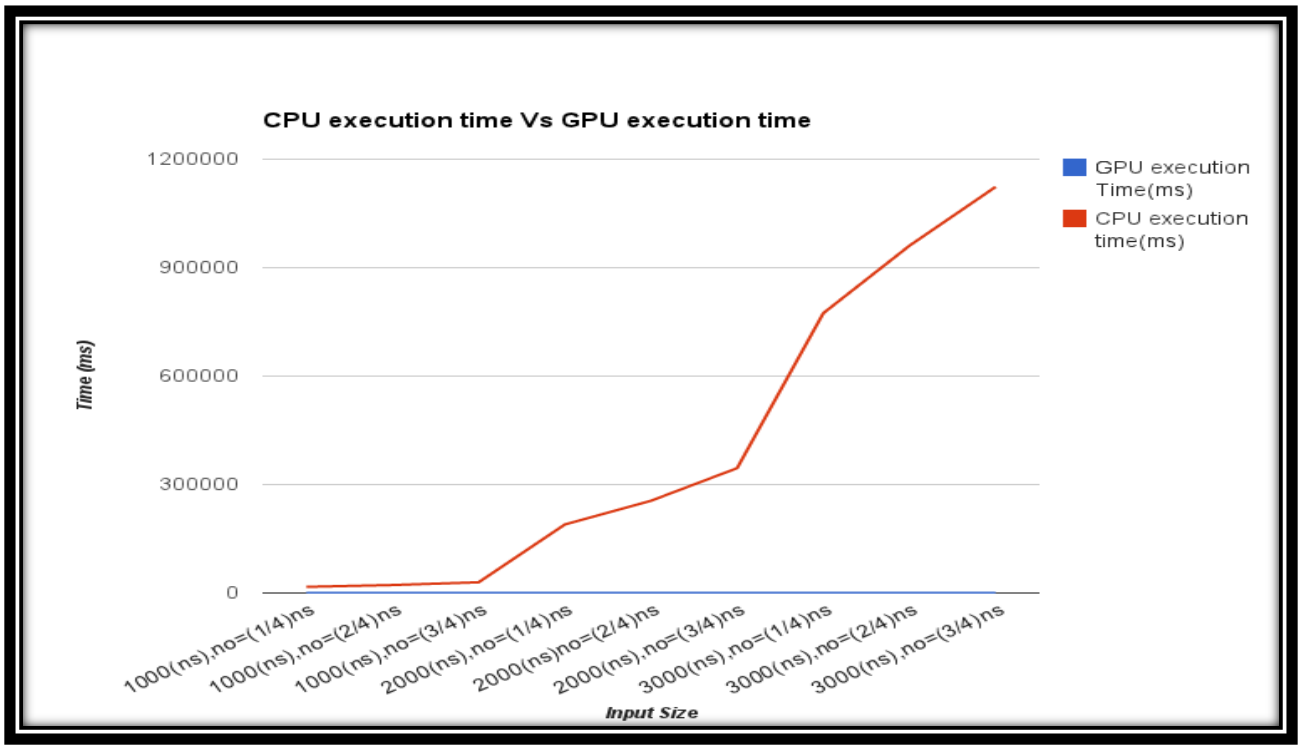

Figure 3: Comparison of CPU and GPU execution time

Figure 3 shows the comparison of the CPU and the GPU execution time for input ranges. Here we can observe that the difference between the execution times increases at very fast rate with the increases of the sate dimension input.

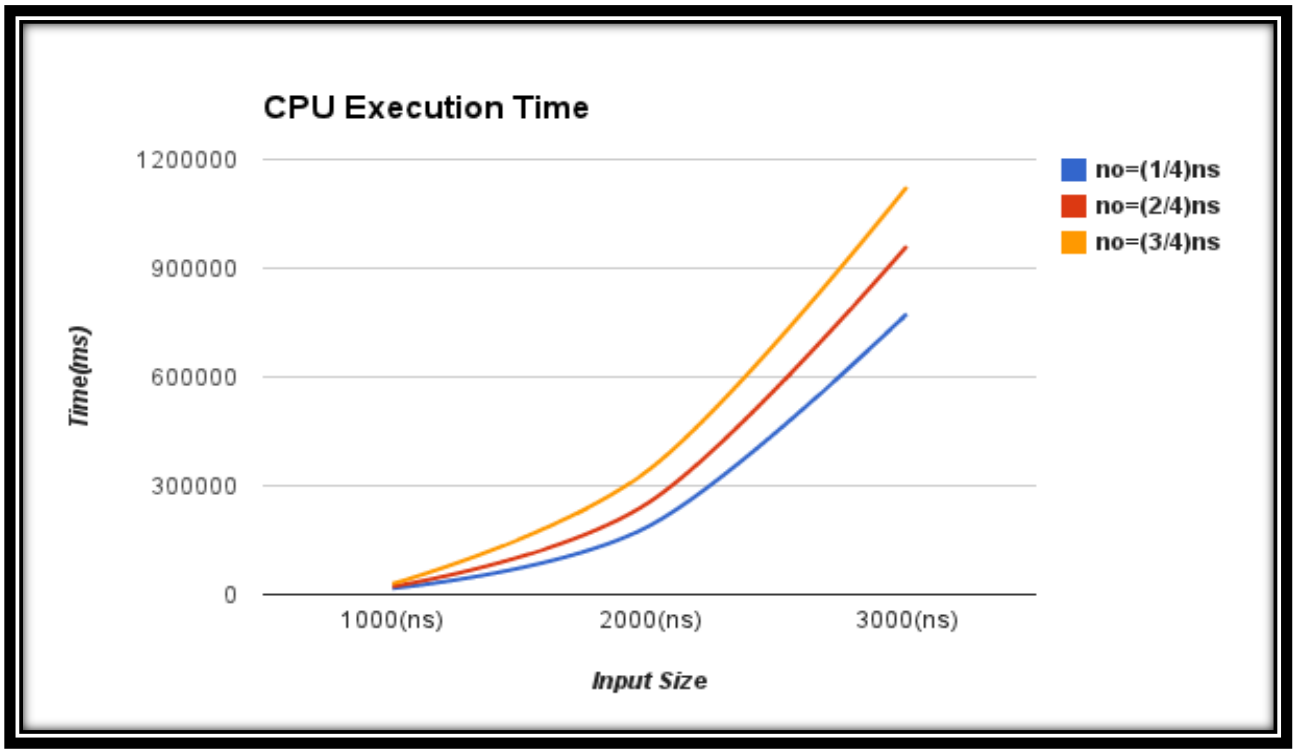

Figure 4 : CPU execution time for different input size

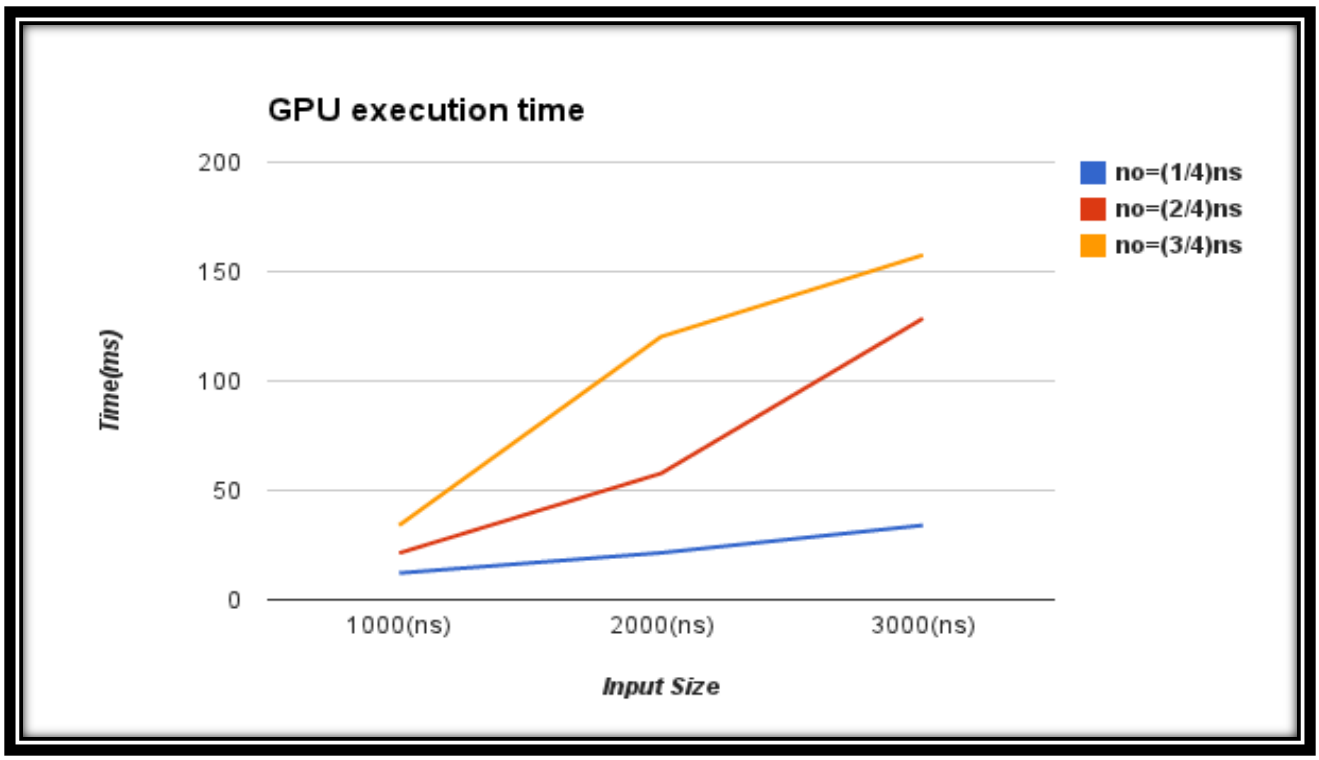

Figure 5 : GPU Execution time for different input size

Figure 4 and 5 shows the CPU and GPU execution time for the kalman filter execution. The state dimension (ns) is varied from 1000 to 3000 in steps of 100 while the observation dimension (no) is fixed at 1/4, 2/4, ¾ of the state dimension.

The computation time increases with the increase of state dimension and the observation dimension in both the cases. Also, the CPU execution time increases almost linearly with the increases in the input size. However, the GPU execution time increases at a lower rate when compared to CPU execution time.

7 CONCLUSION AND FUTURE WORK

Kalman filter is used to compute the new estimate of the of the system state based on its previous state, observation, error co-variance and the other system models. With the two phases of operation, the predict phase and the update phase it has overall 19 matrix operations including addition, subtraction, multiplication, transpose and inverse.

In our experiment, we have tried to evaluate the overall execution time for all the operations involved in kalman filter on GPU for different size of input state matrix. Also, we have compared the GPU and CPU execution times. It can be seen that GPU speed up is in the order of 1000s for different input size. The CPU computation time grows linearly with the input size whereas the GPU computation time grows sub linearly.
Further, we observed that the matrix inverse is the most computationally intensive operation amongst all the other matrix operations consuming nearly 30% of overall execution time.

For our future work, we would like to experiment with the real applications of the kalman filter mainly targeting the challenges involved in object tracking. Here, we would like to analyze how effective the use of GPU acts when it comes to different iterations of prediction and update phase barring the memory transfer overhead.

REFERENCES:

[1] An Introduction to the Kalman Filter by Greg Welch and Gary Bishop.
[2] http://en.wikipedia.org/wiki/Kalman_filter
[3] http://en.wikipedia.org/wiki/GPGPU
[4] http://www.nvidia.com/object/what-is-gpu-computing.html
[5] GPGPU origins and GPU hardware architecture by Stephan Soller
[6] http://en.wikipedia.org/wiki/CUDA
[7] Matrix Multiplication with CUDA-A basic introduction to the CUDA programming model by Robert Hochberg.
[8] Min-Yu Huang; Shih-Chieh Wei; Bormin Huang; Yang-Lang Chang; , "Accelerating the Kalman Filter on a GPU," Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on , vol., no., pp.1016-1020, 7-9 Dec. 2011
[9] R. Bouckaert, Matrix inverse with Cuda and CUBLAS, available at http://www.cs.waikato.ac.nz/~remco/.
[10] G. H. Golub and C.F. Van Loan, Matrix Computations, John Hopkins University Press, 1996.