Efficient GPU-Accelerated Implementation of Particle and Particle Flow Filters for Target Tracking

VESSELIN P. JILKOV
JIANDE WU

Particle filtering is a very popular method for nonlinear/non-Gaussian state estimation, however, implementation of particle filters (PFs) with a high state dimension in real-time is a very challenging practical task because the computation is prohibitive. Parallel & distributed (P&D) computing is a natural way to deal with the computational challenges of PF methods in order to make them practical for large scale problems, such as multitarget multisensor tracking. This paper presents results on development, implementation and performance evaluation of computationally efficient parallel algorithms for particle and particle flow filters (PFFs) utilizing a Graphics Processing Unit (GPU) as a parallel computing environment. Proposed are state-of-the-art parallel PF and PFF implementations which are optimized for GPU architecture and capabilities. The proposed algorithms are applied and tested, via simulation, for tracking multiple targets using a pixelized sensor, and for a high-dimensional nonlinear density estimation problem. It is demonstrated by the obtained simulation results that the proposed parallel GPU implementations can greatly accelerate the computation of both PFs and PFFs, and thereby bring them closer to practical applications.

Manuscript received on December 30, 2014; released for publication March 23, 2015.

Refereeing of this contribution was handled by Paolo Braca.

Supported in part by Louisiana BoR through grant LEQSF(2009–12)-RD-A-25, ARO through grant W911NF-08-1-0409, ONR-DEPSCoR through grant N00014-09-1-1169, and NASA/LEQSF (2013–15) through grant NNX13A29A.

Authors'address: Department of Electrical Engineering, University of New Orleans, New Orleans, LA 70148, USA (e-mail: vjilkov@uno.edu).

1557-6418/15/$17.00

I. INTRODUCTION

For more than two decades now, particle filtering [1] has become the most popular approach for nonlinear/non-Gaussian state estimation problems. Particle filters (PFs) have found numerous applications in areas that involve nonlinear filtering of noisy signals (data) [2], most notably in target tracking, e.g., [3]. However, while the literature reporting capabilities of PFs to solve hard nonlinear estimation problems from different application areas is abundant, most practical implementations are limited to small-scale problems with low dimensional state vectors, such as single target tracking. Implementation of high dimensional PFs in real-time for large-scale problems is still quite challenging. Parallel & distributed (P&D) computing is a natural way to overcome/mitigate this limitation. Development of P&D algorithms and architectures that fully exploit the spatial and temporal concurrency of the computations is a great potential to make PFs (and density-based nonlinear filtering, in general) practical for large scale problems, such as multitarget multisensor tracking. Considerable research effort has been going on along this direction, e.g., [4]–[15].

In recent years a new class of nonlinear filters has been gaining momentum—the particle flow filters (PFFs), proposed by Daum & Huang [16]–[22], which overcome the well known problem of particle degeneracy of the PFs (and other, e.g., deterministic sampling methods for density-based nonlinear filtering [23]) caused by the pointwise multiplication of the Bayes rule. The method for samples' update is deterministic and is conceptually based on a natural homotopy relating the prior and posterior filter densities which induces a flow of the samples from the prior density towards a set of samples from the posterior. A flow of each particle from prior to posterior is governed by a flow equation—an ordinary differential equation (ODE)—which in general satisfies a linear partial DE (PDE) with constant coefficients. This PDE is central in this approach. While a numerical integration of the PDE is prohibitive for real-time computation, the good news is that for some special distributions, e.g. Gaussian and exponential families it is analytically tractable. The explicit solution for (unnormalized) Gaussian prior and likelihood, referred to as the exact PFF [19] is straightforward to implement and fast for computation. Plenty of simulation results have been reported by the authors of the PF method, e.g., [18], [21], [22], showing that PFFs can outperform other nonlinear filters in computation and accuracy for difficult nonlinear problems. Another nice property of the PFFs is that they are essentially "embarassingly" parallel—(almost) all computations are independent and can be conducted in a parallel/distrubuted manner as opposed to PFs where resampling is a bottleneck. This makes PFFs even more attractive and promising for P&D computing. It also motivated us to pursue parallel implementations of PFF

TABLE I
Generic SIS/R PF Algorithm [26]

- *Importance Sampling* (IS)
  - — For $i = 1,\ldots,\bar{N}$
    Draw a sample (particle): $\bar{x}_k^i \sim \pi(x_k \mid x_{k-1}^i, z^k)$
    Evaluate importance weights
    $\bar{w}_k^i = w_{k-1}^i \dfrac{p(z_k \mid x_k^i) p(x_k^i \mid x_{k-1}^i)}{\pi(x_k^i \mid x_{k-1}^i, z^k)}$
  - — For $i = 1,\ldots,\bar{N}$
    Normalize importance weights: $w_k^i = \dfrac{\bar{w}_k^i}{\sum_{j=1}^{\bar{N}} \bar{w}_k^j}$
- *Resampling* (R)
  - — Effective sample size estimation: $\hat{N}_{eff} = \dfrac{1}{\sum_{j=1}^{N} (w_k^j)^2}$
  - — If $\hat{N}_{eff} < N_{th}$
    Sample from $\{\bar{x}_k^j, w_k^j\}_{j=1}^{N}$ to obtain
    a new sample set $\left\{ x_k^i = \bar{x}_k^{j_i}, w_k^i = \dfrac{1}{N} \right\}_{i=1}^{N}$

and make a quantitative performance evaluation and comparison with parallel PF algorithms studied by us before [11]–[13].

At present, there are many types of parallel hardware available such as multicore processors, field-programmable gate arrays (FPGAs), computer clusters, and graphics processing units (GPUs). A GPU is a single instruction multiple data (SIMD) parallel processor intended originally to meet the demands of computationally intensive tasks for real-time high-resolution 3D-graphics. Nowadays, GPUs are highly parallel multicore systems that can process very efficiently large blocks of data in parallel [24]. For highly parallelizable algorithms GPUs are becoming more efficient than the sequential central processing unit (CPU) [10]. On the other hand, GPUs are easily accessible and inexpensive—most new personal computers have GPU card. Hence, GPUs offer an attractive opportunity for speeding up PFs and PFFs for real-time applications.

In this paper we present results on development, implementation and performance evaluation of computationally efficient parallel algorithms for particle and particle flow filters by utilizing a Graphics Processing Unit (GPU) as a parallel computing environment. Proposed are state-of-the-art parallel PF and PFF implementations which are optimized for GPU architecture and capabilities. The proposed algorithms are applied and tested, via simulation, for tracking multiple targets using a pixelized sensor (up to 20 targets), and for a high-dimensional nonlinear density estimation problem (up to 40-dimensional state vector). Comprehensive simulation results are presented which illustrate that the proposed parallel GPU implementations can greatly accelerate the computation of both PF and PFF, and thereby bring them closer to practical applications.

The paper is organized as follows. Sect. II provides background information on basic particle & particle flow filtering, and GPU concepts, needed for the rest of the paper. Sect. III proposes parallel PF and PFF algorithms, optimized for GPU implementation. Sect. IV presents application of the parallel GPU PFs for tracking multiple ground targets with a pixelized sensor by a Joint Multitarget Probability Density (JMPD) algorithm and analyzes their performance based on the obtained simulation results. Sect. V presents implementation and performance evaluation/comparison of parallel PF vs. PFF over a high dimensional density estimation problem. Sect. VI provides a summary and conclusions.

This paper is an outgrowth of our previous conference papers [13] and [14], wherein some preliminary parts of this paper have been presented. However the presentation for this journal paper has been unified, improved, and extended with significant new results, e.g., the enhanced parallel PF presented in Sect. III-B and its performance evaluation in Sect IV-B.2.

II. BACKGROUND

This section outlines very briefly the computational procedures of the generic Particle Filter (PF) and the Exact Particle Flow Filter (PFF), and provides some basic GPU computing concepts, needed for the rest of the paper. Parallelization of the filtering algorithms and their GPU implementation are discussed in Section III.

A. Bayesian Recursive Filter

Let $\{x_k\}_{k=1,2,\ldots}$ be a vector valued discrete-time Markov process with state transition probability density function (pdf) $p(x_k \mid x_{k-1})$, and $\{z_k\}_{k=1,2,\ldots}$ be another process, stochastically related to $\{x_k\}_{k=1,2,\ldots}$ through the likelihood $p(z_k \mid x_k)$. x_k and z_k are the state and the measurement, respectively, and $p(x_k \mid x_{k-1})$ and $p(z_k \mid x_k)$ are the state and measurement models. The exact Bayesian recursive filter (BRF) provides the posterior density $p(x_k \mid z^k)$ via the following prediction-update scheme [25]:

$$p(x_{k-1} \mid z^{k-1}) \rightarrow p(x_k \mid z^{k-1}) \rightarrow p(x_k \mid z^k) \qquad (1)$$

given $p(x_0)$ and measurements $z^k = \{z_1,\ldots,z_k\}$.

B. Generic Particle Filter

In particle filtering the pdfs are represented approximately through a set of random samples (particles) and the BRF (1) is performed directly on these samples. Most PFs are based on two principal components: sequential importance sampling (SIS) and resampling (R) as given in Table I [26].

The importance distribution $\pi(\cdot)$ must contain the support of the posterior and is subject to design. One possibility is to choose $\pi = p(x_k \mid x_{k-1}^i)$ [1] which is often referred to as the sampling importance resampling (SIR) PF, or the bootstrap PF. Many other choices are possible [2]. Resampling is in effect discarding of samples that have small probability and concentrating on samples that are more probable. The resampling step is critical in every implementation of PF because

TABLE II
Gaussian Exact Particle Flow Algorithm

- *Particle Prediction*
  — For $i = 1,\ldots,\bar{N}$
  Draw a predicted particle: $\bar{x}_k^i \sim p(x_k \mid x_{k-1}^i)$
- *Particle Update*
  — Compute predicted estimate $\bar{x}_k$, its error covariance P, and linearized measurement model matrix H
  — Compute parameters of flow velocity (exact computation):
  $A(\lambda) = -\frac{1}{2}PH'(\lambda HPH' + R)^{-1}H$
  $b(\lambda) = (I + 2\lambda A)[(I + \lambda A)PH'R^{-1}z_k + A\bar{x}_k]$
  — For $i = 1,\ldots,N$
  Solve the particle flow ODE
  $\frac{dx}{d\lambda} = f(x,\lambda) = A(\lambda)x + b(\lambda)$
  for $\lambda \in [0\ \ 1]$ with initial condition $x(0) = \bar{x}_k^i$;
  Let $x_k^i := x(1)$.

without it the variance of the particles weights quickly increases leading to inference degradation.

C. Gaussian Exact Particle Flow Filter

In this paper we limit our consideration to the Gaussian Exact PFF [19]. We summarize one time-step, $k-1 \to k$, of the algorithm in Table II.

First, note that the prior and posterior at each time-step k are also represented through samples: $\{\bar{x}_k^i\}_{i=1}^{\bar{N}}$ and $\{x_k^i\}_{i=1}^{N}$, respectively. In Table II we give the prediction step in terms of random sampling (as in the bootstrap PF) which amounts to passing each particle from the posterior through the stochastic state dynamic model. However the prediction sampling need not be random in general—the prior can be approximated by deterministic samples as well, e.g., based on optimal Dirac mixture approximations [27]. In our GPU implementation we use random sampling but deterministic sampling for prediction is of further interest for future implementation because generating random numbers by GPU is not straightforward and may introduce some latency.

Second, the computation of the state prediction estimate $\bar{x}_k$, error covariance matrix $\bar{P}_k$, and linearized measurement model H_k is not explicitly given because it can be done in different ways, e.g., $\bar{x}_k$ and $\bar{P}_k$ can be computed as the sample mean an covariance of the particles and then H_k can be obtained via linearization of the measurement model about $\bar{x}_k$, or the computation can be done via EKF/UKF equations.

Third, the flow velocity parameters $A(\lambda)$, $b(\lambda)$ are common for all particles and, therefore, their computation is given outside the `for`-loop for solving the ODE in Table II. However, in a parallel computer implementation this is not necessarily the fastest arrangement—Assuming each "processor" is dedicated to one particle or a groups of particles (as in our GPU-implementation, discussed in Sect. III), it could be better to compute $A(\lambda)$, $b(\lambda)$ for each particle (as if within the particles' `for`-loop) or group of particles in parallel, and thus eliminate the time for transferring data among "processors."

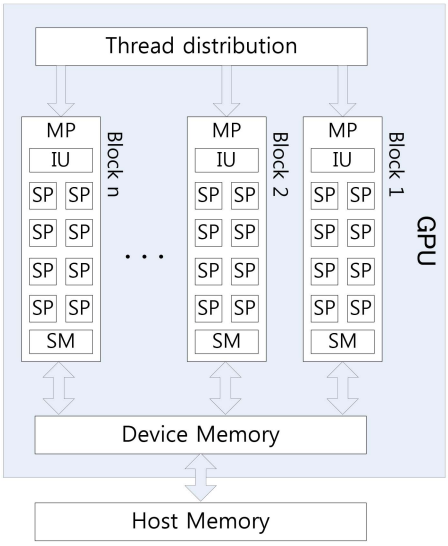

Fig. 1. Nvidia GPU Architecture [28]

D. GPU & CUDA Computing

A typical GPU is a collection of multiprocessors (MPs) where each of them has several scalar processors (SP), also referred to as cores [28]. At any given clock cycle, each SP executes the same program (one or a set of instructions) on different data by following the single instruction/program multiple data (SIMD/SPMD) models. Each SP has access to different memory levels. Fig. 1 gives a simplified illustration of the architecture of Nvidia GPU, where SM stands for shared memory, and IU—for instruction unit. This type of architecture is ideally suited to data-parallel computation since large quantities of data can be loaded into shared memory for the cores to process in parallel.

Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by Nvidia as a computing engine in GPUs. It allows access to the virtual instruction set and memory of a GPU's parallel computational elements. By using CUDA the GPUs can be used for computation like CPUs.

A detailed description of CUDA is given in [29]. The MP model used in CUDA is called single-instruction multiple-thread (SIMT). In SIMT, the MP maps each thread to one SP core, and each thread executes independently with its own instruction address and register state. The MP creates, manages, and executes concurrent threads in hardware with no scheduling overhead. The threads are logically organized in blocks and grids. A block is a set of threads, while a grid is a set of blocks. The sizes of the blocks and grids can be programmatically controlled. To optimize an execution configuration the first parameters to choose are the grid size (number of blocks per grid) and block size (number of threads per block). As a general guideline, provided by [29], the primary concern is keeping the entire GPU busy—the

number of blocks in a grid should be larger than the number of multiprocessors so that all multiprocessors have at least one block to execute, and there should be multiple active blocks per multiprocessor so that blocks which are not waiting for thread synchronization can keep the hardware busy.

CUDA devices use several memory spaces, which have different characteristics that reflect their distinct usages in CUDA applications. These memory spaces include global, local, shared, texture, constant, and registers. There is a 16 KB per thread limit on local memory, a total of 64 KB of constant memory, and a limit of 16 KB of shared memory, and either 8,192 or 16,384 32-bit registers per multiprocessor. Global, local, and texture memory have the greatest access latency, followed by constant memory, registers, and shared memory. Memory optimizations are key for performance. The best way to maximize bandwidth is to use as much fast-access memory and as little slow-access memory as possible.

III. PARALLEL PF AND PFF IMPLEMENTATION ON GPU

The general idea of implementing both particle algorithms (PF and PFF) on GPU is to map (dedicate) a thread to a particle and organize the algorithms in terms of groups of particles, mapped correspondingly to GPU blocks, in order to take advantage of the GPU architecture. The key is to use the shared memory for fast particle data communication within each block and avoid/reduce communications between different blocks, as much as possible. Optimizing any particular implementation is crucially dependent on the capabilities of the available GPU hardware, as well as, on the filtering problem dimension (size). For our application studies this issue is discussed further in Sect. IV.

Other important issues arising in the GPU implementation for both filters are the cumulative summation (needed for normalization and sample mean computation) and random number generation. The cumulative sum can be implemented using a multipass scheme similar to that of [10]. This multipass scheme is a standard method for parallelizing seemingly sequential algorithms based on the scatter and gather principles. The CURAND library [30] provides facilities to efficiently generate high-quality pseudorandom and quasirandom numbers. It is used to create several generators at the same time. Each generator has a separate state and is independent of all other generators.

A. Parallel PF

The computation for importance sampling is independent across particles and can be executed in parallel without any data communication among particle-dedicated processors (or threads). Parallelization of the resampling part, however, is quite nontrivial because generating a single resampled particle requires information from all particles of the sample set. Thus, resampling becomes a bottleneck in parallel implementations. Several parallel/distributed resampling schemes have been already proposed and studied in the literature, e.g., [4], [7]. In this paper we implement distributed resampling with nonproportional allocation (RNA) [7]. Briefly, the idea is as follows (more details, including some high level pseudocode, can be found in [7] and our previous papers [11]–[13]).

Distributed RNA is a modification of the distributed resampling with proportional allocation (RPA) which is essentially based on stratified sampling [2]. The sample space is partitioned into several strata (groups) and each stratum corresponds to a processing node (PN)—mapped to a GPU block of threads in our GPU implementation. Proportional allocation among strata is used, which means that more samples are drawn from the strata with larger weights. After the weights of the strata are known, the number of particles that each stratum replicates is calculated at a head processing node (HN)—a dedicated block of threads in a GPU implementation—using residual systematic resampling, and this process is referred to as inter-resampling since it treats the PNs as single particles. Finally, resampling is performed inside the strata (at each PN, in parallel) which is referred to as intra-resampling. Therefore, the resampling algorithm is accelerated by having an inner loop that can run in parallel on the PNs (intra-resampling) with small centralized processing (inter-resampling) at HN. RPA requires a complicated scheme for particle routing. In the parallel RNA particle routing is deterministic and planned in advance by the designer. The number of particles within a group after resampling is fixed and equal to the number of particles per group as opposed to the RPA where this number is random (proportional to the weight of the stratum). This introduces an extra approximation in the resampling (in statistical sense) but allows to execute resampling in parallel by each group (GPU block).

In our implementation each group of particles, as defined in RNA, is naturally mapped into a GPU block as each particle of the group corresponds to a thread of the block. Thus, in terms of the GPU architecture, the importance sampling is thread-parallel while the intra-resampling part is only block-parallel, and the inter-resampling and weight normalization are centralized (Table III). The centralized parts can be computed at the CPU or at some of the blocks of the GPU (designated as a head-node). Both options are implemented in our simulation study, presented in Sect. IV.

B. Enhanced Parallel PF

A significantly enhanced alternative to the presented, RNA-based, PPF is proposed based on the following two reasons. First, each thread can access all particles'

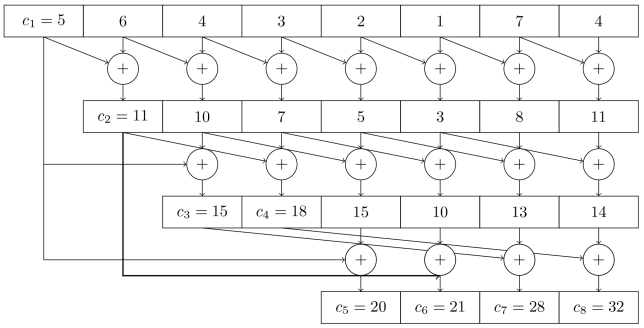

Fig. 2. Parallel Cumulative Sum

TABLE III
Parallel PF Algorithm for GPU

- *Importance Sampling* (IS)
  - — For $i = 1,\ldots,N$ (**Thread Parallel**)
    Sample: $\bar{x}_k^i \sim \pi(x_k \mid x_{k-1}^i, z^k)$
    Evaluate importance weights $\bar{w}_k^i$
  - — For $i = 1,\ldots,N$ (**Centralized**)
    Normalize importance weights: $w_k^i = \dfrac{\bar{w}_k^i}{\sum_{j=1}^{N} \bar{w}_k^j}$
- *Resampling* (R)
  $\{\bar{x}_k^j, w_k^j\}_{j=1}^N \Rightarrow \left\{ x_k^i = \bar{x}_k^{j_i}, w_k^i = \dfrac{1}{N} \right\}_{i=1}^N$
  - — *Inter Resampling* (**Centralized**)
  - — *Intra Resampling* (**Block Parallel**)

information concurrently through the shared and device memory and, thus, replicating particles could be parallelized in all threads. Second, the cumulative sum of weights, which is key for resampling, can be also executed in parallel in contrast to the generic (RNA-based) PPF where it is done in a centralized manner.

When implementing a cumulative weight sum on a single processor, the time complexity is $O(n)$ addition operations for an array of length n. This is the minimum number of additions required. A parallel algorithm is presented in [31]. Fig. 2 shows the operation. The time complexity is $O(\log n)$ if enough processors are available. This is fine for small arrays. In our application, particle filter algorithms use large number of particles, so large arrays are used.

We modify the PPF by using a parallel cumulative weighted sum algorithm. The enhanced PPF algorithm is shown in Table IV.

Additional illustration of the operation of the EPPF is given in Fig. 3. Note that Step 3 can be done thread parallel because all threads can access all cumulative sums concurrently.

After optimizing the particle filter algorithm, the main bottleneck left is the global memory access. To better cover the global memory access latency and improve overall efficiency, we should let each thread do more computation. So we could let each thread process two or four particles instead of one particle. Each thread performs a sequential access of four particles and stores

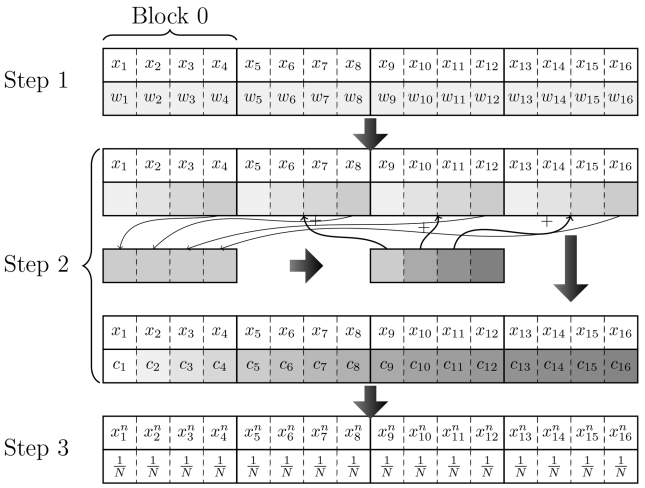

Fig. 3. Illustration of the Enhanced GPU PF

TABLE IV
Enhanced PPF Algorithm for GPU

- *Importance Sampling* (IS)
  - — For $i = 1,\ldots,N$ (**Thread Parallel**)
    Sample: $\bar{x}_k^i \sim \pi(x_k \mid x_{k-1}^i, z^k)$
    Evaluate importance weights $\bar{w}_k^i$
- *Unnormalized Cumulative Sum of Weights* (UCSW)
  - — For $j = 1,\ldots,N$ (**Block Parallel**)
    $\bar{c}_k^j = \bar{w}_k^1 + \cdots + \bar{w}_k^j$
    Execute UCSW for each block
    Execute UCSW for all blocks
    Get all UCSWs
- *Resampling* (R)
  - — For $i = 1,\ldots,N$ (**Thread Parallel**)
    Normalize UCSW: $c_k^i = \dfrac{\bar{c}_k^i}{\sum_{j=1}^{N} \bar{c}_k^j}$
    Sample from $\{x_k^j\}_{j=1}^N$ with $\{c_j\}_{j=1}^N$ to obtain a new sample set $\left\{ x_k^i = x_k^{j_i}, w_k^i = \dfrac{1}{N} \right\}_{i=1}^N$

them in registers. This method is more than twice as fast as the code which only processes one particle at each thread.

C. Parallel PFF

GPU parallelization of PFF is much easier than that of the PF. It is apparent from Table II that the prediction part and solving the ODE for each particle is independent across particles and can be executed completely in parallel by mapping a particle to a GPU thread. For computing $\bar{x}_k$, P_k, H_k, and the flow velocity parameters $A(\lambda)$, $b(\lambda)$ there are different options because they are common for all particles. One way is to have them computed by the CPU ("externally" to the GPU) by a point estimator like EKF/UKF and sent to each particle. Another way is to collect all predicted particles in the CPU, compute sample mean $\bar{x}_k$, and based on it, compute P_k, H_k, $A(\lambda)$, $b(\lambda)$ and send the results to each particle. Involvement of the CPU imposes a communication time overhead (due to the need

to access the slow memory), and does not take advantage of the GPU block architecture. That is why, as in the GPU-PF, we divide all particles in groups and map each group to a GPU block wherein each particle corresponds to a thread. A predicted estimate $\bar{x}_{k,b}$ is computed for each block b, as the sample mean of its particles, and based on it, flow velocity parameters for this block are computed. Thus, even though some extra approximation is incurred, any communication with the CPU and among different block is avoided. An outline of the Parallel GPU PFF is shown in Table V.

For computing a block $\bar{x}_{k,b}$ we use the sample mean of the particles within the block. Then $P_{k,b}$, $H_{k,b}$ are computed using the EKF equations, based on $\bar{x}_{k,b}$.

For numerical computation of the ith particle's flow ODE we use the finite difference forward Euler method (as given in Table V), where $L > 1$ is an integer defining the discretization step $\Delta\lambda = 1/L$ of the interval [0 1].

In accordance with the limitation of shared memory (our device is 48K bytes/block), the number of particles we include in each block is given in Table VIII. We also use the same block sizes for the GPU-PF implementation.

Note that, except for the small block sample mean part (which is block parallel), this GPU-PFF implementation is thread parallel as opposed to the GPU-PF implementation (Table III) wherein a significant part (the resampling) is only partially (block) parallel. This clearly gives an explanation to the fact that the implemented GPU-PFF is much faster than the GPU-PF for the same number of particles, as seen in the simulation results presented in Sect. V.

TABLE V
Gaussian Exact Parallel PFF Algorithm

- *Particle Prediction*
  - — For $i = 1,\ldots,N$ (**Thread Parallel**)
    Sample: $\bar{x}_k^i \sim p(x_k \mid x_{k-1}^i)$
- *Particle Update*
  - — For $b = 1,\ldots,N_b$ (**Block Parallel**)
    Compute $\bar{x}_{k,b}$, $P_{k,b}$, $H_{k,b}$, $f_{k,b}$
  - — For $i = 1,\ldots,N$ (**Thread Parallel**)
    Compute particle flow
    $x_{[0]} = \bar{x}_k^i$
    $x_{[l+1]} = x_{[l]} + \Delta\lambda f_b(x_{[l]}, l\Delta\lambda)$, $l = 0,\ldots,L-1$
    $x_k^i = x_{[L]}$

IV. SIMULATION STUDY I: GROUND MULTITARGET TRACKING USING IMAGE SENSOR

The main purpose of this simulation is to evaluate the computational feasibility and performance of the GPU PPFs (given in Tables III & IV) for a realistic target tracking scenario of high dimension. The parallel algorithms developed and implemented are based on the Joint Multitarget Probability Density (JMPD) algorithm for multiple target tracking [32]–[37].

A. JMPD Tracking

1) Problem Formulation:

In the JMPD approach to multiple target tracking the uncertainty about the number of targets present in a surveillance region as well as their individual states is represented by a single composite pdf. That is, the state of all targets is described by a meta-target state vector $X_k = (x_k^1, x_k^2, \ldots, x_k^T)$ where x_k^i is the state of target $i = 1,\ldots,T$. The number of targets at time k, $T_k \in [0,1,2,\ldots,\infty)$, is also assumed random. The posterior distribution of interest[1] is

$$p(X_k, T_k \mid Z^k) = p(X_k \mid T_k, Z^k) p(T_k \mid Z^k)$$

where $Z^k = \{Z_1, Z_2, \ldots, Z_k\}$ is the cumulative measurement set of the surveillance region up to time k, and $Z_l, l = 1,\ldots,k$ is the measurement set at time l. The model of target state and number evolution over time is given by $p(X_k, T_k \mid X_{k-1}, T_{k-1})$ and is referred to as the kinematic prior. It includes models of target motion, target birth and death, and any additional prior information on kinematics that may be available, e.g., terrain and road maps. The measurement model over the surveillance region is given by the likelihood $P(Z_k \mid X_k, T_k)$.

For the purpose of our implementation and performance study we adopt the kinematic prior and measurement models of [34].

2) Multitarget Model:

Each target $i = 1,\ldots,T$ is assumed to follow a nearly constant velocity motion model

$$x_k^i = F x_{k-1}^i + w_k^i \tag{2}$$

where $x = (\mathrm{x}, \dot{\mathrm{x}}, \mathrm{y}, \dot{\mathrm{y}})'$ is the state vector, $w \sim \mathcal{N}(0, Q)$ is white process noise with given covariance, and

$$F = \mathrm{diag}\left\{ \begin{bmatrix} 1 & \Delta \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & \Delta \\ 0 & 1 \end{bmatrix} \right\}$$

where Δ is the sampling interval. To account for maneuvers a mode variable can be also added [36]. The number of target in this paper is considered constant and known. Unknown number of targets using the transitional model of [36] is of interest for future work.

3) Sensor Model:

It is assumed that a pixelized sensor provides raw (unthresholded) measurements data from the surveillance region according to the following association-free model [34] used often for track-before-detect (TBD) problems. A sensor scan at time k consists of the

[1]Note that in this formulation the so-called "mixed labeling" [38] is not addressed. It is assumed that no track extraction is needed and, consequently, the ordering of x^i within X is irrelevant as far as only the density is of interest. In [34] this assumption is referred to as a symmetry under permutation.

outputs of M pixels (cells of the region), i.e., $Z = \{z[1],\ldots,z[M]\}$[2] where $z[i]$ is the output of pixel i. The likelihood $P(Z \mid X,T)$ is given by

$$P(Z \mid X,T) \propto \prod_{i \in i_X} \frac{p_{n_{[i]}(X)}(z[i])}{p_0(z[i])} \quad (3)$$

where i_X is the set of all pixels that couple to X, $n_{[i]}(X)$ is the occupation number of pixel i (number of targets from X that lie in i). The output z of each pixel is assumed to follow the Rayleigh model

$$p_n(z) = \frac{z}{1+n\lambda} \exp\left(-\frac{z^2}{2(1+n\lambda)}\right), \quad n = 0,1,2,\ldots \quad (4)$$

where λ denotes the signal-to-noise ratio (SNR).

4) Particle Filter:

With the definition of the transitional density $p(X_k,T_k \mid X_{k-1},T_{k-1})$ and likelihood $P(Z_k \mid X_k,T_k)$ the solution to the multitarget tracking problem formally boils down to the BRF (1) and the standard SIS/R PF given above can be applied. However, with large number of target the computation requirements become prohibitive. The first step to improve the efficiency of the multitarget PF is to choose an appropriate importance (proposal) distribution for sampling that takes into account the specifics of the multitarget problem. Along with the SIR algorithm's Kinematic Prior (KP) proposal $\pi = p(X_k,T_k \mid X^i_{k-1},T^i_{k-1})$, where $i-$ particle index, [34] suggested three more sophisticated schemes for choosing the importance distribution for multitarget PF, referred to as Independent Partition (IP), Coupled Partition (CP), and Adaptive Partition (AP) [34]. The second step is to parallelize as much as possible the resulting multitarget algorithms. Next we propose parallel implementation of these schemes and incorporate them in the parallel structures of the corresponding overall multitarget parallel algorithms.

5) JMPD Parallel PF:

In the JMPD SIR PF the proposal is just the kinematic prior and the IS step is completely decoupled with respect to particles $\{X^i\}$. Consequently, both versions of the above generic parallel PF (Tables III & IV) work without any modification. The significantly more efficient proposal schemes IP, CP, and AP of JMPD PF have intrinsic coupling among particles introduced by the dependence of the proposal on the current measurement data. By more careful inspection, however, IP and CP can be parallelized as given next.

Each particle i for T_i targets is $X^i = (x^{i,1},x^{i,2},\ldots,x^{i,T^i_k})$ and $x^{i,j}$ is referred to as a partition j of particle i.

The IS step of the JMPD with IP can be done as follows:

A) For each partition $j = 1,\ldots,T^i_k$ (in parallel)

$$x^{ij}_k, w^{ij}_k = \mathsf{IP}[\{x^{ij}_{k-1}, w^{ij}_{k-1}\}^N_{i=1}, Z_k]$$

[2]Time index k is omitted here to lighten notation.

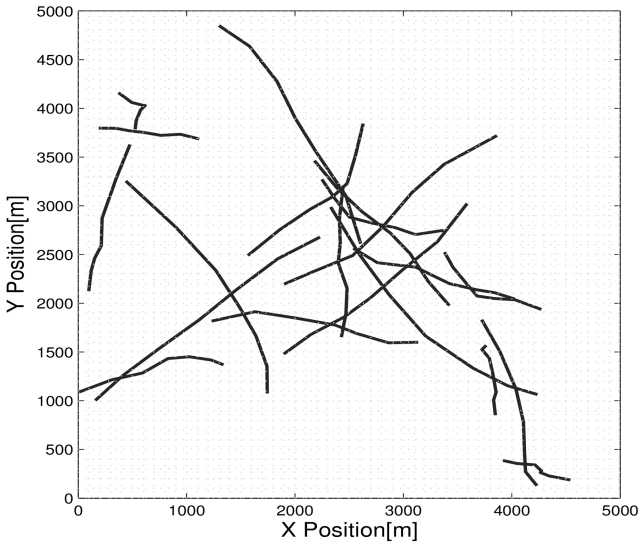

Fig. 4. Scenario with 20 Targets

B) For each particle $1 = 1,\ldots,N$

$$\text{Importance weights } \bar{w}^i_k = w^i_{k-1} \frac{p(Z_k \mid X^i_k)}{\Pi^T_{\theta=1} w^{ij}_k}$$

where IP denotes the IP subroutine of [34] which practically implements the SIR algorithm for each partition. The IS step of the JMPD with CP can be done similarly, except for IP being replaced by a subroutine CP which practically implements the known auxiliary SIR particle filter [2] for each partition but only outputs one resampled partition.

The local importance weights $w^{i,j}_k$ are data dependent and their inclusion in the calculation of importance weights $\bar{w}^i_k$ amounts to improving the proposal π, i.e., bias the proposal towards the optimal importance density $\pi(\cdot,Z_k)$.

Part A) is integrated easily with the RNA resampling scheme used in the PPF (Table III). Part B) can also be computed at the head-node (at the CPU or at the GPU) at the expense of an extra communication between head-node and all threads, or it can be computed locally at each thread but this incurs extra pairwise (thread-to-thread) communication between all threads through the device memory. The latter option is parallel but not necessarily faster due to the communication overhead. In our GPU implementation we use the former option—compute B) at the head-node (implemented alternatively at the GPU or at the GPU).

B. Simulation Experiments & Results

Two simulation experiments were performed using two different hardware computing platforms and parallel algorithms tailored to each one of them, respectively.

Scenarios: For both experiments, the same tracking scenarios were simulated. The parameters of these scenarios are as follows. The targets move in a 5000 m × 5000 m surveillance area. They have a nearly constant

TABLE VI
Hardware Configuration 1

CPU	Model:	Intel Core(TM)2 Duo
	Clock Rate:	1.40 GHz
	Memory:	2.0 G
	Operating System:	Windows 7
GPU	Model:	NVIDIA GeForce 8400M GS
	CUDA Driver:	3.20
	Clock Rate:	0.80 GHz
	Cores:	2 (MP) x 8 (Cores/MP) = 16 (Cores)
	Global Memory:	115M bytes
	Constant Memory:	64K bytes
	Shared Memory:	16K bytes/block

velocity motion, according to the state model (2) with $Q = \text{diag}\{20, 0.2, 20, 0.2\}$. The initial position and velocity of each target, for each Cartesian coordinate x and y are generated randomly from the uniform distributions $\mathcal{U}(0, 5000)$ and $\mathcal{U}(-10, 10)$, respectively. The sensor scans a fixed rectangular region of 50×50 pixels, where each pixel represents a 100 m $\times$ 100 m area on the ground plane. The sensor returns Rayleigh-distributed measurements in each pixel, depending on the number of targets that occupy the pixel according to the measurement model (3). The sensor sampling interval $\Delta = 1s$ and SNR $\lambda = 15$. Scenarios with different number of targets T were simulated. Fig. 4 shows a realization of a scenario with $T = 20$.

1) Experiment 1:

a) Algorithms: Three PFs (CPU-CR, GPU-CR, and GPU-DR)[3] were implemented with different number of particles for each scenario, as follows (see also Sect. III-A).

- *CPU-CR:* CPU performs prediction (draw predicted samples and calculate importance weights) and resampling (using the residual systematic resampling method [2]). This algorithm does not use GPU. It is implemented just for comparison.
- *GPU-CR:* GPU (in parallel) performs importance sampling and CPU performs centralized resampling.
- *GPU-DR:* Both IS & R are performed in parallel on the GPU. DR is as described in Sect. III-A, Table III. After the weights of the blocks are known, the number of particles that each block replicates is calculated at CPU using residual systematic resampling (inter resampling). Finally, intra-resampling is performed inside the block in parallel.

b) Computing Platform: The hardware used in this experiment is presented in Table VI.

Note that there are 8 parallel pipelines and the number of particles chosen $\gg$ number of pipelines. Also, the GPUs has a maximal texture size that limits the number of particles that can be resampled as a single unit (block).

[3]CR stands for Centralized Resampling and DR—for Distributed Resampling

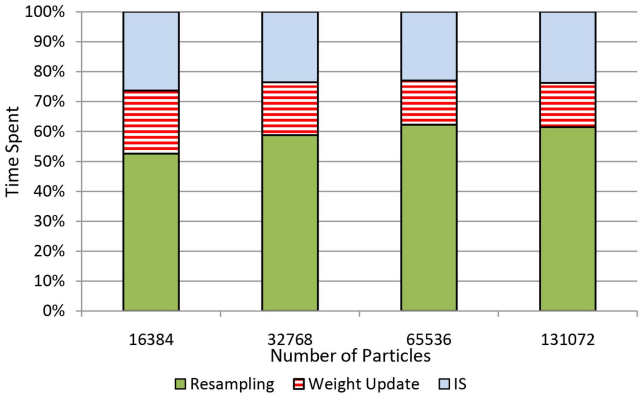

Fig. 5. Relative Times Spent in the Different Steps Using GPU-DR

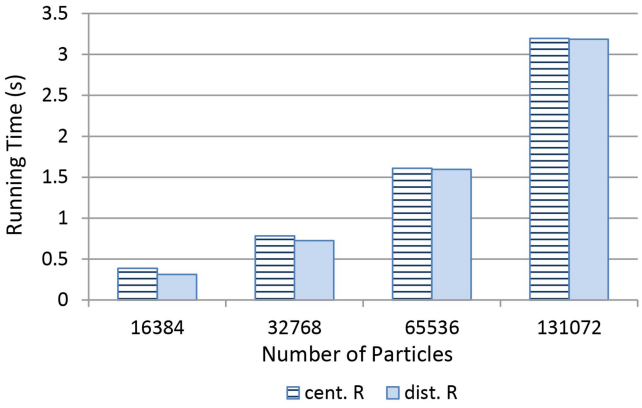

Fig. 6. Computation Times of CR & DR

The code for all implementations is written in C++ and compiled using Visual C++ 2008.

c) Results & Comparison: Due to space limitation, only the most representative results are reported.

First, Fig. 5 shows the computation times spent on different parts of the PF algorithm in the GPU-DR implementation. The resampling step incurs the highest computational cost. Quantitatively, resampling is from two to four times more costly (depending on the number of particles) than importance sampling, and about 2.5 times than generating the estimates.

Second, Fig. 6 shows a comparison between the times for resampling only (CR and DR) with different number of particles. Even though DR is in parallel, the improvement over CR is not significant because the clock rate of GPU is much lower (almost two times) than that of the CPU. Also, as the number of particles increases the efficiency of DR decreases due to the limited pipelines of GPU.

Next, Fig. 7 and Fig. 8 show the position *time-average root-mean square errors* (TARMSE) and execution times of the three PF algorithms for 3 targets, respectively. Fig. 9 and Fig. 10 are for 20 targets. Fig. 7 indicates that, for 3 targets, using slightly more than 60K particles (in all filters) is a reasonable choice for practical purposes. For 20 targets, Fig. 9 indicates that more than 130k particles are needed. These figures also

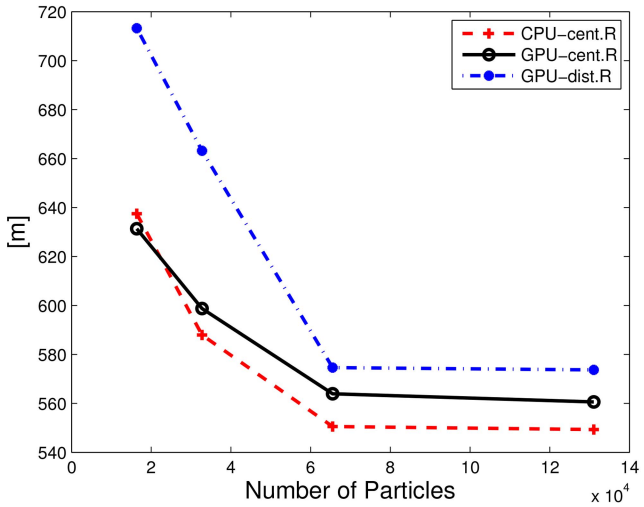

Fig. 7. Position TARMSE; 3 Targets; 100 MC Runs

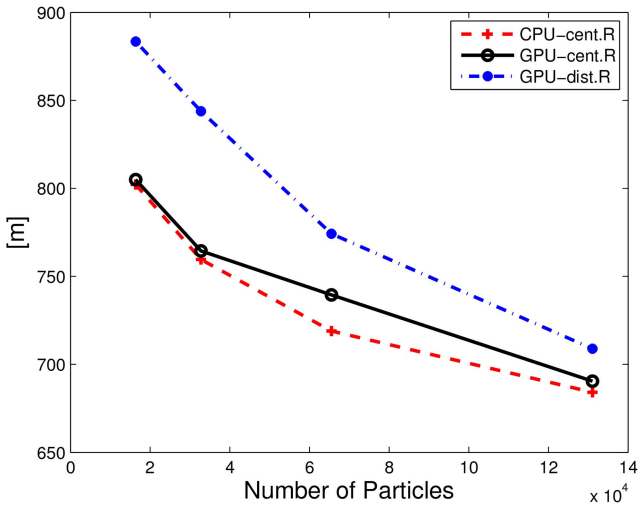

Fig. 9. Position TARMSE; 20 Targets; 100 MC Runs

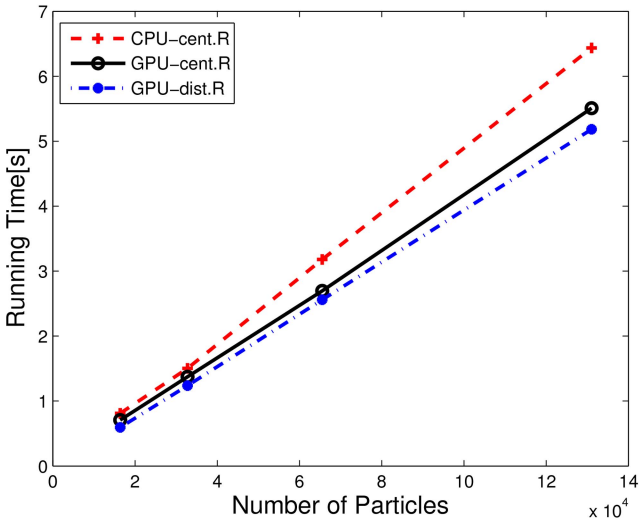

Fig. 8. Average Execution Time; 3 Targets; 100 MC Runs

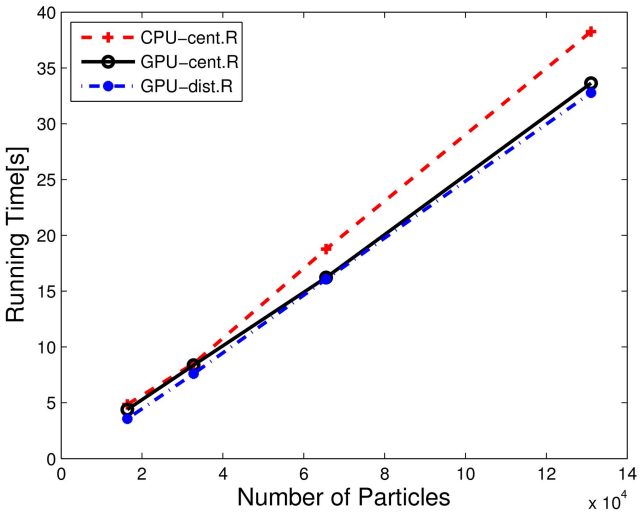

Fig. 10. Average Execution Time; 20 Targets; 100 MC Runs

illustrate that CPU-CR and GPU-CR are better than GPU-DR in terms of accuracy (for the same number of particles). This is clear because CR has better utilization of particles than DR—the former implements the resampling exactly as opposed to the latter which is approximate.

On the other hand, Fig. 8 and Fig. 10 show the execution times for one computational cycle of the tracking filter. Now the order of performance is reversed with CPU-CR being considerably slower than both GPU-CR and GPU-DR (which are close in computation times). Quantitatively (based on all simulations with 130K particles), CPU-CR is about 30% slower than GPU-DR.

The fully centralized algorithm (CPU-CR) is the best in terms of accuracy at a given number of particles but its computation time is worst. The fully distributed algorithm (GPU-DR) has shown the best running time but its accuracy is the worst. The partially distributed algorithm (GPU-CR), for the considered scenarios, has shown a computation time close to that of the fully distributed (GPU-DR) and accuracy not much worse than that of the fully centralized (CPU-CR). It appears that, for the considered hardware configuration, the partially distributed implementation may provide a reasonable tradeoff between filter accuracy and computation time as compared to the other two implementations.

2) Experiment 2:

a) Algorithms: Three PFs (CPU-CR, GPU-DR, and GPU-DR (new)) were implemented with different number of particles for each scenario. CPU-CR and GPU-DR were the same as described in Experiment 1. GPU-DR (new) implemented the novel enhanced PPF proposed in Sect. III-B

- *GPU-DR (new):* Both IS & R are performed in parallel on the GPU. DR is as described for the enhanced PPF in Table IV.

b) Computing Platform: The hardware used in this experiment is presented in Table VII.

c) Results & Comparison: Here we evaluate the novel EPPF proposed in Sect. III-B, Table IV in comparison

TABLE VII
Hardware Configuration 2

CPU	Model:	Intel(R) Core(TM) i5-3210M
	Clock Rate:	2.50 GHz
	Memory:	4.0 G
	Operating System:	Windows 7
GPU	Model:	NVS 5400M
	CUDA Driver:	5.0
	Clock Rate:	0.95 GHz
	Cores:	2 (MP) x 48 (Cores/MP) = 96
	Registers:	32K bytes/block
	Constant Memory:	64K bytes
	Shared Memory:	48K bytes/block

with CPU-CR and GPU-DR, described in Sect. III-A, Table III.

Fig 11 shows the computation time of the CPU-centralized resampling (cent.R), GPU-distributed resampling with RNA (dist.R (RNA)) and our enhanced algorithm (dist.R (new)). It is seen that for different number of particles our updated algorithm has better performance than that of the generic algorithms. The new algorithm is up to 1.5 times faster than dist.R (RNA). Fig. 12 shows relative times spent in the different steps of three PF algorithms. The resampling step of dist.R (new) has almost the same computational cost as the step of importance sampling. It is a significant improvement of the resampling step as compared with cent.R and dist.R (RNA).

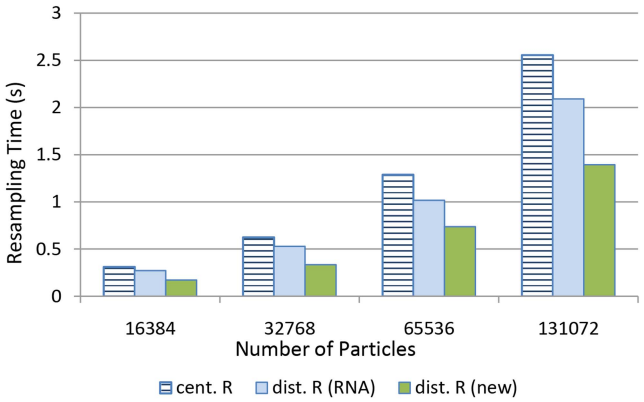

Fig. 11. Computation Times of CR & DR

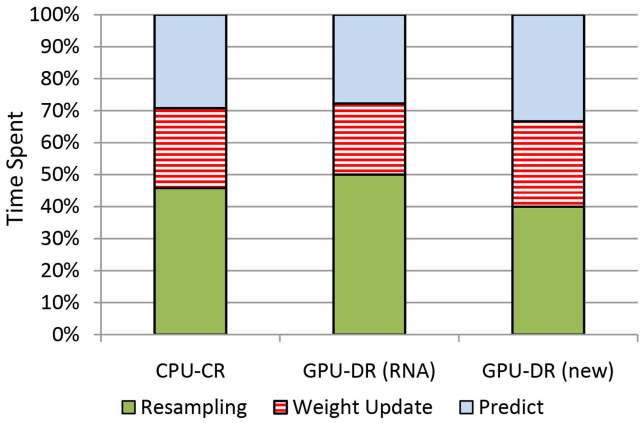

Fig. 12. Relative Times Spent in the Different Steps Using GPU-DR

V. SIMULATION STUDY II: GPU PPF VS. PFF FOR HIGH-DIMENSIONAL FILTERING PROBLEM

The purpose of this simulation study is to evaluate and compare the performances of the GPU-accelerated parallel PF (Table III) and PFF (Table V) for a high dimensional nonlinear filtering problem.

A. Model

We consider nonlinear filtering for the following model with cubic measurement nonlinearities[4]

$$x_{k+1} = \Phi x_k + w_k \tag{5}$$

$$z_k = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \mathsf{x}_{k,1}^3 \\ \mathsf{x}_{k,2}^3 \\ \mathsf{x}_{k,3}^3 \end{bmatrix} + v_k \tag{6}$$

where $k = 0, 1, \ldots$ is time index, $x_k = [\mathsf{x}_{k,1}\ \mathsf{x}_{k,2}\ \ldots, \mathsf{x}_{k,d}]'$ is the state vector of dimension $d \geq 3$, Φ is a positive definite transition matrix that is generated randomly for each scenario in the simulation, $w_k \sim \mathcal{N}(0, 0.04^2 I_d)$ and $v_k \sim \mathcal{N}(0, 1.0^2 I_3)$ are zero-mean Gaussian white process and measurement noises, respectively, and the initial state vector $x_0 \sim \mathcal{N}(0.8, 25000 I_d)$ is randomly generated.

Four scenarios with different state dimension (i.e., $d = 10, 20, 30, 40$), each with different number of particles as specified in Table VIII, are simulated.

B. Algorithms

Implemented are the following four filter algorithms: PFF-CPU, PFF-GPU, PF-CPU and PF-GPU where CPU stands for a sequential implementation (needed for the purposes of comparison). The hardware used in our experiments is presented in Table VII. The code for all implementations is written in C++ and compiled using Visual C++ 2008. The particular configuration parameters of our GPU implementations are given in Table VIII.

C. Performance Measures

In order to obtain statistically significant evaluation of the performance metrics, $R = 50$ Monte Carlo (MC) runs are performed for each scenario.

The filters' estimation accuracy at each time step k is measured in terms of the average dimension-free error

$$e_k = \frac{1}{R} \sum_{r=1}^{R} e_k^{(r)}, \tag{7}$$

[4]The target dynamics need not be nonlinear for the purpose of comparison because both nonlinear filters, PF and PFF, differ only in the measurement update part.

TABLE VIII
GPU Blocks' Specification

Num of Particles =		1536	12288	49152	98304	196608	393216	786432	1572864
XDim = 10	Threads/Block = 256	6	48	192	384	768	1536	3072	6144
XDim = 20	Threads/Block = 128	12	96	384	768	1536	3072	6144	12288
XDim = 30	Threads/Block = 96	16	128	512	1024	2048	4096	8192	16384
XDim = 40	Threads/Block = 64	24	192	768	1536	3072	6144	12288	24576

where

$$e_k^{(r)} = \frac{1}{d}(\hat{x}_k^{(r)} - x_k^{(r)})'(\hat{x}_k^{(r)} - x_k^{(r)}) \tag{8}$$

and $x_k^{(r)}$ and $\hat{x}_k^{(r)}$ are the true and estimated state vectors, respectively, in MC run $r = 1,2,\ldots,R$.

The filters' overall accuracy is measured in terms of the time-averaged error (TAE), defined as follows

$$\varepsilon_{[m,n]} = \frac{1}{n-m+1}\sum_{k=m}^{n} e_k \tag{9}$$

where $[m,n]$ is the time interval of averaging. TAE is sometimes used in target tracking as a single measure of "steady-state" filter accuracy. In the simulation $m = 21$, $n = 50$.

The computational performance of all four filters is measured in terms of average running time per one filter time-step. The computational performance of the parallel GPU filters is measured in terms of speedup with respect to the corresponding CPU sequential filter, i.e.,

$$Speedup = \frac{T_{CPU}}{T_{GPU}} \tag{10}$$

where T_{CPU} and T_{GPU} denote the average running time of the filter (PF or PFF) on CPU and GPU, respectively. The speedup characterizes the scalability of a parallel algorithm.

D. Results

Fig. 13 shows the dimension-free error plots of the filters[5] with 10 dimensional state vectors and 800K particles. It illustrates that PFF-CPU is the best in terms of accuracy (for the same number of particles), followed by PFF-GPU, PF-CPU and PF-GPU. Very similar results were obtained for different state vector dimensions (up to 40) and different number of particles. Based on all results, the GPU versions of both PFF and PF are apparently less accurate than their corresponding CPU versions. This is because the parallel GPU versions are actually approximations of the fully centralized CPU versions. For the PF this approximation is in the resampling step: it is global (uses all particles) in PF-CPU and local (uses only the particles within a block) in PF-GPU. A similar effect happens with the PFF: the computation of $\bar{x}_k$ in PFF-CPU is global (the sample mean of all particles), while the computation of $\bar{x}_{k,b}$ in PFF-GPU is local and consequently less accurate.

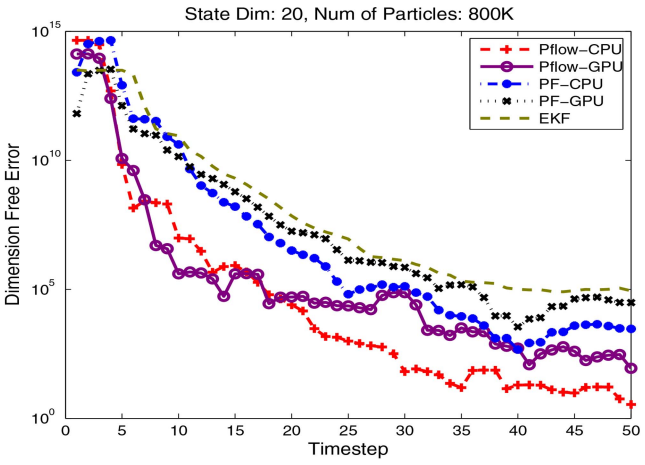

Fig. 13. Dimension-Free Errors

More detailed overall accuracy comparison can be made based on the plots in Fig. 14 (left) that show the time-averaged dimension-free errors of the four filters with different dimensions of the state vector versus the number of particles. The differences in accuracy are quite significant. In particular, PFF-GPU is several orders of magnitude more accurate than both PF-GPU and PF-CPU. Of course, PFF-GPU is less accurate than PFF-CPU (for reasons explained above), but this does not seem significant, given the fact (discussed next) that the former is much faster than the latter.

The plots in Fig. 14 (right) show the average running times—the "prices" paid to achieve the accuracies shown in the corresponding plots of Fig. 14 (left). With the same number of particles, PFF-CPU is about 1.5–2 times faster than PF-CPU (depending on the number of particles), and PFF-GPU is about 4–5 times faster than PF-GPU. Table IX is provided for more accurate comparison. Even though it might seem that PFF-CPU requires more computation time per particle than PF-CPU, it actually appears otherwise. PFF update includes computing $A(\lambda)$, $b(\lambda)$ and solving the ODE via the Euler scheme (Table V). Computing A, b is common for all particles and independent from the number of particles (except for $\bar{x}_k$). Therefore, for large number of particles it has insignificant contribution to the "per particle" computation time. The main portion of computation time per particle is spent on the ODE which is a fairly simple and fast computation for small L. In the simulation $L = 10$ and the time spent on it is up to twice

[5]EKF's error plot is shown as a baseline only and is excluded from further comparison.

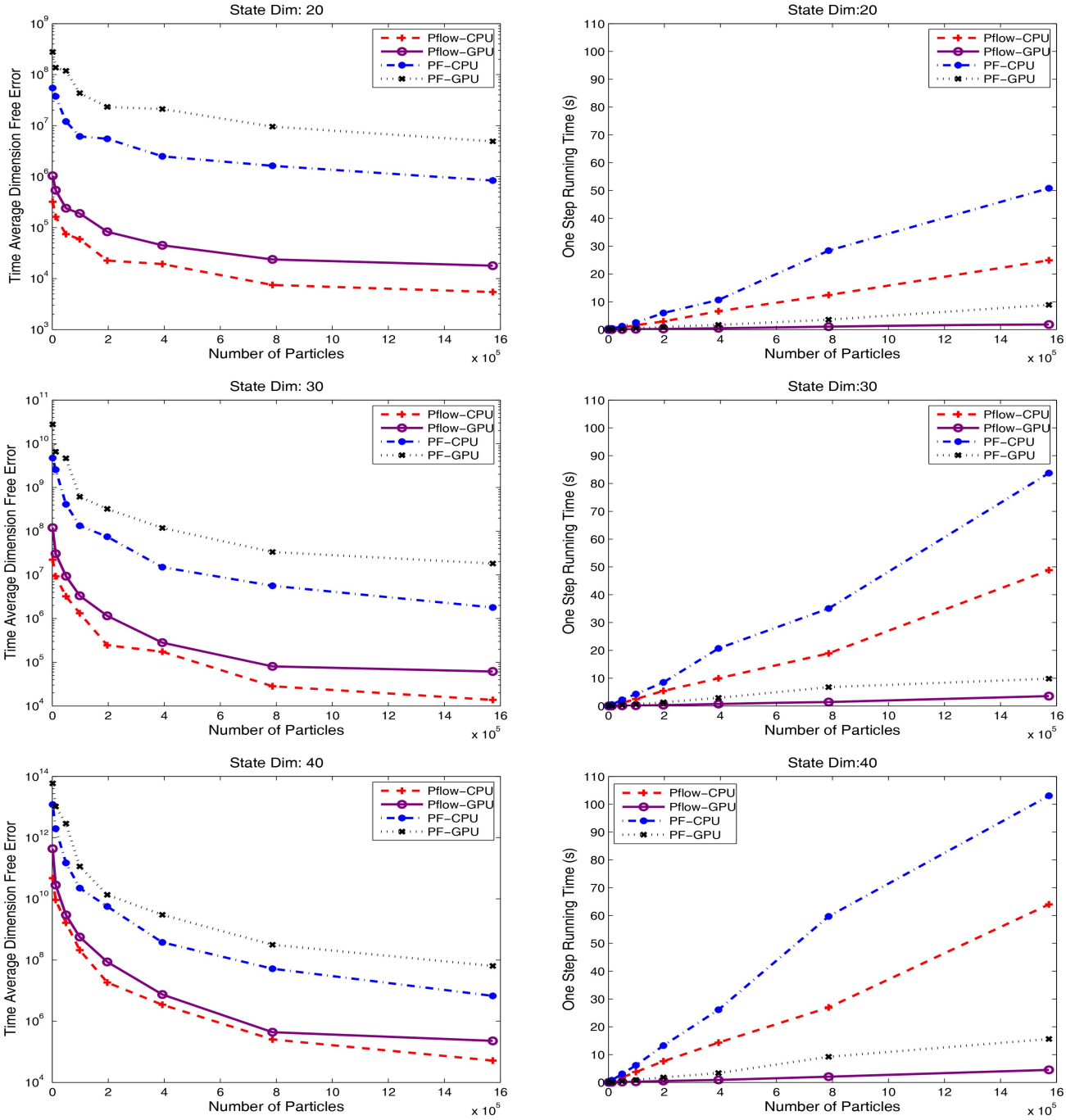

Fig. 14. Time-Averaged Dimension-Free Error (left) & Running Time (right)

smaller than the time PF spends to draw a resampled particle from a large set of particles using stratified sampling. PFF with $L = 5$ and 20 was also run, but $L = 10$ was chosen as the best tradeoff between accuracy and speed. For $L = 20$ the computation times of PFF-CPU and PF-CPU are closer but the advantage of PFF in accuracy increases. The time results regarding PFF-GPU vs. PF-GPU are not surprising given the fact that PFF-GPU is almost completely (thread) parallel while the resampling of PF-GPU is only partially (block) parallel.

Fig. 15 illustrates the effect of the state dimension on the running time with different number of particles of both parallel filters: PF-GPU (left) and PFF-GPU (right).

Fig. 16 shows the speedup of PF-GPU (left) and PFF-GPU (right) with different state dimension and different number of particles. It appears that for both GPU filters the speedup most often increases with the state dimension (for the same number of particles) which supports using GPU for highly dimensional problems. On the other hand, the speedup for both GPU filters does not seem to vary very significantly with the number of particles (for the same state dimension). Finally, Fig. 17 compares the speedup of PF-GPU and PFF-GPU for

TABLE IX
Running Time (s)

Num of Particles =		1536	12288	49152	98304	196608	393216	786432	1572864
XDim = 10	PFF-CPU	0.0099	0.0792	0.3403	0.6682	1.2630	3.1733	5.0851	14.4671
	PFF-GPU	0.0007	0.0054	0.0218	0.0439	0.1000	0.2187	0.4625	1.0493
	PF-CPU	0.0172	0.1418	0.5856	1.2602	2.5559	4.6397	9.1571	17.8682
	PF-GPU	0.0034	0.0281	0.1129	0.2438	0.5337	1.0696	1.7700	5.5668
XDim = 20	PFF-CPU	0.0218	0.1760	0.7338	1.5952	2.9421	6.6210	12.4409	24.9088
	PFF-GPU	0.0014	0.0117	0.0469	0.0989	0.2175	0.4928	1.0343	1.7933
	PF-CPU	0.0382	0.3139	1.2319	2.5249	5.9612	10.6548	28.3821	50.8195
	PF-GPU	0.0063	0.0513	0.2023	0.4362	0.9371	1.6812	3.5717	8.8979
XDim = 30	PFF-CPU	0.0359	0.2925	1.1410	2.4776	5.4436	9.9433	18.9128	48.8515
	PFF-GPU	0.0023	0.0181	0.0722	0.1464	0.2984	0.6978	1.4003	3.5066
	PF-CPU	0.0643	0.5213	2.2102	4.2976	8.4932	20.7041	35.0581	83.7303
	PF-GPU	0.0092	0.0743	0.3132	0.5864	1.3253	2.9221	6.7780	9.8334
XDim = 40	PFF-CPU	0.0520	0.4127	1.7799	3.7476	7.6032	14.2963	26.9597	64.0159
	PFF-GPU	0.0033	0.0272	0.1092	0.2148	0.5102	0.8640	2.0259	4.4173
	PF-CPU	0.0947	0.7756	3.0457	6.1521	13.2333	26.1284	59.6716	103.0515
	PF-GPU	0.0124	0.0996	0.4036	0.8578	1.7792	3.3625	9.2275	15.5961

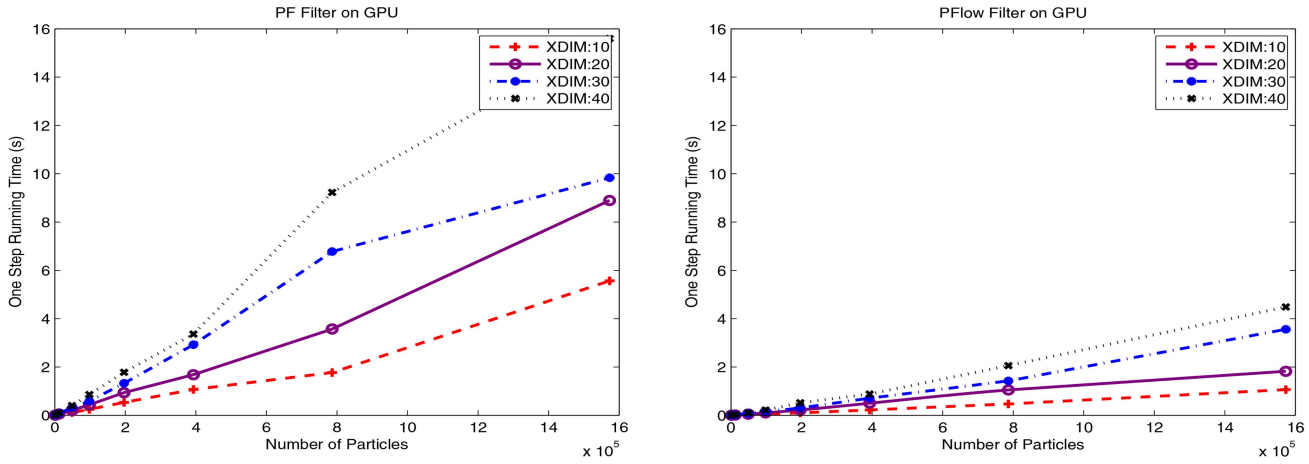

Fig. 15. One-Step Running Time

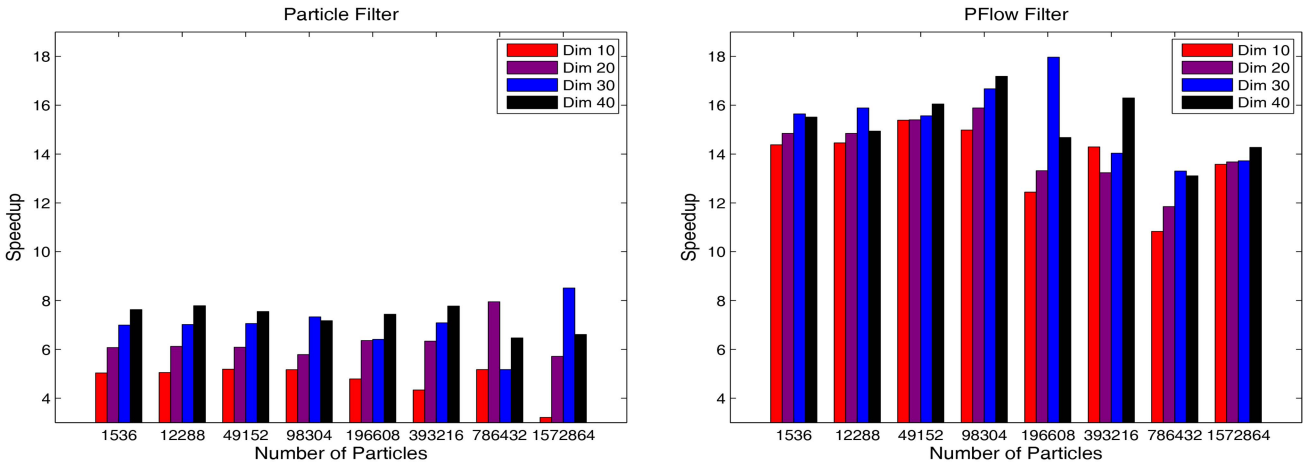

Fig. 16. Speedup of PF & PFF

different number of particles (for state dimension 40). PF-GPU provides a speedup of about six times with respect to PF-CPU and PFF-GPU provides a speedup of about fifteen times with respect to PF-CPU. PFF-GPU outperforms PF-GPU more than two times in terms of speedup due to its higher level of parallelization.

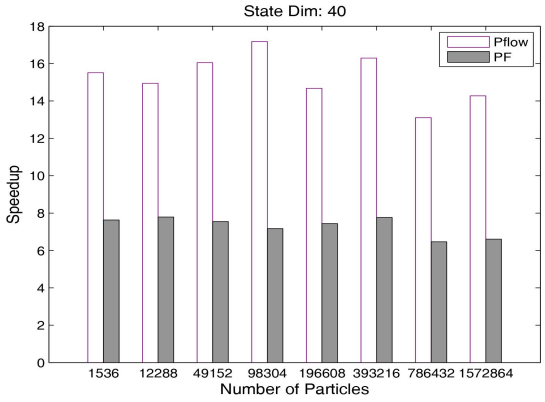

Fig. 17. Speedup of PF *vs.* PFF

VI. SUMMARY & CONCLUSIONS

Three efficient parallel particle and particle flow filter implementations, optimized for GPU architecture, have been proposed and studied. They have been applied and tested, via simulation, for tracking multiple targets using a pixelized sensor, and for a high-dimensional nonlinear density estimation problem.

Overall, the obtained simulation results have demonstrated that using GPU can significantly accelerate the computation of particle filters and particle flow filters through parallelization of the computational algorithms at a tolerable loss of accuracy, and thereby bring them closer to practical applications.

Specifically:

- For the multitarget target tracking problem, the newly proposed Enhanced PPF GPU implementation has shown superior computational performance and the same accuracy as compared to the previous (RNA-based) PPF implementation.
- For the high-dimensional nonlinear estimation problem, the parallel particle flow filter has shown superior performance in comparison with the parallel particle filter implementation in both estimation accuracy and computational efficiency.

ACKNOWLEDGMENT

The authors would like to thank Fred Daum and Jim Huang for providing the relevant papers on particle flow and keeping us updated on their latest results.

REFERENCES

[1] N. Gordon, D. Salmond, and A. Smith
"Novel Approach to Nonlinear/Non-Gaussian Bayesian State Estimation,"
IEE Proceedings-F., vol. 140, no. 2, pp. 107–113, April 1993.

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

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

[4] A. S. Bashi, V. P. Jilkov, X. R. Li, and H. Chen
"Distributed Implementations of Particle Filters,"
in *Proc. 2003 International Conf. on Information Fusion*, Cairns, Australia, July 2003, pp. 1164–1171.

[5] V. Teulière and O. Brun
"Parallelisation of the Particle Filtering Technique and Application to Doppler-Bearing Tracking of Maneuvering Sources,"
Parallel Computing, vol. 29, no. 8, pp. 1069–1090, 2003.

[6] S. Sutharsan, A. Sinha, T. Kirubarajan, and M. Farooq
"An Optimization-Based Parallel Particle Filter for Multitarget Tracking,"
in *Proc. 2005 SPIE Conf. on Signal and Data Processing of Small Targets*, vol. 5913, San Diego, CA, Aug. 2005.

[7] M. Bolic, P. M. Djuric, and S. Hong
"Resampling Algorithms and Architectures for Distributed Particle Filters,"
IEEE Trans. Signal Processing, vol. 53, no. 7, pp. 2442–2450, Jul. 2005.

[8] G. Hendeby, J. D. Hol, R. Karlsson, and F. Gustafsson
"A Graphics Processing Unit Implementation of the Particle Filter,"
in *Proc. 15th European Signal Processing Conference*, 2007.

[9] A. Lee, C. Yau, M. B. Giles, A. Doucet, and C. C. Holmes
"On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods,"
Journal of Computational and Graphical Statistics, vol. 19, no. 4, pp. 769–789, 2010.

[10] G. Hendeby, R. Karlsson, and F. Gustafsson
"Particle Filtering: The Need for Speed,"
EURASIP Journal on Advances in Signal Processing, no. 181403, 2010.

[11] V. P. Jilkov, J. Wu, and H. Chen
"Parallel Particle Filter Implementation on Cluster Computer for Satellite Tracking Application,"
in *Proc. 4th International Conference on Neural, Parallel & Scientific Computations*, Atlanta, GA, USA, August 2010, pp. 179–186.

[12] V. P. Jilkov and J. Wu
"Implementation and Performance of a Parallel Multitarget Tracking Particle Filter,"
in *Proc. 2011 International Conf. on Information Fusion*, Chicago, IL, USA, July 2011, pp. 298–305.

[13] J. Wu and V. P. Jilkov
"Parallel Multitarget Tracking Particle Filters using Graphics Processing Unit,"
in *Proc. 44th IEEE Southeastern Symposium on System Theory*, Jacksonville, FL, USA, March 2012, pp. 151–155.

[14] V. P. Jilkov, J. Wu, and H. Chen
"Performance Comparison of GPU-Accelerated Particle Flow and Particle Filters,"
in *Proc. 2013 International Conf. on Information Fusion*, Istanbul, Turkey, July 2013.

[15] J. Wu
"Parallel Computing of Particle Filtering Algorithms for Target Tracking Applications,"
Ph.D. dissertation, University of New Orleans, 2014.

[16] F. Daum and J. Huang
"Nonlinear Filters with Log-Homotopy,"
in *Proc. 2007 SPIE Conf. on Signal and Data Processing of Small Targets*, vol. 6699, no. 669918, 2007.

[17] F. Daum, J. Huang, M. Krichman, and T. Kohen
"Seventeen Dubious Methods to Approximate the Gradient for Nonlinear Filters with Particle Flow,"
in *Proc. 2009 SPIE Conf. on Signal and Data Processing of Small Targets*, vol. 7445, no. 74450V, 2009.

[18] F. Daum and J. Huang
"Numerical Experiments for Nonlinear Filters with Exact Particle Flow Induced by Log-Homotopy,"
in *Proc. 2010 SPIE Conf. on Signal and Data Processing of Small Targets*, vol. 7698, no. 769816, 2010.

[19] ———
"Exact Particle Flow for Nonlinear Filters,"
in *Proc. SPIE Conf. on Signal Processing, Sensor Fusion, and Target Recognition XIX*, vol. 7697, no. 769704, 2010.

[20] ———
"Exact Particle Flow For Nonlinear Filters: Seventeen Dubious Solutions to a First Order Linear Underdetermined PDE,"
in *Proc. of the 44th Asilomar Conference on Signals, Systems and Computers*, November 2010, pp. 64–71.

[21] ———
"Particle Degeneracy: Root Cause and Solution,"
in *Proc. SPIE Conf. on Signal Processing, Sensor Fusion, and Target Recognition XX*, vol. 8050, no. 80500W, 2011.

[22] ———
"Hollywood Log-Homotopy: Movies of Particle Flow for Nonlinear Filters,"
in *Proc. SPIE Conf. on Signal Processing, Sensor Fusion, and Target Recognition XX*, vol. 8050, no. 80500X, 2011.

[23] X. R. Li and V. P. Jilkov
"A Survey of Maneuvering Target Tracking—Part VIb: Approximate Nonlinear Density Filtering in Discrete Time,"
in *Proc. 2012 SPIE Conf. on Signal and Data Processing of Small Targets*, vol. 8393, Baltimore, MD, USA, Apr. 2012.

[24] M. D. McCool
"Signal Processing and General-Purpose Computing and GPUs [Exploratory DSP],"
IEEE Signal Processing Magazine, vol. 24, no. 3, pp. 109–114, May 2007.

[25] X. R. Li and V. P. Jilkov
"A Survey of Maneuvering Target Tracking—Part VIa: Density-Based Exact Nonlinear Filtering,"
in *Proc. 2010 SPIE Conf. on Signal and Data Processing of Small Targets*, vol. 7698, Orlando, FL, USA, Apr. 2010.

[26] S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp
"A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking,"
IEEE Trans. Signal Processing, vol. 50, no. 2, pp. 174–188, 2002.

[27] O. C. Schrempf and U. D. Hanebeck
"Recursive Prediction of Stochastic Nonlinear Systems Based on Optimal Dirac Mixture Approximations,"
in *Proc. American Control Conference*, 2007.

[28] S. Chikkagoudar, K. Wang, and M. Li
"GENIE: A Software Package for Gene-Gene Interaction Analysis in Genetic Association Studies Using Multiple GPU,"
BMC Research Notes, vol. 4:158, 2011.

[29] NVIDIA
CUDA C Programming Best Practices Guide,
2012. [Online]. Available: http://docs.nvidia.com/cuda.

[30] ———
CUDA CURAND Library,
2010. [Online]. Available: http://docs.nvidia.com/cuda.

[31] W. D. Hillis and G. L. Steele Jr
"Data Parallel Algorithms,"
Communications of the ACM, vol. 29, no. 12, pp. 1170–1183, 1986.

[32] C. Kreucher, K. Kastella, and A. Hero
"Tracking Multiple Targets using a Particle Filter Representation of the Joint Multitarget Probability Density,"
in *Proc. SPIE: Signal and Data Processing of Small Targets*, O. Drummond, Ed., no. 5204, 2003, pp. 258–269.

[33] C. Kreucher, M. Morelande, K. Kastella, and A. Hero
"Particle Filtering for Multitarget Detection and Tracking,"
in *2005 IEEE Aerospace Conference*, March 2005, pp. 2101–2116.

[34] C. Kreucher, K. Kastella, and A. O. Hero
"Multitarget Tracking Using the Joint Multitarget Probability Density,"
IEEE Trans. Aerospace and Electronic Systems, vol. 41, no. 4, pp. 1396–1414, Oct. 2005.

[35] M. Morelande, C. Kreucher, and K. Kastella
"A Study of Factors in Multiple Target Tracking with a Pixelized Sensor,"
in *Proc. SPIE: Signal and Data Processing of Small Targets*, O. Drummond, Ed., vol. 5913, 2005, pp. 155–167.

[36] ———
"A Bayesian Approach to Multiple Target Detection and Tracking,"
IEEE Trans. on Signal Processing, vol. 55, no. 5, pp. 1589–1604, 2007.

[37] C. Kreucher, A. Hero, K. Kastella, and M. Morelande
"An Information-Based Approach to Sensor Management in Large Dynamic Networks,"
Proceedings of the IEEE, vol. 95, no. 5, pp. 978–999, May 2007.

[38] Y. Boers, E. Sviestins, and H. Driessen
"Mixed Labelling in Multitarget Particle Filtering,"
IEEE Trans. on Aerospace and Electronic Systems, vol. 46, no. 2, pp. 792–802, 2010.

Vesselin P. Jilkov received the B.S./M.S. degree in Mathematics from the University of Sofia, Bulgaria in 1982, the Ph.D. degree in the Technical Sciences (Electrical Engineering) in 1988, and the academic rank Senior Research Fellow of the Bulgarian Academy of Sciences in 1997.

From 1982 to 1988, he was a research scientist with the R&D Institute of Special Electronics, Sofia, and, from 1988 to 1999, with the Institute for Parallel Processing, Bulgarian Academy of Sciences. Since 1999, he has been with the Department of Electrical Engineering, University of New Orleans, New Orleans, LA, where he is, at present, an Associate Professor and Riley/Ruby Parker Endowed Professor. His main research interests include stochastic systems, nonlinear filtering, applied decision, estimation & control, target tracking, information fusion, parallel processing.

Dr. Jilkov is author/coauthor of one book and over 100 journal articles and conference papers. He is a member of IEEE, ISIF, and SIAM.

Jiande Wu received the B.E. and M.S. degrees in Computer Science and Technology from the North China Electric Power University, Beijing, China in 1998 and 2005, respectively. He received the M.S. and Ph.D. degrees in Electrical Engineering from the University of New Orleans, New Orleans, LA, USA in 2012 and 2014, respectively.

His research interests include parallel computing, Monte Carlo methods, applied decision and estimation, and nonlinear filtering.