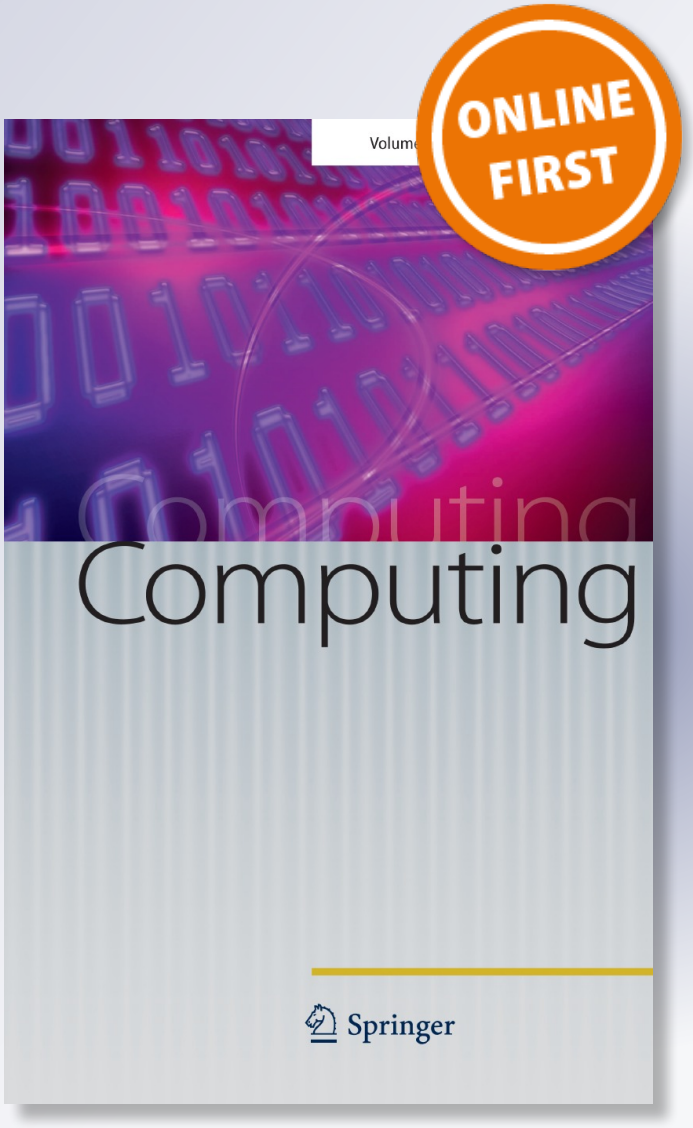
GPU accelerated novel particle filtering method

Subhra Kanti Das, Chandan Mazumdar & Kumardeb Banerjee

Computing
Archives for Scientific Computing

ISSN 0010-485X

Computing
DOI 10.1007/s00607-014-0400-2



Springer

Computing
DOI 10.1007/s00607-014-0400-2

GPU accelerated novel particle filtering method

Subhra Kanti Das · Chandan Mazumdar · Kumardeb Banerjee

Received: 30 August 2013 / Accepted: 10 April 2014

Abstract In this paper, a graphics processor unit (GPU) accelerated particle filtering algorithm is presented with an introduction to a novel resampling technique. The aim remains in the mitigation of particle impoverishment as well as computational burden, problems which are commonly associated with classical (systematic) resampled particle filtering. The proposed algorithm employs a priori-space dependent distribution in addition to the likelihood, and hence is christened as dual distribution dependent (D3) resampling method. Simulation results exhibit lesser values for root mean square error (RMSE) in comparison to that for systematic resampling. D3 resampling is shown to improve particle diversity after each iteration, thereby affecting the overall quality of estimation. However, computational burden is significantly increased owing to few excessive computations within the newly formulated resampling framework. With a view to obtaining parallel speedup we introduce a CUDA version of the proposed method for necessary acceleration by GPU. The GPU programming model is detailed in the context of this paper. Implementation issues are discussed along with illustration of empirical computational efficiency, as obtained by executing the CUDA code on Quadro 2000 GPU. The GPU enabled code has a speedup of 3 and 4 over the sequential executions of systematic and D3 resampling methods respectively. Performance both in terms of RMSE and running time have been elaborated with respect to different selections for threads per block towards effective implementations. It is in this context that, we further introduce a cost to performance metric (CPM) for assessing

S. K. Das (✉)
CSIR-CMERI, MG Avenue, Durgapur 713209, India
e-mail: subhrakanti.das82@gmail.com

C. Mazumdar
Department of CSE, Jadavpur University, Kolkata 700032, India

K. Banerjee
Department of EIE, Jadavpur University, Kolkata 700098, India

Published online: 28 April 2014

the algorithmic efficiency of the estimator, involving both quality of estimation and running time as comparative factors, transformed into a unified parameter for assessment. CPM values for estimators obtained from all such different choices for threads per block have been determined and a final value for the chosen parameter is resolved for generation of a holistic effective estimator.

Keywords Particle filters · Resampling · Dual distribution · Parallel · GPU

Mathematics Subject Classification (2010) 68W10 · 62L20

1 Introduction

Particle filters are numerical approximated Bayesian estimators, working on the general principles of Monte Carlo technique [1–4]. Despite being effective in handling system nonlinearities and non-Gaussian assumptions for noise densities, particle filters are computationally intractable owing to the large number of particles required for reducing the variance of estimation error. Although such a requirement is strictly from the theoretical perspective (following the law of large numbers [5]), practically the need for a large number of samples originates from resampling the current population.

Conventional resampling methods for particle filters [6–9] typically ignore the extraneous bias of likelihood density in generating a new population. The resampled population thus generated contains statistically dependent particles with poor diversity, which in succession demands an exorbitant population size. The aim in this paper remains to introduce a new resampling concept for particle filters. In contrast to systematic resampling, which is performed by considering only the likelihood distribution of the particles, the proposed algorithm employs a priori-space dependent distribution in addition to the likelihood, and hence is the name dual distribution dependent (D3R) resampling method. The so-called priori-space distribution is defined in terms of individual particle measure of spread from the expected mean of the priori population. This in turn is recursively obtained by propagating a posteriori estimated mean from previous sampling instant through the state transition model in the present sampling time. Resampled particles are subsequently obtained from the significant regions of dual distributions thus defined. Furthermore diversity among particles is enhanced with the addition of simulated noise much like in classical roughening [10], with the exception that, the noise distributions are local to each resampled particle and the variance is defined by two replicating samples (nominated by prior and likelihood spaces), instead of one.

However, computational burden is significantly increased owing to few excessive computations within the newly formulated resampling framework. Reducing the sequential time complexity thus involved inevitably leaves scope for necessary parallelization of the proposed method. As obtained from literature, distributed particle filters in particular have been dealt with in different modalities. Distributed particle filtering has been carried out in the works of [11–14], wherein dedicated hardware design and distribution mechanisms have been elaborated. Detailed comparative analysis for distributed resampling schemes evaluated on multi-core processor, has been presented

in [15]. State decomposition as in [16] has also been adopted for decentralizing the estimation process using particle filters, while the approach can be intractable in cases where model equations define coupled relations for state variables. In this regard it may be noted that all such methods rely on distributed architectures which are concurrent rather than strictly parallel. Moreover, such methods are constrained by inter-process communications, as a result of which speedup gets affected by communication delays. Contrastingly, the other aim in this paper remains to present a parallel variant of the proposed resampling method, thereby exploiting the massive parallelism and operational intensity provided by generic graphics processing units (GPUs). Necessary parallel algorithm is developed in CUDA C for its flexibility in programming on GPUs. In the present context of the paper, the parallel methodology thus adopted is detailed with discussions on implementation and computational effectiveness. Although a load balanced GPU computing approach for particle filters has been proposed in [17], the work is sensitively defined only from the perspective of parallel computing architecture. There is little reference to quality of estimation obtained after parallelizing the filter. Proper analysis related to the effect of different CUDA parameters on overall performance of the method is lacking in the context of the referred literature. However, the present paper describes a naive approach towards distributed generation of replication index, obtained purely from algorithmic modifications.

An overall organization of the paper may be presented with Sect. 2 introducing the basics of systematic resampling particle filter, followed by formal elaboration on consequence of likelihood biased resampling for classical particle filters. Subsequently, Sect. 3 describes the proposed D3 resampling method, and comments on computational complexity involved an quality of estimation as obtained with the newly formulated particle filter. The proposed method is shown to outperform systematic resampling in terms of particle diversity and subsequently with low root mean square error (RMSE) of estimation. Section 4 covers up a detailed discussion on parallel resampling with CUDA. With a brief introduction to CUDA framework, the GPU programming model adopted in the context of this paper, is described in this section. Implementation issues are also discussed along with illustration of empirical computational efficiency, as obtained by executing the CUDA code on Quadro 2000 GPU. The GPU enabled code has a speedup of 3 and 4 over the sequential executions of systematic and D3 resampling methods respectively. Performance both in terms of RMSE and running time have been elaborated with respect to different selections for threads per block towards effective implementations. It is in this context that, we further introduce a cost to performance metric (CPM) for evaluating the algorithmic efficiency of the estimator, involving both quality of estimation and running time as comparative factors, transformed into a unified parameter for assessment. CPM values for estimators obtained from all such different choices for threads per block have been determined and a final value for the chosen parameter is resolved for generation of a holistic effective estimator. Finally, Sect. 5 is conclusive of the work presented in this paper.

2 Generic systematic resampling particle filter

In the present section, a generic resampling particle filter algorithm is presented as background knowledge. A population of N particles is initialized with random values

considering an arbitrary initial variance about a pre-designated mean of the state vector X_0, (the population distribution of which is unknown). Weights for all the particles are initialized to $\frac{1}{N}$. Together these form the probability mass function (PMF) of the sample population, after normalization. Given the current measurement Z_t available at a sampling instant t, the steps of the algorithm may be described as follows:

1) FOR i = 1, 2, …, N propagate particles according to the state transition density:

$$P(\tilde{x}_t(i)|\tilde{x}_{t-1}(i)) = f(\tilde{x}_{t-1}(i))$$

END

2) FOR i = 1, 2, …, N compute likelihood weights:

$$w_t(i) = w_{t-1}(i)P(Z_t|\tilde{x}_t(i))$$

END

where $P(Z_t|\tilde{x}_t(i))$ is the observation density, given by

$$\frac{1}{\sqrt{2\pi\sigma_z^2}} \exp - \frac{Z_t - \tilde{x}_t(i)^2}{2\sigma_z^2}$$

3) Normalize the weights so that the likelihood weight form a probability distribution function:

$$w_t(i) = \frac{w_t(i)}{\sum_{i=1}^{N} w_t(i)}$$

4) Calculate the state estimates,

$$E[x_t] = \hat{x}_t = \sum_{i=1}^{N} w_t(i)\tilde{x}_t(i)$$

The state transition model f (.) usually being nonlinear, we assume measurement noise with σ_z^2 as variance. Weight normalization ensures that the aggregate of all the weights is equal to unity thereby qualifying the weight function to represent appropriately the probability mass function for the sampled population. Conditional resampling is performed when the sampled population falls below a measure of degeneracy [18] determined by the effective sample size defined on the basis of coefficient of variation of the weights:

$$N_{eff} = \frac{1}{E[w_t(i)^2]} \leq N$$

Discrete approximation gives estimate of the effective particle size as:

$$N_{eff} = \sum_{i=1}^{N} \frac{1}{w_t^2(i)}$$

Systematic resampling is the simplest and most straightforward method for resampling [19]. In this particular method a random number is drawn from a uniform distribution $U[0, \frac{1}{N}]$ which forms the basis for resampling. The cumulative density of each particle is compared with the chosen random value. The particle having a cumulative density greater than the chosen random value is selected to replicate the original predicted particle. Subsequently, weight of each resampled particle is re-initialized to $\frac{1}{N}$. This ensures an equi-probable distribution of the resampled population. Finally the resampled population consists of few samples with larger weights being replicated a finite number of times and the particles having smaller weights eliminated.

In this section, an implicit shortcoming of conventional resampling method shall be discussed. An introspect into the population distribution at subsequent sampling instants shall be presented to establish the bias of systematic resampling towards likelihood. The internal mechanism of resampling technique may be realized with the following set of derivations: Let $X(i)_0 \sim N(0, Q)$ be the initial distribution of particles, Q being variance for population initialization, and $W(i)_0$ is the initial weight distribution and is equal to $\frac{1}{N}$. This leads to the subsequent propagation of particle values and their weights:

$$X(i)_1 = f(X(i)_0)$$
$$W(i)_1 = W(i)_0 P(Y_1|X(i)_1)$$

Hence the above can be further defined as follows:

$$W(i)_1 = \frac{1}{N} P(Y_1|X(i)_1)$$

Therefore the estimate of the expected mean of posterior density (refer to step 4 in the present section) may be defined as follows:

$$X_1 = \sum_{i=1}^{N} W(i)_1 X(i)_1 = \frac{1}{N} \sum_{i=1}^{N} P(Y_1|X(i)_1) X(i)_1$$

Let us assume, systematic resampling in the present context, generates k surviving particles having higher weights which replicate themselves thereby eliminating the other (N–k) particles of lower weights in the present population of sampling instant t = 1. Evidently each such surviving particle assumes its own unique replicating factor (i.e. the number of particles it is copied into) m_j such that: $\sum_{j=1}^{k} m_j = N$ wherein the surviving particles are $X(j_1)_1 \sim X(j_k)_1$. Since the set of these surviving particles forms the prior population for the next sampling instant t = 2, therefore the set of particles after being propagated through the state model is defined as follows:

$$X(i)_2 = m_1.f(X(j_1)_1), .m_k.f(X(j_k)_1).$$

Following the previous steps as regards computing normalized weights the estimated expected mean of the posterior density for t = 2 may be approximated as:

$$X_2 \approx \frac{1}{N}[m_1.f(X(j_1)_1)P(Y_2|f(X(j_1)_1)) + \cdots + m_k.f(X(j_k)_1)P(Y_2|f(X(j_k)_1))].$$

However, the surviving particles $X(j_1)_1 \sim X(j_k)_1$ form a small set of the prior population of the previous sampling instant which fall within significant regions of the likelihood. Moreover, during the next sampling instant, the weight distribution of the a posteriori particle population becomes largely dependent on the resampled a priori particle distribution. This is because after every resampling the weights of all the particles is re-initialized to $\frac{1}{N}$, which is a constant. Henceforth resampling at each sampling instant leads to likelihood biased a priori population for the next sampling instant. It is beyond doubt that estimation of the posterior mean gets significantly biased towards likelihood progressively. Furthermore, the depletion of particles severely affects the quality of estimation as it reduces the effective population size into a smaller set of particles having significant likelihood. Thus particle impoverishment is introduced. Moreover, such a partial population diverges the posterior trajectory from the true density.

3 Description of D3 resampling scheme

Assuming the posterior density to be asymptotically represented by the Monte Carlo estimate of expectation and furthermore that the posterior distribution is almost invariant in time over a given small sampling period, the estimated expectation of the posterior distribution for the state or parameter from the previous sampling instant, can be reliably considered to be the mean of prior distribution of particles in the current sampling instant. Following this the priori distribution for the present population of particles can be formulated as follows:

$$\mu_k = E[\hat{x}(i)_k^-] \simeq f(\hat{x}(i)_{k-1}^-) \simeq f\left(\sum_{i=1}^{N}[\hat{x}(i)_{k-1}^-.w(i)_{k-1}]\right) \quad (1)$$

$$Var[\hat{x}(i)_k^-] = \sigma_k^2 = \frac{1}{N-1}\sum_{i=1}^{N}(\hat{x}(i)_k^- - \mu_k)^2 \quad (2)$$

The probability mass function following a Gaussian distribution can be formally defined in terms of normalized priori weights as:

$$wp(i)_k = \frac{wp(i)_k}{\sum_{i=1}^{N}(wp(i)_k)} \quad (3)$$

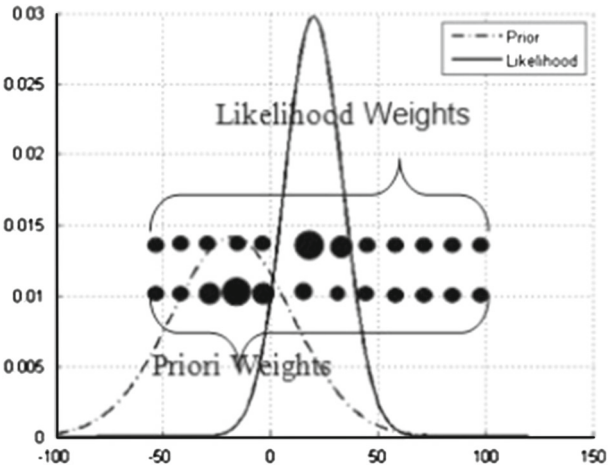


Fig. 1 Typical probability spaces for prior and likelihood distributions

where the unnormalized individual particle priori-weights $wp(i)_k$ are defined using a Gaussian model as follows:

$$wp(i)_k = \frac{1}{\sigma_k\sqrt{2\pi}} \exp^{-} \frac{(\hat{x}(i)_k^- - \mu_k)^2}{2\sigma_k^2} \tag{4}$$

Such a definition is instrumental in determining significant and weak regions of the prior population distribution even in the absence of any knowledge about the associated process noise. Figure 1 is illustrative of a physical relaization of priori as well as likelihood weights for a typical distribution having its likelihood falling in non-significant region of the prior.

3.1 Mechanism for D3 resampling method

In order to formally describe the proposed algorithm let us consider a snapshot of the population having N particles during resampling. Using a systematic approach we select a uniform random variable $u1 \sim U(0, \frac{1}{N})$, and increment the same by $\frac{i-1}{N}$ for each ith iteration over the population (i.e. for each ith particle x(i)). Abiding by the first criterion for resampling (i.e. selecting a high likelihood particle), let x(j) be the particle for which the CDF (approximated by the cumulative weight of x(j)) is just greater than u1. Subsequently according to classical resampling approach, x(j) needs to be copied into x(i). However at this point, the proposed approach differs from the conventional one.

Let us consider the sub-population of particles $x(m)|m = i, i+1, \ldots, j-1, j$. As shown in Fig. 2, the particles within this subset can probabilistically have some particle(s) having likelihood weights greater than that of the selected particle x(j). Let x(l) be such a particle with maximum weight in the above defined subset. Corresponding to the same subset let us define x(p) to be the particle having maximum priori

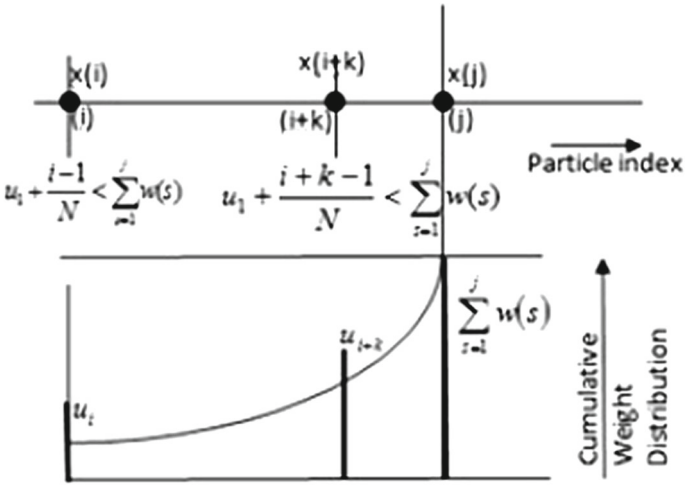


Fig. 2 Illustration of the distribution of u_i for particle indices (i) and (i+k). The particles are shown as spots on the index axis

weight (which is as defined in the previous sub-section). However, since x(i) falls in the significant region of the likelihood it can very well be pushed to the significant region in the prior instead of blind replacement by x(j). Contrastingly if $l \neq j$ then x(j) can be pulled in towards more significant region of likelihood. The new particles thus generated are defined as follows:

$$x(i) = x(p) + r_i, r_i \sim U(0, |x(i) - x(p)|) \tag{5}$$

$$x(j) = x(l) + r_j, r_j \sim U(0, |x(j) - x(l)|) \tag{6}$$

Equations (5) and (6) defined above imply the obvious exception from classical roughening. Significant regions of prior are preserved in (5), by randomly warping the replicated particle towards peak prior space using a uniform distribution. The uniform distribution is defined by difference (in state-space) between the replicated particle and one with highest priori weight in the subset. Contrastingly in (6), the replicating particle thus selected from systematic resampling is pushed towards significant likelihood. In this regard, the measures of spread for the resampled particles may be elaborated as follows:

$$E[x(i)] = x(p) + E[r_i] = x(p) + \frac{|x(i) - x(p)|}{2} \tag{7}$$

$$Var[x(i)] = x(p)^2 + Var[r_i] = x(p)^2 + \frac{(x(i) - x(p))^2}{12} \tag{8}$$

$$E[x(j)] = x(l) + E[r_j] = x(l) + \frac{|x(j) - x(l)|}{2} \tag{9}$$

$$Var[x(j)] = x(l)^2 + Var[r_j] = x(l)^2 + \frac{(x(j) - x(l))^2}{12} \tag{10}$$

The underlying objective behind the proposed method remains in maintaining the statistical independence among the resampled particles together with distinctly identified significant regions of both the prior distribution as well as likelihood induced distribution.

3.2 Algorithmic illustration and estimation quality

A formal algorithm may be presented in the light of the proposed method of resampling. Since the D3 resampling particle filter does not differ from the mainstream estimation method of particle filtering, we restrict to the description of only the resampling part. Structurally the proposed algorithm differs from the classical counterpart in the introduction of a third inner loop for determining indices for particles having highest priori as well as likelihood weights as shown in Table 1. A closer look at the innermost loop reveals that (j–i) iterations are performed, thereby making the total number of iterations O(N(j–i)). Let us consider a limiting case where j = N.

Since the cumulative weight for all the particles is unity therefore CSW[N] = 1. This makes j to be chosen for all particle indices i from i = 1 to N. the total number of iterations subsequently becomes N(N–1)/2, which gives an asymptotic complexity $O(N^2)$ to the proposed algorithm.

In this section, we take a small instance for testing the proposed method. As for the problem, we need to estimate the velocities of an unmanned underwater vehicle in the global frame of reference, considering inputs (accelerations and angular rates) obtained in the vehicle frame, through some INS (inertial navigation system). The state model in a very compact form (for details see [20]) may be described as follows:

$$\dot{X}_{t+1} = \dot{X}_t - rt\dot{Y}_t + f_x cos(\psi_t)t - f_y sin(\psi_t)t \tag{11}$$

$$\dot{Y}_{t+1} = \dot{Y}_t - rt\dot{X}_t + f_x sin(\psi_t)t - f_y cos(\psi_t)t \tag{12}$$

where the left hand side variables in (11) and (12) are global frame velocities, and f_x, f_y and r are the accelerations and heading rate obtained in the body-fixed frame of reference. Measurements arising from an acoustic navigation aid like Doppler velocity log (DVL) sensor are simulated with Gaussian noise having typical mean of 0.0123 m/s and variance as 0.151. The simulation is run with a population size of 50,000 particles over duration of 360 s with a simulated sampling time of 0.01 s. The choice of such simulation parameters is justified by realization of the overall simulation within a decent computational run-time.

As shown in Fig. 3, systematic resampling particle filter suffers from significant root mean square error (RMSE) of estimation as 2.9535 m/s. The quality of estimation degrades progressively after t = 200 s, with error growing exponentially. Contrastingly, by referring to Fig. 4, we find that estimate obtained by using the D3 resampling algorithm has a comparably low RMSE of 0.2681 m/s. The estimator is consistent with minimal bias 0.025 m/s and convergent to the true state progressively. This ensures the runtime efficiency of the proposed algorithm over the classical resampling particle filter. Discrepancy in the quality of estimation between the two PF estimators

Table 1 Pseudo-code for D3 resampling method

```
u1  U(0,1/N)
j = 1
FOR i = 1 to N
                    u = u1 + (i–1)/N
                    WHILE (u greater than CSW[j])
                    j = j + 1
                    END
                    maxlw = w[1]
                    maxpw = pw[1]
                    p = 1
                    l = 1
                                        FOR m = i to j
                                        IF (w[m] greater than maxlw)
                                        maxlw = w[m]
                                        l = m
                                        END
                                        IF (pw[m] greater than maxpw)
                                        maxpw = pw[m]
                                        p = m
                                        END
                                        END
                    DP = x[p] – x[i]
                    DL = x[l] – x[j]
                    x[i] = x[p] + U(0,DP)
                    x[j] = x[l] + U(0,DL)
                    w[i] = 1/N
END
```

lies mainly in the difference of particle diversity maintained by the two resampling schemes. Since quality of estimation falls closely after t = 200 s for the classical resampling filter, we inspect a snapshot of the particle distribution after resampling is done at t = 200th second. From Fig. 5 it is evident that the systematic resampling fails to maintain observable diversity among the particles. As evident from the figure, posterior mean is effectively approximated by only the first 50 particles, which accounts for only 10 % of the overall population. However, particles generated from D3 resampled distribution are subject to greater variations and the effective population size is almost the entire set of particles, as illustrated in Fig. 6. The variation in values are of the order $O(10^{-2})$ in comparison to the order of variation $O(10^{-3})$ in the case of systematic resampling, thereby reducing particle depletion. Taking a different note on the computational effectiveness of the algorithm, it is well understood that runtime efficiency of D3 resampling is lower than that of classical systematic method. As a result, the other aim in this paper remains to achieve for the proposed method,

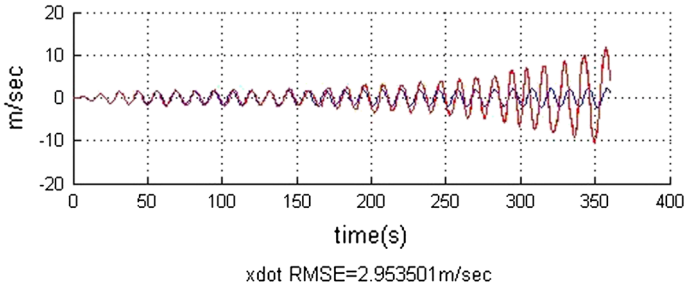


Fig. 3 State estimates given by systematic resampling particle filter; *red lines* represent the estimated state (velocities in X and Y axes of Earth Referenced Frame)whereas *blue line* represents true state (color figure online)

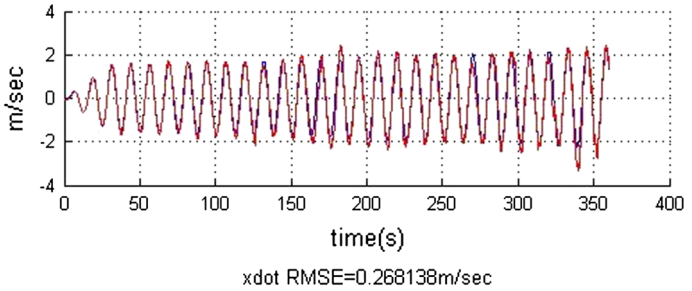


Fig. 4 State estimates given by D3 resampling particle filter; *red lines* represent the estimated state (velocities in X and Y axes of Earth Referenced Frame) whereas *blue line* represents true state (color figure online)

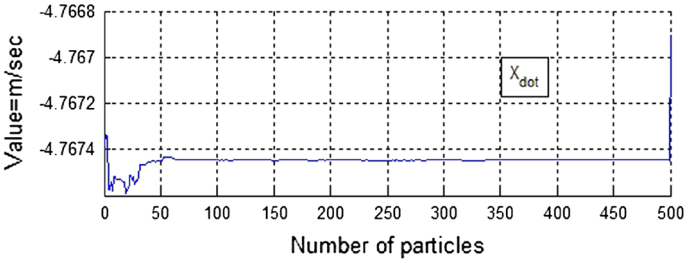


Fig. 5 Particle values from systematically resampled distribution at t = 200 s

greater computing effectiveness without sacrificing quality of estimation. Subsequent sections present an implementation model using GPU computing towards reducing runtime overhead of D3 resampling. In the present context a new metric is formulated (and presented towards the end of this paper) in order to assess overall algorithmic quality as regards both runtime and estimation accuracy.

4 Code acceleration on GPU using CUDA

CUDA stands as an acronym for compute with unified device architecture and has been introduced by NVidia as a C/C++ based application programming framework

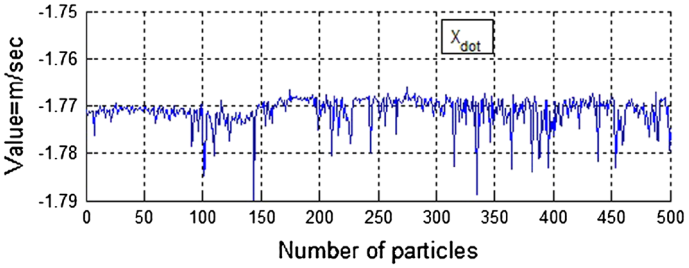


Fig. 6 Particle values from D3 resampled distribution at t = 200 s

for a particular class of NVidia GPUs. CUDA offers a scalable parallel programming environment essentially following the SIMD model. Parallel computing is realized through different levels of abstraction like dividing a problem into few subproblems and executing each sub-problem in parallel by few blocks of threads whereas sub-dividing each sub-problem into smaller units which are executed cooperatively by threads within each block [21,22]. Figure 7 illustrates a typical CUDA code snippet thereby demonstrating the abstraction of threads into blocks and grid. The organization of CUDA threads is essentially realized in the way they access various units of the memory hierarchy as defined within the CUDA model. The device memory consists of the following memory units:

1. Global memory, which is accessible by a grid as well as threads in different blocks. This memory unit is also accessible by the CPU (usually termed the host in the CUDA context). Therefore data transfer between the device and the host is largely achieved through the global memory.
2. Constant memory, which is accessible by all threads in a grid with a read only access.
3. Shared memory, which is shared by all threads executing within a single block context. Hence, a variable declared as shared in a kernel has duplicate copies for each block in execution.
4. Private memory, which is privately accessed by each unique thread in execution within a single block. Hence, a variable declared as private in a kernel has duplicate copies for each thread in execution within a single active block. Private memory consists of the registers available with a multiprocessor.

In the present section, we shall describe in details the overall partitioning of the sequential D3 resampling algorithm into parallel units of execution, along with a discussion on requisite issues for implementation. Computational performance and estimation of quality for the GPU version are elaborately discussed. Finally, towards the end of this section, a computational metric is proposed to assess the overall algorithmic effectiveness of the GPU version over the sequential one considering both estimation quality and CPU time.

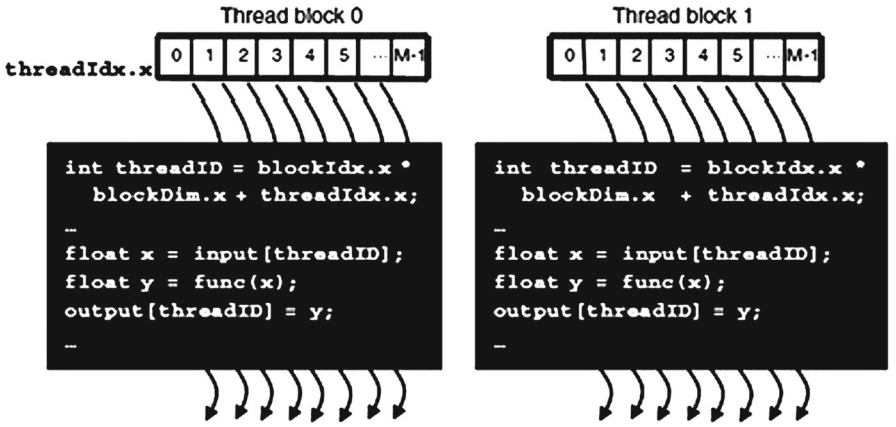


Fig. 7 Overall thread abstraction; the threadID is represented with reference to the blockID as well as gridID

4.1 Parallelization methodology

The overall parallelization for the proposed particle filter algorithm is described as a step-by-step partitioning of sequential computations into parallel blocks of thread level execution.

1. Propagation of particles according to state transition density is carried out using N threads representing N particles. The data parallel model may be defined as follows:

FOR ALL i=1∼N (CUDA Threads)
$\tilde{x}_k(i) = f(\tilde{x}_{k-1}(i))$
END

2. Computation of likelihood weights for N particles is performed by each of N CUDA threads independently and asynchronously:

FOR ALL i=1∼N (CUDA Threads)
$w_k(i) = w_{k-1}(i).P(\tilde{x}_k(i)|z_k)$
END

where $P(\tilde{x}_k(i)|z_k)$ shall usually be defined considering a Gaussian form of measurement noise. Refer to step (2) as defined in Sect. 2.

3. The estimation step is implicitly sequential operation as defined in Eq. (13). The sequential process needs to be carried out by a single CUDA thread which may be treated as the *master* while others remain idle throughout the former's execution period. However, it may be transformed in a way such that the computational load for the *master* thread gets reduced.

$$\hat{x_k} = \sum_{i=1}^{N} \tilde{x}_k(i) W_k(i) \tag{13}$$

$$\Rightarrow \hat{x_k} = \sum_{i=1}^{N} \tilde{x}_k(i) \frac{w_k(i)}{\sum_{i=1}^{N} w_k(i)} \tag{14}$$

$$\Rightarrow \hat{x_k} = \sum_{i=1}^{N} \frac{\tilde{x}_k(i) w_k(i)}{\sum_{i=1}^{N} w_k(i)} \tag{15}$$

From a closer look at the transformations (14) and (15) of Eq. (13), it may be realized that the state estimate $\hat{x}_k$ can be obtained in 2N–1 Floating Point operations, with N–1 additions toward the sum of products of $\tilde{x}_k(i)$ and $w_k(i)$, N–1 additions of N un-normalized likelihood weights (in order to find out the sum total of all weights for normalizing each individual particle's weight), followed by a single division operation. This is invariably comparable with the computational requirements of step (1) which involves N–1 additions for finding normalized weights, N multiplications of $\tilde{x}_k(i)$ and $w_k(i)$, and N–1 additions of the N products $\tilde{x}_k(i).w_k(i)$, totaling to 3N–2 operations. Therefore, the operational transformation from Eq. (13) to Eq. (15) reduces the computational load by an order of O(N).

4. Finally, resampling may be summarized as a sequence of the following sequential operations: a. Computing the uniform random variable

b. Computing CDF for N particles over the entire likelihood weight space.

c. Iterating over the CDF in order to find out the replication index j whose CDF immediately exceeds the value of the sampled random variable.

d. Select particles having indices m and l respectively, between indices i and j, such that x(m) and x(l) have the maximum priori and likelihood weights respectively.

e. Update x(i) and x(j) with x(m) and x(l) respectively along with additive noise as defined in Eqs. (5) and (6) for D3 resampling algorithm described in Sect. 3.1. In the present context of the paper, following modifications have been carried out with the sequential code of D3 resampling Particle Filter, in order to achieve computational effectiveness and correctness with GPU execution. Following steps elaborate the transformations of each sequential operation involved in resampling.

Step (a): Computing the uniform random variable: Let us look at the value assignments for the random variable u(i) in successive iterations during resampling. $u(i) = u(0) + \frac{i-1}{N}$ Where, u(0) is the initial random value U(0,1/N) which is systematically updated in each subsequent stratum. Now, for N concurrent threads, the increment becomes trivial with a prior initialization of u(0) as a shared variable within each block. Each thread maintains a private copy of the random variable allowing for asynchronous and independent update of each stratum. Contrastingly, a global allocation of u(0), enforces all the threads to synchronize with the thread initializing the variable, such that subsequent stratified sample values can be defined by each thread consistently. Hence shared memory approach eliminates un-necessary global synchronization across blocks (Table 2).

The above implementation allows for parallel stratified sampling of the variable u. However, the above defined implementation becomes non-trivial with multiple CUDA blocks which can be attributed to the limited scope pf the shared variable u(0) is defined only within block 0. Hence for effective stratification by t threads /block we introduce a quasi initialization approach within each block coded as shown in Fig. 8. u(0) is

Table 2 Local initialization of 'u'

Thread (id)	Scope of 'u'	Value of 'u'
Thread (0)	Shared	$U(0, \frac{1}{N})$
Thread (i)	Private	$u(0)+\frac{1}{N}$
.	.	.
.	.	.
.	.	.
Thread (N–1)	Private	$u(0)+\frac{N-1}{N}$

```
if(tid==blockIdx.x*blockDim.x){  // thread 0 of each block
      float temp=1/(float)N;
      u1=(((temp/2)+(temp/2)*curand_uniform(&state))/N)/2+((tid/N)/2);
      u=u1; // copying u1 into shared variable u
}
__syncthreads();// all threads in a block wait for thread 0 to complete
                  initialization of u1
if(tid<N){
      if(tid!=blockIdx.x*blockDim.x){
            u=(u1/N)/2+((tid/N)/2);  // all threads update u in their
                                        private scope
      }
}
```

Fig. 8 Quasi-initialization of random variable 'u' in the private scope of each thread using shared variable 'u1' defined by thread 0 of each block

represented by u1 and u(i) by u defined in the private scope of each thread. Since value of u1 is not defined across multiple blocks, therefore thread (0) of each block initializes u1 as follows:

$$u1 = U\left(0, \frac{1}{N}\right) + \frac{blockid * tpb}{N}$$

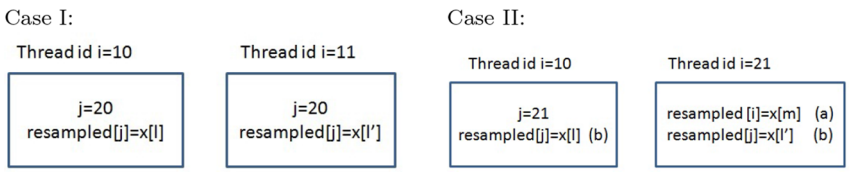
The above initialization by a particular thread is a close approximation to the initial value of u1 defined within the scope of block 0 and subsequent stratification by thread id = blockid*threads/block. However, what follows is an error between the locally stratified u1 within each block and u1 as defined in the context of thread (0) within block 0, which may be defined as proportional to the second order moment of the variable u1:

$$u_{error} \propto Var[u]$$

Since u1 is uniformly distributed, therefore the error also has the form of uniform distribution, which redefines the previous proportionality as follows:

$$\Rightarrow u_{error} \propto \frac{1}{12N^2}$$

Table 3 Execution snapshot of two distinct threads within the same block



Now for a large N, the error approximately degenerates to 0.

$$\lim_{N\to\infty} u_{error} = 0$$

Hence for large values of N, the quasi-initialization valid within each block of CUDA threads, approaches global initialization which is valid across the entire grid.

Steps (b) and (c)

For the conventional sequential resampling algorithm CDF for the particle-weight space is computed as follows:

$$CDF[i] = CDF[i-1] + w[i]$$

The loop carried data dependency in the above sequential code inevitably reduces the overall speedup of the parallel program according to Amdahls Law [23]. However from step (c) it may be realized that the sole purpose of computing a CDF is to find out a particle with replication index j such that u for a particular thread having id i is immediately lesser than CDF(j). For an unequally distributed likelihood weight space W(0,…, N), the comparison with u may not extend up to the Nth CDF in which case, a significant computing effort is wasted in finding the full CDF = $\sum_{i=1}^{N} W(i)$. Contrastingly, we propose a local CDF computation within each thread context, wherein the CDF can be conditionally computed as follows, wherein thread i stops computing CDF as it reaches index m which necessarily has a higher value than the locally defined u, within the concerned thread's context.

$$CDF(i) = \sum_{m=1}^{N} W(m_i)$$

The proposed approach enforces all the threads to iterate over the global likelihood weight space, constrained only by the value of u defined in their individual contexts. Thus load on a single thread in computing the entire CDF is distributed among all the threads through local CDF computations, which results in load-balanced load based concurrent computation. Step (d) is carried out subsequently with $|j - i|$ iterations for each thread context.

From the structure of the code as defined at step (e), it may well be observed that parallel update for each x_k[i] (i being a particular thread context), with x_k[m] is in

direct contention with the update of $x_k[m]$ by thread having id m. Hence, this may lead to a race condition between the threads and the update might remain inconsistent. the inconsistency may be resolved by the introduction of a global array resampled [0,…,N], as follows.

Step (a): resampled [i] = $x_k[m_i]$.

Step (b): resampled [j] = $x_k[l_i]$.

While the first assignment is thread-safe due to non-overlapping write operations, data-parallel execution of the latter is non-trivial. This because of the nondeterministic index j. An example as shown in Table 3, instantiates the contention involved.

Case I highlights two uniquely identified threads within the same block defining the same replication index j. Hence, both of them intend to update resampled[j] which leads to a concurrent write and hence a race condition. Thread id = 10 and 11 are taken as instances.

Case II exhibits a situation, wherein one thread (id = 10) has already finished executing step (a) and is about to execute step (b), whereas the other (thread id = 21) is at the execution of step (a). Since, j defined in the context of thread 10 coincides with i defined in the context of thread 21, therefore the two threads are in contention with each other in executing steps (a) and (b). The contention of Case II may be explained with the help of an example. Let us assume a thread having an identifier such that it is located close to the boundary of its block (as defined by the number of threads/block). From an understanding of how systematic resampling works, the thread is probable to find some j value which coincides with the identifier of some thread within the context of adjacent block. Since threads are not synchronized across blocks in the CUDA framework, the threads in our example are in contention with respect to the execution of step (a) in one's own context and step (b) in the other's. Hence race condition as shown in Case II is also possible for threads across blocks.

The problem as depicted in Case I may be solved by using a "quasi synchronization" approach, wherein the thread id with j is expected to execute step b. Since, both the indices j and l are independent of the thread context i and both the variables x_k and resampled are global therefore thread j can independently and asynchronously execute steps (b) defined in the context of both threads 10 and 11. Since, thread j is the only thread allowed to execute step (b) defined within any other thread context, steps (a) and (b) remain sequential for thread j and hence are definitely synchronized. Contrastingly, problem defined in Case II demands the completion of step (a) before the execution of step (b) begins. This may easily be achieved by synchronizing the threads within a block. For a look into the synchronization code we can refer to Fig. 9.

While contention in Case I is resolved by quasi-synchronization whereby only thread identified with j is enforced to execute step (b), contention in Case II can be easily resolved by putting barrier synchronization after step (a) within the context of a thread. This shall synchronize all the threads within a block before execution of step (b). Thus completion of step (a) is ensured before execution of step (b) for all threads within a particular block. However, the quasi-synchronization mechanism works even for avoiding race condition of Case II among threads across blocks. This can be realized as step (b) being executed by a unique thread having identifier j, which in turn

```
if(tid<N){
  //resample for xdot
  if(tid!=blockIdx.x*blockDim.x){
        u=(u1/(float)N)/2.0+(float)((tid/(float)N)/2.0); //all threads update u in their
        private scope
  }
        ...
  resampled[tid]=x_k[m]+(diffi/2.0)+((diffi/2.0)*curand_uniform(&state));
}
__syncthreads();// wait till all threads in a block write to resampled []
if(tid==j){
  // only the thread identified as 'j' shall commit
  resampled[j]=x_k[l]+(diffj/2.0)+((diffj/2.0)*curand_uniform(&state));
}
```

Fig. 9 Synchronization of threads within the same block for update of 'resampled[i]' and 'resampled[j]' where 'i' is the same as 'tid' standing for thread id

is carried out only after step (a) has been executed by that particular thread. Hence, quasi synchronization along with intra-block synchronization is instrumental towards resolving both the cases of contention that can possibly occur during resampling.

4.2 Concern for implementation

Performance of a GPU within a CUDA framework depends largely on the effective thread level parallelism obtained by issuing each instruction to a single warp (consisting of 32 threads). Since the main execution units in a CUDA architecture are warps, therefore it is extremely important, that at any point of execution, the multiprocessor should have a maximum of warps that it can sustain. This is primarily because latency hiding can be achieved with a large number of active warps, i.e. the entire execution becomes transparent to the waiting of few for some resources. In this context, it may be realized that concurrent execution is achieved within a block of threads and true parallelism is obtained often at the block level, since multiple blocks can be scheduled on different multiprocessors. Therefore an increase in block size can be beneficial to GPU performance, wherein multiprocessor occupancy increases. Contrastingly, with a further increase in the number of blocks, overall performance can degrade due to excess blocks waiting to get allocated to a multiprocessor. Hence there should be an optimum choice for number of threads and consequently the number of blocks. A blindfolded increase in the number of threads cannot be a good choice for speedup since the threads are subject to contention among themselves regarding resources like registers and shared memory. In the present context, multiprocessor utilization (as a factor contributing in overall performance) is evaluated using the system model as provided in [24].

The total number of warps W_{block} in a block is as follows:

$$W_{block} = \left\lceil \frac{T}{W_{sz}} \right\rceil$$

where, T is the number of threads per block, W_{sz} is the warp size, which is equal to 32

The total number of registers R_{blk} allocated for a block is as follows:

For devices of compute capability 1.x:

$$R_{block} = \left\lceil \frac{\left\lceil \frac{W_{block}}{G_W} \right\rceil \times W_{sz} \times R_k}{G_T} \right\rceil$$

For devices of compute capability 2.x:

$$R_{block} = \left\lceil \frac{W_{sz} \times R_k}{G_T} \right\rceil \times W_{block}$$

where, G_W is the warp allocation granularity, R_k is the number of registers used by the kernel and G_T is the thread allocation granularity.

The total amount of shared memory S_{block} in bytes allocated for a block is as follows:

$$S_{block} = \left\lceil \frac{S_k}{G_S} \right\rceil$$

where, S_k is the amount of shared memory used by the kernel in bytes, G_S is the shared memory allocation granularity.

Figure 10 is representative of how multiprocessor utilization varies with the chosen number of threads/block. Table 4 summarizes the model parameters for Quadro 2000 GPU. It may well be figured out that increase in the number of threads increases contention for resources which is reflected by the lower utilization levels obtained for more than 800 threads/block. The maximum and minimum number of threads for which utilization exceeds 80 % are 160 and 736 respectively.

4.3 Empirical study of computational efficiency

Implementation of the proposed GPU accelerated particle filtering has been carried out on Nvidia Quadro 2000 which has a compute capability of 2.1 and 192 CUDA cores, while the host processor used for running the serial version is a Core i5 machine having CPU frequency of 3.2 GHz and 2 hyper-threaded physical (i.e. 4 logical) cores. Performance on a GPU under the CUDA architecture is largely defined by the degree of utilization of multiprocessors present on the device (refer to the previous section). In this section, we shall discuss the pivotal aspects of performance as obtained for the proposed D3 resampling Particle filter, from the perspective of both temporal effectiveness as well as quality of estimation (in terms of root mean square of error). As a means of assessing the overall algorithmic quality of the proposed estimator, a cost to performance metric (CPM) is introduced towards the end of this section. A holistic comparison is done between sequential and GPU versions of the proposed PF algorithm. Finally, the GPU accelerated version is shown to be a better choice in comparison to the sequential one in terms of CPM.

Table 5 summarizes for different population sizes, execution times of GPU and CPU versions for both the systematic and D3 resampling particle filter algorithms. The

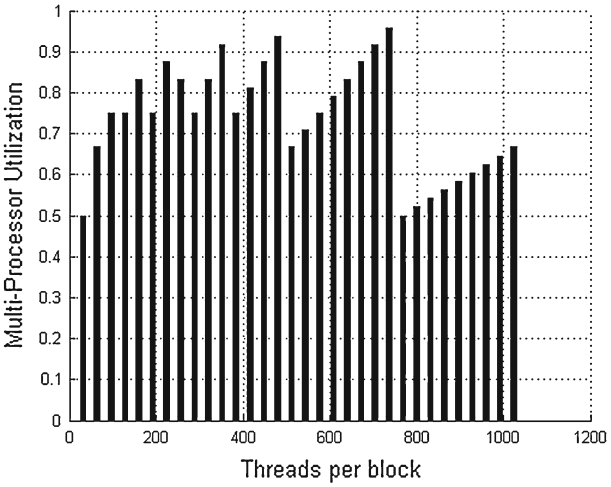


Fig. 10 Multi-processor utilization plot in terms of active resident warps obtained for different values of threads/block

Table 4 CUDA parameters for Nvidia Quadro 2000

Compute capability	G_W	G_T	G_S	Shared memory
2.1	2	64	128	28 Bytes

Table 5 Comparative summary of execution times

Particles	Sys. resampling (CPU)	D3 resampling (CPU)	D3 resampling (GPU)
500	1.92156	2.8468	0.58757
1,000	3.88116	5.67684	0.712095
5,000	19.5552	28.7204	2.0303
10,000	38.5844	57.5684	3.85415
15,000	57.8555	85.9349	6.29413
20,000	81.8349	114.363	10.4578
50,000	204.738	285.763	61.6328

number of threads/block has been chosen as 160. The GPU version consumes lesser running time in comparison to the sequential version of both proposed approach and the classical one. The execution time is estimated as average over a set of 2,400 runs for each population size. We get approximate speedups of 3 and 4 for the GPU version over the sequential versions of classical systematic resampling and D3 resampling particle filters respectively. Variations in computation time on GPU for changes in population size follow the asymptotic complexity O(N) for the proposed particle filter.

However, as discussed previously, the computation of CDF over the entire likelihood weight space for all the particles, being a sequential operation is a perfect instance of unbalanced load for a single thread to execute. The method as adopted in the paper towards effective distribution of overall computation of CDF by all the CUDA threads within a grid is instrumental in increasing the relative occupancy of all the threads in comparison to execution within the context of a single thread. Figure 11 is illustrative of a snapshot of parallel CDF computation by all the CUDA threads over a single run,

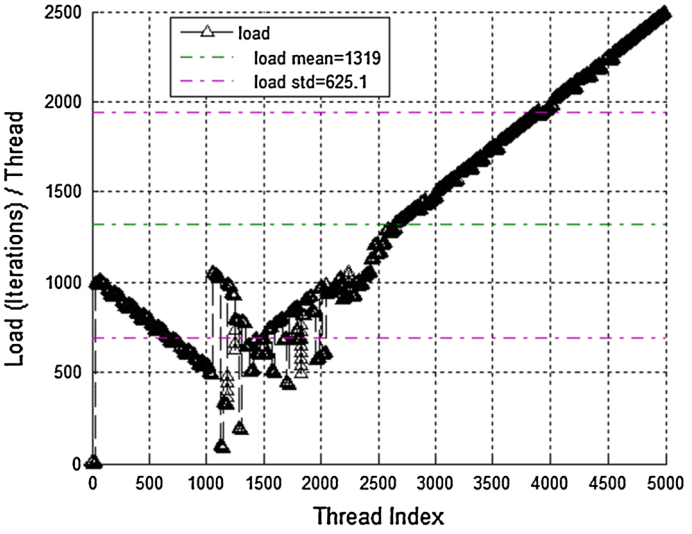


Fig. 11 Load distribution graph in terms of iteration-count/thread as obtained in the context of different CUDA threads

thereby showing a plot for load/thread versus each individual thread context. Most of the threads are seen to compute CDF only partially, with the mean number of iterations (=1,315) being less than the total number of threads (=5,000) within the grid. The floating point operations remaining same for all threads in order to compute the CDF, load on a particular thread is proportional to the number of iterations it has to sweep over the global likelihood weight space. Computing CDF with the quasi-initialization approach evidently improves upon load sharing among the CUDA threads.

An elaborate comparison for running time of GPU execution having different threads/block over varying sizes of population, is presented in Table 6. Apparently, the best running time efficiency is obtained with 160 threads/block. We now need to look upon the quality of estimation as achieved through GPU execution. Table 7 summarizes the estimation accuracy (in terms of root mean square error) for the state variables $\dot{x}$ and $\dot{y}$. Accuracy tends to improve with larger threads/block, which is mainly due to better synchronization at the block level, such that the adopted brute-force synchronization works effectively, when number of threads/block is significantly greater in comparison to the number of blocks. Since improper synchronization among the blocks shall lead to an improper computation of normalized weights and state estimates, comparable differences in quality of estimation are observed for different choices of threads/block. Table 7 shows that best results are obtained when there are 628 threads/block. However, this is in direct contention with the previously chosen value of 160, for which overall GPU execution has the least running time. We now present a cost to performance metric, such that both running cost and quality of estimation can be treated as a consolidated metric for assessing overall algorithmic quality for different choices of parameters, namely threads/block and population size. However, theoretical convergence of particle filters to the true posterior is proved for large population sizes in [25]. Therefore, we are left only with threads/block which

Table 6 Execution time summary for different threads per block (TPB)

TPB	Particles								
	500	1,000	1,500	2,000	5,000	10,000	15,000	20,000	50,000
160	0.568	0.716	0.893	1.080	2.281	5.002	8.387	14.195	61.877
196	0.763	0.984	1.068	1.399	3.115	5.968	9.393	15.702	61.864
232	0.800	1.119	1.443	1.689	2.994	5.616	8.693	11.472	65.206
268	1.018	1.300	1.609	1.894	3.855	7.157	10.492	14.956	65.665
304	1.121	1.405	1.928	2.139	4.244	7.982	11.403	16.433	72.903
340	1.258	1.789	1.999	2.526	4.672	8.368	12.427	16.332	68.030
376	1.371	1.879	2.395	2.596	5.068	9.109	13.669	17.658	74.743
412	1.508	2.048	2.482	3.064	5.487	9.804	14.764	19.268	66.738
448	1.655	2.204	2.645	3.146	5.955	10.571	16.006	20.805	72.264
520	2.461	2.936	3.450	3.937	6.977	12.099	17.994	23.921	82.288
556	2.652	3.130	3.599	4.064	7.826	13.477	19.156	25.630	67.097
592	2.874	3.329	3.815	4.276	7.967	14.307	20.063	27.124	70.792
628	3.088	3.547	4.025	4.487	8.951	15.097	21.447	27.779	74.617
664	3.300	3.768	4.258	4.712	9.099	15.221	22.498	28.929	78.254
700	3.532	4.007	4.496	5.723	9.231	16.245	23.566	30.674	82.377
736	3.775	4.272	4.761	5.971	10.567	17.611	24.748	31.885	87.491

Table 7 Root mean squared error (RMSE) in estimation for $\dot{X}$ $\dot{Y}$ obtained with different threads per block (TPB)

TPB	RMSE ($\dot{X}$)	RMSE ($\dot{Y}$)
160	1.2375	1.1937
196	2.0805	1.2555
232	0.5484	0.5345
268	1.0248	1.0582
340	0.9501	0.9759
376	0.4939	0.4901
412	0.9692	0.9972
448	1.0359	1.0735
520	0.4594	0.4538
556	0.4600	0.4543
592	0.4599	0.4542
628	0.4592	0.4536
664	1.0127	1.0454
700	0.5204	0.5314
736	0.9548	0.9815

significantly affect both running time as well as estimation quality for the proposed algorithm. With this we introduce the cost to performance metric (CPM) subsequently in this section.

Let us consider two estimators E1 and E2 working with the same number of particles say N with E1 having better performance with greater cost measures in comparison to E2. Let us assume P1 and P2 to be the performance measures of the estimators

respectively in terms of σ^{-1} of estimation error, and let C1 and C2 be the cost of the algorithm in Milli-secs of running time. Contextually, effectiveness of the estimators can be classified into two categories viz. one with better performance and the other with better speedup, since only these two criteria constitute what may collectively be called quality. As a result cost to performance metric (CPM) should reflect a collective effect of these two parameters. In this regard, we define two dimensionless entities in the form of cost and performance ratios between two estimators as follows:

$$P_{12} = \frac{P_1}{P_2}$$
$$C_{12} = \frac{C_1}{C_2}$$

where the subscripts in P_{12} and C_{12} designate which algorithm is being assessed with respect to the other. Considering the definitions of comparative performance and cost measures P_{12} and C_{12} respectively, we are now in a position to define the cost-to-performance metric for any two estimators (one exhibiting better estimation accuracy and the other consuming less execution time). As for the better performing estimator, we define CPM as the ratio of P_{12} to C_{12}, i.e. whether $P_{12} > C_{12}$. The higher the value of this ratio, the greater is the quality of the estimator. On the other hand, for the faster estimator with relatively poorer performance, CPM is defined as the ratio of C_{21} to P_{21}, i.e. whether $C_{21} < P_{21}$. Lower goes the value of this ratio, better is the quality of the estimator. It may be realized that the two definitions for quality of two distinct types of estimators, are absolutely in consistence with each other. If one is of good quality then it is inevitably better than the other. With this criteria, the CPM may formally be defined as follows:

$$CPM_{12} = \frac{P_{12}}{C_{12}} \quad \text{for } P_{12} > 1 \text{ and } C_{12} > 1$$
$$CPM_{21} = \frac{C_{21}}{P_{21}} \quad \text{for } P_{21} < 1 \text{ and } C_{21} < 1$$

Referring back to Table 7, we find the GPU based PF estimator has comparatively lower RMSE of estimation for values of threads/block varying between 520 and 628. However, by referring to Table 6, we find that the lowest running time occurs when there are 160 threads/block. It is therefore of good interest to find out the best of the estimators among the performing group (as with 520–628 threads/block) as well as the fast running algorithm (with 160 threads/block), using CPM as introduced above in the present section. CPM for the performing estimators are computed with respect to the fast estimator having 160 threads/block. The values for running times and performance are obtained by referring to Tables 6 and 7 respectively. Performance is computed as

reciprocal of the norm of RMSE ($\dot{X}$) and RMSE($\dot{Y}$) for each of the estimators.

$$CPM_{520|160} = \frac{P_{520}XC_{160}}{P_{160}XC_{520}} = 2.003$$
$$CPM_{556|160} = \frac{P_{556}XC_{160}}{P_{160}XC_{556}} = 2.46$$
$$CPM_{592|160} = \frac{P_{592}XC_{160}}{P_{160}XC_{592}} = 2.32$$
$$CPM_{628|160} = \frac{P_{628}XC_{160}}{P_{160}XC_{628}} = 2.21$$

Evidently having 556 threads/block optimizes the D3 resampling particle filter in terms of running time as well as estimation quality, for our chosen problem.

5 Conclusion

The present paper aims at introducing a dual distribution dependent (D3) resampling scheme for particle filters as an effective means in solving particle impoverishment otherwise associated with conventional resampling methods. Subsequently, the significant running time for the proposed algorithm is shown to be reduced by using CUDA computing for harnessing the massive parallelism and computational power of generic GPUs. The proposed resampling is instrumental in improving particle diversity by driving the population towards significant regions of likelihood without eliminating the prior completely. The method generates two resampled particles from the previous sample in comparison to a single sample in the case of conventional resampling schemes. Hence, the resampled population retains dual significance with improved diversity. Thus the mechanism of D3 resampling is effective in mitigating particle impoverishment. However, a drawback of the D3 algorithm is that it suffers from greater running time as compared to systematic resampling. With a view to reduce the sequential time involved, a parallel variant of the method is proposed using CUDA C. Detailed discussions regarding implementation of the multi-threaded parallel code on Quadro 2000 GPU are presented along with illustrations of empirical computational performance. The GPU accelerated code is shown to have 3 times speedup over the sequential one. However, it is shown that both estimation quality as well as running time vary with the choice of threads/block in the adopted CUDA model. As a matter of conflict between the two measures of performance, a unified cost to performance metric (CPM) is introduced in this context. The CPM is shown to be effective in determining a non-conflicting choice for threads/block in order to design an optimized GPU accelerated D3 resampling particle filter code.

References

1. Gordon N, Salmond D (Apr. 1993) Novel approach to nonlinear/non-Gaussian Bayesian state estimation. IEE Proc F 140(2):107–113

2. Sanjeev Arulampalam M, Maskell Simon, Gordon Neil, Clapp Tim (2002) A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. IEEE Trans Signal Proc 50:2
3. James V (2009) Candy, Bayesian signal processing classical, modern, and particle filtering methods. Wiley, New Jersey
4. Simon Dan (2006) Optimal state estimation Kalman H and nonlinear approaches. Wiley, New Jersey
5. Kak Marc (1964) Statistical independence in probability analysis and number theory, mathematical association of America. Wiley, New Jersey
6. Pitt M, Shephard N (1999) Filtering via simulation: auxiliary particle filter, Jasa, vol 94
7. Karlsson R, Gustafsson F (2003) Particle filter for underwater terrain navigation, Tech Rep LiTH-ISY-R-2530, Sweden
8. Musso C, Oudjane N, LeGland F (2001) Improving regularised particle filters, Sequential Monte Carlo methods in practice. In: Doucet A, de Freitas JFG, Gordon NJ (eds) Springer, New York
9. de Freitas N, Andrieu C, Hojen-Sorensen P, Niranjan M, Gee A (2001) Sequential Monte Carlo methods for neural networks, Sequential Monte Carlo methods in practice. In: Doucet A, de Freitas N, Gordon N (eds) Springer, New York, pp 359379
10. Fearnhead P (1998) Sequential Monte Carlo methods in filter theory, PhD thesis, University of Oxford, Oxford
11. Brun Olivier, Teuliere Vincent, Garcia Jean-Marie (2002) Parallel particle filtering. J Parallel Distrib Comput 62:1186–1202
12. Bolic Miodrag, Djuric Petar M, Hong Sangjin (2005) Resampling algorithms and architectures for distributed particle filters. IEEE Trans Signal Proc 53:7
13. Hong S, Djuric PM (2006) High-throughput scalable parallel resampling mechanism for effective redistribution. In: Proceedings of IEEE transactions on signal processing of particles, vol 54, no 3, 2006
14. Rosn O, Medvedev A, Ekman M (2010) Speedup and tracking accuracy evaluation of parallel particle filter algorithms implemented on a multicore architecture. In: Proceedings of IEEE international conference on control applications part of 2010 IEEE multi-conference on systems and Control, Japan, 8–10 September 2010
15. Rosn Olov, Medvedev Alexander (2013) Efficient parallel implementation of state estimation algorithms on multicore platforms. IEEE Trans Control Syst Technol 21:1
16. Chen Tianshi, Schn Thomas B, Ohlsson Henrik, Ljung Lennart (2011) Decentralized particle filter with arbitrary state decomposition. IEEE Trans Signal Proc 59:2
17. Hwang Kyuyeon, Sung Wonyong (2013) Load balanced resampling for real-time particle filtering on graphics processing units. IEEE Trans Signal Proc 61:2
18. Bolic M, Djuric PM, Hong S (2003) New resampling algorithms for particle filters. In: Proceedings of IEEE international conference on acoustics, speech, and signal processing, 2003, (ICASSP '03)
19. Liu J (2001) Monte Carlo strategies in scientific computing. Springer, New York
20. Fossen TI (2002) Marine control systems. Marine Cybernetics AS, Norway, pp 19–38
21. Sanders J, Kandrot E (2011) CUDA by example. Addison-Wisley, USA
22. Cecilia JM, Garcia JM, Ujaldon M (2010) The GPU on the matrix-matrix multiply: performance study and contributions. Parallel computing: from multicores and GPUs to Petascale. doi:10.3233/978-1-60750-530-3-331, IOS Press
23. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. AFIPS Conf Proc 30:483–485. doi:10.1145/1465482.1465560
24. Cuda C Programming Guide, NVIDIA Corporation, 2013, URL:http://docs.nvidia.com/cuda/pdf/CUDA C Programming Guide.pdf
25. Pequito S (2009) From particle filters to Malliavin filtering with application to target tracking, URL: http://users.isr.ist.utl.pt/pjcro/courses/def0910/docs/report_SP.pdf