A GRAPHICS PROCESSING UNIT IMPLEMENTATION OF THE PARTICLE FILTER

Gustaf Hendeby, Jeroen D. Hol, Rickard Karlsson, Fredrik Gustafsson

Department of Electrical Engineering
Automatic Control
Linköping University, Sweden
{hendeby, hol, rickard, fredrik}@isy.liu.se

ABSTRACT

Modern graphics cards for computers, and especially their graphics processing units *(GPUs), are designed for fast rendering of graphics. In order to achieve this GPUs are equipped with a parallel architecture which can be exploited for* general-purpose computing on GPU *(GPGPU) as a complement to the* central processing unit *(CPU). In this paper GPGPU techniques are used to make a parallel GPU implementation of state-of-the-art recursive Bayesian estimation using* particle filters *(PF). The modifications made to obtain a parallel particle filter, especially for the resampling step, are discussed and the performance of the resulting GPU implementation is compared to one achieved with a traditional CPU implementation. The resulting GPU filter is faster with the same accuracy as the CPU filter for many particles, and it shows how the particle filter can be parallelized.*

1. INTRODUCTION

Modern *graphics processing units* (GPUs) are designed to handle huge amounts of data about a scene and to render output to screen in real time. To achieve this, the GPU is equipped with a *single instruction multiple data* (SIMD) parallel instruction architecture. GPUs are developing rapidly in order to meet the ever increasing demands from the computer game industry, and as a side-effect, *general-purpose computing on graphics processing units* (GPGPU) has emerged to utilize this new source of computational power [1–3]. For highly parallelizable algorithms the GPU may even outperform the sequential *central processing unit* (CPU).

The *particle filter* (PF) is an algorithm to perform recursive Bayesian estimation [4–6]. Due to its nature, a large part consists of performing identical operations on many particles (samples), so it is potentially well suited for parallel implementation. Successful parallelization may lead to a drastic reduction of computation time and open up for new applications requiring large state space descriptions with many particles. Nonetheless, filtering and estimation algorithms have only recently been investigated in this context, see for instance [7, 8]. There are many types of parallel hardware available nowadays; examples include multicore processors, *field-programmable gate arrays* (FPGAs), computer clusters, and GPUs. GPUs are low cost and easily accessible SIMD parallel hardware — almost every new computer comes with a decent graphics card. Hence, GPUs are an interesting option for speeding up a PF and to test parallel implementations. A first GPU implementation of the PF was reported in [9] for a visual tracking computer vision application. In contrast, in this paper a general PF GPU implementation is developed. To the best of the authors' knowledge no successful complete implementation of a general PF on a GPU has yet been reported and this article aims to fill this gap: GPGPU techniques are used to implement a PF on a GPU and its performance is compared to that of a CPU implementation.

The paper is organized as follows: In Section 2 GPGPU programming is briefly introduced and this is used in Section 3 to discuss various aspects of the PF requiring special attention for a GPU implementation. Results from CPU and GPU implementations are compared in Section 5, and concluding remarks are given in Section 6.

This work has been funded by the Swedish Research Council (VR), the EU-IST project MATRIS, and the Strategic Research Center MOVIII, funded by the Swedish Foundation for Strategic Research, SSF.

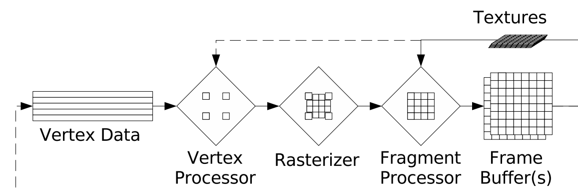

Figure 1: The graphics pipeline. The vertex and fragment processors can be programmed with user code which will be evaluated in parallel on several pipelines. (See Section 2.1.)

2. GENERAL PURPOSE GRAPHICS PROGRAMMING

GPUs operate according to the standardized *graphics pipeline* (see Figure 1), which is implemented at hardware level [2]. This pipeline, which defines how the graphics should be processed, is highly optimized for the typical graphics application, *i.e.*, displaying 3D objects.

The vertex processor receives vertices, *i.e.*, corners of the geometrical objects to display, and transform and project them to determine how the objects should be shown on the screen. All vertices are processed independently and as much in parallel as there are pipelines available. In the rasterizer it is determined what fragments, or potential pixels, the geometrical shapes may result in, and the fragments are passed on to the fragment processor. The fragments are then processed independently and as much in parallel as there are pipelines available, and the resulting color of the pixels is stored in the frame buffer before being shown on the screen.

At the hardware level the graphics pipeline is implemented using a number of processors, each having multiple pipelines performing the same instruction on different data. That is, GPUs are SIMD processors, and each processing pipeline can be thought of as a parallel sub-processor.

2.1 Programming the GPU

The two steps in the graphics pipeline open to programming are the vertex processor (working with the primitives making up the polygons to be rendered) and the fragment processor (working with fragments, *i.e.*, potential pixels in the final result). Both these processors can be controlled with programs called *shaders*, and consist of several parallel pipelines (subprocessors) for SIMD operations.

Shaders, or GPU programs, were introduced to replace, what used to be, fixed functionality in the graphics pipeline with more flexible programmable processors. They were mainly intended to allow for more advanced graphics effects, but they also got GPGPU started. Programming the vertex and fragment processors is in many aspects very similar to programming a CPU, with limitations and extensions made to better support the graphics card and its intended usage, but it should be kept in mind that the code runs in parallel on multiple pipelines of the processor.

Some prominent differences include the basic data types which are available; most operations of a GPU operate on *colors* (represented by one to four floating point numbers), and data is sent to and from the graphics card using *textures* (1D–3D arrays of color data). In newer generations of GPUs 32 bit floating point operations are supported, but the rounding units do not fully conform to the IEEE floating point standard, hence providing somewhat poorer numerical accuracy.

In order to use the GPU for general purpose calculations, a typical GPGPU application applies a program structure similar to Algorithm 1. These very simple steps make sure that the fragment program is executed once for every element of the data. The workload is automatically distributed over the available processor pipelines.

Algorithm 1 GPGPU skeleton program[1]

1. Program the fragment shader with the desired operation.
2. Send the data to the GPU in the form of a texture.
3. Draw a rectangle of suitable size on the screen to start the calculation.
4. Read back the resulting texture to the CPU.

2.2 GPU Programming Language

There are various ways to access the GPU resources as a programmer, including *C for graphics* (Cg), [10], and OpenGL [11] which includes the *OpenGL Shading Language* (GLSL), [12]. This paper will use GLSL that operates closer to the hardware than Cg. For more information and alternatives see [1, 2, 10].

To run GLSL code on the GPU, the OpenGL *application programming interface* (API) is used [11, 12]. The GLSL code is passed as text to the API that compiles and links the different shaders into binary code that is sent to the GPU and executed the next time the graphics card is asked to render a scene.

[1] The stream processing capabilities of the upcoming GPU generations might change this rather complicated method of performing GPGPU.

3. RECURSIVE BAYESIAN ESTIMATION

The general nonlinear filtering problem is to estimate the state, x_t, of a state-space system

$$x_{t+1} = f(x_t, w_t), \tag{1a}$$
$$y_t = h(x_t) + e_t, \tag{1b}$$

where y_t are measurement and $w_t \sim p_w(w_t)$ and $e_t \sim p_e(e_t)$ are process and measurement noise, respectively. The function f describes the dynamics of the system, h the measurements, and p_w and p_e are *probability density functions* (PDF) for the involved noise. For the important special case of linear-Gaussian dynamics and linear-Gaussian observations the Kalman filter, [13, 14], solves the estimation problem in an optimal way. A more general solution is the *particle filter* (PF), [4–6], which approximately solves the Bayesian inference for the posterior state distribution, [15], given by

$$p(x_{t+1}|\mathbb{Y}_t) = \int p(x_{t+1}|x_t) p(x_t|\mathbb{Y}_t)\, dx_t, \tag{2a}$$
$$p(x_t|\mathbb{Y}_t) = \frac{p(y_t|x_t) p(x_t|\mathbb{Y}_{t-1})}{p(y_t|\mathbb{Y}_{t-1})}, \tag{2b}$$

where $\mathbb{Y}_t = \{y_i\}_{i=1}^t$ is the set of available measurements. The PF uses statistical methods to approximate the integrals. The basic PF algorithm is given in Algorithm 2.

Algorithm 2 Basic Particle Filter [5]

1. Let $t := 0$, generate N particles $\{x_0^{(i)}\}_{i=1}^N \sim p(x_0)$.
2. Measurement update: Compute the particle weights $\omega_t^{(i)} = p(y_t|x_t^{(i)}) \,/\, \sum_j p(y_t|x_t^{(j)})$.
3. Resample:
   (a) Generate N uniform random numbers $\{u_t^{(i)}\}_{i=1}^N \sim \mathscr{U}(0,1)$.
   (b) Compute the cumulative weights: $c_t^{(i)} = \sum_{j=1}^i \omega_t^{(j)}$.
   (c) Generate N new particles using $u_t^{(i)}$ and $c_t^{(i)}$: $\{x_t^{(i\star)}\}_{i=1}^N$ where $\Pr(x_t^{(i\star)} = x_t^{(j)}) = \omega_t^j$.
4. Time update:
   (a) Generate process noise $\{w_t^{(i)}\}_{i=1}^N \sim p_w(w_t)$.
   (b) Simulate new particles $x_{t+1}^{(i)} = f(x_t^{(i\star)}, w_t^{(i)})$.
5. Let $t := t+1$ and repeat from 2.

4. GPU BASED PARTICLE FILTER

To implement a parallel PF on a GPU there are several aspects of Algorithm 2 that require special attention. Resampling and weight normalization are the two most challenging steps to implement in a parallel fashion since in these steps all particles and their weights interact with each other. The main difficulties are cumulative summation, and selection and redistribution of particles. In the following sections, solutions suitable for parallel implementation are proposed for these tasks, together with a discussion on issues with random number generation, likelihood evaluation as part of the measurement update, and state propagation as part of the time update.

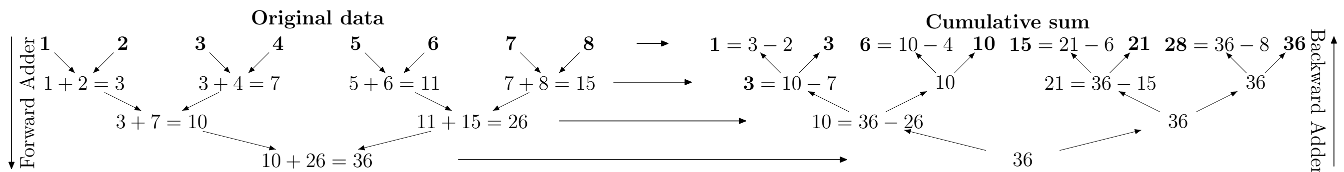

Figure 2: Illustration of a parallel implementation of cumulative sum generation of the numbers $1,2,\ldots,8$. First the sum is calculated using a forward adder tree. Then the partial summation results are used by the backward adder to construct the cumulative sum; $1,3,\ldots,36$.

4.1 Random Number Generation

At present, state-of-the-art graphics cards do not have sufficient support for random number generation for usage in a PF, since the statistical properties of the built-in generators are too poor. The algorithm in this paper therefore relies on random numbers generated on the CPU to be passed to the GPU. This introduces quite a lot of data transfer as several random numbers per particle are required for one iteration of the PF. Uploading data to the graphics card is rather quick, but still some performance is lost.

Generating random numbers on the GPU suitable for use in Monte Carlo simulation is an ongoing research topic, see *e.g.*, [16–18]. Doing so will not only reduce data transport and allow a standalone GPU implementation, an efficient parallel version will improve overall performance as the random number generation itself takes a considerable amount of time.

4.2 Likelihood Evaluation and State Propagation

Both likelihood evaluation (as part of the measurement update) and state propagation (in the time update), Steps 2 and 4b of Algorithm 2, can be implemented straightforwardly in a parallel fashion since all particles are handled independently. As a consequence of this, both operations can be performed in $\mathcal{O}(1)$ time with N parallel processors, *i.e.*, one processing element per particle. To solve new filtering problems, only these two functions have to be modified. As no parallelization issues need to be addressed, this is easily accomplished.

In the presented GPU implementation the particles $x^{(i)}$ and the weights $\omega^{(i)}$ are stored in separate textures which are updated by the state propagation and the likelihood evaluation, respectively. Textures can only hold four dimensional state vectors, but using multiple rendering targets the state vectors can easily be extended when needed. When the measurement noise is low-dimensional the likelihood computations can be replaced with fast texture lookups utilizing hardware interpolation. Furthermore, as discussed above, the state propagation uses externally generated process noise, but it would also be possible to generate the random numbers on the GPU.

4.3 Summation

Summations are part of the weight normalization (during measurement updates) and cumulative weight calculation (during resampling), Steps 2 and 3b of Algorithm 2. A cumulative sum can be implemented using a multi-pass scheme, where an adder tree is run forward and then backward, as illustrated in Figure 2. Running only the forward pass the

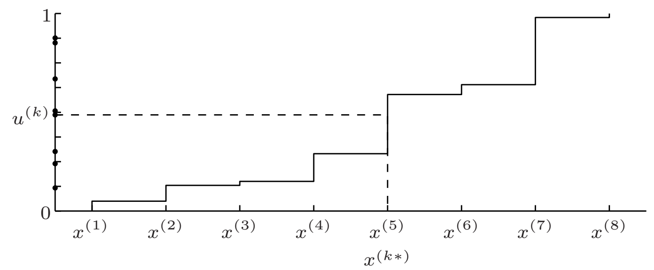

Figure 3: Particle selection by comparing uniform random numbers $(\cdot)$ to the cumulative sum of particle weights (–).

total sum is computed. This multi-pass scheme is a standard method for parallelizing seemingly sequential algorithms based on gather and scatter principles. The reference [2] describes these concepts in for the GPU setting. In the forward pass partial sums are created that are used in the backward pass to compute the missing partial sums to complete the cumulative sum. The resulting algorithm is $\mathcal{O}(\log N)$ in time given N parallel processors and N particles.

4.4 Particle Selection

To prevent sample impoverishment, the resampling step, Steps 3 of Algorithm 2, replaces the weighted particle distribution with a unweighted one. This is done by drawing a new set of particles $\{x^{(i\star)}\}$ with replacement from the original particles $\{x^{(i)}\}$ in such a way that $\Pr(x^{(i\star)} = x^{(j)}) = \omega^{(j)}$. Standard resampling algorithms [4, 19, 20] select the particles by comparing uniform random numbers $u^{(k)}$ to the cumulative sum of the normalized particle weights $c^{(i)}$, as illustrated in Figure 3. That is, assign

$$x_t^{(k\star)} = x_t^{(i)}, \text{with } i \text{ such that } u^{(k)} \in [c_t^{(i-1)}, c_t^{(i)}), \tag{3}$$

which makes use of an explicit expression for the generalized inverse cumulative probability distribution.

Different methods are used to generate the uniform random numbers [20]. Stratified resampling, [19], generates uniform random numbers according to

$$u^{(k)} = \frac{(k-1) + \tilde{u}^{(k)}}{N}, \text{ with } \tilde{u}^{(k)} \sim \mathcal{U}(0,1), \tag{4}$$

whereas systematic resampling, [19], uses

$$u^{(k)} = \frac{(k-1) + \tilde{u}}{N}, \text{ with } \tilde{u} \sim \mathcal{U}(0,1), \tag{5}$$

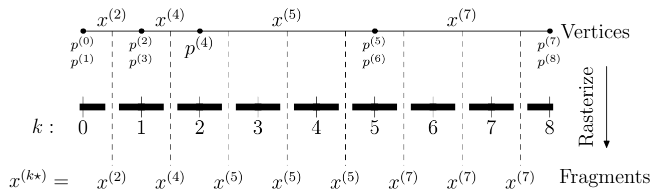

Figure 4: Particle selection on the GPU. The vertices p, cumulative weights snapped to an equidistant grid, define a line where every segment represents a particle. Some vertices may coincide, resulting in line segments of zero length. The rasterizer creates particles x according to the length of the corresponding line segments.

where $\mathscr{U}(0,1)$ is a uniform distribution between 0 and 1. Both methods produce ordered uniform random numbers which have exactly one number in every interval of length N^{-1}, reducing the number of $u^{(k)}$ to be compared to $c^{(i)}$ to a single one. This is the key property enabling a parallel implementation.

Utilizing the rasterization functionality of the graphics pipeline, the selection of particles can be implemented in a single render pass: calculate vertices $p^{(i)}$ by assigning the cumulative weights $c^{(i)}$ to an equidistant grid depending on the uniform random numbers $u^{(i)}$. That is,

$$p^{(i)} = \begin{cases} \lfloor Nc^{(i)} \rfloor, & \text{if } Nc^{(i)} - \lfloor Nc^{(i)} \rfloor < u^{(\lfloor Nc^{(i)} \rfloor)}, \\ \lfloor Nc^{(i)} \rfloor + 1, & \text{otherwise,} \end{cases} \quad (6)$$

where $\lfloor x \rfloor$ is the floor operation. Drawing a line connecting the vertices $p^{(i)}$ and associating a particle to every line segment, the rasterization process creates the resampled set of particles according to the length of each segment. This procedure is illustrated with an example in Figure 4 based upon the data in Figure 3. The computational complexity of this is $\mathscr{O}(1)$ with N parallel processors, as the vertex positions can be calculated independently. Unfortunately, the current generation of GPUs has a maximal texture size limiting the number of particles that can be resampled as a single unit. To solve this, multiple subsets of particles are simultaneously being resampled and then redistributed into different sets, similarly to what is described in [21]. This modification of the resampling step does not seem to significantly affect the performance of the particle filter as a whole.

4.5 Complexity Considerations

From the descriptions of the different steps of the PF algorithms it is clear that the resampling step is the bottleneck that gives the time complexity of the algorithm, $\mathscr{O}(\log N)$ compared to $\mathscr{O}(N)$ for a sequential algorithm.

The analysis of the algorithm complexity above assumes that there are as many parallel processors as there are particles in the particle filter, *i.e.*, N parallel elements. Today this is a bit too optimistic, a modern GPU has an order of ten parallel pipelines, hence much less than the typical number of particles. However, the number of parallel units is constantly increasing so the degree of parallelization is improving.

Especially the cumulative sum suffers from a low degree of parallelization. With full parallelization the time complexity of the operation is $\mathscr{O}(\log N)$ whereas the sequential algorithms is $\mathscr{O}(N)$, however the parallel implementation uses $\mathscr{O}(N \log N)$ operations in total. As a result, with few pipelines and many particles the parallel implementation will be slower than the sequential one. However, as the degree of parallelization increases this will be less and less important.

Table 1: Hardware used for the evaluation.

GPU	
Model:	NVIDIA GeFORCE 7900 GTX
Driver:	2.1.0 NVIDIA 96.40
Bus:	PCI Express, 14.4 GB/s
Clock speed:	650 MHz
Processors:	8/24 (vertex/fragment)
CPU	
Model:	Intel Xeon 5130
Clock speed:	2.0 GHz
Memory:	2.0 GB
Operating System:	CentOS 4.4 (Linux)

5. FILTER EVALUATION

To evaluate the designed PF on the GPU two PF have been implemented; one standard PF running on the CPU and one implemented as described in Section 4 running on the GPU. (The code for both implementations is written in C++ and compiled using `gcc` 3.4.6.) The filters were then used to filter data from a constant velocity tracking model, measured with two distance measuring sensors. The estimates obtained were very similar with only small differences that can be explained by the different resampling method (one, or multiple sets) and the presence of round off errors. This shows that the GPU implementation does work, and that the modification of the resampling step is acceptable. The hardware is presented in Table 1. Note that there are 8 parallel pipelines in which the particle selection and redistribution is conducted and that the rest of the steps in the PF algorithm is performed in 24 pipelines, *i.e.*, $N \gg$ number of pipelines.

To study the time complexity of the PF, simulations with 1000 time steps were run with different numbers of particles. The time spent in the particle filters was recorded, excluding the generation of the random numbers which was the same for both filter implementations. The results can be found in Figure 5. The maximum number of particles (10^6) may seem rather large for current applications, however, it helps to show the trend in computation time and to show that it is possible to use this many particles. This makes it possible to work with large state dimensions and open up for PFs in new application areas.

Some observations should be made: for few particles the overhead from initializing and using the GPU is large and hence the CPU implementation is the fastest. The CPU complexity follows a linear trend, whereas at first the GPU time hardly increases when using more particles; parallelization pays off. For even more particles there are not enough parallel processing units available and the complexity becomes linear, but the GPU implementation is still faster than the CPU. Note that the particle selection is performed on 8 processors and the other steps on 24, see Table 1, and that hence the degree of parallelization is not very high for many particles.

A further analysis of the time spent in the GPU implementation shows in what part of the algorithm most of the time is spent. Figure 6, shows that most of the time is spent in the resampling step, and that the portion of time spent there

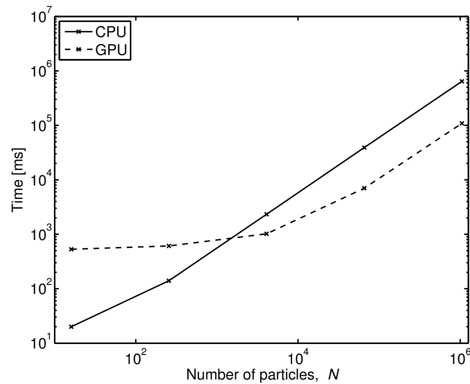

Figure 5: Time comparison between CPU and GPU implementation. The number of particles is large to show that the calculation is tractable, and to show the effect of the parallelization. Note the log-log scale.

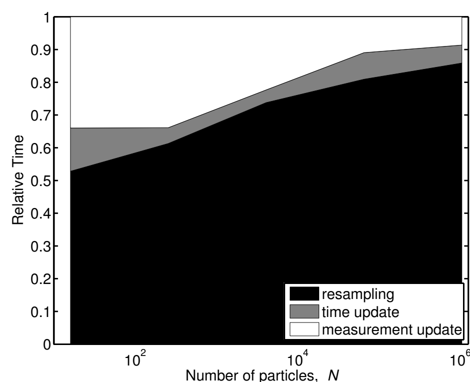

Figure 6: Relative time spent in different parts of GPU implementation.

increases with more particles. This is quite natural since this step is the least parallel in its nature and requires multiple passes. Hence, optimization efforts should be directed into this part of the algorithm.

6. CONCLUSIONS

In this paper, the first complete parallel general particle filter implementation in literature on a GPU is described. Using simulations, the parallel GPU implementation is shown to outperform a CPU implementation on computation speed for many particles while maintaining the same filter quality. The techniques and solutions used in deriving the implementation can also be used to implement particle filters on other similar parallel architectures.

References

[1] "GPGPU programming web-site," 2006, `http://www.gpgpu.org`.

[2] M. Pharr, Ed., *GPU Gems 2. Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley, 2005.

[3] M. D. McCool, "Signal processing and general-purpose computing on GPUs," *IEEE Signal Process. Mag.*, vol. 24, no. 3, pp. 109–114, May 2007.

[4] A. Doucet, N. de Freitas, and N. Gordon, Eds., *Sequential Monte Carlo Methods in Practice*, Statistics for Engineering and Information Science. Springer-Verlag, New York, 2001.

[5] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," *IEE Proc.-F*, vol. 140, no. 2, pp. 107–113, Apr. 1993.

[6] B. Ristic, S. Arulampalam, and N. Gordon, *Beyond the Kalman Filter: Particle Filters for Tracking Applications*, Artech House, Inc, 2004.

[7] S. Maskell, B. Alun-Jones, and M. Macleoad, "A single instruction multiple data particle filter," in *Proc. Nonlinear Statistical Signal Processing Workshop*, Cambridge, UK, Sept. 2006.

[8] A. S. Montemayor, J. J. Pantrigo, A. Sánchez, and F. Fernández, "Particle filter on GPUs for real time tracking," in *Proc. SIGGRAPH*, Los Angeles, CA, USA, Aug. 2004.

[9] A. S. Montemayor, J. J. Pantrigo, R. Cabido, B. R. Payne, Á. Sánchez, and F. Fernáandez, "Improving GPU particle filter with shader model 3.0 fir visual tracking," in *Proc. SIGGRAPH*, Boston, MA, USA, Aug. 2006.

[10] "NVIDIA developer web-site," 2006, `http://developer.nvidia.com`.

[11] D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Language. The official guide to learning OpenGL, Version 2*, Addison-Wesley, 5 edition, 2005.

[12] R. J. Rost, *OpenGL Shading Language*, Addison-Wesley, 2 edition, 2006.

[13] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Trans. ASME*, vol. 82, no. Series D, pp. 35–45, Mar. 1960.

[14] T. Kailath, A. H. Sayed, and B. Hassibi, *Linear Estimation*, Prentice-Hall, Inc, 2000.

[15] A. H. Jazwinski, *Stochastic Processes and Filtering Theory*, vol. 64 of *Mathematics in Science and Engineering*, Academic Press, Inc, 1970.

[16] C. J. K. Tan, "The PLFG parallel pseud-random number generator," *Future Generation Computer Systems*, vol. 18, pp. 693–698, 2002.

[17] A. De Matteis and S. Pagnutti, "Parallelization of random number generators and long-range correlation," *Numer. Math.*, vol. 53, no. 5, pp. 595–608, 1988.

[18] M. Sussman, W. Crutchfield, and M. Papakipos, "Pseudorandom number generation on the GPU," in *Graphics Hardware. Eurographics Symp. Proc*, Vienna, Austria, Aug. 2006, pp. 87–94.

[19] G. Kitagawa, "Monte Carlo filter and smoother for non-gaussian nonlinear state space models," *J. Comput. and Graphical Stat.*, vol. 5, no. 1, pp. 1–25, Mar. 1996.

[20] J. D. Hol, T. B. Schön, and F. Gustafsson, "On resampling algorithms for particle filters," in *Proc. Nonlinear Statistical Signal Processing Workshop*, Cambridge, UK, Sept. 2006.

[21] M. Bolić, P. M. Djurić, and S. Hong, "Resampling algorithms and architectures for distributed particle filters," *IEEE Trans. Signal Process.*, vol. 53, no. 7, pp. 2442–2450, July 2005.