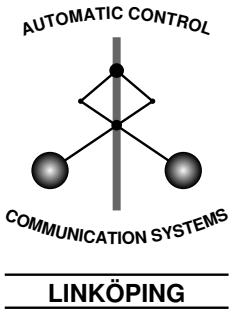
Graphics Processing Unit Implementation of the Particle Filter

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

Division of Automatic Control
Department of Electrical Engineering
Linköpings universitet, SE-581 83 Linköping, Sweden
WWW: http://www.control.isy.liu.se
E-mail: hendeby@isy.liu.se, hol@isy.liu.se, rickard@isy.liu.se, fredrik@isy.liu.se

9th October 2006



Report no.: LiTH-ISY-R-2749
Submitted to IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Hawaii, USA, 2007

Technical reports from the Control & Communication group in Linköping are available at http://www.control.isy.liu.se/publications.

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 have a parallel architecture which can be exploited for *general-purpose computing on graphics processing units* (GPGPU) as a complement to the *central processing unit* (CPU). In this paper GPGPU techniques are used to implement state-of-the-art recursive Bayesian estimation using *particle filters* (PF). The main steps of the PF are highly parallel except for the resampling step, and this paper discuss one way to handle this. The performance of this implementation is compared to that achieved with a traditional CPU implementation. The resulting GPU filter is faster with the same performance as the CPU filter for many particles.

Keywords: Parallel programming, Monte Carlo methods, Estimation, Particle filtering, Graphics processing unit

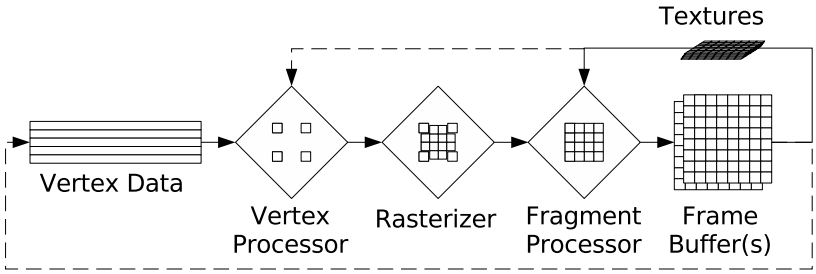


Figure 1: The graphics pipeline. The vertex and fragment processors can be programmed with user code.

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 parallel architecture with *single instruction multiple data* (SIMD) type instructions. GPUs are developing rapidly in order to meet the ever increasing demands from computer games 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, 2]. For highly parallelizable algorithms the GPU may even outperform the serial *central processing unit* (CPU).

The *particle filter* (PF) is an algorithm to perform recursive Bayesian estimation [3, 4, 5]. Due to its nature, performing identical operations on many particles, it is potentially well suited for parallel implementation if it had not been for the resampling step. GPUs are low cost and easily accessible SIMD parallel hardware, hence they are an interesting option for speeding up a PF and for testing parallel implementations. Non-the-less, filtering and estimation algorithms have only recently been investigated in this context, see for instance [6, 7]. To the best of the authors knowledge no successful complete implementation of the PF on a GPU has yet been reported. In this paper GPGPU techniques are used to implement a PF on a GPU. Its performance is compared to a CPU implementation.

The paper is organized as follows: In Section 2 GPGPU programming is briefly described and this is used to discuss the PF in terms of GPGPU in Section 3. Results from CPU and GPU are compared in Section 4, and concluding remarks are given in Section 5.

2 General Purpose Graphics Programming

2.1 Graphics pipeline

GPUs operate according to the standardized graphics pipeline (see Fig. 1), which is implemented at hardware level [2]. This pipeline is highly optimized for the typical graphics application, *i.e.*, displaying 3D objects. Geometric primitives are presented as vertex data to the pipeline and processed consecutively by the vertex processor, the rasterizer, and the fragment processor before the result is

stored in a frame buffer and displayed on the screen. It is also possible to read back results from the GPU. Until recently, before the introduction of the PCI Express bus, this has been a quite slow operation.

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 for SIMD operations.

2.2 Programming the GPU

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 started GPGPU. Programming the vertex and fragment processors is in many aspects very similar to programming a CPU, with limitations and extensions to better support the graphics card and its intended usage, but it should be kept in mind that the code runs in parallel.

One GPU limitation is the basic data types available; most operations operates on *colors* (represented by 1–4 floating point numbers), and data is sent to 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, providing somewhat poorer numeric performance.

2.3 GPU Programming Language

There are various ways to access the GPU resources as a programmer, including *C for graphics* (Cg), [8], and OpenGL [9] which includes the *OpenGL Shading Language* (GLSL), [10]. This paper will use GLSL that operates closer to the hardware than Cg. For more information and alternatives see [1, 2, 8].

To run GLSL code on the GPU, the OpenGL *application programming interface* (API) is used [9, 10]. The GLSL code is passed as text to the API that compiles and links the different shaders into binary code that is sent the GPU and executed the next time the graphics card is asked to render a scene.

3 Recursive Bayesian Estimation

3.1 Particle Filtering

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 $w_t \sim p_w(w)$ and $e_t \sim p_e(e)$ are process and measurement noise, respectively. For the important special case of linear-Gaussian dynamics and linear-Gaussian observations the Kalman filter, [11, 12] solves the

problem in an optimal way. A more general solution is the *particle filter* (PF), [3, 4, 5], which approximately solves the Bayesian inference [13] given by

$$p(x_{t+1}|\mathbb{Y}_t) = \int_{\mathbb{R}^n} p(x_{t+1}|x_t)p(x_t|\mathbb{Y}_t)\,dx_t, \tag{2a}$$

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

where $\mathbb{Y}_t = \{y_i\}_{i=1}^t$, using statistical methods. The basic PF algorithm is given in Alg. 1.

Alg. 1 Basic Particle Filter

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 random numbers $\{u_t^{(i)}\}_{i=1}^N \sim \mathcal{U}(0,1)$.
   (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.

3.2 GPU Based Particle Filter

To implement a parallel PF on a GPU there are at least three important aspects of Alg. 1 needing special attention: random number generation, time and measurement updates, and resampling.

Random number generation At present, state-of-the-art graphics cards do not have sufficient support for random numbers for usage in a PF. The statistical properties are too poor. The algorithm in this paper therefore rely on random numbers generated on the CPU to be passed to the GPU. Uploading data to the graphics card is rather quick, and this should not impose a bottleneck in the implementation. However, it is an interesting future extension to generate random numbers on the GPU to get a stand-alone GPU implementation.

Time and measurement updates Both time update and measurement updates are straightforwardly implemented using Steps 2 and 4 of Alg. 1 since all particles are handled independently. As a consequence of this, both time and measurement updates can be performed in $\mathcal{O}(1)$ time with N parallel processors. When the measurement noise is low dimensional this can be utilized by replacing the likelihood computations with fast texture lookups utilizing hardware interpolation. Also, as discussed above, the time update uses externally

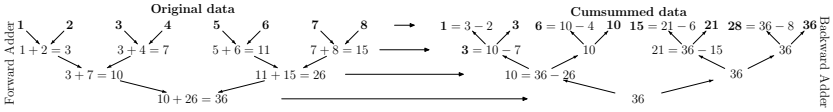


Figure 2: Illustration of a parallel implementation of cumulative sum generation. Note how the partial results of the forward adder are used in the backward adder.

generated process noise, but it would also be possible to generate the random numbers on the GPU.

Another observation is that since the underlying data type, color, uses 4 components, implementing models with more than 4 states forces the state to be stored in more than one color element. This imposes some extra bookkeeping, but should not be considered a limitation of the state to merely 4 components.

Resampling The resampling step is the most difficult step to parallelize because all particles and weights interact. The implementation used in this paper consists of two steps; computing the cumulative sum (cumsum) of the particle weights, and selecting and redistributing the particles according to stratified or systematic resampling (slight variations of Step 3, Alg. 1) [3, 14, 15]. Fortunately, both steps can be parallelized on the GPU at the price of a few more operations.

The cumsum can be implemented using a two pass scheme, where an adder tree is run forward and then backward, as illustrated in Fig. 2. This method is a standard method for parallelizing seemingly sequential algorithms. In the forward pass partial sums are created that are then used in the backward pass to compute the missing partial sums to complete the cumsum. The resulting algorithm is $\mathcal{O}(\log N)$ in time given N parallel processors, and easily extends to the 2D algorithm used on the GPU.

Stratified and systematic resampling perform particle selection and redistribution by comparing the cumsum of the particle weights $c^{(i)}$ to uniform random numbers $u^{(i)}$. This can be implemented in a single render pass making explicitly use of the graphics pipeline: lines are drawn connecting the vertices $p^{(i)}$, with

$$p^{(i)} = \begin{cases} \lfloor Nc^{(i)} \rfloor, & \text{if } Nc^{(i)} - \lfloor Nc^{(i)} \rfloor < u^{(i)} \\ \lfloor Nc^{(i)} \rfloor + 1, & \text{otherwise,} \end{cases} \tag{4}$$

and then the rasterization process creates copies of the particles based on length of the line segments. See Fig. 3. This scheme is $\mathcal{O}(1)$ with N parallel processors. Unfortunately, the maximal texture size limits the number of particles that can be resampled as one unit. To solve this multiple subsets of particles are resampled separately, simultaneously and then distributed into the different sets, in a way similar to what is described in [16]. This modification of the resampling step does not seem to significantly affect the performance of the particle filter as a whole.

To conclude the discussion about the parallel PF, with N processors it would be possible to implemented the PF with $\mathcal{O}(\log N)$ time complexity. The prize to pay for this is that the total number of operations increases. This is not a problem as long as the number of processors is large. State-of-the-art GPUs today have tens of parallel processors, and the number is growing fast.

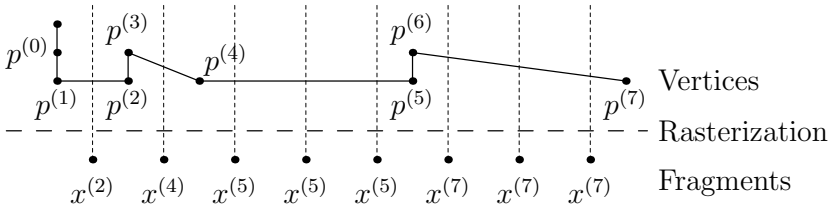


Figure 3: Illustration of the GPGPU resampling implementation. Note that several steps of the graphics pipeline interact.

Table 1: Hardware used for the evaluation.

GPU	
Model:	NVIDIA GeFORCE 6600 GT
Driver:	2.0.2 NVIDIA 87.74
Bus:	PCI Express, 14.4 GB/s
Clock speed:	500 MHz
Processors:	3/8 (vertex/fragment)
CPU	
Model:	AMD Athlon 3000+
Clock speed:	1.8 GHz
Memory:	1.0 GB
Operating System:	CentOS 4.4 (Linux)

4 Filter Evaluation

To evaluate the designed PF on the GPU two PF have been implemented; one standard SIR PF running on the CPU and one implemented as described in Sec. 3.2 running on the GPU. (The code for both implementations are 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 close, with only small differences that should be possible to be explain with the different resampling methods and round off errors. This shows that the GPU implementation in fact works, and that the modification of the resampling step is acceptable. The hardware used is presented in Table 1.

To study the time complexity as a function of particles in the PF, simulation followed by estimation of 1000 time steps were run with different numbers of particles. The time spent in the particle filters were recorded. The result can be found in Fig. 4. Some observations can be made, for few particles the overhead from initializing and using the GPU is large and hence the CPU implementation the fastest. With many particles the CPU has a slightly more favorable complexity trend, this because the parallelism in the GPU is too low. However, outside the extremes the GPU is faster than the CPU, showing that the parallelization pays off. With more processors in the GPU the positive GPU trend should be expected to continue. Hence, this shows the possibility of parallelization.

Further detailing the time spent in the GPU implementation shows in what part of the algorithm most the time is spent. See Fig. 5. Most the time is spent in the resampling step, and that the portion of time spent there increases with more particles. This is quite natural since this step is the least parallel in its nature and more effort should be spent trying to optimize this part of the

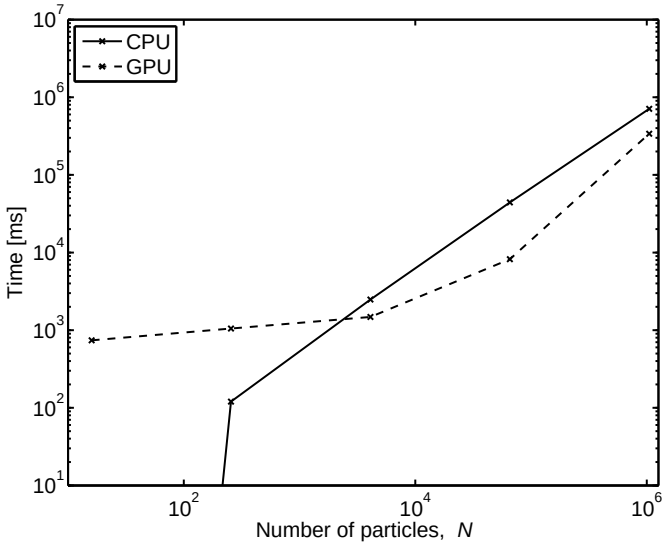


Figure 4: Time comparison between CPU and GPU implementation Note the log-log scale.

algorithm. The plot also shows that the measurement update is faster than the time update. This may be because the process noise is sent to the GPU instead of generated there.

5 Conclusions

In the paper general-purpose computing on graphics processing has been used to implement a particle filter using the OpenGL Shading Language. The implemented filter is shown in simulations to outperform a CPU implementation for many particles while maintaining the same filter performance. These ideas can also be used to implement particle filters on other similar parallel architectures.

Acknowledgment

This work has been funded by the Swedish Research Council (VR) and the EU-IST project MATRIS.

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] 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.

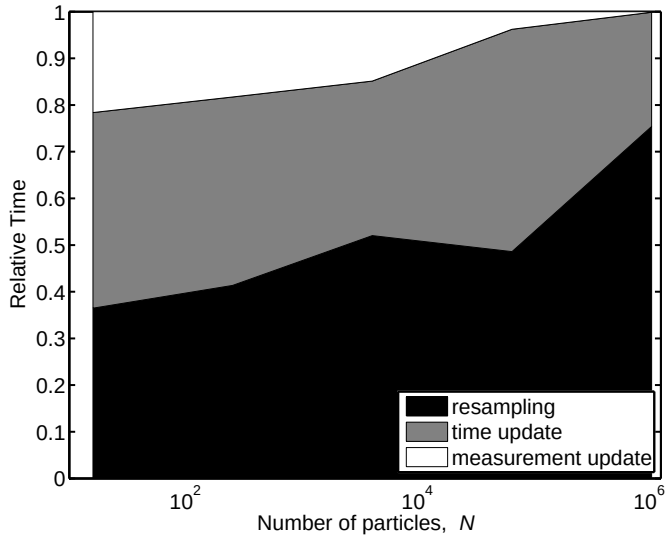


Figure 5: Relative time spent in different parts of GPU implementation.

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

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

[6] 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.

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

[9] 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.

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

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

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

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

[14] 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.

[15] 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.

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

Avdelning, Institution
Division, Department

Division of Automatic Control
Department of Electrical Engineering

Datum
Date

2006-10-09

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☐ Examensarbete
☐ C-uppsats
☐ D-uppsats
☒ Övrig rapport
☐ _____

ISBN
—

ISRN
—

Serietitel och serienummer
Title of series, numbering

ISSN
1400-3902

LiTH-ISY-R-2749

URL för elektronisk version

`http://www.control.isy.liu.se`

Titel
Title

Graphics Processing Unit Implementation of the Particle Filter

Författare
Author

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

Sammanfattning
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 have a parallel architecture which can be exploited for *general-purpose computing on graphics processing units* (GPGPU) as a complement to the *central processing unit* (CPU). In this paper GPGPU techniques are used to implement state-of-the-art recursive Bayesian estimation using *particle filters* (PF). The main steps of the PF are highly parallel except for the resampling step, and this paper discuss one way to handle this. The performance of this implementation is compared to that achieved with a traditional CPU implementation. The resulting GPU filter is faster with the same performance as the CPU filter for many particles.

Nyckelord
Keywords

Parallel programming, Monte Carlo methods, Estimation, Particle filtering, Graphics processing unit