Received March 7, 2019, accepted April 1, 2019, date of current version April 19, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2910163

Improved Parallel Resampling Methods for Particle Filtering

MATTHEW A. NICELY, (Member, IEEE), AND B. EARL WELLS, (Member, IEEE)
Department of Electrical and Computer Engineering, The University of Alabama in Huntsville, Huntsville, AL 35805-1911, USA
Corresponding author: Matthew A. Nicely (man0003@uah.edu)

ABSTRACT Particle filter techniques are common methods used to estimate the evolving state of nonlinear, non-Gaussian time-variant systems by utilizing a periodic sequence of noisy measurements. The accuracy of particle filter methods has often been shown to be superior to other state estimation techniques, such as the extended Kalman filter (EKF), for many applications. Unfortunately, the high computational cost and highly nondeterministic runtime behavior of particle filters often preclude their use in hard, real-time environments, where filter response must meet the strict timing requirements of the application. Particle filter algorithms are composed of three main stages: prediction, update, and resampling. General purpose graphics processing units (GPGPUs) have been successfully employed in previous research to accelerate the computation of both the prediction and update stages by exploiting their natural fine-grain parallelism. This research focuses on accelerating the resampling stage for GPGPU execution, which has been much more difficult to parallelize due to it's apparent inherent sequentially. This paper introduces a novel GPGPU implementation of the systematic and stratified resampling algorithms that exploit the monotonically increasing nature of the prefix-sum and the evolutionary nature of the particle weighting process to allow the re-indexing portion of the algorithms to occur in a two-phase, multi-threaded manner. This resulting measured factor of performance improvement for the systematic and stratified algorithms was 15x and 32x, respectively, over the serial implementations.

INDEX TERMS Graphics processing units, parallel algorithms, parallel architectures, parallel programming, particle filters, state estimation, resampling.

I. INTRODUCTION

The concept of particle filtering [1] was introduced in 1993 to numerically solve general nonlinear, non-Gaussian state-space estimation problems. Since that time, the base particle filtering approach has been expanded and refined as it has been successfully applied to such applications as navigation [2], image and signal processing [3], [4], robotics [5], economics [6] and self-localization [7]. Particle filtering is an iterative method that utilizes a finite set of particles to represent the posterior distribution of some stochastic process. It is composed of three main phases: 1) prediction, the generation of new particles, 2) update, the computation of particle weights, and 3) resampling, removing weights with little significance. A major issue with particle filtering has been the large amount of computation that is required.

Over the years, prediction and update steps have been modified to utilize hardware that allows for parallelization. Unfortunately, due to its serial nature, resampling has remained a bottleneck of the particle filter. Common resampling algorithms from literature include multinomial, residual, stratified, and systematic. There have been attempts to mitigate this issue, such as Metropolis [8], a parallel resampling method, and the Rao-Blackwellized [9] filter, which is a hybrid of Kalman and particle filters reducing the number of particles required. Stratified and systematic resampling were chosen for this research because of their underlying *for-while* loop structure. The only difference between the two being random number generation. To demonstrate the performance of this novel parallel approach, timing results are compared to those of Metropolis and Rejection, which were chosen because they are a natural fit for graphics processing unit (GPU).

GPUs were not always the parallel processing powerhouses they are today. At their conception, GPUs were to provide the current hardware with a more efficient work flow for graphics processing. The original GPUs were modeled after the concept of a graphics pipeline and used fixed

The associate editor coordinating the review of this manuscript and approving it for publication was Mouloud Denai.

purpose hardware. The graphics pipeline refers to the process and steps of rendering images to a computer display. Even though transferring more of the graphics pipeline to the GPU progressed significantly through the 1980's and 1990's, they still required much help from the central processing unit (CPU). It was not until 1999, when NVIDIA implemented the last step of the pipeline in hardware, that reliance on the CPU was eliminated and thus the first consumer GPU was created. NVIDIA was the first to coin the term GPU.

One pitfall of the graphics pipeline was that it only allowed one pixel output per clock cycle, meaning CPUs could still send more triangles, the basic texture facet used in graphics processing, to the GPU than it could handle. This leads the way to introducing multiple pipelines in parallel on a GPU. Another downside to early GPUs is that the pipeline was a fixed-function pipeline meaning once graphics data entered the pipeline it could not be modified. This was fixed with the introduction of a programmable pipeline. A programmable pipeline allows a programmer access to manipulate and operate on data while it is in the pipeline. application programming interfaces (APIs) such as Brook and Sh removed the need for programmers to reformulate computational problems into terms of graphics primitives. Newer languages such as NVIDIAâĂŹs Compute Unified Device Architecture (CUDA) allows the user to focus on high-performance computing concepts and less on earlier basic graphical concepts.

This paper is organized as follows: Sections II provides a background of particle filters along with information on systematic and stratified resampling algorithms. Section III introduces GPGPUs, the programming model of CUDA and discusses some recently developed resampling algorithms designed for GPU processing. Section IV dives into the novel parallel implementation of the systematic and stratified algorithms and covers some inherent limitations. Section V discusses coding experiments, including a performance comparison, and ends with the conclusion in Section VI.

II. PARTICLE FILTERS

A particle filter is a recursive Bayesian technique for estimating the state of a dynamic system. The Bayesian method constructs a probability density function (PDF) of the state based on all available information, including received measurements and contains available statistical information. This is useful because for most nonlinear/non-Gaussian problems there is not a general analytic expression for the desired PDF.

A. BACKGROUND

The particle filter offers an alternative way of representing and recursively generating an approximation to the state PDF. The idea is that any PDF can be represented as a set of samples, or particles. Particle filters are considered optimal as $N \to \infty$, where N is the number of particles. The general dynamic system consists of a system model with process noise and a measurement model with measurement noise expressed as

$$x_{t+1} = f_t(x_t, w_t) \tag{1}$$

$$z_t = h_t(x_t, e_t) \tag{2}$$

Here x_{t+1} is the propagated state variable at time t, z_t is the update measurement, w_t is the process noise, e_t is the measurement noise, and $\boldsymbol{f},\boldsymbol{h}$ are two arbitrary nonlinear functions, representing the dynamic system and incoming measurement data, respectively. The noise densities are independent and are assumed to be known. In a Bayesian setting there is a two-step framework, prediction and update.

The prediction step, or *a priori*, $p(x_t|z_{1:t-1})$ is computed from the filtering distribution $p(x_{t-1}|z_{1:t-1})$ at time $t-1$.

$$p(x_t|z_{1:t-1}) = \int \overbrace{p(x_t|x_{t-1})}^{\substack{\text{system}\\\text{model}}} \overbrace{p(x_{t-1}|z_{1:t-1})}^{\substack{\text{previous}\\\text{posterior}}} dx_{t-1} \tag{3}$$

where $p(x_{t-1}|z_{t-1})$ is assumed known due to recursion and $p(x|x_{t-1})$ is given by Equation 1. During the update step, or *a posteriori*, the *a priori* is updated with new measurement data z_t.

$$p(x_t|z_{1:t}) = \frac{\overbrace{p(z_t|x_t)}^{\substack{\text{measurement}\\\text{model}}} \overbrace{p(x_t|z_{1:t-1})}^{\substack{\text{current}\\\text{prior}}}}{\underbrace{p(z_t|z_{1:t-1})}_{\text{normalization constant}}} \tag{4}$$

A mathematical tutorial of a particle filter is presented by Arulampalam [10].

Initially, particle filters consisted of only two steps, prediction and update. During the prediction step, particles are propagated through the system model to obtain *a priori* distribution at a given time step. The update step takes new measurement information and evaluates the likelihood of the prior samples and then obtains the normalized weight of each sample. The recursive execution of the two step method makes up what is commonly called sequential importance sampling (SIS). SIS is a modification of importance sampling [11] without changing the past simulated trajectories [12]. Unfortunately, it suffers from a phenomenon known as *weight degeneracy* where the variance between particle weights increases with every time step, typically exponentially with N [13]. Over time, almost all of the particles have weights equal to zero while one or a few particles contain most of the weight causing poor approximation of the filtering estimates.

B. RESAMPLING

While early forms of particle filters offered an alternative for nonlinear/non-Gaussian state estimation, it was the introduction of the resampling step [1] that made them a viable option in real-world applications. Before resampling, weights of the particles would be updated iteratively in time as the next observation became available. With no method to discard weights with low or no discernible effect, the variance between particle weights will increase with time [13].

After a series of iterations, most particles retain a negligible weight and resources will be wasted propagating useless particles. A simple approach is to add an exorbitant number of particles, but the computational workload makes this impractical for most applications.

Since its conception, resampling has proven to be beneficial both practically and theoretically [5]. Resampling is performed by removing particles with small weights and replacing them with neighboring particles with high weights. Once the remaining particles have been redistributed, all weights are set to a constant value of $1/N$. A visual aid is provided in Figure 1.

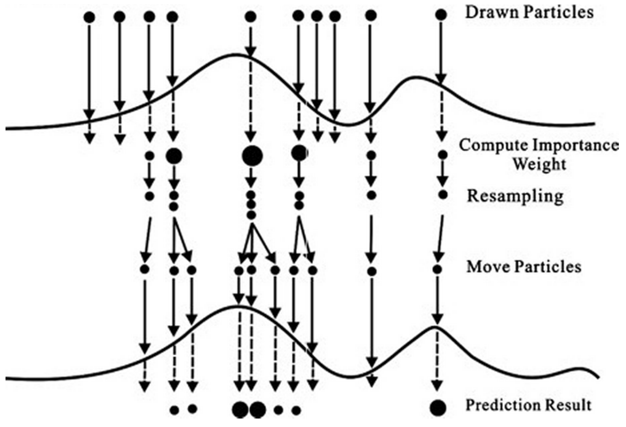


FIGURE 1. Traditional resampling.

A common measure of the degeneracy of the algorithm is the effective sample size (ESS), where N_{eff} is given by:

$$N_{eff} = \frac{1}{\sum_{i=1}^{N}(w_t^i)^2} \tag{5}$$

and w_t is the particle weight [5]. Traditionally, resampling is executed when the ESS drops below a given threshold, N_T. The threshold is expressed as a proportion of the number of particles and is sometimes defaulted to 50% [14]. This is because most resampling algorithms are serial by nature and create a bottleneck. Historically, this has caused programmers to balance performance and accuracy.

It is important to note that the act of resampling can have the adverse effect of introducing *sample impoverishment* when the system process noise is small. This is especially true for the *total sampling* technique, where the resulting particle set after resampling only contains the set of new-born particles. One approach to solve this problem is the resample-move algorithm described by Gilks and Berzuini [15]. The resample-move algorithm has a move step after the resampling step based on Markov chain Monte Carlo (MCMC) sampling. The move step is performed on each particle to rejuvenate the diversity. Other approaches to alleviate sample impoverishment can be found in [16]–[18].

Besides effects on accuracy, resampling also reduces the parallelism of a particle filter. Popular unbiased resampling algorithms include multinomial [1], residual [19], stratified [20], and systematic [21]. For these, the computational workload increases with the number of particles. A popular parallel resampling algorithm is Metropolis [8]. While the algorithm is more computationally intensive, it can benefit significantly from a GPGPU implementation. Metropolis works well under certain assumptions but can be biased if the search depth is chosen poorly. A recent survey by Li *et al.* [22] provides greater details on various resampling algorithms which are not in the scope of this paper. The primary focus of this paper is to introduce a novel parallel approach for implementing the stratified and systematic algorithms on a GPU.

1) STRATIFIED

Stratified resampling was first proposed by Kitagawa [20]. In the algorithm, it is assumed that division into strata, or layers, is performed. In each stratum, random numbers are drawn independently. The stratified method has a complexity of $\mathcal{O}(N)$ due to the prefix-sum at line 2, random number generation at line 3, and *for* loop iteration at line 5 in Algorithm 1.

Algorithm 1 Stratified Resampling

```
Input: $w \leftarrow$ Particle Weights
Output: $idx \leftarrow$ Resample Index
1:  $N \leftarrow$ count $(w)$
2:  $c \leftarrow$ Inclusive-Prefix-Sum $(w)$
3:  $u \leftarrow ((n-1)+u_n)/N$    $\triangleright\ u_n \sim U[0,1); n = 1, \ldots, N$
4:  $k \leftarrow 1$
5:  for $i \leftarrow 1, N$ do
6:      while $c(k) < u(i)$ do
7:          $k \leftarrow k+1$
8:      end while
9:      $idx(i) \leftarrow k$
10: end for
```

2) SYSTEMATIC

Systematic resampling is very similar to stratified except that it tries to reduce the discrepancy of particles by choosing the strata and their number of samples more effectively. This is achieved by choosing only one uniform random number and adding it to the entire ordered set. The samples are no longer independent and at the same position in the stratum as shown at line 3 of Algorithm 2. Systematic also has a complexity of $\mathcal{O}(N)$. It is more efficient and often the preferred algorithm due to its simplistic implementation and its ability to minimize Monte Carlo variations.

III. PARALLEL PROCESSING

With the introduction of programmable pipelines and streaming multiprocessors (SMs), GPUs began to be utilized as GPGPUs, performing calculations in applications conventionally performed by the CPU. An SM has multiple cores that can access and execute multiple threads or operations simultaneously. Although GPUs can only process independent fragments, it can do them in parallel. Where a CPU might

Algorithm 2 Systematic Resampling

```
Input: $w \leftarrow$ Particle Weights
Output: $idx \leftarrow$ Resample Index
1: $N \leftarrow$ count $(w)$
2: $c \leftarrow$ Inclusive-Prefix-Sum $(w)$
3: $u \leftarrow ((n-1)+u_0)/N$    $\triangleright\ u_0 \sim U[0,1); n = 1,\ldots,N$
4: $k \leftarrow 1$
5: for $i \leftarrow 1, N$ do
6:     while $c(k) < u(i)$ do
7:         $k \leftarrow k+1$
8:     end while
9:     $idx(i) \leftarrow k$
10: end for
```

have 4, 8, 16, or 32 cores, a GPU can have up to a couple of thousand. These cores are accessed through kernels, which are functions that work on each element. GPUs are extremely efficient at single instruction, multiple data (SIMD) or data parallelism. GPU processing can perform mathematically intensive computations on very large data sets, while a CPU can run the operating system and perform traditional serial tasks. This is an example of heterogeneous processing, which refers to systems that use more than one kind of processor.

A. GPU PROGRAMMING MODEL

With the increasing popularity of GPUs outside of the graphics domain, the CUDA API was introduced by NVIDIA in 2006. CUDA offers a mature development environment via an extension to the C/C++ programming language. An alternative to CUDA is OpenCL, provided by the Khronos Group, in 2008. It is an open and royalty-free standard that can be utilized on a wide selection of hardware including multi-core CPUs, GPUs (AMD and NVIDIA), field-programmable gate arrays (FPGAs), and digital signal processors (DSPs). While the programming scheme is similar, it does not provide the same performance as CUDA on NVIDIA GPUs, since CUDA is tied closer to the hardware. This can become important when trying to achieve optimal performance.

CUDA provides a heterogeneous environment where programs are divided between the *host* and the *device*. CUDA allows programmers to define special functions, called kernels, that are called by the host code to be executed in parallel on the GPU by a collection of threads. Kernels are launched in a *grid* made up of a group of *blocks* that contain numerous threads. Once blocks are distributed to SMs they are divided into *warps*. All warps contain 32 threads and are executed concurrently through a series of warp schedulers. Warps within a block can be launched randomly, and threads with a warp can execute random. Execution context (program counters, registers, etc.) for each warp processed by a multiprocessor is maintained on-chip during the entire lifetime of the warp.

While the number of threads per warp has stayed constant through the evolution of GPGPU development, the number of warp schedulers vary between architecture. Also, entire blocks are required to reside on a SM; therefore, the number of threads in a block are limited by resources. Threads are organized in a block, and blocks are organized in a grid in a one-, two-, or three-dimensional fashion. Each thread and block are designated with a unique ID that can be accessed in the kernel by a built-in variable `threadIdx.x (,y,z)` and `blockIdx.x (,y,z)`, respectively. With the release of CUDA 9.X, threads within a block are synchronized with Cooperative Groups. The compute capability of a device is represented by a version number, which identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU.

B. ALTERNATIVE GPU RESAMPLING ALGORITHMS

To understand the full capability of the novel parallel implementation, it will be compared to the serial implementation along with multiple alternative resampling algorithms, which are designed embarrassingly parallel and specifically for GPUs.

1) METROPOLIS

Metropolis resampling was designed specifically for GPUs as a way to accelerate resampling through parallelization [8]. Like multinomial and stratified, it requires the creation of a relatively large set of uniformly distributed random numbers. With Metropolis, weights are not summed cumulatively or normalized; therefore, removing dependency between weights and improving parallelization. The concept is to execute N threads in parallel over B iterations comparing randomly selected particles. If the weight of a randomly selected particle is greater, then it shall be selected for further comparisons, see line 8 in Algorithm 3. However, if the weight of the randomly selected particle is smaller, it shall be selected with probability equal to the ratio of the weights. After B comparisons, the current particle is passed for replication. Careful consideration must be taken when choosing B. If B is too small, the sample size will be more biased and possibly not converge. The selection of B is a trade-off of performance and accuracy.

2) REJECTION

In the same paper, Murray *et al.* [8], states that if an upper bound of the particle weights is known, rejection sampling is possible. Similar to Metropolis, rejection sampling does not require collective operation, such as prefix-sum, and is not affected by particle sets larger than 2^{20}. It also offers the following advantages.

1) it is unbiased
2) it permits a first deterministic proposal that $a^i = i$

A major difference between Metropolis and rejection resampling methods is thread execution. Thread execution in Metropolis is deterministic because the number of iterations of the inner for-loop is set to B. On the other hand, the inner loop of rejection is a while-loop, at line 6 of Algorithm 4.

Algorithm 3 Metropolis Resampling

```
Input: $w \leftarrow$ Particle Weights
Output: $idx \leftarrow$ Resample Index
1:  $N \leftarrow$ count $(w)$
2:  $B \leftarrow$ Number of iterations
3:  for $i \leftarrow 1, N$ do
4:      $p \leftarrow i$
5:      for $j \leftarrow 1, B$ do
6:          $u \sim U[0, 1]$
7:          $q \sim U\{1, \dots, N\}$
8:          if $u \leq w(q)/w(p)$ then
9:              $p \leftarrow q$
10:         end if
11:     end for
12:     $idx(i) \leftarrow p$
13: end for
```

Algorithm 4 Rejection Resampling

```
Input: $w \leftarrow$ Particle Weights
Output: $idx \leftarrow$ Resample Index
1:  $N \leftarrow$ count $(w)$
2:  $B \leftarrow$ Number of iterations
3:  for $i \leftarrow 1, N$ do
4:      $p \leftarrow i$
5:      $u \sim U[0, 1]$
6:      while $u \leq w(p)/w_{max}$ do
7:          $p \sim U\{1, \dots, N\}$
8:          $u \sim U[0, 1)$
9:      end while
10:     $idx(i) \leftarrow p$
11: end for
```

This causes the run-time of independent threads to vary, which is an example of a *variable task-length problem* [23]. Thread execution efficiency will be covered in greater detail in Section IV-A.2. It is important to ensure that the upper bound, w_{max}, is tight. Otherwise the method may perform poorly. While empirical calculations of w_{max} can be performed, $w_{max} = max\{w^1, \dots, w^N\}$, it would defeat the purpose of the approach by introducing a collective operation. Due to the variable task-length, Metropolis may be the preferred choice if its bias is acceptable and an appropriate B is selected.

3) COALESCED METROPOLIS

Dülger *et al.* [24] recently improved performance of Metropolis by implementing it on a GPU with coalesced memory accesses to global memory. While Metropolis resampling is well suited to utilize the multi-core architecture of a GPU, it does not perform efficient global memory accesses. This is because of the way it randomly generates an index in line 7, of Algorithm 3, and reads that element from the particle weight array. This has a negative impact on performance because the device tries to coalesce global memory loads and stores issued by threads of a *warp* into as few transactions as possible to minimize DRAM bandwidth. A warp is a maximal subset of threads from a single cooperative thread array (CTA), such that the threads execute the same instructions at the same time. This will be explained in detail in Section IV-A. When concurrent threads simultaneously access memory addresses that are very far apart in physical memory, there is no chance for the hardware to combine the accesses. Uncoalesced accesses can cause up to a 57.0% performance loss [25].

Dülger provides two methods, both of which are faster than Metropolis, but at the expense of quality. The two techniques are designated as Metropolis-C1 (abbreviated C1) and Metropolis-C2 (abbreviated C2). From this point forward, the original, or uncoalesced, Metropolis will be referred to simply as Metropolis. Because read/write operations to global memory are performed in segments, these modifications constrain threads within a warp to only read and write within these segments. A segment is defined as a fixed number of contiguous elements. All the threads in the warp select random weights within a segment.

Algorithm 5 Metropolis-C1 Resampling

```
Input: $w \leftarrow$ Particle Weights
Output: $idx \leftarrow$ Resample Index
1:  $N \leftarrow$ count $(w)$
2:  $B \leftarrow$ Number of iterations
3:  for $i \leftarrow 1, N$ do
4:      $p \leftarrow i$
5:      $s \sim U\{1, \dots, SC\}$
6:      for $j \leftarrow 1, B$ do
7:          $u \sim U[0, 1]$
8:          $q \sim U\{(s-1) * DC + 1, \dots, s * DC\}$
9:          if $u \leq w(q)/w(p)$ then
10:             $p \leftarrow q$
11:         end if
12:     end for
13:     $idx(i) \leftarrow p$
14: end for
```

C1 is presented in Algorithm 5, SC is the number of s-segments, and DC is the number of elements in each segment. All threads in a warp will select the same s, where s is the index of the select segment drawn from a uniform distribution. This can be ensured by using the warp index in the random number generator. Next, q is a random index between the first and last elements of the segment. Although not explicitly stated in the paper, if DC is greater than the size of a warp, currently 32, then it is possible that global memory accesses for that warp will not be coalesced. By definition, a coalesced read occurs when a warp can access all required memory locations in a single access. This is explained in detail in the CUDA Programming Guide [26] under the Best Practices section.

C2 in Algorithm 6 has the same parameters as Algorithm 5. The only difference is when the selection of s is performed.

Algorithm 6 Metropolis-C2 Resampling

```
Input: $w \leftarrow$ Particle Weights
Output: $idx \leftarrow$ Resample Index
1:  $N \leftarrow$ count $(w)$
2:  $B \leftarrow$ Number of iterations
3:  for $i \leftarrow 1, N$ do
4:      $p \leftarrow i$
5:      for $j \leftarrow 1, B$ do
6:          $u \sim U[0, 1]$
7:          $s \sim U\{1, \ldots, SC\}$
8:          $q \sim U\{(s-1) * DC + 1, \ldots, s * DC\}$
9:          if $u \leq w(q)/w(p)$ then
10:             $p \leftarrow q$
11:         end if
12:     end for
13:     $idx(i) \leftarrow p$
14: end for
```

For C2, it is performed in the inner-loop during each iteration of B. C2 is slower than C1 because of the additional random numbers generated in the inner loop, but provides higher quality results because it encounters more variety selection of weights. To reiterate, while C1 and C2 are faster than Metropolis, they produce worse quality because they select weights from a limited portion of the particle weight array. Therefore, C1 and C2 variations of Metropolis provide a spectrum of speed versus quality trade-off for users.

IV. PARALLEL SYSTEMATIC/STRATIFIED RESAMPLING

While systematic resampling is the algorithm most preferred due to its easy implementation and ability to minimize Monte Carlo variation [5], [10], [27] because of its similarities with stratified the implementation presented below can easily be applied to both. They can be divided into three sections: 1) a *prefix-sum*, 2) random number generation, 3) comparing prefix-sum and random numbers. Unfortunately, systematic and stratified, as well as some other traditional resampling algorithms, requires collective operations across particles and weights due to data dependencies. This collective operation is the cumulative summation of particle weights, also called a prefix-sum. Major contributions have been made to parallelize prefix-sum algorithms on GPUs [28]–[31]. The parallel inclusive-prefix-sum goes to $\mathcal{O}(\log N)$; therefore, it effectively disappears as N continues to grow. Next, generating random numbers for these resampling algorithm can have a complexity of $\mathcal{O}(N)$ on a CPU. On GPUs, random number generation can be distributed among threads bringing its complexity to $\mathcal{O}(1)$. This provides a more pronounced improvement to stratified resampling. The third portion of systematic and stratified resampling, which requires marching through particle weights and comparing the prefix-sum to the uniform ordered random numbers, still remains a serial process.

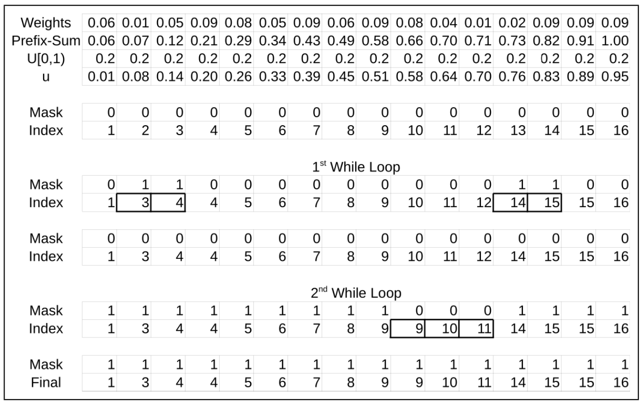
To efficiently utilize the GPU, work must be distributed to many threads. Taking a closer look at systematic resampling, it is a while-loop wrapped in a for-loop iterated over

Algorithm 7 Parallel Stratified (Systematic)

```
Input: $w \leftarrow$ Particle Weights
Output: $idx \leftarrow$ Resample Index
1:  $c \leftarrow$ Inclusive-Prefix-Sum $(w)$
2:  for all $t \leftarrow thread$ do
3:      $N \leftarrow$ count $(w)$
4:      $u_t \leftarrow (t + u_n(u_0))/N$          ▷ $u_n(u_0) \sim U[0, 1)$; $n = 1, \ldots, N$
5:      $m_t \leftarrow \texttt{true}$             ▷ Bit mask for thread $t$
6:      $\ell_t \leftarrow 0$
7:      while $m_t \neq \texttt{false}$ do
8:          if $t > (N - \ell_t)$ then
9:              $m_t \leftarrow \texttt{false}$
10:         else
11:             $m_t \leftarrow c(t + \ell_t) < u_t$
12:         end if
13:         if $m_t = \texttt{true}$ then
14:             $idx_t \leftarrow idx_t + 1$
15:         end if
16:         $\ell_t \leftarrow \ell_t + 1$
17:     end while ▷ All mask threads must be FALSE to exit
18:     $\ell_t \leftarrow 1$
19:     while $m_t \neq \texttt{true}$ do
20:         if $t < \ell_t$ then
21:             $m_t \leftarrow \texttt{true}$
22:         else
23:             $m_t \leftarrow c(t - \ell_t) < u_t$
24:         end if
25:         if $m_t = \texttt{false}$ then
26:             $idx_t \leftarrow idx_t - 1$
27:         end if
28:         $\ell_t \leftarrow \ell_t + 1$
29:     end while ▷ All mask threads must be TRUE to exit
30: end for
```

a zero-based *consecutive strictly monotonic* index of the prefix-sum set comparing elements to a random number. This process can be split into two subprocesses; a *middle-out* approach where each process executes a while-loop on each element in the set simultaneously. One process increments through the prefix-sum until each and all threads have satisfied the comparator. The second process then decrements through the prefix-sum. Comparator results are stored in a bit mask. This implementation produces a variable task-length problem similar to that of rejection resampling, in Section III-B.2; therefore, it has a maximum complexity of $\mathcal{O}(\ell)$, where ℓ is the number of strides from *c[thread]*. Algorithm 7 provides pseudocode of the parallel implementation; for brevity, stratified and systematic implementation have been combined. Notice that line 8 during the incrementing process and line 20 in the decrementing process ensures that the while-loops do not access memory out of bounds.

The parallelized systematic and stratified are identical except for creation of uniform random numbers. Both parallel methods should have similar timing since random number generation is performed by each thread.



Weights	0.06	0.01	0.05	0.09	0.08	0.05	0.09	0.06	0.09	0.08	0.04	0.01	0.02	0.09	0.09	0.09
Prefix-Sum	0.06	0.07	0.12	0.21	0.29	0.34	0.43	0.49	0.58	0.66	0.70	0.71	0.73	0.82	0.91	1.00
U[0,1)	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
u	0.01	0.08	0.14	0.20	0.26	0.33	0.39	0.45	0.51	0.58	0.64	0.70	0.76	0.83	0.89	0.95
Mask	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1st While Loop																
Mask	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0
Index	1	$\boxed{3}$	$\boxed{4}$	4	5	6	7	8	9	10	11	12	$\boxed{14}$	$\boxed{15}$	15	16
Mask	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	1	3	4	4	5	6	7	8	9	10	11	12	14	15	15	16
2nd While Loop																
Mask	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1
Index	1	3	4	4	5	6	7	8	9	$\boxed{9}$	$\boxed{10}$	$\boxed{11}$	14	15	15	16
Mask	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Final	1	3	4	4	5	6	7	8	9	9	10	11	14	15	15	16

FIGURE 2. **A simplified example of the parallel systematic resampling approach. The index for each particle (column) can be computed by individual threads in parallel.**

A simplified output of Algorithm 7 is provided in Figure 2. In the first while-loop, 4 out of 16 elements satisfy the comparator at line 11; therefore, only those elements are incremented. When all threads fail the while condition, the elements proceed to the next while-loop. In the second loop, only 3 elements fail the comparator at line 23 and must be decremented. Once all threads satisfy the second while-loop, all that remains is the resampled index. Summing the total operations, what would have taken 23 operations using serial methods now can be completed in 4. Another advantage of this new method is that it allows the data to remain on the GPU without expensive copies to and from the CPU.

A. LIMITATIONS OF PARALLEL IMPLEMENTATION

While the parallel implementation provides a speedup over the serial versions, both kernels suffer from the following three issues that impede optimal performance:

1) Memory dependency stalls
2) Thread execution efficiency
3) Sub-optimal occupancy

In the following subsections, the issues will be described and the approach used to alleviate them.

1) MEMORY DEPENDENCY STALLS

When programming on GPUs, different implementation techniques can indirectly affect performance. There are two popular techniques to implement kernels using CUDA, *monolithic kernels* and *grid-stride loops*. A monolithic kernel utilizes a single large grid containing one thread per element and processes the entire array in one pass. Grid-stride loops deploy a small grid of threads and loops over the data one grid at a time. They also provides additional tuning capability by allowing configurable block grid sizes per kernel per device. It is often the preferred technique because it reduces the overhead of launching, maintaining and destroying additional blocks. It is also regarded as being more flexible, scalable, and portable. For simplicity, examples, using SAXPY, of both are shown in Technique 1 and Technique 2.

Technique 1 Monolithic Kernel

```
Input: $n \leftarrow$ size of array
1: tid = blockIdx.x * blockDim.x + threadIdx.x
2: if $tid < n$ then
3:     y[i] = a * x[i] + y[i]
4: end if
```

Technique 2 Grid-Stride Loop

ht!]

```
Input: $n \leftarrow$ size of array
1: tid = blockIdx.x * blockDim.x + threadIdx.x
2: gridSize = blockDim.x * gridDim.x
3: for i = tid; i<n; i + = gridSize do
4:     y[i] = a * x[i] + y[i]
5: end for
```

Unfortunately, the performance of grid-stride loops can break down if kernels are too heavily memory bound. Memory loads have a much higher latency than computations. Through the hardware warp scheduler, when a warp executes a memory load, it can be removed from the scheduler and another warp can be executed while data is retrieved from memory. This approach helps hide latency, or stalls, if there are enough blocks and/or arithmetic operations. If there are not enough blocks to occupy the warp scheduler, all warps can stall due to memory dependencies. This is the case for the naive implementation of Algorithm 7. There is one memory load, one comparator operation, and one arithmetic operation per thread. As ℓ, Algorithm 7, increases, the kernel must switch from grid-stride loops to a monolithic kernel avoid negative performance impact. Determining when to choose grid-stride loops or monolithic kernels are kernel specific, but starting with grid-stride loop is the preferred method.

Another solution to minimize memory stalls, in conjunction to adjusting kernel technique, is to reduce memory loads from global memory while maximizing compute operations per thread. This can be achieve by performing multiple warp-length loads, of consecutive elements, from the prefix-sum into shared memory. Now threads can perform comparator and arithmetic operations with less global memory loads. This shifts the cause of warp stalls from memory dependencies to pipe utilization. The optimal number loads is hardware dependent but should be a multiple of 32 so global memory accesses remain coalesced. For the improved implementation, a block size of 32 and 2 coalesced loads of into shared memory, 64 consecutive elements in total, was combined with a grid-stride loop with grid size equal to the number of maximum active blocks per SM multiplied by the number of SMs to achieve maximum performance. Assigning the number of blocks proportional to the number of SMs provides scalability to multiple GPU architectures [26], and is different per kernel and application. Further rationale for setting block size to 32 will be given in the Section IV-A.3.

It is important to note that care should be taken when utilizing shared memory on the CUDA. Because it is on-chip, shared memory has a much higher bandwidth and lower latency than local or global memory. To achieve this bandwidth, memory is divided into equally sized memory *banks* that can be accessed simultaneously. However, if multiple memory requests access the same bank, the request will be serialized. The one exception is when multiple threads within a warp access the same shared memory location, the result is broadcast to those threads. Devices with compute capability 3.x and higher have two banking mode options, successive 32-bit or 64-bit words. Because the calculations in this paper deal with single precision, 32-bit mode is chosen. The implementation presented in the previous paragraph does not introduce any bank conflicts. This is because the block size is equal to that of a warp, 32, and each thread is accesses independent consecutive elements.

2) THREAD EXECUTION EFFICIENCY

As mention previously, threads from a block are bundled into fixed-size warps. Threads within a warp must follow the same execution trajectory. Maximum thread execution efficiency is achieved when 100% of a warp's threads are active. Less than 100% means threads are inactive due to sub-optimal launch, early return, or predicated off due to control flow divergence. Due to the stochastic nature of the parallel implementation, it's efficiency is lessened through early return of thread through control flow divergence. As threads satisfy the comparator in a given while-loop, they enter an inactive state. This is critical to GPU performance because a warp cannot exit until all threads have finished the last instruction, and a block can not exit a SM until all warps have finished.

In the worst case, where there are 1024 threads, 32 warps with 32 threads, in a block, if only one thread is active that means 31 warps are active and consuming resources. This decreases the number of eligible warps per scheduler. Eligible warps are are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps either increase the number of active warps or reduce the time the active warps are stalled. The stochastic nature allows for the possibility of one thread executing while the rest of the block remains idle, limiting available resources. Due to each thread having a variable-task length caused by the while-loop, control flow divergence and early return are simply inherent to the implementation. For the naive parallel implementation, optimal performance is seen at 64 threads per block.

3) SUB-OPTIMAL OCCUPANCY

Another important consideration when programming on a GPU is *occupancy*. Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps and helps measure a kernel's ability to utilize resources of a SM. The number of blocks that can execute concurrently on an SM is limited by the multiple factors such as the number of blocks, warps, registers, and the amount of shared memory. A subset of resources available on a GTX 1080 is given in Table 1.

TABLE 1. NVIDIA GTX 1080 hardware specifications.

Property	Value
Architecture	Pascal
Clock Rate	1860 MHz
CUDA Cores	2560
Global Memory	8 GB
Shared Memory	48 KB
Register Count	65,536
SM Count	20
Max Threads Per SM	2048
Max Blocks Per SM	32

If a block requires too much of any one resource, it limits the number of active blocks on that SM. It is evident that this is the case with the parallel implementation. As previously stated, in order to minimize the effect of low thread execution efficiency, one must increase the number of active warps or reduce the time the active warps are stalled. For the improved parallel implementation both suggestions can be implemented by setting the block size to be equal to that of a warp, or 32 threads. Doing so, eliminates the scenario of one active thread in a warp in a block of multiple warps. Now if a block has only one active thread it only affects an individual warp and not many. The increases the ratio of eligible warps to active warps; therein boosting performance. One downside to this approach is that is causes the kernel to run at sub-optimal occupancy. This block size, in combination with the maximum number of blocks allowed per SM, limits the number of possible active threads to 1024. Because there are 2048 threads available per SM, the occupancy of each kernel is theoretically 50%. Note that while higher occupancy does not always mean better performance, lower occupancy always constrains the ability to hide memory latency [26]. Memory latency issues caused are minimized by utilizing shared memory as suggested in Section IV-A.1.

The novel parallel implementation presented in this paper consists of two kernels that are mutually exclusive. Fortunately, CUDA provides an additional level of concurrency in the form of *streams*. A stream is a sequence of operations that executes in issue-order on the GPU. Different streams may execute their commands, or kernels, concurrently or out of order with respect to each other. It is important to note that any kernel not explicitly designated to a specific stream is launched in the *default* stream, which is synchronous, or blocking. Running the two kernels in independent asynchronous, or non-blocking, streams will not increase the occupancy, but now the GPU has an opportunity to run blocks from both kernels concurrently as resources become available. A visual aid is provided in Figure 3.

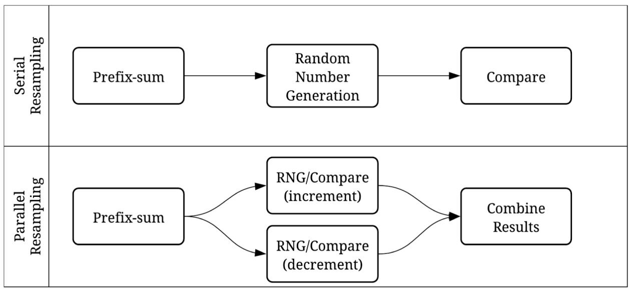


FIGURE 3. Algorithm flowchart: Serial vs. parallel.

TABLE 2. Intel i7-5960X hardware specifications.

Property	Value
Architecture	Haswell
Clock Rate	3.0 GHz
Overclock Rate	4.4 GHz
Cores	8
Threads	16
Cache	20 MB
Lithography	22 nm

When splitting the parallel implementation into two streams, care must be taken to ensure that both kernels use the same uniform distribution set and the output indexes are merged correctly before replacing insignificant particle weights. Once both kernels are complete, the particle filter can continue to the next step. A comparison of the naive and improved implementations of Algorithm 7 will be presented in Section V.

V. RESULTS

To examine the performance of the novel parallel approach introduced in this paper, the parallel implementation is compared to other alternatives using a general nonlinear/non-Gaussian four state-space model from research provided by Schon *et al.* [2] and can be found under software and data sets - Rao-Blackwelled particle filter at http://user.it.uu.se/~thosc112/research/. For simplicity, a bootstrap particle filter was used to run 100 Monte Carlos of 2500 samples with 2^{16} and 2^{20} particles, with resampling performed every time step. Particle set sizes do not exceed 2^{20} to eliminate possible numerical instabilities produced during the prefix-sum, as pointed out by Murray *et al.* [8] while using single-precision on GPUs. Double-precision on consumer-level cards and embedded systems is significantly slower.

These tests were performed using a desktop computer running Kubuntu 18.04 with a NVIDIA GTX 1080 and an Intel i7-5960X. Specifications are provided in Table 1 and Table 2, respectively. Serial versions of the systematic and stratified are run on the CPU with optimization level *-O3*. Code was written and tested in C++ GNU 7.3.0 and CUDA 10.0. The *–use_fast_math* flag was set at compile time to improve the use of any special functions by forcing the use of intrinsics. Intrinsic functions are faster as they map to fewer native instructions. A XORWOW generator was used from cuRAND [26] to generate all random number on the device. To ensure both while-loops of the parallelized systematic and stratified methods had the random numbers, a seed was generated on the CPU and sent to the GPU as a parameter to each kernel.

The prefix-sum, at line 1 can implemented in parallel using the library CUDA UnBound (CUB) [32]. It is an NVIDIA library containing collective kernel-level primitives designed around reusable software components such as warps and blocks for the single instruction, multiple thread (SIMT) paradigm. This library not only provides the benefit of simplified coding, but is optimized by NVIDIA to use the latest hardware features and provide sustainability when porting this method to different architectures.

To access and compare the quality of the serial and parallel resampling algorithms, the root mean squared error (RMSE) [33] is used in the following form

$$RMSE_{fo} = \sqrt{\sum_{i=1}^{M}\sum_{n=1}^{S}(f_n^i - o_n^i)^2/(M * S)} \tag{6}$$

where M is the number of Monte Carlo runs, S is the number of samples in each run, f is the expected results, or truth data, and o is the observed filter estimates.

Due to the stochastic nature of the resampling process, execution time is determined by averaging the execution time of all the Monte Carlos. Also, the first executed Monte Carlo was considered a warm-up run and the timing of that run was discarded.

TABLE 3. Metric comparison of systematic/stratified: 2^{16} particles.

Type	Speed	RMSE for States			
	(μs)	x_1	x_2	x_3	x_4
Systematic (CPU)	456	0.2061	0.1770	0.1662	0.1540
Stratified (CPU)	936	0.2061	0.1770	0.1662	0.1540
Systematic (N-GPU)	57	0.2060	0.1769	0.1661	0.1540
Stratified (N-GPU)	57	0.2060	0.1769	0.1661	0.1540
Systematic (I-GPU)	29	0.2060	0.1769	0.1661	0.1540
Stratified (I-GPU)	29	0.2060	0.1769	0.1661	0.1540

Table 3 shows timing and RMSE for all systematic and stratified implementations. The naive GPU (N-GPU) implementations of Algorithm 7 is 8× and 16.42× faster than the serial CPU versions of systematic and stratified, respectively. The improved GPU (I-GPU) implementations are 15.72× and 32.28× faster. Stratified execution times are similar to the systematic on the GPU because the random number generation can be performed on each thread in parallel. All three produced nearly identical RMSE. This is expected because the GPU implementations produce the exact resampling index as

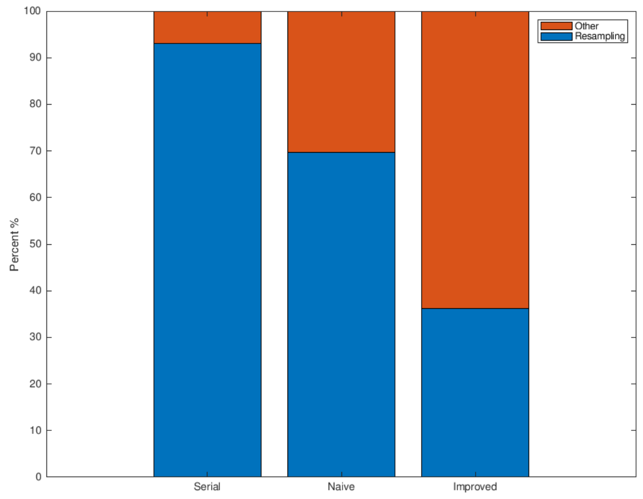


FIGURE 4. Workload percentages: Systematic resampling implementations.

serial methods. Differences can be contributed to rounding error produced during the prefix-sum computation. Timing results include the prefix-sum computation, random number generation, and resampling index search.

The additional speedup of the improved implementations over the naive is a direct correlation to the decreased number of global memory loads. Load quantities for 2^{20} particles for over 1000 samples are presented in Table 4. The improved implementation reduces the number of global memory loads over $10\times$ by utilizing shared memory.

TABLE 4. 32bit global memory loads: naive vs. improved.

2^{20} Particles	Naive	Improved
while_loop_increment	154,365,478	14,996,374
while_loop_decrement	152,593,046	13,611,845

As mentioned earlier, the historical bottleneck of particle filtering lied within the resampling step. This is evident in Figure 4, which shows that when running the traditional systematic resampling on the CPU it consumes roughly 93% of workload performed in a time step. While the naive parallel implementation is able to shift the 70%, it's the improved parallel implementation, with a workload of 32%, that transfers the majority of the computation performed from resampling to the prediction and update steps of particle filtering.

Table 5 shows speed and RMSE comparison between Metropolis, C1, C2; rejection; and the parallelized systematic and stratified methods for a particle set of 2^{20}. The results show that the parallelized implementations of the systematic and stratified methods provide at least a $11.64\times$ speed

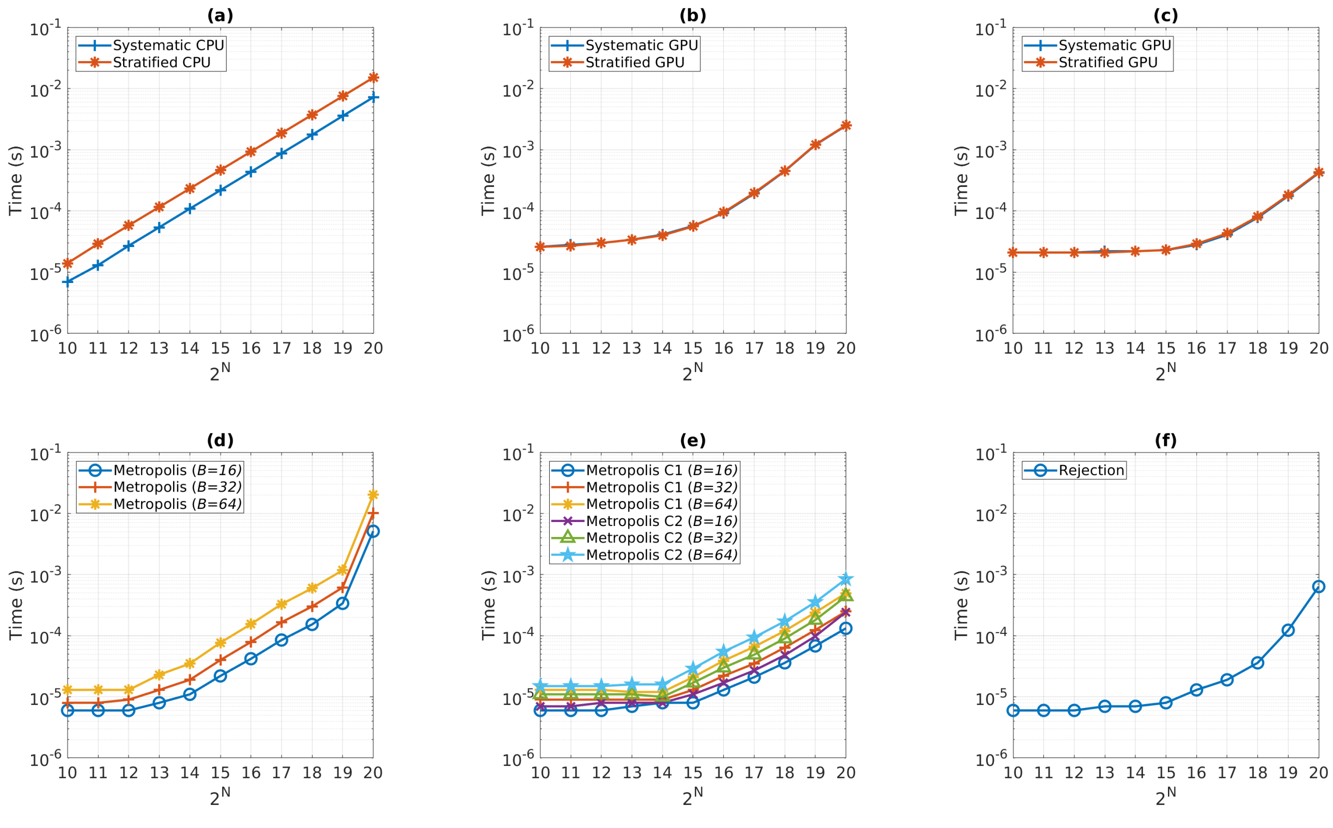


FIGURE 5. Execution times of the resampling algorithms in single-precision floating point arithmetic verses increasing number of particles (2^N): (a) serial systematic and stratified on CPU with −O3, (b) naive parallel systematic and stratified on GPU, (c) improved parallel systematic and stratified on GPU, (d) metropolis on GPU, (e) metropolis: C1 and C2 on GPU, (f) rejection resampling on GPU.

TABLE 5. Metric comparison of resampling techniques: 2^{20} particles.

Type	Cutoff (B)	Speed (μs)	RMSE for States			
			x_1	x_2	x_3	x_4
Systematic		425	0.2059	0.1769	0.1661	0.1539
Stratified		440	0.2059	0.1769	0.1661	0.1539
Metropolis	16	5122	0.2062	0.1772	0.1663	0.1540
	32	10158	0.2061	0.1771	0.1662	0.1540
	64	20226	0.2059	0.1769	0.1662	0.1540
Rejection		631	0.3352	0.2613	0.2123	0.1807
Metropolis (C1)	16	135	0.2192	0.1865	0.1725	0.1551
	32	259	0.2207	0.1871	0.1728	0.1551
	64	518	0.2216	0.1874	0.1730	0.1552
Metropolis (C2)	16	245	0.2062	0.1771	0.1663	0.1540
	32	452	0.2061	0.1771	0.1662	0.1540
	64	881	0.2061	0.1770	0.1662	0.1540

over Metropolis, and as B increases, that speed up becomes more pronounced. As expected, systematic and stratified techniques provide the best quality, with Metropolis in second. Although C1 provides the fastest execution time of all Metropolis implementations, it has the worst quality because it focuses on local selections, and the expected number of repetitions of the particles becomes different than that in Metropolis [24]. C2 provides a balance between speed and quality; speed because it is coalesced and quality because it is more similar to Metropolis. The parallelized systematic and stratified methods are slightly faster than C2 with $B \geq 32$, but nearly double the execution time with $B = 16$. With C2 producing quality only slightly worse for this particular test problem, it seems to be the best trade-off between speed and quality. It is evident that while rejection resampling is faster than Metropolis and C2, at $B \geq 64$, it has the worst quality out of all the methods, for this data set.

Variance of the particle weight array will play a significant roll in the performance of these resampling methods. For a given B, as the variance increases the execution time will remain constant while the quality will worsen. Conversely, execution time of the systematic and stratified parallel implementations will most certainly increase as threads are required to traverse further through the particle weight array to satisfy the while-loop condition.

Figure 5 compares execution times of all methods over a sweep of particle set sizes from 2^{10} to 2^{20}. For small particle sets, the CPU will perform better than all GPU methods. Execution times on a GPU will plateau for small particles sets because overhead is a larger contributor than computational workload. It can be seen that C1 and C2 are faster alternatives to the improved systematic and stratified parallel versions when B is small and might be an appealing choice if the impact to quality is not severe.

VI. CONCLUSION

In this article, a novel parallel method of the systematic and stratified resampling algorithms is presented that is well suited for GPU architecture. Improvements to thread execution efficiency, memory dependency stall and sub-optimal occupancy are then provided to decrease execution times of the systematic and stratified by 15.72× and 32.28× respectively over the serial method. With this parallel approach, all steps of the particle filter can be implemented in a parallel fashion. This shifts the historical workload particle filtering from resampling to the prediction and update steps. For a thorough comparison, the performance was compared to a popular GPU method known as Metropolis resampling. Three versions of the Metropolis are presented; the original version and two coalesced versions that performed more efficient memory reads to global memory. The parallelized systematic and stratified methods are significantly faster than Metropolis on large particle sets. But they are slightly slower than the coalesced versions of Metropolis when B is small. Therefore, the specific method that is chosen will be a trade-off between performance and accuracy. For future work, these techniques can be applied to embedded systems such as the TK1, TX1, TX2, or Xavier.

REFERENCES

[1] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," *IEE Proc. F-Radar Signal Process.*, vol. 140, no. 2, pp. 107–113, Apr. 1993.
[2] T. B. Schön, F. Gustafsson, and P.-J. Nordlund, "Marginalized particle filters for mixed linear/nonlinear state-space models," *IEEE Trans. Signal Process.*, vol. 53, no. 7, pp. 2279–2289, Jul. 2005.
[3] G. Hendeby, R. Karlsson, F. Gustafsson, and N. Gordon, "Performance issues in non-Gaussian filtering problems," in *Proc. IEEE Nonlinear Stat. Signal Process. Workshop*, Sep. 2006, pp. 65–68.
[4] K. Nummiaro, E. Koller-Meier, and L. Van Gool, "An adaptive color-based particle filter," *Image Vis. Comput.*, vol. 21, no. 1, pp. 99–110, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0262885602001294
[5] A. Doucet and A. M. Johansen, "A tutorial on particle filtering and smoothing: Fifteen years later," *Handbook Nonlinear Filtering*, vol. 12, nos. 656–704, p. 3, 2008.
[6] T. Flury and N. Shephard, "Bayesian inference based only on simulated likelihood: Particle filter analysis of dynamic economic models," *Econ. Theory*, vol. 27, no. 5, pp. 933–956, 2011.
[7] D. Weikersdorfer and J. Conradt, "Event-based particle filtering for robot self-localization," in *Proc. IEEE Int. Conf. Robot. Biomimetics (ROBIO)*, Dec. 2012, pp. 866–870.
[8] L. M. Murray, A. Lee, and P. E. Jacob, "Parallel resampling in the particle filter," *J. Comput. Graph. Statist.*, vol. 25, no. 3, pp. 789–805, 2016. [Online]. Available: https://doi.org/10.1080/10618600.2015.1062015
[9] G. Hendeby, R. Karlsson, and F. Gustafsson, "A new formulation of the Rao-Blackwellized particle filter," in *Proc. IEEE/SP 14th Workshop Stat. Signal Process.*, Aug. 2007, pp. 84–88.
[10] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking," *IEEE Trans. Signal Process.*, vol. 50, no. 2, pp. 174–188, Feb. 2002.
[11] J. Geweke, "Bayesian inference in econometric models using Monte Carlo integration," *Econometrica*, vol. 57, no. 6, pp. 1317–1339, 1989. [Online]. Available: http://www.jstor.org/stable/1913710
[12] A. Doucet, N. de Freitas, and N. Gordon, "An introduction to sequential Monte Carlo methods," in *Sequential Monte Carlo Methods in Practice*. New York, NY, USA: Springer, 2001, pp. 3–14. doi: 10.1007/978-1-4757-3437-9_1.
[13] A. Kong, J. S. Liu, and W. H. Wong, "Sequential imputations and Bayesian missing data problems," *J. Amer. Stat. Assoc.*, vol. 89, no. 425, pp. 278–288, 1994.
[14] P. E. Jacob, "Sequential Bayesian inference for implicit hidden Markov models and current limitations," *ESAIM, Proc. Surv.*, vol. 51, pp. 24–48, Oct. 2015.

[15] W. R. Gilks and C. Berzuini, "Following a moving target—Monte Carlo inference for dynamic Bayesian models," *J. Roy. Stat. Soc., B (Stat. Methodol.)*, vol. 63, no. 1, pp. 127–146, 2001. [Online]. Available: http://www.jstor.org/stable/2680638

[16] J. Zuo, "Dynamic resampling for alleviating sample impoverishment of particle filter," *IET Radar, Sonar Navigat.*, vol. 7, no. 9, pp. 968–977, Dec. 2013.

[17] T. Fetzer, F. Ebner, F. Deinzer, and M. Grzegorzek, "Recovering from sample impoverishment in context of indoor localisation," in *Proc. Int. Conf. Indoor Positioning Indoor Navigat. (IPIN)*, Sep. 2017, pp. 1–8.

[18] T. Li, S. Sun, T. P. Sattar, and J. M. Corchado, "Fight sample degeneracy and impoverishment in particle filters: A review of intelligent approaches," *Expert Syst. Appl.*, vol. 41, no. 8, pp. 3944–3954, Jun. 2014. doi: 10.1016/j.eswa.2013.12.031.

[19] E. R. Beadle and P. M. Djurić, "A fast-weighted Bayesian bootstrap filter for nonlinear model state estimation," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 33, no. 1, pp. 338–343, Jan. 1997.

[20] G. Kitagawa, "Monte Carlo filter and smoother for non-Gaussian nonlinear state space models," *J. Comput. Graph. Statist.*, vol. 5, no. 1, pp. 1–25, 1996. [Online]. Available: http://www.jstor.org/stable/1390750

[21] J. Carpenter, P. Clifford, and P. Fearnhead, "Improved particle filter for nonlinear problems," *IEE Proc.-Radar, Sonar Navigat.*, vol. 146, no. 1, pp. 2–7, Feb. 1999.

[22] T. Li, M. Bolić, and P. M. Djurić, "Resampling methods for particle filtering: Classification, implementation, and strategies," *IEEE Signal Process. Mag.*, vol. 32, no. 3, pp. 70–86, May 2015.

[23] L. Murray, "GPU acceleration of Runge-Kutta integrators," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 1, pp. 94–101, Jan. 2012.

[24] Ö. Dülger, H. Oğuztüzün, and M. Demirekler, "Memory coalescing implementation of metropolis resampling on graphics processing unit," *J. Signal Process. Syst.*, vol. 90, no. 3, pp. 433–447, 2018. doi: 10.1007/s11265-017-1254-6.

[25] D.-H. Kim, "Evaluation of the performance of GPU global memory coalescing," *J. Multidisciplinary Eng. Sci. Technol.*, vol. 4, no. 4, p. 5, 2017.

[26] *NVIDIA CUDA C Programming Guide, Version 9.1.85*, Nvidia Corporation, 2018.

[27] B. G. Sileshi, C. Ferrer, and J. Oliver, "Particle filters and resampling techniques: Importance in computational complexity analysis," in *Proc. Conf. Design Architectures Signal Image Process.*, Oct. 2013, pp. 319–325.

[28] M.-A. Chao, C.-Y. Chu, C.-H. Chao, and A.-Y. Wu, "Efficient parallelized particle filter design on CUDA," in *Proc. IEEE Workshop Signal Process. Syst.*, Oct. 2010, pp. 299–304.

[29] G. Hendeby, R. Karlsson, and F. Gustafsson, "Particle filtering: The need for speed," *EURASIP J. Adv. Signal Process.*, vol. 2010, Feb. 2010, Art. no. 22. [Online]. Available: https://dl.acm.org/citation.cfm?id=1928437. doi: 10.1155/2010/181403.

[30] P. Gong, Y. O. Basciftci, and F. Ozguner, "A parallel resampling algorithm for particle filtering on shared-memory architectures," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp. Workshops PhD Forum*, May 2012, pp. 1477–1483.

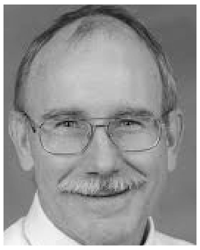
[31] S. Maskell, B. Alun-Jones, and M. Macleod, "A single instruction multiple data particle filter," in *Proc. IEEE Nonlinear Stat. Signal Process. Workshop*, Sep. 2006, pp. 51–54.

[32] D. Merrill. *CUDA Unbound*. [Online]. Available: http://nvlabs.github.io/cub/

[33] A. G. Barnston, "Correspondence among the correlation, RMSE, and Heidke forecast verification measures: Refinement of the heidke score," *Weather Forecasting*, vol. 7, no. 4, pp. 699–709, 1992. doi: 10.1175/1520-0434(1992)007<0699:CATCRA>2.0.CO;2.

MATTHEW A. NICELY (M'18) was born in Tuscaloosa, AL, USA, in 1985. He received the BSEE degree from Auburn University, Auburn, AL, USA, in 2009, and the MSEE degree from the University of Alabama in Huntsville (UAH), Huntsville, AL, USA, in 2014, where he is currently pursuing the Ph.D. degree in computer engineering focusing on algorithm optimizations on GPUs.

Since 2007, he has been with the U.S. Army Aviation and Missile Research Development and Engineering Center, Huntsville, AL, USA. He conducts research into parallel algorithm development and optimizations and embedded GPU environments for real-time hardware-in-the-loop for missile and UAV applications.



B. EARL WELLS (M'79) received the B.S., M.S., and Ph.D. degrees in electrical engineering from the University of Alabama, Tuscaloosa, in 1983, 1988, and 1992, respectively.

He is currently a Professor with the Department of Electrical and Computer Engineering, The University of Alabama in Huntsville, Huntsville. His research interests include re-configurable hardware systems, and parallel/distributed processing architectures and applications.