Rethinking resampling in the particle filter on graphics processing units

Lawrence M. Murray, Anthony Lee and Pierre E. Jacob

***Abstract*—Modern parallel computing devices such as the graphics processing unit (GPU) have gained significant traction in scientific computing, and are particularly well-suited to data-parallel algorithms such as the particle filter. Of the components of the particle filter, the resampling step is the most difficult to implement well on such devices, as it often requires a collective operation, such as a sum, across weights. We present and compare a number of resampling algorithms in this work, including rarely-used alternatives based on Metropolis and rejection sampling. We find that these alternative approaches perform significantly faster on the GPU than more common approaches such as the multinomial, stratified and systematic resamplers, a speedup attributable to the absence of collective operations. Moreover, in single-precision (particularly relevant on GPUs due to its faster performance), the common approaches are numerically unstable for plausibly large numbers of particles, while these alternative approaches are not. Finally, we provide a number of auxiliary functions of practical use in resampling, such as for the permutation of ancestry vectors to enable in-place propagation of particles.**

***Index Terms*—Particle filter, sequential Monte Carlo, state-space models, resampling, graphics processing unit**

I. Introduction

For some sequence of time points $t = 1, \ldots, T$ and observations at those times $\mathbf{y}_1, \ldots, \mathbf{y}_T$, the particle filter [1], [2] uses N weighted samples, or *particles*, to recursively estimate the time marginals $p(\mathbf{x}_t|\mathbf{y}_{1:t})$ of the latent states $\mathbf{x}_1, \ldots, \mathbf{x}_T$ of the state-space model

$$p(\mathbf{x}_{0:T}, \mathbf{y}_{1:T}) = p(\mathbf{x}_0) \prod_{t=1}^{T} p(\mathbf{y}_t|\mathbf{x}_t) p(\mathbf{x}_t|\mathbf{x}_{t-1}), \qquad (1)$$

depicted in Figure 1.

Pseudocode for the simplest *bootstrap* particle filter [1] is given in Code 1. The initialisation, propagation and weighting steps are readily parallelised, being independent operations on each particle $\mathbf{x}_t^i$ and its weight w_t^i. Resampling, on the other hand, is a collective operation across particles and weights, so that parallelisation is more difficult. It is here that the present work concentrates.

The resampling step can be encoded by a randomised algorithm Ancestors that accepts a vector, $\mathbf{w}_{t-1} \in \mathbb{R}^N$, of particle weights and returns a vector, $\mathbf{a}_t$, of integers between 1 and N, where each a_t^i is the index of the particle at time $t-1$ which is to be the *ancestor* of the ith particle at time t. Alternatively, the resampling step may be encoded by a randomized algorithm Offspring that also accepts a vector, $\mathbf{w}_{t-1} \in \mathbb{R}^N$, of particle weights, but instead returns a vector, $\mathbf{o}_t$, of integers between between 0 and N, where each o_t^i is the number of *offspring* to be created from the ith particle at time $t-1$ for propagation to time t. As we shall see, each resampling algorithm more naturally takes one form or the other, and ancestry vectors are readily converted to offspring vectors and vice-versa (§III provides functions to achieve this).

L.M. Murray is with CSIRO Mathematics, Informatics & Statistics.
A. Lee is with the Department of Statistics, University of Warwick.
P.E. Jacob is with the Department of Statistics and Applied Probability, National University of Singapore.

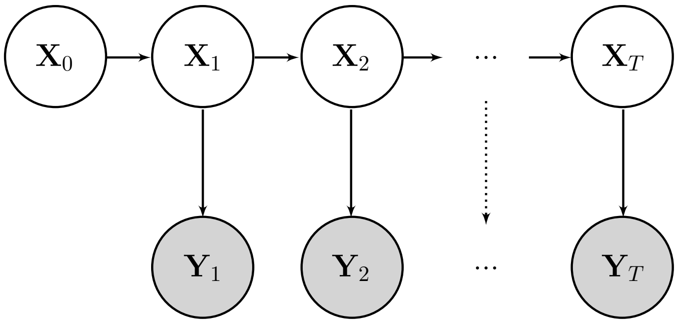

Fig. 1. The state-space model.

Code 1 Pseudocode for the bootstrap particle filter.

```
Particle-Filter($N \in \mathbb{N}_+, T \in \mathbb{N}_0$)
1  foreach $i \in \{1, \ldots, N\}$
2      $\mathbf{x}_0^i \sim p(\mathbf{x}_0)$ // initialise particle $i$
3      $w_0^i \leftarrow 1/N$ // initialise weight $i$
4  for $t = 1, \ldots, T$
5      $\mathbf{a}_t \leftarrow$ Ancestors($\mathbf{w}_{t-1}$) // resample
6      foreach $i \in \{1, \ldots, N\}$
7          $\mathbf{x}_t^i \sim p(\mathbf{x}_t|\mathbf{x}_{t-1}^{a_t^i})$ // propagate particle $i$
8          $w_t^i \leftarrow p(\mathbf{y}_t|\mathbf{x}_t^i)$ // weight particle $i$
```

Following the resampling step is the propagation step. In the implementation of this there are two options in arranging memory. The first is to have an *input* buffer holding the particles at time $t-1$, and a separate *output* buffer into which to write the propagated particles at time t. The second is to work *in-place*, reading and writing particles from and to the same buffer. The second option is more memory efficient by a factor of two (for a fixed number of particles), but places more stringent conditions on the output of the resampling algorithm. It is sufficient that the ancestry vector, $\mathbf{a}_t$, satisfies $\forall i \in \{1, \ldots, N\}$:

$$o_t^i > 0 \implies a_t^i = i. \qquad (2)$$

With this, it is possible to insert a copy step immediately before each propagation step, setting $\mathbf{x}_{t-1}^{i} \leftarrow \mathbf{x}_{t-1}^{a_t^i}$ for all $i = 1, \dots, N$, but $a_t^i \neq i$. Each particle can then be propagated in-place by reading from and writing to the same buffer. Importantly, (2) ensures that the copies can be done concurrently without read and write conflicts, as each particle is either read from or written to, but not both. The focus of this work is on this in-place mode, such that algorithms are timed up to the delivery of an ancestry vector that satisfies (2). We also present auxiliary functions for the permutation of arbitrary ancestry vectors to achieve this.

The challenge in programming a parallel device such as a graphics processing unit (GPU) is that, to occupy its full capacity, an algorithm must be executable with tens of thousands of threads. Furthermore, the threads with which it executes are not entirely independent, grouped into teams (of size 32 on current hardware [3]) called *warps*. The threads of a warp should ideally follow the same path through an algorithm. Where they diverge to different branches of a conditional, or different trip counts of a loop, their execution becomes serialised and parallelism is lost (referred to as *warp divergence*). A further consideration is that, for all the threads of a warp, memory reads and writes should be from or to neighbouring addresses for best performance (referred to as the *coalescing* of reads and writes). Considerations such as these shape the development of the algorithms in this work. The reader is referred to the OpenCL standard [4] for general information, and the Compute Unified Device Architecture (CUDA) [5] for more detail on optimising for NVIDIA GPUs in particular.

Parallel implementation of resampling schemes has been considered before, largely in the context of distributed memory machines [6], [7]. Likewise, implementation of the particle filter on GPUs has been considered [8], [9], [10], although with an emphasis on low-level implementation issues rather than algorithmic adaptation as here.

Section II presents algorithms for resampling based on multinomial, stratified, systematic, Metropolis and rejection sampling. Section III presents auxiliary functions for converting between offspring and ancestry vectors, and the permutation of ancestry vectors to satisfy (2). Empirical results for execution time and variance in outcomes are obtained for all algorithms in Section IV, with concluding discussion in Section V.

II. Resampling algorithms

We present parallel resampling algorithms suitable for both multi-core central processng units (CPUs) and many-core graphics processing units (GPUs). Where competitive serial algorithms exist, these are also presented. Resampling algorithms are nicely visualised by arranging particles by weight in a circle, as in Figure 2, which may help to elucidate the various approaches.

The algorithms presented here are described using pseudocode with a number of conventions. In particular, we make use of *primitive* operations such as searches, transformations, reductions and prefix sums. Such operations will be familiar to programmers of, for example, the C++ standard template library (STL) or Thrust library [11], and their implementation on GPUs has been well-studied [12]. The advantage of describing algorithms in this way is that the efficient implementation of these primitives in both serial and parallel contexts is well understood, and a single pseudocode description in terms of primitives will often suffice for both. Code 2 defines the particular primitives used throughout this work.

Code 2 Pseudocode for various primitive functions.

```
INCLUSIVE-PREFIX-SUM($\mathbf{w} \in \mathbb{R}^N$) $\to \mathbb{R}^N$
1  $W^i \leftarrow \sum_{j=1}^{i} w^j$
2  return $\mathbf{W}$

EXCLUSIVE-PREFIX-SUM($\mathbf{w} \in \mathbb{R}^N$) $\to \mathbb{R}^N$
1  $W^i \leftarrow \begin{cases} 0 & i = 1 \\ \sum_{j=1}^{i-1} w^j & i > 1 \end{cases}$
2  return $\mathbf{W}$

ADJACENT-DIFFERENCE($\mathbf{W} \in \mathbb{R}^N$) $\to \mathbb{R}^N$
1  $w^i \leftarrow \begin{cases} W^i & i = 1 \\ W^i - W^{i-1} & i > 1 \end{cases}$
2  return $\mathbf{w}$

SUM($\mathbf{w} \in \mathbb{R}^N$) $\to \mathbb{R}$
1  return $\sum_{i=1}^{N} w^i$

LOWER-BOUND($\mathbf{W} \in \mathbb{R}^N, u \in \mathbb{R}$) $\to \mathbb{N}_+$
1  requires
2      $W$ is sorted in ascending order
3  return
4      the lowest $j$ such that $u$ may be inserted into
       position $j$ of $W$ and maintain its sorting.
```

We distinguish between the **foreach** and **for** constructs. The former is used where the body of the loop is to be executed for each element of a set, with the order unimportant. The latter is used where the body of the loop is to be executed for each element of a sequence, where the order must be preserved. The intended implication is that **foreach** loops may be parallised, while **for** loops may not be.

In what follows, we omit the subscript t from weight, offspring and ancestry vectors, as all of the algorithms presented behave identically at each time in the particle filter.

A. Multinomial resampling

Multinomial resampling proceeds by drawing each a^i independently from the categorical distribution over $\mathcal{C} = \{1, \dots, N\}$, where $P(a^i = j) = w^j / \textsc{Sum}(\mathbf{w})$. Pseudocode is given in Code 3. The algorithm is dominated by the N calls of LOWER-BOUND, which if implemented with a binary search, will give $\mathcal{O}(N \log_2 N)$ complexity overall.

The INCLUSIVE-PREFIX-SUM operation on line 1 of Code 3 is not numerically stable, as large values may be added to relatively insignificant ones during the procedure (an issue

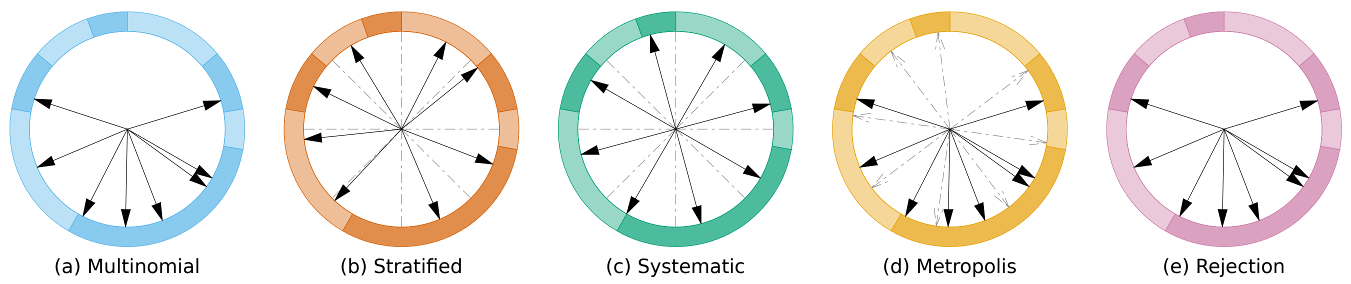

Fig. 2. Visualisation of the resampling algorithms considered. Arcs along the perimeter of the circles represent particles by weight, arrows indicate selected particles and are positioned **(a)** uniformly randomly in the multinomial resampler, **(b & c)** by evenly slicing the circle into strata and randomly selecting an offset (stratified resampler) or using the same offset (systematic resampler) into each stratum, **(d)** by initialising multiple Markov chains and simulating to convergence in the Metropolis resampler, or **(e)** by rejection sampling.

Code 3 Pseudocode for parallel multinomial resampling.

```
MULTINOMIAL-ANCESTORS($\mathbf{w} \in \mathbb{R}^N$) $\rightarrow \mathbb{R}^N$
$\mathbf{W} \leftarrow$ INCLUSIVE-PREFIX-SUM($\mathbf{w}$)
foreach $i \in \{1, \dots, N\}$
    $u^i \sim \mathcal{U}[0, W^N)$
    $a^i \leftarrow$ LOWER-BOUND($\mathbf{W}, u^i$)
return a
```

Code 4 Pseudocode for serial, single-pass multinomial resampling.

```
MULTINOMIAL-ANCESTORS($\mathbf{w} \in \mathbb{R}^N$) $\rightarrow \mathbb{R}^N$
$\mathbf{W} \leftarrow$ EXCLUSIVE-PREFIX-SUM($\mathbf{w}$)
$W \leftarrow W^N + w^n$ // sum of weights

$lnMax \leftarrow 0$
$j \leftarrow N$
for $i = N, \dots, 1$
    $u \sim \mathcal{U}[0, 1)$
    $lnMax \leftarrow lnMax + \ln(u)/i$
    $u \leftarrow W \exp(lnMax)$
    while $u < W^j$
        $j \leftarrow j - 1$
    $a^i \leftarrow j$
return a
```

Code 5 Pseudocode for stratified resampling.

```
1 STRATIFIED-CUMULATIVE-OFFSPRING($\mathbf{w} \in \mathbb{R}^N$) $\rightarrow \mathbb{R}^N$
1 $\mathbf{u} \in \mathbb{R}^N \sim$ i.i.d.$\mathcal{U}[0, 1)$
2 $\mathbf{W} \leftarrow$ INCLUSIVE-PREFIX-SUM($\mathbf{w}$)
3 foreach $i \in \{1, \dots, N\}$
4     $r^i \leftarrow \frac{NW^i}{W^N}$
5     $k^i \leftarrow \min\left(N, \lfloor r^i \rfloor + 1\right)$
6     $O^i \leftarrow \min\left(N, \left\lfloor r^i + u^{k^i} \right\rfloor\right)$
7 return O
```

intrinsic to any large summation). With large N, assigning the weights to the leaves of a binary tree and summing with a depth-first recursion over this will help. With large variance in weights, pre-sorting will also help. While log-weights are often used in the implementation of particle filters, these need to be exponentiated (perhaps after rescaling) for the INCLUSIVE-PREFIX-SUM operation, so this does not alleviate the issue.

Serially, the same approach may be used, although a single-pass approach of complexity $\mathcal{O}(N)$ is enabled by generating sorted uniform random variates [13]. Code 4 details this approach. There is scope for a small degree of parallelism here by dividing N amongst a handful of threads, but not enough to make this a useful algorithm on the GPU. A drawback is the use of relatively expensive logarithm functions, but we nevertheless find it superior to Code 3 on CPU.

B. Stratified resampling

The variance in outcomes produced by the multinomial resampler may be reduced [14] by stratifying the cumulative probability function of the same categorical distribution, and randomly drawing one particle from each stratum. This stratified resampler [15] most naturally delivers not the ancestry or offspring vector, but the *cumulative offspring* vector, defined with respect to the offspring vector $\mathbf{o}$ as $\mathbf{O} =$ INCLUSIVE-PREFIX-SUM($\mathbf{o}$). Pseudocode is given in Code 5. The algorithm is of complexity $\mathcal{O}(N)$.

As for multinomial resampling, the INCLUSIVE-PREFIX-SUM operation on line 2 of Code 5 is not numerically stable. The same strategies to ameliorate the problem apply.

Line 6 of Code 5 is more problematic. Consider that there may be a j such that, for $i \geq j$, u^{k^i} is not significant against r^i under the floating-point model, so that the result of $r^i + u^{k^i}$ is just r^i. For such i, no random sample is being made within the strata. Furthermore, rounding up on the same line might easily deliver $O^N = N + 1$, not $O^N = N$ as required, if not for the quick-fix use of min! This is particularly relevant in the present context because contemporary GPUs have significantly faster single-precision than double-precision floating-point performance, so that the use of single precision is always tempting. In single precision, seven significant figures (in decimal) can typically be expected.

Code 6 Pseudocode for systematic resampling.

```
SYSTEMATIC-CUMULATIVE-OFFSPRING($\mathbf{w} \in \mathbb{R}^N$) → $\mathbb{R}^N$
1  $u \sim \mathcal{U}[0,1)$
2  $\mathbf{W} \leftarrow$ INCLUSIVE-PREFIX-SUM($\mathbf{w}$)
3  foreach $i \in \{1, \dots, N\}$
4      $r^i \leftarrow \frac{NW^i}{W^N}$
5      $O^i \leftarrow \min\left(N, \lfloor r^i + u \rfloor\right)$
6  return O
```

Consider that, with N around one million, almost certainly no $u^{k^i} \in [0,1)$ is significant against r^i at high i. One million particles is not an unrealistic number for some contemporary applications of the particle filter. In double-precision, however, where 15 significant figures (in decimal) is expected, it is unlikely that N is sufficiently large for this to be a problem in current applications. Note that while pre-sorting weights and summing over a binary tree can help with the numerical stability of the INCLUSIVE-PREFIX-SUM operation, it does not help with this latter issue.

C. Systematic resampling

The variance in outcomes of the stratified resampler may often, but not always [14], be further reduced by using the same random offset when sampling from each stratum. This is the *systematic resampler* (equivalent to the *deterministic* method described in the appendix of [15]). Pseudocode is given in Code 6, which is a simple modification to Code 5. The same complexity and numerical caveats apply to the systematic resampler as for the stratified resampler.

D. Metropolis resampling

The preceding resamplers all suffer from two problems:

1) they exhibit numerical instability for large N or large weight variance, and
2) they require a collective operation over the weights, specifically INCLUSIVE-PREFIX-SUM, which is less readily parallelised than independent operations on weights.

We present two alternative approaches, not typically considered in the literature, which do not suffer from these problems. They are more numerically stable, more readily parallelised, and consequently better suited to the breadth of parallelism offered by GPU hardware.

The first produces a multinomial resample via the Metropolis algorithm [16] rather than by direct sampling. This was briefly studied by the first author in [17], but a more complete treatment is given here. Instead of the collective operation, only the ratio between any two weights is ever computed, and only when needed. Code 7 describes the approach. The complexity of the algorithm is discussed later.

The Metropolis resampler is parameterised by B, the number of iterations to be performed before convergence is assumed and each particle may settle on its chosen ancestor.

Code 7 Pseudocode for Metropolis resampling.

```
METROPOLIS-ANCESTORS($\mathbf{w} \in \mathbb{R}^N, B \in \mathbb{N}$) → $\mathbb{R}^N$
1  foreach $i \in \{1, \dots, N\}$
2      $k \leftarrow i$
3      for $n = 1, \dots, B$
4          $u \sim \mathcal{U}[0,1]$
5          $j \sim \mathcal{U}\{1, \dots, N\}$
6          if $u \leq w^j / w^k$
7              $k \leftarrow i$
8      $a^i \leftarrow k$
9  return a
```

As B must be finite, the algorithm produces a biased sample. Selecting B is a tradeoff between speed and bias, with smaller B giving faster execution time but larger bias. This bias may not be much of a problem for filtering applications, but does violate the assumptions that lead to unbiased marginal likelihood estimates in a particle Markov chain Monte Carlo (PMCMC) framework [18], so care should be taken.

We provide guidance as to the selection of B by bounding the bias ϵ for the maximum normalised weight p^* [17]. One may have both an upper bound on unnormalised weights, $\sup w$, and an expected weight, $\mathbb{E}(w)$, from which this might be computed as $p^* = \sup w/(N\mathbb{E}(w))$ (see §IV for one such example, albeit contrived). Otherwise, this might be thought of as a tolerance threshold on weight degeneracy and a value specified accordingly. Convergence of a Metropolis chain depends on a sufficient number of steps being taken such that the probability of a particle of this maximum weight being selected is within ϵ of p^*.

Following [19][§2.1], for N particles, construct a binary 0-1 process where state 1 represents being at a particle with the maximum weight, and state 0 represents being at any other particle. The transition matrix is

$$T = \begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix}, \tag{3}$$

where

$$\alpha = \frac{1}{N} \cdot \frac{1-p^*}{p^*} \tag{4}$$

is the probability of moving from state 1 to 0, and

$$\beta = \frac{1}{N} \tag{5}$$

is the probability of moving from state 0 to 1, with a uniform proposal over all N particles and the Metropolis acceptance rule. The B-step transition matrix is then:

$$T^B = \frac{1}{\alpha+\beta}\begin{pmatrix} \alpha & \beta \\ \alpha & \beta \end{pmatrix} + \frac{\lambda^B}{\alpha+\beta}\begin{pmatrix} \alpha & -\alpha \\ -\beta & \beta \end{pmatrix}, \tag{6}$$

where $\lambda = (1-\alpha-\beta)$. We require that the second term satisfy the bias bound:

$$\frac{\lambda^B}{\alpha+\beta}\begin{pmatrix} \alpha & -\alpha \\ -\beta & \beta \end{pmatrix} < \epsilon, \tag{7}$$

Code 8 Pseudocode for rejection resampling.

```
REJECTION-ANCESTORS($\mathbf{w} \in \mathbb{R}^N$) $\to \mathbb{R}^N$
1  foreach $i \in \{1, \dots, N\}$
2      $j \leftarrow i$
3      $\beta \sim \mathcal{U}[0,1]$
4      while $\beta > w^j / \sup w$
5          $j \sim \mathcal{U}\{1, \dots, N\}$
6          $\beta \sim \mathcal{U}[0,1]$
7      $a^i \leftarrow j$
8      $w^i \leftarrow 1$
9  return $\mathbf{a}$
```

so

$$\frac{\lambda^B}{\alpha + \beta} \max(\alpha, \beta) < \epsilon, \tag{8}$$

which is satisfied when

$$B > \log_\lambda \frac{\epsilon(\alpha + \beta)}{\max(\alpha, \beta)}. \tag{9}$$

Thus, by specifying a p^* and some small ϵ, one proceeeds by calculating α and β, then λ, then finally B as a suggested number of steps for the Metropolis resampler. An appropriate value for ϵ might be relative to p^*, such as $p^* \times 10^{-2}$. The experiments in §IV provide some empirical evidence to support this as a reasonable choice for ϵ, and ultimately B.

The complexity of the Metropolis resampler is $\mathcal{O}(NB)$, but B may itself be a function of N and weight variance, as in the analysis above.

E. Rejection resampling

When an upper bound on the weights is known *a priori*, rejection sampling is possible. Compared to the Metropolis resampler, the rejection resampler also avoids collective operations and their numerical instability, but offers several additional advantages:

1) it is unbiased,
2) it is simpler to configure, and
3) it permits a first deterministic proposal (see, e.g., [20]) that $a^i = i$, improving the variance in outcomes.

Pseudocode is given in Code 8. If the third item above is removed, rejection resampling is an alternative implementation of multinomial resampling. The complexity of the algorithm is $\mathcal{O}(\textsc{Sum}(\mathbf{w})/\sup w)$.

The rejection resampler will perform poorly if $\sup w$ is not a tight upper bound, i.e. if $\max(w^1, \dots, w^N) \ll \sup w$. While one can empirically set $\sup w = \max(w^1, \dots, w^N)$, this would require a collective operation over weights that would defeat the purpose of the approach.

Performance can be tuned if willing to concede a weighted outcome from the resampling step, rather than the usual unweighted outcome. To do this, choose some $\sup v < \sup w$, then form a categorical importance proposal using the weights $v^1, \dots, v^N$, where $v^i = \min(w^i, \sup v)$. Clearly $\sup v$ forms an upper bound on these new weights. Sample from this as a proposal distribution using REJECTION-ANCESTORS, then importance weight each particle i with $w^i \leftarrow w^{a^i}/v^{a^i}$. Note that each weight is 1 except where $w^{a^i} > \sup v$. The procedure may also be used when no hard upper bound on weights exists ($\sup w$), but where some reasonable substitute can be made ($\sup v$).

An issue unique to the rejection resampler is that the computational effort required to draw each ancestor varies, depending on the number of rejected proposals before acceptance. This is an example of a *variable task-length problem* [21], particularly acute in the GPU context where it causes warp divergence, with threads of the same warp tripping the loop on line 4 of Code 8 different numbers of times. A persistent threads strategy [22], [21] might be used to mitigate the effects of this, although we have not been successful in finding such an implementation that does not lose more than it gains through additional overhead in register use and branching.

F. Other algorithms

The resampling algorithms presented here do not constitute an exhaustive list of those in use, but are reasonably representative, and tend to form the building blocks of more elaborate schemes. A notable example is *residual resampling* [23], which deterministically draws each particle $\lfloor Nw^i/\textsc{Sum}(\mathbf{w})\rfloor$ number of times before making up the deficit in the number of particles by randomly drawing additional particles with probabilities proportional to the residuals $Nw^i/\textsc{Sum}(\mathbf{w}) - \lfloor Nw^i/\textsc{Sum}(\mathbf{w})\rfloor$. The first stage may be implemented similarly to the systematic resampler, and the second using either a multinomial, Metropolis or rejection approach.

III. Auxiliary functions

The multinomial, Metropolis and rejection resamplers most naturally return the ancestry vector $\mathbf{a}$, while the stratified and systematic resamplers return the cumulative offspring vector $\mathbf{O}$. Conversion between these is reasonably straightforward.

An offspring vector $\mathbf{o}$ may be converted to a cumulative offspring vector $\mathbf{O}$ via the INCLUSIVE-PREFIX-SUM primitive, and back again via ADJACENT-DIFFERENCE. A cumulative offspring vector may be converted to an ancestry vector via Code 9, and an ancestry vector to an offspring vector via Code 10. These functions perform well on both CPU and GPU. An alternative approach to CUMULATIVE-OFFSPRING-TO-ANCESTORS, using a binary search for each ancestor, was found to be slower.

An ancestry vector may be permuted to satisfy (2). A serial algorithm to achieve this is straightforward and given in Code 11. This $\mathcal{O}(N)$ algorithm makes a single pass through the ancestry vector with pair-wise swaps to satisfy the condition.

The simple algorithm is complicated in a parallel context as the pair-wise swaps are not readily serialised without heavy-weight mutual exclusion. In parallel we propose Code 12. This algorithm does not perform the permutation in-place, but instead produces a new vector $\mathbf{c} \in \mathbb{R}^N$ that is the permutation of the input vector $\mathbf{a}$. It introduces a new vector $\mathbf{d} \in \mathbb{R}^N$, through which, ultimately, $c^i = a^{d^i}$. In the first stage of the algorithm, PREPERMUTE, the thread for element i attempts

Code 9 Pseudocode conversion of an offspring vector $\mathbf{o}$ to an ancestry vector $\mathbf{a}$.

```
CUMULATIVE-OFFSPRING-TO-ANCESTORS($\mathbf{O} \in \mathbb{R}^N$) $\to \mathbb{R}^N$
1  foreach $i \in \{1, \dots, N\}$
2      if $i = 1$
3          $start \leftarrow 0$
4      else
5          $start \leftarrow O^{i-1}$

6      $o^i \leftarrow O^i - start$
7      for $j = 1, \dots, o^i$
8          $a^{start+j} \leftarrow i$
9  return $\mathbf{a}$
```

Code 10 Pseudocode conversion of an ancestor vector $\mathbf{a}$ to an offspring vector $\mathbf{o}$.

```
ANCESTORS-TO-OFFSPRING($\mathbf{a} \in \mathbb{R}^N$) $\to \mathbb{R}^N$
1  $\mathbf{o} \leftarrow \mathbf{0}$
2  foreach $i \in \{1, \dots, N\}$
3      atomic $o^i \leftarrow o^i + 1$
4  return $\mathbf{o}$
```

to claim position a^i in the output vector by setting $d^{a^i} \leftarrow i$. By virtue of the min function on line 3, the element of lowest index always succeeds in this claim while all others contesting the same place fail, and the outcome of the whole permutation procedure is deterministic. This is desirable so that the results of a particle filter are reproducible for the same pseudorandom number seed. For each element i that is not successful in its claim, the thread for i instead attempts to claim d^i, if unsuccessful again then d^{d^i}, then recursively $d^{d^{d^i}}, \dots$ etc, until an unclaimed place is found. Note that this procedure is guaranteed to terminate (a proof is given in Appendix A).

Two other procedures are worth mentioning, as they are often used in conjunction with resampling. The first, the sorting of weights for improved numerical stability, has already been mentioned. The second is the computation of *effective sample size* (ESS) [24], often used to decide whether or not to resample at any given time in the particle filter. The ESS is given by $\textsc{Sum}(\mathbf{w})^2/\mathbf{w}^T\mathbf{w}$. Note that both sorting and ESS are necessarily collective operations. We include these two procedures in our timing results to lend additional perspective.

Code 11 Serial algorithm for permuting an ancestry to ensure (2).

```
PERMUTE($\mathbf{a} \in \mathbb{R}^N$)
1  for $i = 1, \dots, N$
2      if $a^i \neq i$ and $a^{a^i} \neq a^i$
3          $\mathrm{swap}(a^i, a^{a^i})$
4          $i \leftarrow i - 1$ // repeat for new value
5  ensures
6      $\forall i (i \in \{1, \dots, N\} : o^i > 0 \implies a^i = i)$
```

Code 12 Parallel algorithm for the permutation of an ancestry vector to ensure (2).

```
PREPERMUTE($\mathbf{a} \in \mathbb{R}^N$) $\to \mathbb{R}^N$
   // claim places to satisfy (2)
1  Let $\mathbf{d} \in \mathbb{R}^N$ and set $d^i \leftarrow N + 1$ for $i = 1, \dots, N$.
2  foreach $i \in \{1, \dots, N\}$
3      atomic $d^{a^i} \leftarrow \min(d^{a^i}, i)$
4  ensures
5      $\forall i (i \in \{1, \dots, N\} : o^i > 0 \implies d^i = \min_j (a^j = i))$
6  return $\mathbf{d}$

PERMUTE($\mathbf{a} \in \mathbb{R}^N$) $\to \mathbb{R}^N$
1   $\mathbf{d} \leftarrow$ PREPERMUTE($\mathbf{a}$)
2   foreach $i \in \{1, \dots, N\}$
3       $x \leftarrow d^{a^i}$
4       if $x \neq i$
5           // claim unsuccessful in PREPERMUTE
6           $x \leftarrow i$
7           while $d^x \leq N$
8               $x \leftarrow d^x$
9           $d^x \leftarrow i$

10  foreach $i \in \{1, \dots, N\}$
11      $c^i \leftarrow a^{d^i}$

12  ensures
13      $\forall i (i \in \{1, \dots, N\} : o^i > 0 \implies c^i = i)$
14  return $\mathbf{c}$
```

IV. Experiments

Weight sets are simulated to assess the speed and accuracy of each resampling algorithm. For each number of particles $N = 2^4, 2^5, \dots, 2^{20}$ and observation $y = 0, \frac{1}{2}, 1, 1\frac{1}{2}, \dots, 4$, a weight set is generated by sampling $x^i \sim \mathcal{N}(0,1)$, for $i = 1, \dots, N$, and setting

$$w^i = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}(x^i - y)^2\right). \tag{10}$$

This is repeated to produce 500 weight sets for each configuration. Note that the construction is analagous to having a prior distribution of $x \sim \mathcal{N}(0,1)$ and likelihood function of $y \sim \mathcal{N}(x,1)$. As y increases, the relative variance in weights does too.

For this set up, the maximum weight is $\sup w = 1/\sqrt{2\pi}$, and the expected weight

$$\mathbb{E}(w) = \mathcal{N}(y; 0, \sigma^2 = 2) = \frac{1}{2\sqrt{\pi}} \exp\left(-\frac{1}{4}y^2\right). \tag{11}$$

The variance of the weights is

$$\mathbb{V}(w) = \frac{1}{\pi\sqrt{12}} \exp\left(-\frac{1}{3}y^2\right) - [\mathbb{E}(w)]^2, \tag{12}$$

and so their *relative* variance is

$$\mathbb{V}\left(\frac{w}{\mathbb{E}(w)}\right) = \frac{2}{\sqrt{3}} \exp\left(\frac{y^2}{6}\right) - 1, \tag{13}$$

which is increasing with y.

These are used to set B for the Metropolis resampler according to the analysis in §II-D. This maximum weight is also used for the rejection resampler. Note that, while presented with weights here, our actual implementation works with log-weights, which we assume is fairly common practice for observation densities from the exponential family.

Experiments are conducted in single-precision on two devices. The first device is an eight-core Intel Xeon E5-2650 CPU, compiling with the Intel C++ Compiler version 12.1.3, using OpenMP to parallelise over eight threads, and using the Mersenne Twister pseudorandom number generator (PRNG) [25] as implemented in the Boost.Random library [26]. The second device is an NVIDIA S2050 GPU hosted by the same CPU, compiling with CUDA 5.0 and the same version of the Intel compiler, and using the XORWOW PRNG [27] from the CURAND library [28]. All compiler optimisations are applied.

Figure 3 shows the surface contours of the mean execution time across N and y for each algorithm, as well as marking up the fastest algorithms on each device, and across both devices. Execution times are taken until the delivery of an ancestry vector satisfying (2), and so include any of the auxiliary functions in §III necessary to achieve this. Note that, as we would expect, execution times of the multinomial, stratified and systematic resamplers are not sensitive to y – or equivalently to the variance in weights – while the Metropolis and rejection resamplers are.

As the resampling step is performed numerous times throughout the particle filter, it is not unreasonable to compare resampling algorithms on mean execution time alone. Nonetheless, some variability in execution time is expected, especially for the rejection resampler. This is shown in Figure 4 by taking a horizontal transect across each surface in Figure 3 at $y = 1$, then plotting N versus execution time separately for each device. The execution time of the sorting and ESS procedures is added to these plots for context.

To assess the variability in resampling draws for each algorithm we compute the mean-square error in the outcome for each weight-set, based on the offspring vector $\mathbf{o}$ [15]:

$$\frac{1}{N}\sum_{i=1}^{N}\left(\frac{o^i}{N} - \frac{w^i}{\textsc{Sum}(\mathbf{w})}\right)^2. \tag{14}$$

We then take the mean of this across all 500 weight sets, and the square-root of this, to arrive at root-mean-square error (RMSE) as in Figure 5. This figure plots the RMSE for both $y = 1$ and $y = 3$. Nothing in this figure is surprising: it is well known that the stratified resampler reduces variance over the multinomial resampler, and that the systematic resampler can, but does not necessarily, reduce it further [14]. Notably

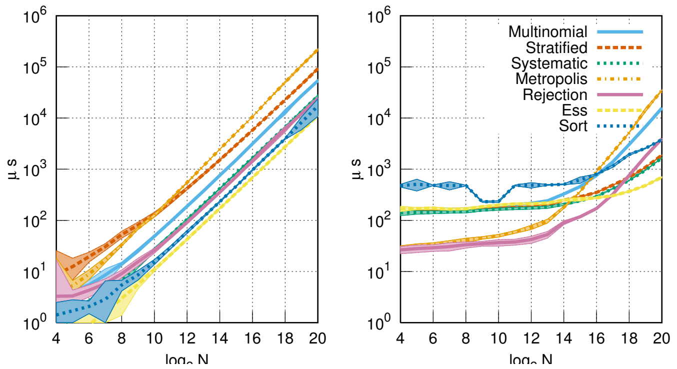

Fig. 4. Execution times against $\log_2 N$ at $y = 1$ for **(left)** the CPU and **(right)** the GPU. All resampling algorithms are shown, along with the Sort and ESS procedures for context.

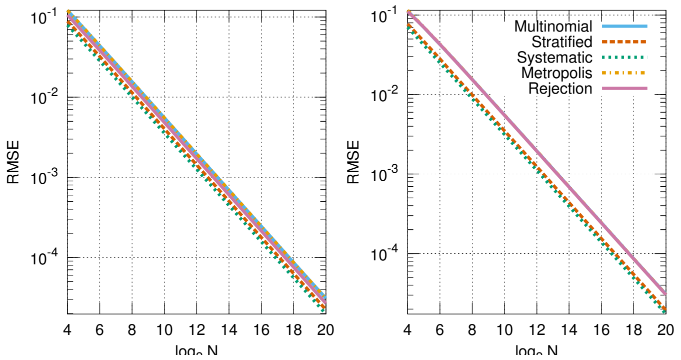

Fig. 5. Root-mean-square error (RMSE) of the various resampling algorithms at **(left)** $y = 1$ and **(right)** $y = 3$.

the RMSE of the Metropolis resampler appears to match that of the multinomial resampler, suggesting that the procedure in §II-D for setting the number of steps, B, is appropriate. Also of interest is that as y increases, the probability of the initial proposal of the rejection resampler being accepted declines, and so its RMSE degrades toward that of the multinomial and Metropolis resamplers.

V. Discussion

On raw speed, the Metropolis and rejection resamplers are worth considering for faster execution times on GPU. This is largely due to their avoidance of collective operations across all weights, which better suits the GPU architecture. They do not scale as well in either N or y, however, so that the stratified and systematic resamplers are faster at larger values of these. The Metropolis resampler scales worst of all in this regard. If there exists a good upper bound on the weights, the rejection resampler is faster, easier to configure and unbiased with respect to the Metropolis resampler. The Metropolis resampler is more configurable, however, and allows the tuning of B to trade off between execution time and bias. This may be particularly advantageous for applications with hard execution time constraints, such as real-time object tracking. Recall that the rejection resampler is somewhat configurable by using an approximate maximum weight, if one is willing to accept a still-weighted output from the resampling step.

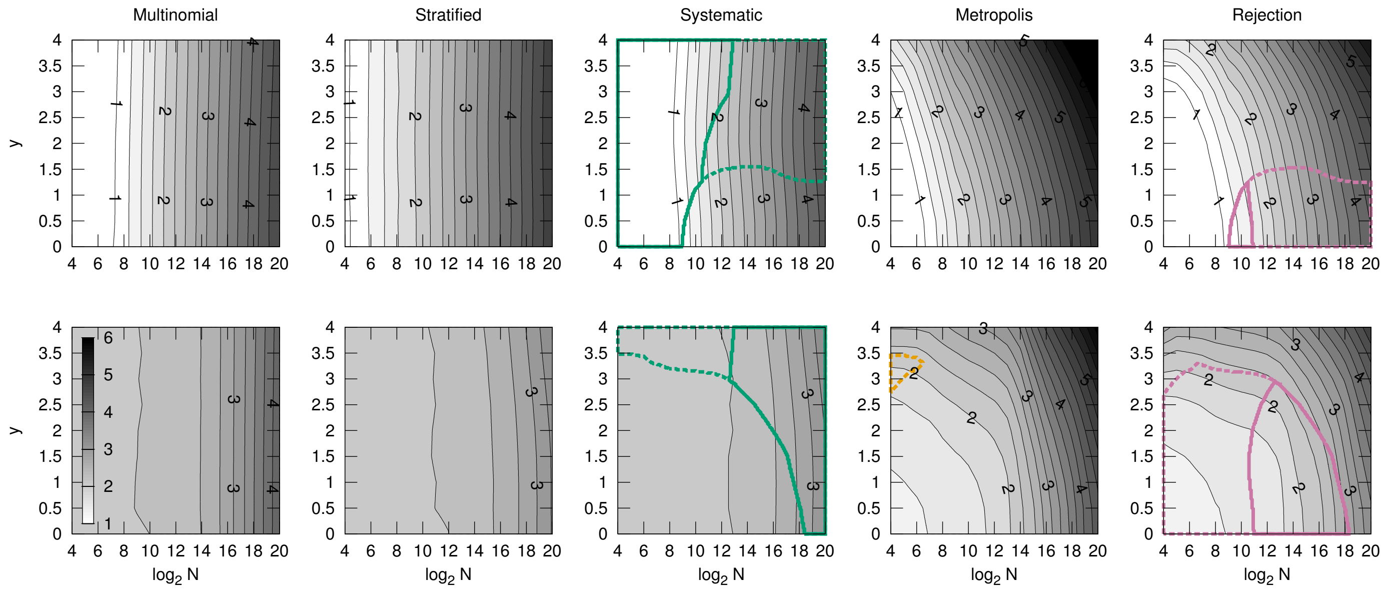

Fig. 3. Mean execution times for algorithms on each device, **(top row)** CPU and **(bottom row)** GPU. Each surface shows the base-10 logarithm of mean execution time across $\log_2 N$ (where N is the number of particles) and observation y, with prior distribution over particles $x \sim \mathcal{N}(0, 1)$ and likelihood function $y \sim \mathcal{N}(x, 1)$. Dashed lines demarcate the region, if any, where an algorithm is the fastest in its row. Solid lines demarcate where each algorithm is the fastest overall.

A further consideration is that the execution time of both the Metropolis and rejection resamplers depends on the PRNG used. This dependence is by a constant factor, but can be substantial. Here, robust PRNGs for Monte Carlo work have been used, but conceivably cheaper, if less robust, PRNGs might be considered as another trade-off between execution time and bias.

One should still be aware that the stratified resampler is variance-reducing with respect to the multinomial resampler, and that the systematic resampler can give further reductions [14]. This may be a more important consideration than raw speed, depending on the application. The rejection resampler, with its first deterministic proposal, has the nice property that it can also improve upon the variance of the multinomial resampler. The Metropolis resampler is expected to match the multinomial resampler on variance, and indeed any variation can only be attributed to the bias introduced by finite B.

A special consideration on GPUs is the use of single-precision floating-point operations to improve execution time. For a number of particles upwards of hundreds of thousands, plausible in modern applications, it is worth emphasising again that great care should be taken in the use of the stratified and systematic resamplers due to numerical instability. The Metropolis and rejection resamplers have better numerical properties, as they compute only ratios of weights. In double precision, the number of particles required to have the same issue far surpasses that which is realistic at present, so numerical stability is unlikely to be an issue.

References

[1] N. Gordon, D. Salmond, and A. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," *IEE Proceedings-F*, vol. 140, pp. 107–113, 1993.

[2] A. Doucet, N. de Freitas, and N. Gordon, Eds., *Sequential Monte Carlo Methods in Practice*. Springer, 2001.

[3] *CUDA C Programming Guide*, NVIDIA Corporation, July 2012.

[4] OpenCL. Khronos Group. [Online]. Available: www.khronos.org/opencl/

[5] CUDA. NVIDIA Corporation. [Online]. Available: www.nvidia.com/cuda

[6] M. Bolić, P. M. Djurić, and S. Hong, "Resampling algorithms for particle filters: A computational complexity perspective," *EURASIP Journal on Applied Signal Processing*, pp. 2267–2277, 2004.

[7] ——, "Resampling algorithms and architectures for distributed particle filters," *IEEE Transactions on Signal Processing*, vol. 53, pp. 2442–2450, 2005.

[8] G. Hendeby, J. D. Hol, R. Karlsson, and F. Gustafsson, "A graphics processing unit implementation of the particle filter," in *15th European Signal Processing Conference*, 2007.

[9] C. Lenz, G. Panin, and A. Knoll, "A GPU-accelerated particle filter with pixel-level likelihood," in *Vision, Modeling, and Visualization 13th International Fall Workshop*, 2008.

[10] 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, pp. 769–789, 2010.

[11] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010. [Online]. Available: thrust.github.com

[12] S. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithms for GPUs," NVIDIA, Tech. Rep. NVR-2008-003, 2008.

[13] J. L. Bentley and J. B. Saxe, "Generating sorted lists of random numbers," Carnegie Mellon University, Computer Science Department, Tech. Rep. 2450, 1979. [Online]. Available: http://repository.cmu.edu/compsci/2450

[14] R. Douc and O. Cappe, "Comparison of resampling schemes for particle filtering," in *Image and Signal Processing and Analysis, 2005. ISPA 2005. Proceedings of the 4th International Symposium on*, 15-17 2005, pp. 64 – 69.

[15] G. Kitagawa, "Monte Carlo filter and smoother for non-Gaussian non-linear state space models," *Journal of Computational and Graphical Statistics*, vol. 5, pp. 1–25, 1996.

[16] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *Journal of Chemical Physics*, vol. 21, pp. 1087–1092, 1953.

[17] L. M. Murray, "GPU acceleration of the particle filter: The Metropolis resampler," in *DMMD: Distributed machine learning and sparse representation with massive data sets*, 2011. [Online]. Available: arxiv.org/1202.6163

[18] C. Andrieu, A. Doucet, and R. Holenstein, “Particle Markov chain Monte Carlo methods,” *Journal of the Royal Statistical Society Series B*, vol. 72, pp. 269–302, 2010.
[19] A. E. Raftery and S. Lewis, “How many iterations in the Gibbs sampler?” in *Bayesian Statistics 4*. Oxford University Press, 1992, pp. 763–773.
[20] P. Del Moral, P. E. Jacob, A. Lee, L. M. Murray, and G. W. Peters, “Feynman-Kac particle integration with geometric interacting jumps.” [Online]. Available: arxiv.org/1211.7191
[21] L. M. Murray, “GPU acceleration of Runge-Kutta integrators,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, pp. 94–101, 2012.
[22] T. Aila and S. Laine, “Understanding the efficiency of ray traversal on GPUs,” in *Proc. High-Performance Graphics 2009*, 2009, pp. 145–149.
[23] J. S. Liu and R. Chen, “Sequential Monte-Carlo methods for dynamic systems,” *Journal of the American Statistical Association*, vol. 93, pp. 1032–1044, 1998.
[24] ——, “Blind deconvolution via sequential imputations,” *Journal of the American Statistical Association*, vol. 90, pp. 567–576, 1995.
[25] M. Matsumoto and T. Nishimura, “Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator,” *ACM Transactions on Modeling and Computer Simulation*, vol. 8, pp. 3–30, 1998.
[26] Boost C++ libraries. [Online]. Available: www.boost.org
[27] G. Marsaglia, “Xorshift RNGs,” *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003.
[28] *CUDA Toolkit 5.0 CURAND Library*, NVIDIA Corporation, July 2012.

Appendix

We offer a proof of the termination of Code 12.

First note that PREPERMUTE leaves $\mathbf{d}$ in a state where, excluding all values of $N+1$, the remaining values are unique. Furthermore, in PERMUTE the conditional on line 4 means that the loop on line 7 is only entered for values of i that are not represented in $\mathbf{d}$.

For each such i, the while loop traverses the sequence $x_0 = i$, $x_n = d^{x_{n-1}}$, until $d^{x_n} = N+1$. For the procedure to terminate this sequence must be finite. Because each x_n is an element of the finite set $\{1,\dots,N\}$, to show that the sequence is finite it is sufficient to show that it never revisits the same value twice. The proof is by induction.

1) As no value of $\mathbf{d}$ is i, the sequence cannot revisit its initial value $\mathbf{x}_0 = i$. The element x_0 is therefore unique.
2) For $k \geq 1$, assume that the elements of $x_{0:k-1}$ are unique.
3) Now, $x_{0:k}$ is *not* unique if there exists some $j \in \{1,\dots,k-1\}$ such that $x_k = d^{x_{k-1}} = x_j = d^{x_{j-1}}$, with $x_{j-1} \neq x_{k-1}$ by the uniqueness of $x_{0:k-1}$. But this contradicts the uniqueness of the (non $N+1$) values of $\mathbf{d}$. Thus the elements of $x_{0:k}$ are unique, the sequence is finite, and the program must terminate. $\square$