EURASIP Journal on Advances in Signal Processing
a SpringerOpen Journal

RESEARCH Open Access

Fast, parallel implementation of particle filtering on the GPU architecture

Anna Gelencsér-Horváth*, Gábor János Tornai, András Horváth and György Cserey

Abstract

In this paper, we introduce a modified cellular particle filter (CPF) which we mapped on a graphics processing unit (GPU) architecture. We developed this filter adaptation using a state-of-the art CPF technique. Mapping this filter realization on a highly parallel architecture entailed a shift in the logical representation of the particles. In this process, the original two-dimensional organization is reordered as a one-dimensional ring topology. We proposed a proof-of-concept measurement on two models with an NVIDIA Fermi architecture GPU. This design achieved a 411-μs kernel time per state and a 77-ms global running time for all states for 16,384 particles with a 256 neighbourhood size on a sequence of 24 states for a bearing-only tracking model. For a commonly used benchmark model at the same configuration, we achieved a 266-μs kernel time per state and a 124-ms global running time for all 100 states. Kernel time includes random number generation on the GPU with *curand*. These results attest to the effective and fast use of the particle filter in high-dimensional, real-time applications.

Introduction

In applications in the field of image processing [1,2], navigation [3], and financial mathematics [4,5] we deal with non-linear state-space models subject to additive noise which is not restricted to Gaussian noise. Even if each state only depends on the previous state (i.e. the sequence follows the Markov dynamics [6]), a Kalman filter [7] is suboptimal for the state estimation due to non-linearity and non-Gaussian noise. Furthermore, an analytic solution is often not available. In contrast, sequential Monte Carlo methods (SMCM) offer a probabilistic framework that is suited to non-linear and non-Gaussian state-space models. In our work, we focus on a particle filter (PF) [8] which is both part of the SMCM algorithm family and can be considered an extension of a Kalman filter. Our main aim is to introduce a fast and reliable PF on a GPU.

We restrict ourselves to particle filters which use sequential importance resampling (SIR) [9]. The PF algorithm (as described in the next section in detail) has a high running time due to the resampling step - according to the *complete* cumulative distribution; therefore, an adequate parallel implementation would fetch a remarkable speed-up. The classical resampling algorithm of this process needs N processors to reduce its computational need from $O(N)$ to $O(\log(N))$, and so, this particle filter was considered unsuitable for parallelism. It should be noted that this statement holds as long as the the complete cumulative distribution is required for any particle.

Whilst the price of a device is relatively low, graphics processing units (GPUs) have a high computational efficiency. Therefore, GPUs represent an attractive implementation platform. Since GPUs are spreading fast, thanks to the game industry, and developing rapidly, computational effort is still increasing by leaps and bounds.

There have been some former implementations to parallel architectures [10-19]. In [11], an implementation strategy is proposed which is parallel; however, it cannot maintain the local connections of the particles. The particles are split into smaller groups (around 100 particles) which perform operations independently. The information exchange among the particle groups is occasional; share ratio is suggested at around 25%. Researchers admit that the reduced flow of information of the distributed particle filter degrades the quality of estimation compared to the original algorithm which resamples according to the complete cumulative distribution.

Besides continuous information sharing, random number generation has a significant effect on the filter result.

*Correspondence: horvath.anna@itk.ppke.hu
Faculty of Information Technology, Pázmány Péter Catholic University, Práter str. 50/a, Budapest H-1083, Hungary

The simple solution is creating random numbers on the CPU using any of the many implemented reliable random libraries proposed in [10]. However, these researchers are aware of the great disadvantage of this technique, mainly the huge delay raised by data transfer. Hence, they prefer a sufficient GPU random sequence generator instead.

In the resampling step, each particle requires information from all other particles. This adds a high computational delay. Resampling is based on the relative importance of the particles, but the technique is not restricted to uniform sweepstake over the weights (systematic resampling).

Metropolis resampler [20] in each resampling step iteratively selects B times a candidate according to a given rule for each particle. Since the aforementioned rule is based on pairwise operations, the efficient mapping on many-core architecture is realizable as follows.

For each $p \in 1,\ldots,N$ particle in each $i \in 1,\ldots,B$ iteration, two main parameters, a_p^i and s_p^i, are used, where a_p^i stands for the actual particle and s_p^i stands for the selected particle; a_p^i is initialized with particle p. With uniform distribution, we draw a u_p^i random number on $[0,1)$ and s_p^i particle from the complete particle set. If the ratio of the weights $\frac{w_{s_p^i}}{w_{a_p^i}}$ is over u_p^i, the selected particle is indicated as actual. This pairwise operation can be performed independently; therefore, efficient implementation is possible on parallel architecture.

Resampling can also be accelerated if the number of particles is decreased. However, there is a trade-off between particle number and estimation accuracy. The spreading-narrowing technique in [12] proposes a solution. A tolerable N number of *basis particles* generates an $N \times P$ large set by propagating each particle based on the system transition model for a sequence of P states. Each P_i subset then delivers a single particle based on a *local particle selection* process. It either uses maximizing importance selection (MIS), taking the highest weighted particle, or it uses systematic resampling (SR) on the weights. SR has a lower complexity than a global resampling on an $N \times P$ size set as P takes values from{10, 20, 50, 100, 200, 500} based on the current application, and for each P_i set, it can be performed parallel to each other. While MIS has an even lower complexity, it is more sensible to the noise introduced by the artificial propagative spread of the particles. For the measurements, they used a bearing-only tracking (BOT) model with 25 time steps; therefore, we can make direct comparison for the estimation error. For [12], the position error is in the range of 0.06245 to 0.06226, which is slightly lower than our error, but still the same range. Execution time, which is the sum of sampling, weight normalization and resampling times in [12], and total runtime in our work (including memory transfers, file I/O, etc.), shall be compared to our proposed algorithm with regard to the different devices, which still indicates that our technique is faster (see Table 1).

We investigated [13-15] from CUDA ZONE, which also addresses particle filtering *on GPU or parallel and GPU*. In [15], only the weight calculation is performed on the GPU as the focus of this paper is not particle filtering. However, this work is slightly out of our scope as the aim was the fast estimation of face tracking with PF; the contribution of parallelism on the GPU is relevant in the total speed-up. Three different case studies are presented for Monte Carlo methods in [13]. They found out that global resampling has a significant influence on the runtime, and they achieved 10 to 37 times speed-up compared to a single threaded CPU implementation. The measurements were made with the use of a factor stochastic volatility model; therefore, direct comparison to our benchmark model is troublesome. However the computations run on the GPU, the resampling - unlike in our proposed work - is not parallel but sequential. In [14], resampling is performed with a technique based on using an offline-created and offline-uploaded texture of uniformly distributed random numbers. The focus of [14] was single- and multiple-object tracking, using skin detection and spreading the region of interest; therefore, the lack of common model encumbers the direct comparison of the result to our proposed method.

However, in [16], the GPU device is different (making the exact time comparison difficult), and measurements were made using the BOT model. The proposed particle filter method is parallel, still the random numbers are generated on the CPU, and there is no information share among local processes (e.g. in resampling). The position errors are in the same range and almost identical. Execution time in [16] is expressed as the sum of sampling, weighting, weight normalization and resampling times, whilst our presented execution time includes all operations (file I/O, memory transfers, etc). A real-world problem is presented with a particle filtering method in [17]. PF is implemented on the GPU with a distributed resampling. The work is based on *sub-filters* which have a limited information share among themselves. Communication can be represented as a graph where sub-filters correspond to nodes and edges are defined arbitrarily (as an attribute). Before resampling, a particle exchange step is performed among neighbouring graph nodes. However, this approach is not completely local; the amount of the exchanged particles is relatively small. Therefore, information flow is not as complete as in the case of the sequential (cumulative distribution-based original algorithm) or even as in our proposed method. Finally, we would like to mention [18] and [19] which both present parallel but non-GPU particle filters. In [18], the particles are split to subsets. Similar to [17], each subset performs the sub-steps of PF independently; however, there is no

Table 1 Highlighted related works

References	GPU type	GPU computes	Model[a]	Number of SMs on GPU	Number of cores	GPU clock	Time included	Runtime data
[12]	GTX 280	All; spreading-narrowing technique	BOT model, 25 time steps	30	240	1.3 GHz	Sampling + weight normalization + resampling	1,050 particles 79.4 ms, position error 0.06245; 2,000 particles 124.8 ms, position error 0.06226
[13]	GTX 280	All; sequential resampling	A factor stochastic volatility model	30	240	1.3 GHz	SMC algorithm: no further details	8,192 particles 82 ms; 16,382 particles 144 ms; 65,536 particles 465 ms
[14]	8800 GTS	All; resampling uses offline-initiated texture	Skin detection + spreading region of interest	12	96	1.2 GHz	Object tracking time; no further details	1.44 - 13.55 speed-up in fps[b] compared to CPU. Best 90 fps for multiple- and 225 fps for single-object tracking.
[15]	8800 GTX	Weight calculation	Face tracking model	16	128	1.35 GHz	Face tracking algorithm time; no further details	No execution time measurements for particle filter
[16]	9400M	All; random numbers from CPU	BOT model, 25 time steps	2	16	450 MHz	Sampling + weighting + weight normalization + resampling	For 2,048 particles: best time 168.3 ms, position 0.078 - 0.083; for 4,096 particles: best time 168.0 ms, position 0.077 - 0.081
[17]	GTX 580	All; distributed resampling	Dynamic equations to model a robotic arm	16	512	2 GHz	Sum of kernels: random number generation + sampling + local sort + global estimate + exchange + resampling	64,000 particles 0.3 ms
Proposed	GTX 550 Ti	All; all parallel	BOT model, 24 time steps	4	192	1.8 GHz	All operations (including memory transfers, PF steps and file I/O)	2,048 particles 77 ms with 0.09 position error; 16,384 particles 77 ms 0.07 position error

Summary of related works including the following parameters: used model, outline of the technique and the GPU if measurements were made on it. Direct comparison is often hardly feasible due to the differences of the mentioned parameters. [a]As given in the references; [b]frames per second.

information share among the subsets. Central estimation is calculated using the results of subsets. In [19], three different techniques are presented, and locally distributed particle filter is considered as giving the best speed-up and estimation. Also, this is the closest from the three presented methods to our approach; however, operations are performed without any information share and only simulation results are given for a discrete time non-linear dynamic model of nearly constant turn. For the summary of related work, please see Table 1 where we highlighted the most relevant GPU-related works. The table also reveals the difficulty of direct comparison of the results due to the different models, data and GPU devices.

We can say that resampling is a key point in SIR particle filtering. Approaches to deal with SIR resampling try to optimize the speed quality trade-off for the given set-up. Cellular particle filter(CPF) [21] introduces a third approach for the resampling problem by changing the logic representation of PF to a two-dimensional (2D), locally connected grid inspired by cellular neural network (CNN) architecture [22]. Each element in the grid is connected to each of its eight neighbours enabling rapid local information flow. The critical resampling step can then be performed on a subset in an r radius neighbourhood.

Our proposed algorithm is based on cellular particle filter [21], using the idea of local neighbourhoods. Due to the CNN-type representation and the decreased

dimension of resampling sets, CPF offers a solution for the problem of reduced local information change, which is stated in [11]. However, this representation is not optimal for a GPU architecture. Hence, we made some further modifications to achieve an efficient implementation which exploits the characteristics of GPUs. Besides, one of our principles is to generate random sequences on the GPU since NVIDIA SDK Mersenne Twister proved to be insufficient at low numbers. Therefore, we explored possible solutions and finally propose two different approaches for random number generation.

This paper is organized as follows: the 'Background and theory' section describes the necessary background and theory for hidden Markov models (HMMs), particle filters, especially cellular particle filter, and architectural perspectives. In the 'Our proposed method' section, we introduce our proposed method in detail. The 'Evaluation and results' section provides information about applied measurement techniques and our results. Finally, The 'Conclusions' section delivers our conclusion.

Background and theory

Hidden Markov models and particle filtering

HMMs consist of two stochastic processes. One of them is the trajectory of hidden states x_t according to $t = 0, 1, \ldots$, determined by Markov dynamics:

$$x_{t+1} = \varphi(x_t, e_1(t+1)) \tag{1}$$

The other contains observations y_t, for $t = 1, 2, \ldots$, depending only on the current hidden state plus an additive noise which is not limited to Gaussian:

$$y_t = \psi(x_t) + e_2(t) \tag{2}$$

These notable extensions transfer the resolution beyond the Kalman filter [7] to the scope of particle filtering. In case of state estimation, it is considered that φ, ψ, functions and distributions of $e_1(t)$ and $e_2(t)$ are given. For more information about hidden Markov models, see [6].

A particle filter is a tool for estimating the hidden states based on the observation. It is not an analytical calculation but the use of a set of particles at each time step that follows the model dynamics. The algorithm is built up from three main steps in each time t (i.e. *state*).

The first step is error calculation which assigns each particle a fitness value. It is performed between the current particle value and the current observation value (same for all particles at a time step) as described in Equation 3. L stands for the likelihood value, for each $i = 1, \ldots, N$ particle, and l is the density function of the noise of the hidden process ($e_1(t)$):

$$L_t^i = l\,(y_t - \psi(x)) \tag{3}$$

Each particle weight is set based on this likelihood:

$$w_t^i = L_t^i \tag{4}$$

where L_t^i is the fitness value of the ith particle, and for simplicity in resampling, each weight is normalized:

$$w_t^i = \frac{w_t^i}{\sum_{j=1}^{N} w_t^j}. \tag{5}$$

The second main step is resampling. While there are many alternations, we focus on a particle filter with sequential importance resampling [23]. We choose a new ξ' particle set from our current particles, $\xi'^i = \xi'^{\eta(U_i)}$, where $\eta(U_i)$ stands for the uniform random sweepstake, using the set of corresponding normalized particle weights w_t (see Figure 1 and Equation 6, for particles $i, j = 1, \ldots, N$):

$$P(\eta(U_i) = j) = w_t^j \tag{6}$$

The resampled set ξ_t' is used for the current estimation (e.g. taking the mean). The last main step of the loop is the iteration, which is identical to sampling at the beginning of the loop. In this last step, we use the model to generate the next time step's initial particle set using the model:

$$\xi_{t+1}^i = \varphi(\xi_t^i, e_1(t+1)) \tag{7}$$

where ξ_t^i are the resampled particles, $i = 1, \ldots, N$. Particle filtering technique has been used since 1962 [24], and SIR particle filter has been used since 1993 [23]. Still, the proof of convergence was published only 18 years later [25]. More information about particle filters can be found in [23,26]. Henceforward, 'original algorithm' stands for the algorithm described in this section.

Cellular particle filter

In the resampling step, for each retake, we have to use the whole particle set. This is highly time-consuming and for a long time was considered not parallelizable. Cellular particle filter [21] offers a solution for this problem, and in contrast to other distributed particle filters [11], it maintains local connectivity, which allows for each particle to access the information of its neighbours in each time t. This ensures the same or, at some parameters, even better quality of approximation. To provide theoretical proofs for our concept is beyond this articles' scope, but see [21] for some experimental validation.

The main idea lies in the logical representation. The set of particles are organized in a CNN-inspired architecture,

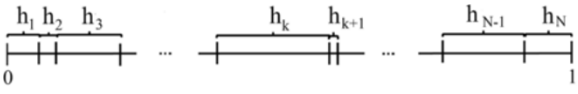

Figure 1 Resampling. Split [0,1] to subintervals for resampling. Each interval has the width of the corresponding normalized weight w_i where $i \in 1 \ldots N$. We generate uniformly distributed random numbers for resampling, which means the bigger w_i value particle i has, the more likely it is to be picked.

namely, a locally connected two-dimensional grid with uniform elements where each element is only connected to its eight neighbours. Based on the connections, we can define a neighbourhood for cell i (ie. cell $C_{i,j}$) with radius r: $C_{k,l} \in N_i$ if $k \in [i-r, i+r]$ and $l \in [j-r, j+r]$ (see Figure 2).

We can retrace the original algorithm if we set the neighbourhood size for each particle to fit the dimension of the grid. However, if we set a smaller r radius, it defines a locally connected N_i neighbourhood for each i particle:

$$W_t^i = \sum_{j \in N_i} L_t^j \tag{8}$$

Using the sum of weights W_t^i of the neighbourhood, in this case, the weights are set to

$$w_t^j(i) = \frac{L_t^j}{W_t^i} \tag{9}$$

where $j \in N_i$.

Now, the resampling step can be performed for each i particle simultaneously within the local N_i neighbourhood according to the local distribution of the weights. Fetching the weights is realized by local communication on the physical device, therefore, it is fast. The random take on all subsets around each i particle produces N resampled particles, respectively. Hence, the time-consuming part is parallelized, and the required computational effort is essentially independent of the number of particles. This method might seem to be similar to distributed particle filters [11], but there are no communication limits among the subsets in any time states.

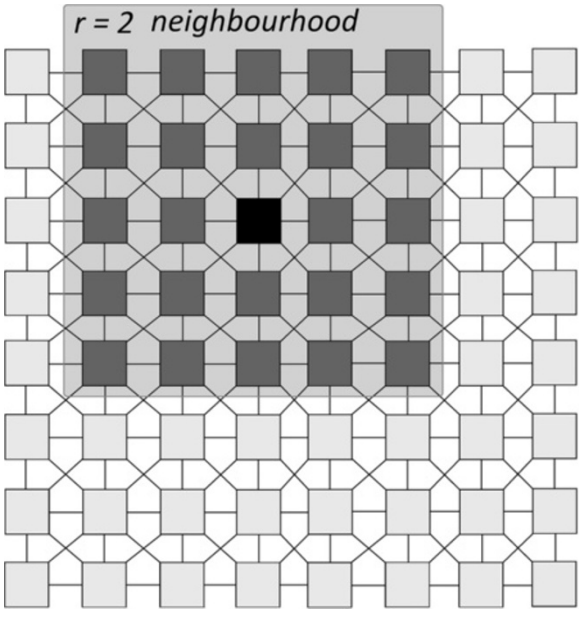

Figure 2 CNN architecture. CNN processor array architecture: two-dimensional fully connected grids. The light gray background highlights the $r = 2$ size neighbourhood around the black cell.

CPF is suited to GPU architecture due to its parallel, locally connected nature. Our aim is to ensure efficient computation and therefore to exploit the properties of the GPU in our adaptation.

GPU details

Our proposed mapping is based on the GPU features summarized in this section. We used NVIDIA CUDA (see [27]) for notations and details. Figure 3 shows the basic architecture of the GPU considered mainly from the view of mapping CPF to this architecture.

In the logical sense, the kernel function is the function executed on the device. It is executed simultaneously by threads. Threads are organized into blocks (typically 32 to 512 in each, based on the current task), in a one-, two-, or three-dimensional array. Blocks are organized in a grid in a one- or two-dimensional array. The number of threads per block and block per grid is called execution configuration.

In the physical sense, the device is built up from streaming multiprocessors (SMP). Each SMP consists of an SD RAM, a number of cuda cores and a scheduling unit. The SD RAM is an on-chip memory, with a few tens of clock cycle delays, its size is 64 kB, and it is divided to L1 cache and shared memory. The GPU has an off-chip global memory to be accessed by each SMP. Its size is usually around 1 to 4 GB, depending on the type of the particular device. Its delay is 400 to 600 clock cycles. Additionally, there are two other types of memory spaces that both reside off-chip and are cached on-chip. The first one is texture memory, and the second one is constant memory. The latter's size is 64 kB.

Blocks are mapped to SMPs. Shared memory of block B_i can only be accessed by the threads which reside in B_i. The communication and data share among the blocks are performed through the global memory. A fixed collection of threads is called warp. Currently, the number of threads in a warp (warp size) is 32, which is physically executed simultaneously. Besides proper memory usage, warp conflict avoidance is essential [27]. The vendor suggests block sizes multiple of the warp size to achieve the most efficient computation. However, in extreme situations, optimal block size can differ from the advised values [28]. If some threads in the warp choose different branches of operation based on the processed data, the threads within the warp may diverge. This is called warp desynchronization and results in the serialization of some threads which adds runtime delay. As a rule of thumb, the block size should be a multiple of 32 and each multiprocessor should execute at least six warps at the same time because the pipeline is six levels deep; therefore, $8 \times 32 = 256$ may be an ideal thread number.

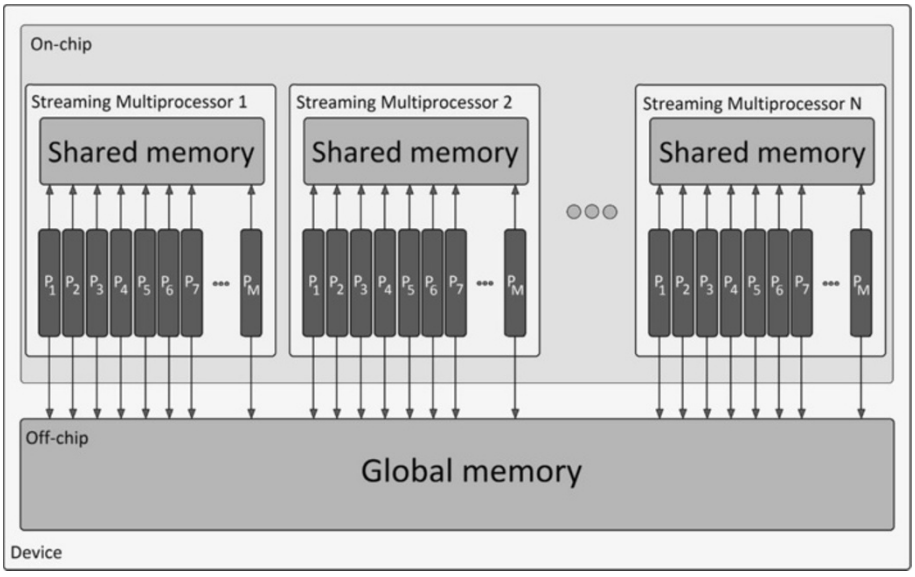

Figure 3 GPU architecture. Simplified architecture of the GPU. Streaming multiprocessors (SMP) contain cuda cores (P_i), an on-chip (shared), fast memory, scheduling unit and special function unit. Besides, all cuda cores in all SMPs have access to the large and slow off-chip memory (global memory).

To achieve a high throughput, on-chip memory (shared memory) should be used if threads require frequent data access. Therefore, the main computational tasks are performed block-wise, in the shared memory of the blocks, and only necessary global synchronization is performed through global memory. Although the two-dimensional texture and surface memory would also be feasible, shared memory throughput can be optimized for a one-dimensional array type representation; thus, structural aspects in CPF algorithm were reconsidered.

In the shared memory, we use arrays with the size 512 for the particles, the error terms, the normalizing sums and the uniform random numbers. In our case study, when taking the highest neighbourhood size and at a single precision 10,250 bytes are occupied. Each shared memory can access a 48-kB memory/multiprocessor at compute capability 2.x [27]. Global memory in recent GPUs is at least 1 GB which restricts the number of states in the observation and estimation, but both memory access is high enough for our computations.

Our proposed method

Random number generation

The SIR in each t time step requires the same amount of random numbers as the number of particles. We intend to generate these random sequences on the GPU device instead of the CPU to spare repetitive data transfer between the main memory of the system and the global memory of the device as recognized in [10]. The distribution of the random numbers in the resampling is critical on the quality of the estimation. If it is not uniform, though the drawing of particles should depend on the weights exclusively, then it would be biased.

Recently GPU random number generation for various purposes has become widely investigated and well tested (see [29-31]). NVIDIA provides two solutions for random number generation. The first option was the Mersenne Twister in the SDK. Unfortunately, we observed that the generated distribution is inappropriate for a small set (hundreds or thousands) of numbers and is primarily admissible for around two million numbers and above. We made delicate modifications to get admissible distribution (see Appendix for details). The second option was *curand*, which proved to be fast and appropriate.

GPU CPF algorithm

As described at CPF subsection, particles are represented as arranged on a 2D grid with local connections and have a given neighbourhood radius. Although GPUs have a different kind of architectural organization compared to the CNN structure, it is possible to map the 2D topology to GPUs' memory hierarchy. However, two modifications were made in the particle topology to fit better to the architectural details of GPUs, namely, instead of 2D, a 1D topology was applied and the neighbourhood was considered circular (see Figure 4) in one direction. There were two reasons for these decisions. First, the proportional size of the neighbourhood is smaller in the 1D case. Second, using only one side of the neighbourhood, the coalesced memory access is ensured.

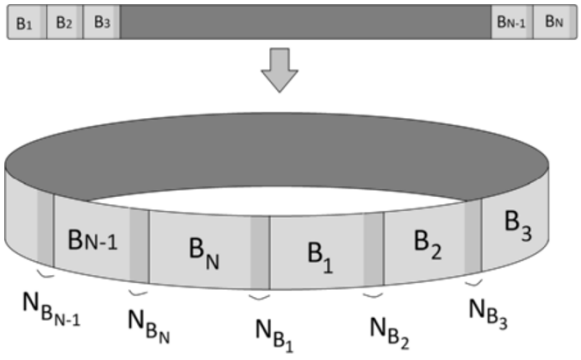

Figure 4 Ring type topology. Restructuring linear representation of N blocks to a ring type topology. B_i stands for the ith block, and N_{B_i} for the corresponding neighbourhood from the previous block, $i \in 1, \ldots, N$.

The local connectivity is preserved by choosing each shared memory array size higher than the thread number exactly with the size of the required neighbourhood. Each thread with index i can obtain its k neighbours at indexes $i-1, i-2, \ldots, i-k$.

In the following, we specify some implementation details. Although the operations are mainly performed in the shared memory because of the synchronization and CPU-GPU data transfer, the following variables are global memory arrays: the observation sequence (Y), the state estimation (X) and the set of particles $x_{\text{particles}}$. The number of particles is denoted by N. Y is naturally given, X is empty, and $x_{particles}$ is initialized with N samples of the same distribution as n_t described in Equation 10. The number of threads in a block was set to 256. The size of the neighbourhood is r, meaning each particle is connected to exactly $r+1$ particle (every particle is connected to itself).

In each block, two shared memory arrays of size $256+r$ are created for particle states x_{shared} and fitness values L_{shared}; additionally, two arrays whose size equal the number of threads in a block are allocated for uniform pseudo-random numbers U_{shared} and normalizing weights w_{shared}.

In each time step, we copy the particle values from the global memory to the shared memory by overlapping split (see Figure 5 for illustration). We load 256 values respectively to each shared memory to the particle's array (x_{shared}) but sparing its first r elements. These positions are filled with the r neighbours in the global memory of the first element in x_{shared}. For the very first element, we use a circular approach by taking the values from the end of the global memory array. There are three kernel calls for each estimated values to provide full synchronization.

The main kernel performs the following operations in each time step t (also see Figure 6 with the same numbering), where global and local thread IDs are defined as follows: $i_{\text{gl}} = \text{blockDim.x} \times \text{blockSize.x} + \text{threadId.x}$ and $i_{\text{loc}} = \text{threadId.x}$, where *threadId.x* is the thread index in the thread block, *blockSize.x* is the number of thread per each block, and *blockDim.x* is the index of the block. For more information about the terminology, see [27]:

I Initialization:

2. $x_{\text{shared}}[\, i_{\text{loc}} + r] \leftarrow x_{\text{particles}}[\, i_{\text{gl}}]$.
2. If $i_{\text{loc}} < r$, then $x_{\text{shared}}[\, i_{\text{loc}}] \leftarrow x_{\text{particles}}[\, i_{\text{gl}} - r + i_{\text{loc}}]$.
2. If $i_{\text{gl}} < r$, then $x_{\text{shared}}[\, i_{\text{loc}}] \leftarrow x_{\text{particles}}[\, N - r + i_{\text{gl}}]$.

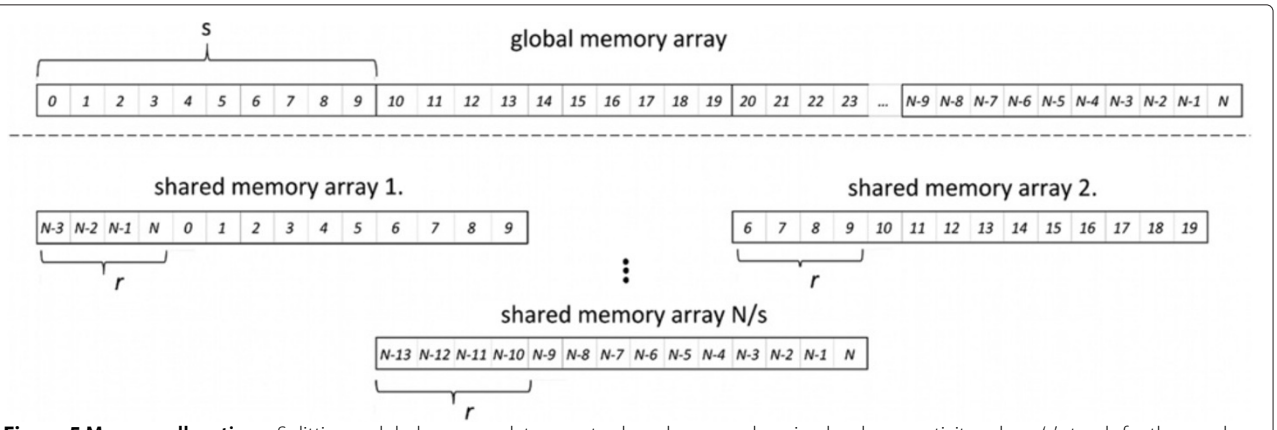

Figure 5 Memory allocations. Splitting a global memory data array to shared memory keeping local connectivity, where 's' stands for the number of threads in each block and r stands for the size of neighbourhood. To fit the architectural details of the GPU and reduce computational time, we applied 1D topology instead of the proposed 2D grid with one-sided neighbourhood.

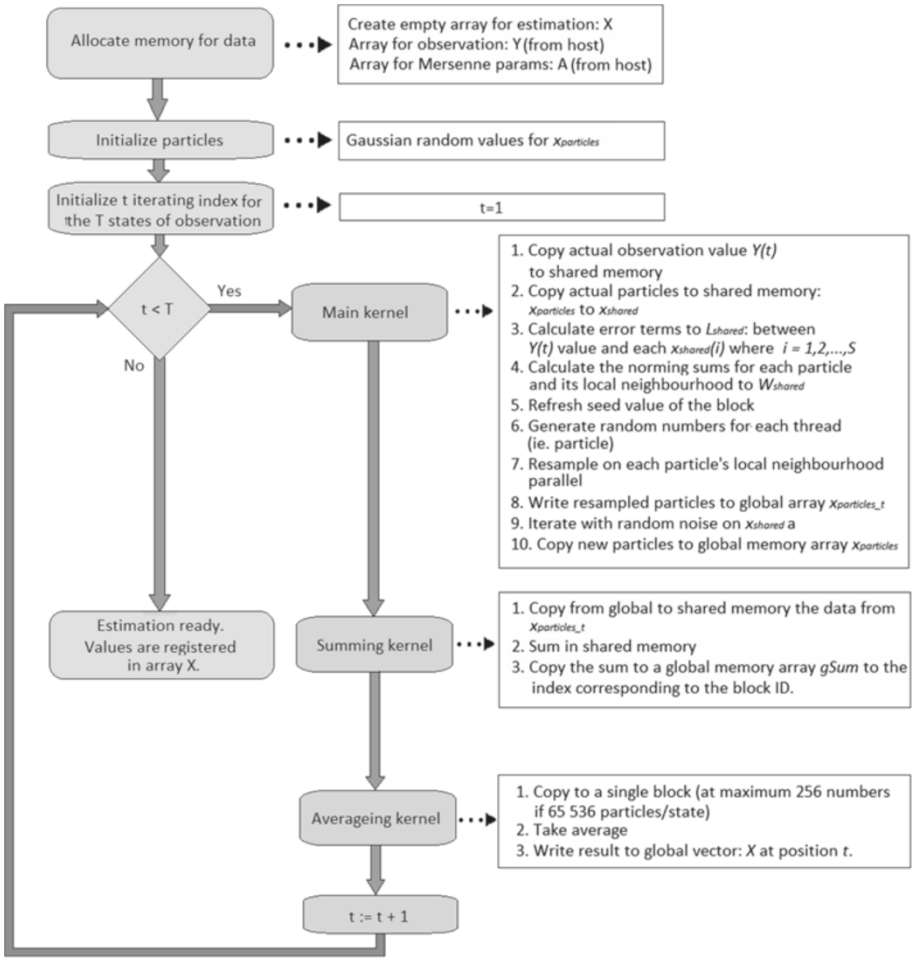

Figure 6 Overview. Flowchart of our implementation on the GPU device. S denotes the number of threads in each block, $x_{\text{particlec}_t}$ denotes the resampled particle set in t state. The observation sequence consists about T states.

II Error calculation:

3. $L_{\text{shared}}[\, i_{\text{loc}} + r] \leftarrow l(Y[\, t] - x_{\text{shared}}[\, i_{\text{loc}} + r]\,)$.
3. If $i_{\text{loc}} < r$, then $L_{\text{shared}}[\, i_{\text{loc}}] \leftarrow l(Y[\, t] - x^s[\, i_{\text{loc}}]\,)$.
4. $w_{\text{shared}}[\, i_{\text{loc}}] \leftarrow L_{\text{shared}}[\, i_{\text{loc}}] + \cdots + L_{\text{shared}}[\, i_{\text{loc}} + r]$ get normalization sums for each particle.

III Resampling:

5. If $i_{\text{loc}} == 0$, refresh seed value of the block.
6. Fill U_{shared} with uniform random numbers.
7. Iteratively sum $L_{\text{shared}}[\, j_{\text{loc}}]\,/w_{\text{shared}}[\, j_{\text{loc}}]$ where $j_{\text{loc}} = i_{\text{loc}}, i_{\text{loc}} - 1, \cdots, i_{\text{loc}} - r$. Stop if adding an $L_{\text{shared}}[\, k_{\text{loc}}]\,/w_{\text{shared}}[\, k_{\text{loc}}]$ term affects the sum to exceed $U_{\text{shared}}[\, i_{\text{loc}}]$ for the first time.
8. From the neighbourhood of i_{loc}, the corresponding k particle is selected to $x_{\text{particles}_t}[\, i_{\text{loc}}]$. Estimation is calculated from these particle values.

IV Iteration on particles (generating samples for next state):

9. Fill U_{shared} with normally distributed random numbers.
9. $x_{\text{shared}}[\, r + i_{\text{loc}}] \leftarrow \varphi(x_{\text{shared}}[\, r + i_{\text{loc}}]\,, n_t)$.
10. $x_{\text{particles}}[\, i_{\text{gl}}] \leftarrow x_{\text{shared}}[\, i_{\text{loc}} + r]$.

The estimation is performed by two kernel calls. The first kernel calculates the sum for each shared memory $x_{\text{particles}_t}$ arrays to a global memory array *gSum*. The second kernel takes the average value of gSum with respect to the number of particles.

Besides, the following optimization techniques were used to achieve optimal efficiency on the GPU: (1) random number generation, resampling and average calculation are performed only on the relevant part of the

shared memory to spare time; (2) the number of *if* statements is minimized as possible and are transformed to ternary expressions; and (3) the shared memory arrays are reused; therefore, even some parameter passes can be spared.

Evaluation and results

Model

The implemented algorithm was tested on two different widely used models. The first was the following benchmark model [8,23,32,33]:

$$x_{t+1} = \frac{x_t}{2} + \frac{25x_t}{1+x_t^2} + 8\cos(1.2t) + n_t \tag{10}$$

$$y_t = \frac{x_t^2}{20} + u_t \tag{11}$$

This model is non-autonomous, non-linear and has a continuous state space; thus, linear tools for state estimation are not applicable. The state of the system is x_t, and the observation is y_t; n_t and u_t are IID Gaussian sequences, $n_t \sim N(0, 10)$ and $u_t \sim N(0, 1)$.

The second model was a bearings-only tracking (BOT) model originally presented in [23] and also analyzed in [16,17]. For the illustration about the trajectories of each model, see Figures 7 and 8.

Measurements

There are two aspects of the measurements, namely, the average quality and the required time for one estimation. These two quantities were monitored with different configurations (number of particles and radius of neighbourhood) of the filter. The number of particles were swept through the following values: 2,048, 4,096, 8,192, and 16,384, while the radius of neighbourhood took the following values: 32, 64, 128, and 256. Consequently, 24 pairs of these are composed; in our terminology, these are called configurations (i.e. *N*,*r* pairs). For the first model, the input observation trajectories (y_t) during the tests were exactly the same as the ones used in [21] to ensure a fair comparison. For the BOT model, we generated 100 trajectories over 24 time steps based on the given state transition equations in [23] and, respectively, the observations.

Measurements were done on a PC with Intel i5-660 (3.33 GHz, 4-MB cache) 4 CPU with 4-GB system memory running Ubuntu Linux 11.04 with kernel version 2.6.38-15 (amd64). We used an NVIDIA GeForce GTX 550 Ti GPU with 1-GB GDDR memory with CUDA toolkit 4.1 with 295.49 driver version. The following nvcc compiler options were used to drive the GPU binary code generation: -arch=sm_20;-use_fast_math. We also made some measurements with -arch=sm_13. The host c code was compiled with gcc 4.5; the compiler flag was -O2. GPU kernel running times were measured with the official profiler provided by the toolkit, and the global times were measured by the OS's own timer. The kernel time measurements include the particle filtering kernel of a single time step; the global times include all operations during the execution for all states (file I/O, memory allocations, computational operations, etc.).

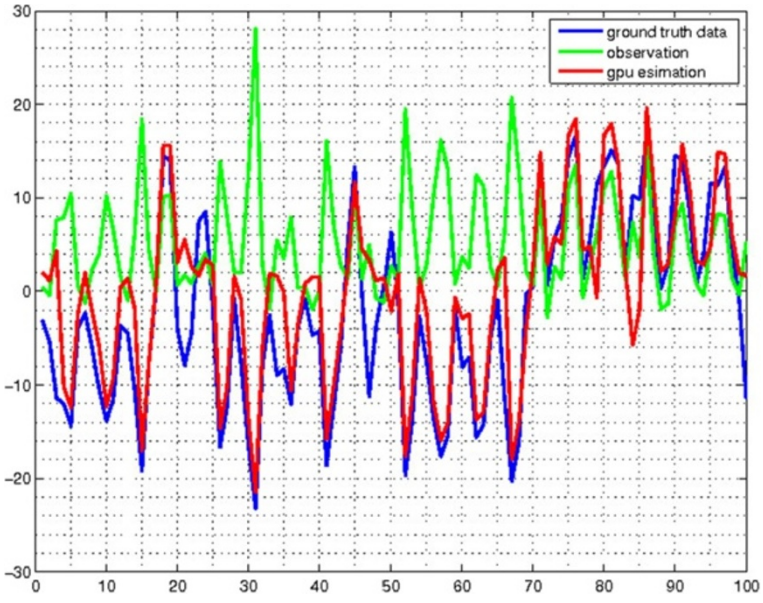

Figure 7 Trajectory for the first model. The hidden states are marked with blue and the observation values of the states are marked with green. The state estimation of our GPU CPF is marked with red.

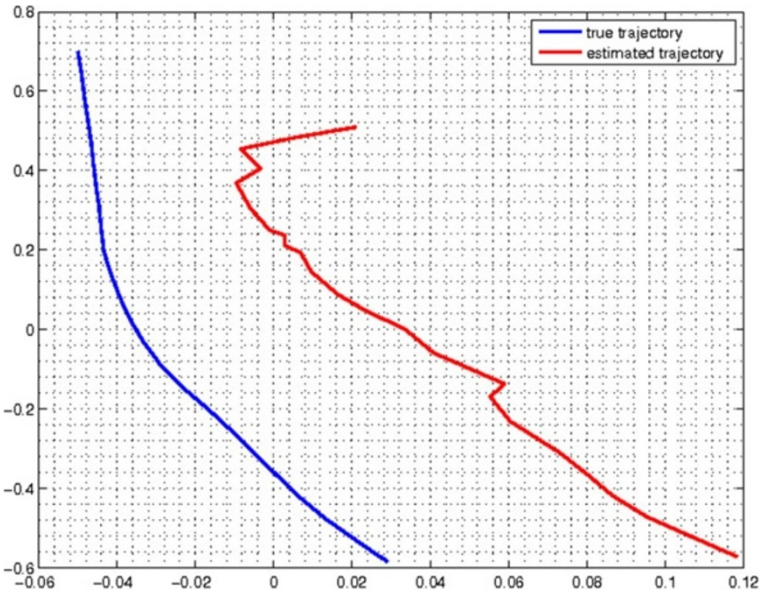

Figure 8 Trajectory for the BOT model. The positions ($x - y$) are the hidden states (blue), the estimation from the GPU CPF is red. The mean of the position error is about 0.08.

The quality of estimation was measured by the mean square error (MSE). In the case as the first model, 1,000 estimations for each configuration of 100 time step long HMMs were made. In the case of the BOT model, the generated 100 trajectories were estimated with each configuration.

Estimation quality

The quality of estimation for the first model was measured by the MSE between each hidden and estimated trajectories, for the BOT model by the position error (ie. Euclidean distance). We used 1,000 and 100 different state sequences (i.e. observation sequences) for each configuration in the first model and the BOT model, respectively.

Figure 9 presents the quality of estimation for the first model; Figure 10, for the BOT model, namely the measurement error with respect to the measurement time. Each point represents a configuration since its x and y coordinate values are the mean of 1,000 and 100 executions for the two models, respectively. For the first model, it can be seen that using more than 4,096 particles slightly improves the quality of estimation.

However, for the BOT model estimations, where the particle number is more or equal to 2,048 (alike [16,17]), provide a fair result. The position error is in the same range as in [17] and our proposed method. The results suggest that the proportion of the neighbourhood size to the particle number realizes an information sharing ratio among the particles. This can be seen in Figure 10: the optimal ratio for the configuration is when the position error is minimal, typically marked with squares except 2,048 particles when marked with triangle.

Time

Figure 11 presents the total runtime of the kernels in the first model. The blue lines represent the running times with compiler option *-arch=sm_13*; the red lines, with *-arch=sm_20;fast_math;*. The first compilation setting will be referred to as old target code; the latter, as new target code. It can be seen that for the neighbourhood size below 64, the old target code performs 20% faster than the new. With the new target code, we can achieve a 40% to 45% improvement in execution time if the old target code is considered as 100%.

Due to the logic of physical mapping of blocks to multiprocessors, the GPU is under-utilized for particle numbers under 2,048. Above this particle number, the scaling of the execution time is nicely illustrated in Figure 11.

For various neighbourhood sizes, we can say that the required time is proportionally increasing to the number of R. This phenomenon is due to the resampling step as it examines the candidates sequentially for resampling. Even if the proper particle is found, the loop does not terminate until the current particle is compared to all of its neighbours to avoid warp desynchronization.

The execution time of particle filter (including file I/O, initialization of random number generation, memory transfer between CPU and GPU, etc.) is 77 ms for the BOT model. However, we used the same model as in [16]; an exact comparison is hardly available due to the differences of the GPUs and it is not specified what their

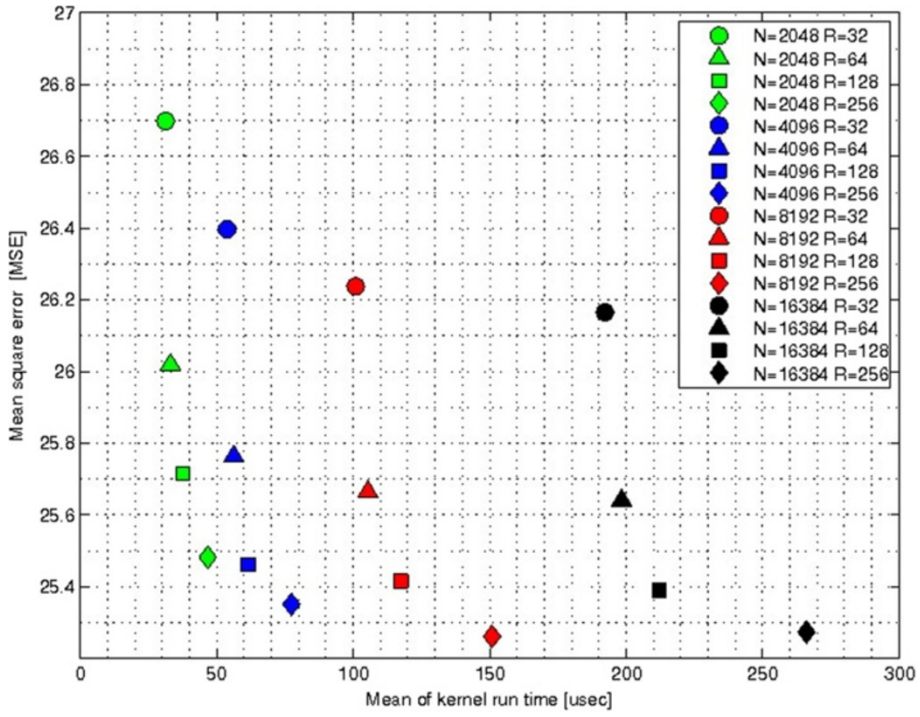

Figure 9 Estimation quality for the first model. This figure shows the mean square error of the estimation as a function of kernel times. It can be seen that at a given particle number with the increase of the neighbourhood size, the estimation quality improves simultaneously.

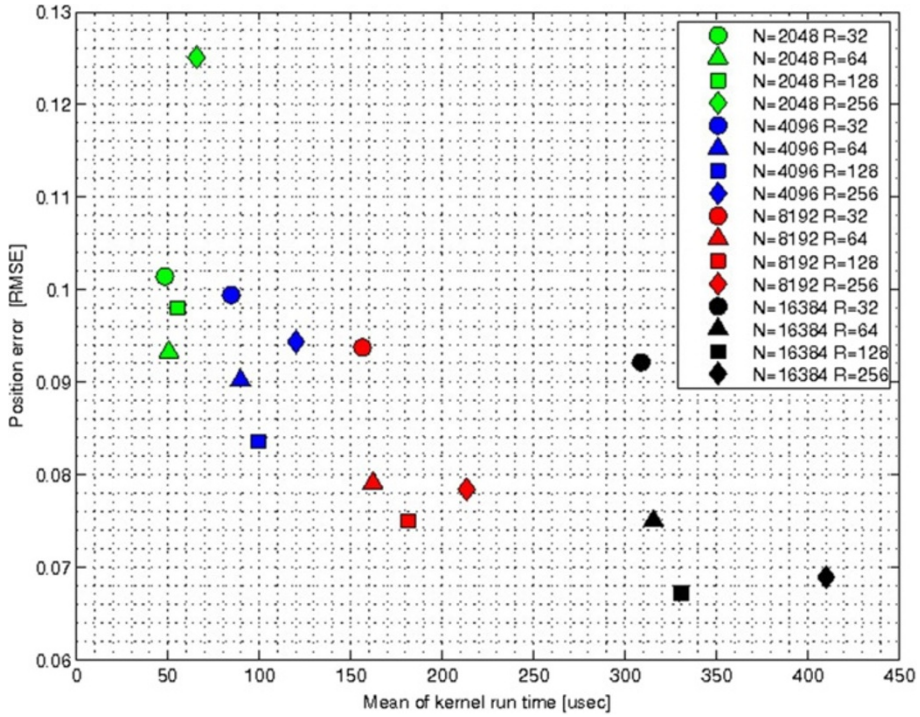

Figure 10 Estimation quality for the BOT model. This figure shows (the root mean of) the position error of the estimation as a function of the kernel times. For this model, it can be seen that there is an optimal information share ratio where the position error is the lowest for a given particle number at a neighbourhood size.

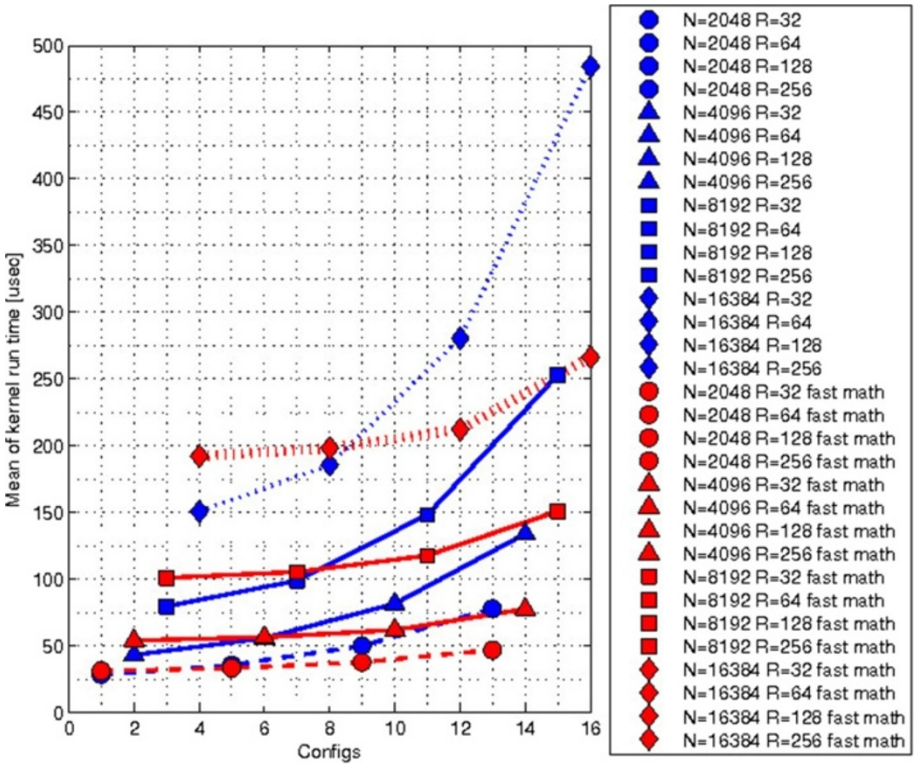

Figure 11 Kernel runtimes with different nvcc flags. This figure shows the difference between the total running times for the different compiler options for the first model. The use of fast math and sm_20 has a significant effect on the kernel times.

time measurements include. For the first model, see total execution times with and without host code optimization (made by compiler) in Figure 12.

Discussion

A key point of our proposed algorithm is the local resampling technique which has a high influence on the estimation quality. This key point can be viewed as diffusion of information inasmuch as every particle with relatively high likelihood attracts all particles which has this likely particle in its neighbourhood. In this way, the other particles not having this very particle in their neighbourhood are not affected in this state estimation time step. Although it is not the traditional full resampling, it enables the algorithm to be sufficient even at high-uncertainty dynamic models. The BOT model is not a highly uncertain model as sharp changes are unlikely, but in the one dimension benchmark model (as you can see in Figure 7), rapid and significant changes are typical. Cellular resampling preserves the diversity of the particles to avoid quality loss. The estimation error for a given particle number changes within a narrow range around the optimal, depending on the neighbourhood size. This modulation is not in direct or inverse ratio to the number of used neighbours but follows a descending and then increasing characteristic (like the shape of letter 'U', see Figure 10). This indicates that for a given model, at any particle number, there exists an optimal share ratio range among particles to achieve the lowest error. In our proposed method, the information sharing ratio is tunable and may be modulated adaptively; therefore, it broadens the range of options than using a predefined information share value. For further details and reasoning, see [21].

To ensure the local diffuse information share, we used shared memory arrays. Due to the high number of writing and reading data arrays (particle samples, weights, likelihoods, resampled values etc), if it were performed in global memory alone, the performance would be worse. However, the use of global memory cannot be totally evaded as synchronization and regular information sharing among blocks are essential. Without this synchronization, a similar information loss and quality degradation would appear as in the case of the distributed particle filter presented in [11] though this synchronization is a time quality trade-off. However, using constant memory (which is cached) would expose an attainable solution; only the observation values could be stored in the constant memory since all the other values are generated during kernel execution. Additionally, it would not improve the performance as reading from the constant memory space requires a fetch from the off-chip memory to cache the value of the current observation (like the current fetch

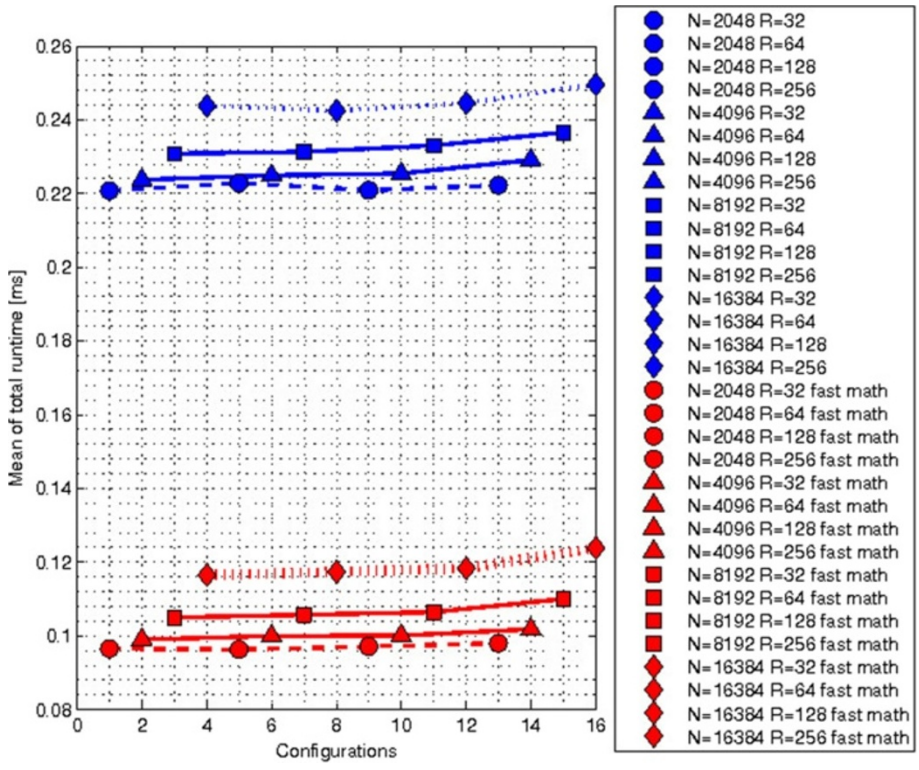

Figure 12 Global runtimes for the first model with and without O2. This figure shows the speed-up of the host code optimization compiler option for the first model.

from the global memory to the shared memory), and the access time of the constant cache is similar to the access time of the shared memory.

Two different models were used in this work. The first one is a synthetic benchmark model. It does not model any physical system of practical interest. It is just a widely used highly non-linear model since both the observation and the state transition is non-linear, unlike the other model (BOT model) which has a linear state transition. This second model describes a bearing-only tracking of an object in the two-dimensional $(x - y)$ plane, with a fixed observer position, where observation (z) is the bearing of the object trajectory. Through the BOT model, we can compare the estimation quality of CPF to GPU particle filters in [11,16,17]. We can see that the error is in the same range with [16] and is better than error in [11]. According to the error and time measurements, we can state that this is a feasible mapping from a virtual machine (CNN UM-inspired architecture) to a state-of-the-art architecture with a mature ecosystem available at a low cost.

In this algorithm, each thread executes roughly 300 to 2,100 floating point operations. This depends on the neighbourhood size. Those operations which are performed through the neighbourhood are additions and unfortunately divisions (calculating the actual weights with the norming sums in the resampling). This amount of divisions are clearly one bottleneck. The speed-up achieved with *fast_math* also supports this explanation. The other bottleneck of the algorithm is the shared memory size and access pattern. If more threads could reside in a block, then the ratio of the overlay among blocks due to the neighbourhood size would be less. In the resampling, branches are unavoidable, and fork and join of threads within a warp are necessary. There are a number of *for-loops* which iterate through the given number of iterations where the end index is unknown at the time of host code compilation. In our framework, it is only known in runtime. Still, for a given application, optimal parameters can be set directly in the code based on measurements, and therefore, loop unroll can be applied. Another approach can be just-in-time (JIT) kernel compilation. This method is effective only if the JIT compilation time is less than the cumulative gain from the loop unrolling through the state predictions.

On the one hand, if we would perform the algorithm as purely sequential, the order would be $O(N \times R)$ without the random number generation which (as mentioned) is not part of the basic task. On the other hand, with a virtual GPU on which all blocks are active simultaneously (totally utilized), the order of the algorithm would be $O(R)$

as the threads are independent from each other, not counting the synchronization points. However, on GTX 550, the time increase starts after 1,536 threads (on 6 SMPs× 256 threads). On GTX 580, there are 16 multiprocessors in comparison with the 6 multiprocessors of GTX 550; therefore, under 4,096 threads, the GPU would be under-utilized (i.e. GTX 580 is the top GPU of the Fermi GPUs).

A direct comparison to a CPU, sequential particle filter, is not entirely adequate; still, we would like to mention differences in the running time. In [21], particle filter was realized and used as a reference for time measurements of the proposed algorithm; therefore, we compare our results to this also. However, time measurements are not presented for many particle numbers; measurements of 4,096 particles already indicates the benefits of our implementation compared to the CPU version. Execution time of the algorithm was 16.417 s for the first model on a dual-core processor PC (Intel T6570) with 2.1 GHz. Compared to our 100-ms execution time, we can say that a 164x speed-up is achieved. We provide all our measurement data as Additional file 1.

Conclusions

In this paper, we introduced the first adaptation of CPF to GPU architecture. Compared to [21], we measured the performance on a real hardware. The strength of this approach is to maintain the local connectivity to prevent information loss, while the position error and the execution time are comparable to those of [16] if we assume that their measurements also include all operations (file I/O, random number generation, etc.). We utilize architectural features: whilst CPF algorithm would demand two-dimensional representation, (e.g. texture or surface), we modified the algorithm to enable one-dimensional processing and still kept the local connectivity and the local neighbourhood-based reduced and parallel processing.

The compute capability of the GPU determines the maxima of the number of threads that can be handled at each state. The sequential operations are performed in the shared memory thus shall make no effect on running time when we increase the number of particles. Still, as the shared memory blocks are arranged to multiprocessors as defined on the device, there is a scheduling which introduces a delay. Therefore, however parallel, the algorithm still will require more time at more particles.

The proposed method is independent from the random generator if the quality of the uniform and normal distribution is acceptable. As the first approach, we used NVIDIA Mersenne Twister for which the corrections to achieve adequate distribution, see details in the Appendix. In the second approach, we used *curand* which generates appropriate distribution quickly. This adaptation of CPF by delicate modifications presents GPU as an excellent platform to solve problems that could not be solved real-time previously.

Appendix

Modification of NVIDIA SDK Mersenne twister

NVIDIA SDK provides an implementation of Mersenne twister (MT) [34,35], which apparently exposes an

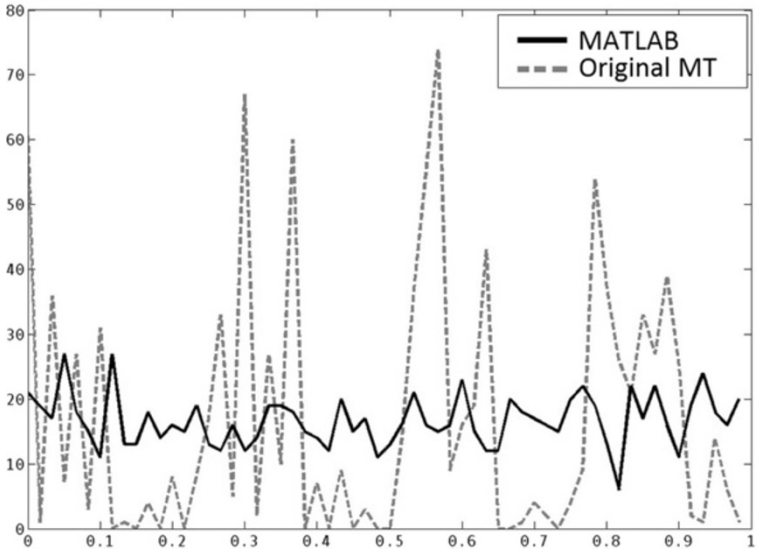

Figure 13 Mersenne Twister distribution. Mersenne Twister uniform distribution (gray) histogram compared to MATLAB uniform distribution (black) with 60 histogram bins on 1,000 random numbers. We can see significant spikes in the histogram of NVIDIA Mersenne Twister which would introduce bias to the resampling. Therefore, it is not suitable for our purpose.

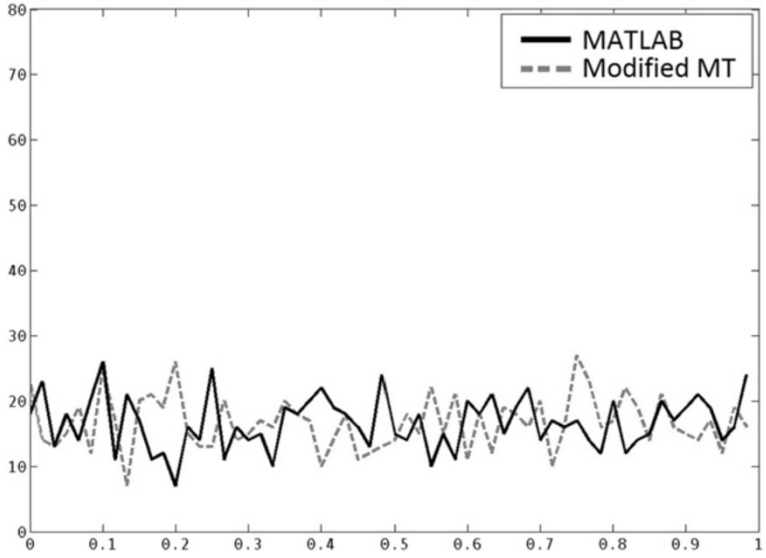

Figure 14 Modified MT distribution. A total of 1,000 random number were generated with the modified implementation of Mersenne Twister compared to MATLAB and original Mersenne Twister distributions on 60 bins. Due to new characteristics, now we have a suitable random number generator.

attainable solution. Still, we observed that the generated distribution is inappropriate for a small set (hundreds or thousands) of numbers and is primarily admissible for around two million numbers and above. As we mentioned earlier, in relation to GPU memory management, the main operations are performed in the shared memory; thus, random number generator is required to comply with the shared memory array size. Consequently, the original NVIDA SDK MT is not feasible for our implementation. Figure 13 shows the difference of the histogram of the MT-generated numbers compared to the histogram of a same number of random values from MATLAB.

To achieve an adequate distribution for the resampling, we made the modifications as follows based on [34] and our empirical experiences. The degree of recursion was changed from 19 to 397; the middle term, from 9 to 624; and the shift value u, from 12 to 11 based on the original values from [34]. Besides, in the tempering transformation, we used the originally defined masks instead of the loaded ones from a predefined file, with hexadecimal values in the bitwise operations. For each thread, the first element of the state array is calculated with a thread and current time-based seed value based on the thread ID and the current system time. The final value is calculated with initializing on the first element of the bit vector for each thread.

With the above modifications, we achieved an acceptable uniform distribution from the Mersenne Twister, which is illustrated in Figure 14, compared to the former MT and the MATLAB distributions.

Additional file

Additional file 1: estimationBOT, estimationFirst, input1000BOT, input1000First, and readme.txt. input1000BOT folder: files containing the 1,000 trajectories for the BOT model. estimationBOT folder: estimation (ie. output) values of the cellular particle filter for the BOT model. Each file corresponds to a configuration, containing 1,000 lines each for a different estimated trajectory. input1000First folder: files containing the 1,000 trajectories for the first model. estimationFirst folder: estimation (ie. output) values of the cellular particle filter for the first model. Each file corresponds to a configuration, containing 1,000 lines each for a different estimated trajectory.

Competing interests
The authors declare that they have no competing interests.

Acknowledgements
This research was supported by the European Union and the State of Hungary, co-financed by the European Social Fund in the framework of TÁMOP 4.2.4.A/-11-1-2012-0001 (National Excellence Program). The support of Furukawa Electric Institute of Technology Ltd. and the grants TÁMOP-4.2.1.B-11/2/KMR-2011-0002 and TÁMOP-4.2.2/B-10/1-2010-0014 are also gratefully acknowledged. The authors would like to thank Miklós Rásonyi for his help and suggestions.

Received: 27 February 2013 Accepted: 8 September 2013
Published: 21 September 2013

References

1. N Azzabou, N Paragios, F Guichard, Image reconstruction using particle filters and multiple hypotheses testing. IEEE Trans. Image Process. **19**(5), 1181–1190 (2010)
2. J Czyz, B Ristic, B Macq, A particle filter for joint detection and tracking of color objects. Image Vis. Comput. **25**(8), 1271–1281 (2007)
3. F Gustafsson, F Gunnarsson, N Bergman, U Forssell, J Jansson, R Karlsson, P Nordlund, Particle filters for positioning, navigation, and tracking. IEEE Trans. Signal Process. **50**(2), 425–437 (2002)

4. H Lopes, R Tsay, Particle filters and Bayesian inference in financial econometrics. J. Forecasting. **30**, 168–209 (2011)
5. S Chib, F Nardari, N Shephard, Markov chain Monte Carlo methods for stochastic volatility models. J. Econometrics. **108**(2), 281–316 (2002)
6. Y Ephraim, N Merhav, Hidden markov processes. IEEE Trans. Inf. Theory. **48**(6), 1518–1569 (2002)
7. RE Kalman, A new approach to linear filtering and prediction problems. Trans. ASME–J. Basic Eng. **82**(Series D), 35–45 (1960)
8. M Arulampalam, S Maskell, N Gordon, T Clapp, A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. Signal Process IEEE Trans. **50**(2), 174–188 (2002)
9. J Candy, Bootstrap particle filtering. Signal Process Mag. IEEE. **24**(4), 73–85 (2007)
10. G Hendeby, K Rickard, F Gustafsson, Particle filtering: the need for speed. EURASIP J. Adv. Signal Process. **2010**, 181403 (2010)
11. M Bolic, P Djuric, S Hong, Resampling algorithms and architectures for distributed particle filters. IEEE Trans. Signal Process. **53**(7), 2442–2450 (2005)
12. C Chu, C Chao, M Chao, A Wu, Multi-prediction particle filter for efficient parallelized implementation. EURASIP J. Adv. Signal Process. **2011**, 1–13 (2011)
13. A Lee, C Yau, MB Giles, A Doucet, CC Holmes, On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods. J. Comput. Graphical Stat. **19**(4), 769–789 (2010)
14. R Cabido, AS Montemayor, JJ Pantrigo, BR Payne, Multiscale and local search methods for real time region tracking with particle filters: local search driven by adaptive scale estimation on GPUs. Mach. Vis. Appl. **21**, 43–58 (2009)
15. K Otsuka, J Yamato, Fast and robust face tracking for analyzing multiparty face-to-face meetings, in *Machine Learning for Multimodal Interaction*, ed. by A Popescu-Belis, R Stiefelhagen. Proceedings of the 5th International Workshop, MLMI 2008, Utrecht, September 8-10 2008. Lecture Notes in Computer Science, vol. 5237 (Springer, Heidelberg, 2008), pp. 14–25
16. MA Chao, CY Chu, CH Chao, AY Wu, Efficient parallelized particle filter design on CUDA, in *2010 IEEE Workshop on Signal Processing Systems (SIPS), San Francisco, 6–8 Oct 2010* (IEEE, Piscataway, 2010), pp. 299–304
17. M Chitchian, A Simonetto, AS van Amesfoort, T Keviczky, Distributed computation particle filters on GPU architectures for real-time control applications. IEEE Trans. Control Syst. Technol. **PP**(99), 1 (2013)
18. O Brun, V Teuliere, JM Garcia, Parallel particle filtering. J. Parallel Distributed Comput. **62**(7), 1186–1202 (2002)
19. AS Bashi, VP Jilkov, XR Li, H Chen, Distributed implementations of particle filters, in *Proceedings of the Sixth International Conference of Information Fusion - FUSION 2003, Cairns 8-11 July 2003* (IEEE, Piscataway, 2003), pp. 1164–1171
20. L Murray, GPU acceleration of the particle filter: the Metropolis resampler, in *Distributed Machine Learning and Sparse Representation With Massive Data-Sets*, (DMDD, 2011). arXiv:1202.6163
21. A Horváth, M Rasonyi, Topographic implementation of particle filters on cellular processor arrays. Signal Process. **93**(7), 1853–1863 (2012)
22. L Chua, L Yang, Cellular neural networks: theory. IEEE Trans. Circuits Syst. **35**(10), 1257–1272 (1988)
23. NJ Gordon, DJ Salmond, AFM Smith, Novel approach to nonlinear/nonGaussian Bayesian state estimation. Radar and Signal Processing, IEE Proc. F **140**, 107–113 (1993)
24. J Halton, Sequential Monte Carlo. Math. Proc. Cambridge Philos Soc. **58**, 57–78 (1962)
25. XL Hu, TB Schön, L Ljung, A general convergence result for particle filtering. IEEE Trans. Signal Process. **59**(7), 3424–3429 (2011)
26. A Doucet, AM Johansen, A tutorial on particle filtering and smoothing: fifteen years later, in *Oxford Handbook of Nonlinear Filtering* (Oxford University Press, Oxford, 2008), pp. 4–6
27. NVIDIA Corporation, NVIDIA CUDA Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html Accessed 24 October 2012
28. G Tornai, G Cserey, I Pappas, Fast DRR generation for 2D to 3D registration on GPUs. Med. Phys. **39**(8), 4795 (2012)
29. V Demchik, Pseudo-random number generators for Monte Carlo simulations on ATI graphics processing units. Comput. Phys. Commun. **182**(3), 692–705 (2011)
30. WB Langdon, A fast high quality pseudo random number generator for nVidia CUDA, in *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers, Montreal, 08-12 July 2009* (ACM, New York, 2009), pp. 2511–2514
31. M Manssen, M Weigel, AK Hartmann, Random number generators for massively parallel simulations on GPU. Eur. Phys. J. Special Top. **210**, 53–71 (2012)
32. BP Carlin, NG Polson, DS Stoffer, A Monte Carlo approach to nonnormal and nonlinear state-space modeling. J. Am. Stat. Assoc. **87**(418), 493–500 (1992)
33. G Kitagawa, Monte Carlo filter and smoother for non-Gaussian nonlinear state space models. J. Comput. Graphical Stat. **5**, 1–25 (1996)
34. M Matsumoto, T Nishimura, Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. (TOMACS). **8**, 3–30 (1998)
35. V Podlozhnyuk, Parallel Mersenne Twister (2007). http://developer.download.nvidia.com/compute/cuda/2_2/sdk/website/projects/MersenneTwister/doc/MersenneTwister.pdf Accessed 16 Dec 2010

doi:10.1186/1687-6180-2013-148
Cite this article as: Gelencsér-Horváth *et al.*: **Fast, parallel implementation of particle filtering on the GPU architecture.** *EURASIP Journal on Advances in Signal Processing* 2013 **2013**:148.