Parallel Resampling for Particle Filters on FPGAs

Shuanglong Liu, Grigorios Mingas, Christos-Savvas Bouganis
Department of Electrical and Electronic Engineering
Imperial College London
London, UK
Email: {s.liu13, g.mingas10, christos-savvas.bouganis}@imperial.ac.uk

***Abstract*—Particle filters (PFs) are a set of algorithms that implement recursive Bayesian filtering, which represent the posterior distribution by a set of weighted samples. Resampling is a fundamental operation in PF algorithms. It consists of taking a population of samples and reconstructing it based on the weights attached to each sample, favouring the samples with large weights. However, resampling is computationally intensive when the number of samples is large and, most importantly, it is not inherently parallelizable like the other steps of the particle filter. Parallel computing devices such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) have been proposed to accelerate resampling. In this paper, we propose novel parallel architectures that map four state-of-the-art resampling algorithms (systematic, residual systematic, Metropolis and Rejection resampling) to a FPGA. FPGA-specific optimisations are introduced to further optimize the performance of the above systems. The proposed architectures are implemented in a Virtex-6 LX240T FPGA device with half-utilization of logic resources. Compared to the respective state-of-the-art implementations on an NVIDIA K20 GPU, the achieved speedups are in the range of 1.7x-49x.**

I. Introduction

Particle Filters (PFs), also known as Sequential Monte Carlo (SMC) methods, are density estimation algorithms which are commonly used to infer the hidden state sequence of a state-space model, given a set of observations. They can efficiently handle non-linearity and/or non-Gaussianity in the model and they exhibit great robustness and accuracy. They have thus been widely used in target tracking, digital signal extraction, air traffic management and robot localization [1]–[4], among other applications. Although powerful, PFs are also computationally intensive, which becomes a major issue in its application to complex models, especially with real-time constraints [1].

Code 1 Particle Filter Algorithm

for $t = 1, \ldots, T$
  for each $i \in \{1, \ldots, N\}$
    1) *Sampling*: $\mathbf{x}_t^i \sim p(\mathbf{x}_t|\mathbf{x}_{t-1}^i)$;
    2) *Importance computation*: $w_t^i = p(\mathbf{y}_t|\mathbf{x}_t^i)$;
    3) *Resampling*:$\{\tilde{\mathbf{x}}_t^i\} \sim \{\mathbf{x}_t^i, w_t^i\}$;
  Output calculation: Calculate desired estimate of the state $\hat{\mathbf{x}}_t = \sum_{i=1}^{N} \tilde{\mathbf{x}}_t^i / N$.

PFs use a set of N particles (i.e. samples) to estimate the density of the state at each time step t. The most common PF algorithm (bootstrap filter) is shown in Code 1. For each time t and for each particle i, the following steps are performed. In the *sampling* step, each particle's state $\mathbf{x}_t^i$ is propagated to the next time step using the transition equation of the model. In the *importance computation* step, the likelihood of each particle given the observation $\mathbf{y}_t$ at the present time step is evaluated. This is the weight of the particle. Based on the values of the weights, a new set of particles is generated in the *resampling* step. In this step, particles with large weights are replicated several times while those with small weights are thrown away. Finally, the resampled particles are used to estimate the new state.

The *sampling* and *importance computation* steps are independent operations for each particle, so they are inherently parallel and straightforward to implement in parallel devices such as GPUs and FPGAs. Resampling however requires a collective operation (either a sum or a cumulative sum of all the weights), which makes it the most challenging step to parallelize. Moreover, resampling is crucial for the stability of the PF because it prevents the filter from weight degeneracy and improves the estimation of states by concentrating particles into domains of higher posterior probability [1]. However, parallelizability of the filter is affected by the *resampling* step, and the major part of time in the GPU implementation of PF is spent on resampling, where the time spent on the other three steps become almost negligible when the number of particles increases considerably [5].

This paper focuses on four state-of-the-art resampling algorithms: Residual Systematic Resampling (RSR), Systematic Resampling (SR), Metropolis Resampling and Rejection Resampling based on recent literature. Novel parallel architectures are proposed for each algorithm. RSR and ISR use a parallel SUM and CUMSUM (cumulative sum or pre-fix sum) operation respectively followed by parallel and pipelined offspring evaluators. On the other hand, Metropolis and Rejection architectures do not require collective operations, but the parallel blocks of these two algorithms must have global access to all the weights. Therefore, memory access strategies which implement a simplified Random Permutation Generator (RPG) are proposed for the parallel Metropolis and Rejection architectures. The main contributions of this work are:

1) The introduction of novel parallel architectures which map four resampling algorithms to a FPGA. This is the first work that presents parallel FPGA architectures for the state-of-the-art resampling algorithms and compares their execution time with that of GPU implementations;

2) Memory access strategies for parallel Metropolis and Rejection architectures. An optimized RPG circuit which uses a cyclic shifter is proposed to randomly forward weights from the memory to the parallel processing blocks and guarantee that all blocks have global access to the weights memory;

978-1-4799-6245-7/14/$31.00 ©2014 IEEE

3) A modified version of SR, which we call Improved Systematic Resampling (ISR), is introduced to save resources and achieve further speedup in hardware. Moreover, the advantages of each algorithm for parallel implementation are summarized.

II. Background

A. Resampling

The *resampling* step in PF aims to regenerate the particle population by removing particles with small weights and replicating particles with large weights. It can be considered as a randomised algorithm that takes as inputs the number of particles (N) and the weights of all particles ($w_i, i = 1, \ldots, N$). The weights can be normalized or non-normalized as the normalization can be performed either in the *importance computation* or *resampling* step. Here, the weights are assumed to be non-normalized. The output of the algorithm can be the number of offsprings of each old particle ($o_i, i = 1, \ldots, N$), i.e. how many times particle i is replicated. Alternatively, the output can be the index of the ancestor of each particle of the new particle population ($a_i, i = 1, \ldots, N$). As we shall see, each resampling algorithm naturally takes one form or the other, and it is easy to convert between the two forms.

The principle of resampling is to ensure that the offspring vector satisfies the following two conditions (1) and (2) [6]:

$$\sum_{i=1}^{N} o_i = N \tag{1}$$

$$E(o_i) = Nw_i/sum(\mathbf{w}) \tag{2}$$

The first equation ensures that the total number of the new resampled particles remains unchanged. The second equation shows that the expected value of the number of replications of a particle should be proportional to the value of its weight. The resampling quality is often quantified by how much the algorithm's result deviates from the expected value of equation (2). This is given by the relative root-mean-square error (RMSE) (3), computed from the offspring vector and weight vector [7].

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\frac{o_i}{N} - \frac{w_i}{sum(\mathbf{w})}\right)^2} \tag{3}$$

B. Related Work

There is exhaustive literature on parallelization of particle filters and resampling using GPUs and FPGAs. FPGA-based implementations of parallel particle filters are presented in [8]–[10] and more recent work can be found in [11], [12]. [8] proposes a new form of systematic resampling (RSR) and ways in which this algorithm can be parallelized. [9] and [12] introduce simplified partial resampling which performs resampling only in part of the particles. This reduces the execution time but negatively affects resampling quality. GPU-based implementations of resampling have been presented in [5], [13]–[15]. Both [13] and [5] employ the traditional systematic resampling (SR) but using different parallel implementations of the cumulative sum of the weights. Most previous works are with an emphasis on low-level implementation issues rather than algorithmic adaptation. In contrast, [15] presents a new form of SR and proposes two novel algorithms (Metropolis and Rejection) which are more readily parallelised in hardware. The main disadvantage for these two algorithms when implemented on a GPU is that they cause warp divergence [15] [16] due to branch statements, or different trip counts of loops. Also, Rejection resampling suffers from possibly frequent rejections and Metropolis from possibly large convergence times.

This work is the first to propose FPGA architectures for systematic, Metropolis and Rejection resampling and also proposes an improved FPGA implementation of RSR. It is the first work to include a comparison of the main state-of-the-art resampling algorithms when implemented on FPGAs, highlighting their advantages and demonstrating how each scales with the number of particles and the variance in weights of particles. Finally, it also provides a comparison on their execution times on FPGAs, GPUs and CPUs.

III. Resampling Algorithms

This section presents the four resampling algorithms, followed by a summary and comparison of these algorithms. The Improved Systematic Resampling (ISR) is also introduced to achieve further speedup in hardware implementation.

A. Residual Systematic Resampling (RSR)

The standard resampling algorithms (e.g. Multinomial, Stratified, Systematic resampling) are based on multinomial selection of o_i, which is equivalent to selecting with replacement N particles $\tilde{x}^j$ from the original set of particles $\{x^i\}$ where $P(\tilde{x}^j = x^i) = w_i, i, j = 1, \ldots, N$. Among these methods, Systematic Resampling (SR) is favourable over the others considering resampling quality and computational complexity [17] and thus it is the method of choice for most implementations, including those in FPGAs and GPUs. The original SR proposed in [17] has non-deterministic runtime which depends on the distribution of the weights, as it needs to precompute a cumulative sum of the weights and do a binary search. [8] proposed residual systematic resampling (RSR) as an alternative form in order to introduce deterministic runtime, and its pseudocode is given in Code 2. In RSR, the number of offsprings of a specific particle is determined in the **for** loop by truncating the product of the number of particles and the normalized weight using uniform random numbers. The random number is updated at each iteration as shown in line 6 of the pseudocode. As a result, the algorithm has a deterministic processing time. However the data dependency inside the resampling loop limits its potential parallelization.

Code 2 Residual Systematic Resampling

$\mathbf{o} = \text{RSR}(N, \mathbf{w}) : \mathbf{w} \in \mathbb{R}^N \rightarrow \mathbb{R}^N$
//non-normalized weights to replication factors

1: $sum = sum(\mathbf{w})$;
2: $u \sim U[0, 1)$;
3: **for** $j = 1$ **to** N **do**
4: $\quad temp = Nw_j/sum - u$;
5: $\quad o_j = \lfloor temp \rfloor + 1$;
6: $\quad u = o_j - temp$;
7: **end for**

B. Improved Systematic Resampling (ISR)

Recently, [15] presented another form of systematic resampling, which delivers the cumulative offspring vector i.e. O_j as shown in Code 3. This algorithm first calculates the cumulative sum of the weights, then truncates the product of the number of particles and the cumulative weight. As a result, it has no data dependency inside the resampling loop. As SR replicates the particle i, $o_j = \lfloor Nw_j \rfloor + 0/1$ times for any values of u in $[0, 1)$, the expected number of replications is consistent with (2). In this subsection, an improved SR algorithm (ISR), shown also in Code 3, is introduced to simplify the calculation of the systematic outputs using $u = 0$. This proposed modification removes the need to generate a uniform random number for each execution of SR, which will translate in resource savings in the FPGA, as will be shown in the next section. However, by setting $u = 0$, the resampling quality is slightly affected, which will be discussed in the Evaluation Section.

Code 3 Improved Systematic Resampling

$\mathbf{o} = \text{ISR}(N, \mathbf{w}) : \mathbf{w} \in \mathbb{R}^N \rightarrow \mathbb{R}^N$
//non-normalized weights to replication factors

1: **cw** $= cumsum(\mathbf{w})$;
2: $sum = cw_N$,$O_0 = 0$;
3: **for** $j = 1$ **to** N **do**
4: $O_j = \lfloor Ncw_j/sum \rfloor$; //$O_j = \lfloor Ncw_j/sum + u \rfloor$ in [15]
5: $o_j = O_j - O_{j-1}$;
6: **end for**

C. Metropolis and Rejection Resampling

[15] proposes two novel resampling algorithms which are more readily parallelized in hardware. The pseudocode of the two algorithms is presented in Code 4 and Code 5. Neither algorithm requires a collective sum or cumsum operation. Metropolis resampling is based on the well-known Metropolis algorithm and it requires the ratio between two weights at each step of the inner loop. After B steps for each weight (outer loop), the algorithm is assumed to have converged to the correct particle distribution implied by (2) (for more details see [15]). The selection of B assumes that the upper bound $sup\, w$ on non-normalized weights and the expected weight value are known. One can always use large B (resulting in increased execution time) to improve the resampling quality. However, this algorithm still produces a biased sample as B must be finite. Rejection resampling is based on the rejection sampling algorithm. The idea of this method is to propose ancestor indexes until an index is accepted based on the ratio in line 4. Compared to Metropolis resampling, Rejection resampling is easier to configure and unbiased but it has a non-deterministic runtime, since the number of **while** loop iterations in line 4 are unknown.

Code 4 Metropolis Resampling

$\mathbf{a} = \text{Metropolis}(N, \mathbf{w}) : \mathbf{w} \in \mathbb{R}^N \rightarrow \mathbb{R}^N$
//non-normalized weights to ancestor indexes

1: **for** $i = 1$ **to** N **do**
2: $k = i$;
3: **for** $n = 1$ **to** B **do**
4: $u \sim U[0, 1]$;
5: $j \sim U\{1, \ldots, N\}$;
6: **if** $u \leq w_j/w_k$ **then**
7: $k = j$;
8: **end if**
9: **end for**
10: $a_i = k$;
11: **end for**

Code 5 Rejection Resampling

$\mathbf{a} = \text{Rejection}(N, \mathbf{w}) : \mathbf{w} \in \mathbb{R}^N \rightarrow \mathbb{R}^N$
//non-normalized weights to ancestor indexes

1: **for** $i = 1$ **to** N **do**
2: $j = i$;
3: $u \sim U[0, 1]$;
4: **while** $u > w_j/sup\, w$ **do**
5: $j \sim U\{1, \ldots, N\}$;
6: $u \sim U[0, 1]$;
7: **end while**
8: $a_i = j$;
9: **end for**

D. Algorithm Summary

Both RSR and ISR require a collective operation over the weights, specifically sum and cumulative sum, which makes them less readily parallelised in hardware. ISR also needs an additional memory space to store the cumulative weights. Another disadvantage of these two algorithms is that the collective operation can exhibit numerical instability for large N or large weight variance [15]. This is more observable when using single-precision arithmetic instead of double-precision arithmetic. On the other hand, Metropolis and Rejection resampling can be parallelized more easily, due to the lack of collective operations (the outer loop iteration of both algorithms are completely independent). However, Metropolis results in increased complexity and a biased result, while Rejection's runtime is non-deterministic. FPGAs are more suitable than GPUs to implement Metropolis and Rejection algorithms, due to the GPU divergence problem mentioned above. Note that the output of the four algorithms takes one of the two forms described in Section II. Even though the form of the output will affect the overall PF architecture, this is out of the scope of this paper which focuses solely on the resampling stage.

IV. Proposed Resampling Architectures

In this Section, we propose optimized FPGA architectures for all four algorithms described in the previous section. First, the parallel architecture of RSR is presented based on [18], and novel parallel architectures are proposed for the other three algorithms. All architectures parallelize resampling by splitting the particles into M sub-sets of N/M particles each and assigning them to M parallel processing blocks. The form of the processing block differs between architectures. Parallel architectures to implement sum and cumulative sum on FPGAs are also introduced for RSR and ISR algorithms respectively. Furthermore, in order to satisfy that the multiple resampling blocks for Metropolis and Rejection architectures have global access to the weights memory at each iteration, an optimized Random Permutation Generator (RPG) is proposed to connect

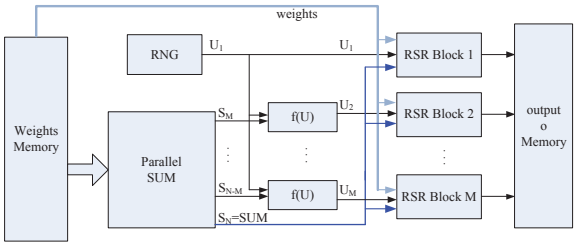

Fig. 1. Parallel architecture for RSR resampling

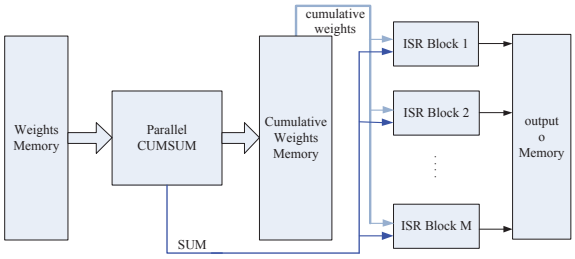

Fig. 2. Parallel architecture for the proposed ISR resampling

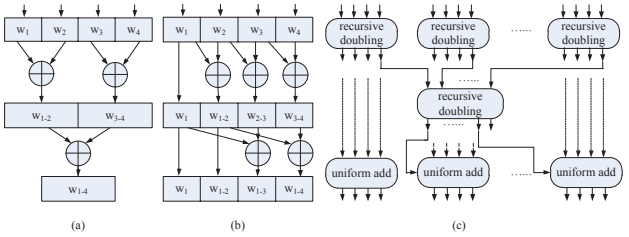

Fig. 3. The parallel sum and cumsum algorithms in the form of block diagrams ($w_{i-j} = \sum_i^j w$): ***(a)*** Tree SUM; ***(b)*** Recursive doubling CUMSUM for a small number of inputs; ***(c)*** Three-step recursive doubling CUMSUM for a large number of inputs.

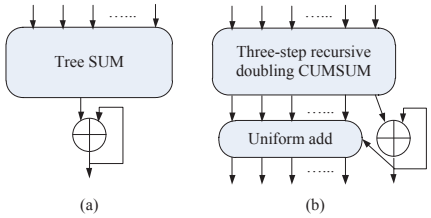

Fig. 4. Block diagram of the parallel sum and cumsum algorithms implemented on the FPGA: ***(a)*** Parallel SUM; ***(b)*** Parallel CUMSUM.

the weights memory and resampling blocks.

A. *Parallel Architectures for RSR and ISR*

The RSR and ISR architectures are shown in Figure 1 and 2 respectively. The M weight sets are stored in one memory unit where each memory address stores M weights (one from each set). This allows us to read M weights in the same cycle. Firstly RSR calculates the sum of the weights while ISR calculates the cumulative sum of weights and stores them in the cumulative weights memory. After the collective operations, M weights are assigned to the parallel RSR blocks and M cumulative weights to ISR blocks at each time for offspring evaluation. Following the evaluation, the offspring results are stored in the respective output memory.

RSR and ISR algorithms need a first step that computes the sum or cumulative sum of weights respectively. Although the sequential computation is straightforward, its parallelization in hardware is challenging due to the output data dependency. Parallel sum (SUM) and cumulative sum (CUMSUM) algorithms are described in [19] and shown in Figure 3. The sum algorithm is as adder tree. With respect to CUMSUM, recursive doubling is a naive parallel scan and needs many data exchange operations. The three-step recursive doubling algorithm is more suitable for a large number of inputs.

The proposed FPGA implementations for the two collective operations are shown in Figure 4. Both are based on using a large amount of parallel pipelined adders and feeding them with a new set of weights at each cycle. For both architectures, an accumulator is placed after the main datapath in order to accumulate the values of each weight set. Note that the SUM and CUMSUM can have different parallel degrees as the offspring evaluation of RSR and ISR. We use M_1 to represent the parallel degree of both SUM and CUMSUM. For CUMSUM, $M_1 - 1$ uniform adders are also necessary to produce the M_1 outputs (cumulative sums) for each weight set, which is illustrated in Figure 4. The execution time (in clock cycles) for SUM and CUMSUM can be reduced from $N + L_{SUM}$ and $N + L_{CSUM}$ cycles for a sequential implementation to $N/M_1 + L_{SUM}$ and $N/M_1 + L_{CSUM}$ cycles respectively, where L_{SUM} and L_{CSUM} are the latency of the SUM and CUMSUM datapaths respectively.

Following the collective operations, both RSR and ISR have to evaluate the number of offsprings of each particle (**for** loop in Code 2 and 3). These operations can also be parallelized. In contrast to ISR, RSR offspring evaluation cannot be straightforwardly implemented in parallel. [18] proposed a way to parallelize RSR by calculating the initial random number used for each block in advance. Assuming M offspring evaluation blocks are implemented for RSR and each block processes N/M weights independently, the random number used for each block is generated using the algorithm in Code 6 which is represented as $U_i = f(U)$ in Figure 1. The algorithm needs to compute the sum of the weights processed by each block, which is achieved from the pipelined SUM algorithm in Figure 4.

ISR is easier to parallelize due to lack of data dependency between loop iterations. The drawback is that one additional memory is needed to store the cumulative sum of the weights. Another difference between ISR and RSR is the absence of RNG in the ISR architecture as shown in Figure 2.

B. *Parallel Architectures for Metropolis and Rejection Resampling with Memory Access Strategies*

The Metropolis and Rejection resampling algorithms can be paralellized more easily due to the lack of collective operations between weights. Nevertheless, the memory access pattern of the two algorithms is different compared to that described in the previous section. While each RSR and ISR offspring evaluation block works on a sub-set of the weights independently, each Metropolis and Rejection resampling block

Code 6 Parallel computation of random numbers used for parallel RSR resampling architecture

1: $u_1 \sim U[0,1]$;
2: $sum = sum(w)$;
3: **for** $i = 2$ **to** M **do**
4: $\quad S_i = \sum_{j=1}^{(i-1)N/M} w_j$;
5: $\quad r_i = NS_i/sum - u_1$;
6: $\quad u_i = \lceil r_i \rceil - r_i$;
7: **end for**

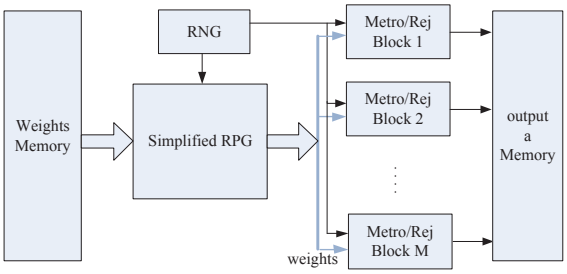

Fig. 5. Parallel architectures for Metropolis and Rejection resampling

requires global and randomized access to all the weights. This Section introduces 1) novel parallel architectures for Metropolis and Rejection resampling; 2) memory access strategies using the proposed Random Permutation Generator (RPG).

The parallel architectures for Metropolis and Rejection resampling based on the proposed memory access strategy are presented in Figure 5. The weights are stored in the same way as described for RSR and ISR. Here, an RPG circuit is introduced to handle the communication between the weights memory and the Metropolis or Rejection resampling blocks. This RPG circuit has to guarantee that each block has random and global access to all the weights, i.e. every block can choose any weight with equal probability (uniform sampling). At each cycle, M weights are read from the memory and they pass through this circuit which, by using the outputs of an RPG, randomly allocates the M weights to the parallel blocks. Then the indexes of the ancestors of the resampled particles are stored in the respective output memory.

The commonly used algorithm for an M-element RPG is the Knuth Shuffle, and its FPGA implementation is presented in [20]. The Knuth Shuffle RPG proceeds through $M-1$ steps and each step has operators including RNG, remainder and swap functions. As the purpose of the RPG here is to randomly distribute the weights to each block and complete permutations implementation is not necessary, we propose a simplified circuit with only one step using a cyclic shifter and $Mlog_2M$ bits RNG to implement an RPG in FPGA. The optimized RPG algorithm to be implemented in FPGA is described in Code 7. It shifts all values by some random number of bits and the result is a permuted sequence of indexes. This simplified RPG takes advantage of the hardware implementation in which all numbers are represented in binary, and thus leads to a reduction in resources and time compared to the Knuth Shuffle RPG. It still satisfies the condition that every element is chosen with equal probability at each place i.e. $p(Q_i = j) = 1/M$, for each $i = 0, \ldots, M-1$ and $j = 0, \ldots, M-1$. The disadvantage of this algorithm when compared to the Knuth Shuffle RPG is that the number of total permutations is reduced from $M!$ to $Mlog_2M$. Nevertheless, this is shown to affect the resampling quality of parallel Metropolis and Rejection algorithms only minimally (see Section V-A). Therefore, this optimized RPG is proposed to randomly allocate the weights read from the memory to each parallel resampling block of Metropolis and Rejection architectures as shown in Figure 5.

Code 7 The simplified RPG in FPGAs

1: Initialization: Q $\sim$ permutation of the numbers $0 : M-1$ in binary (log_2M bits for each and $Mlog_2M$ bits in total);
2: $i \sim U\{0, 1, \ldots, Mlog_2M - 1\}$;
3: Out = Cyclic shifter (Q, i);
4: Output: Out $\sim$ a random permutation of $\{0, \ldots, M-1\}$ as a binary representation.

V. Evaluation and Experiments

The four proposed architectures are implemented on a Xilinx Virtex-6 LX240T FPGA. All arithmetic operators are taken from the Xilinx Coregen library in single floating point precision. Uniform random numbers are generated using the cores described in [21]. As mentioned before, two parameters (M_1, M) are used for the parallel RSR and ISR architectures, where M_1 represents the parallel degree of SUM and CUMSUM, and M represents the parallel degree of the offspring evaluation. Accordingly, only one parallel degree M for the Metropolis and Rejection architectures is considered since no sum or cumulative sum is needed for these two algorithms.

A. Resampling Quality

We first assess the resampling quality of the proposed architectures, i.e. how close the resampled particle set is to the ideal given by (2). We use the same simulated weights as the ones used in [15], which is the standard way to assess the resampling quality, and this allows us to easily compare to the results of the GPU implementations in [15]. The simulated weights are generated based on the following equation:

$$w_i = \frac{1}{\sqrt{2\pi}} exp(-\frac{1}{2}(y_e^i - y_o)^2) \quad (4)$$

y_o represents the time varying observation value and y_e^i represents the estimated values based on the particles $\mathbf{x}^i$ in the *sampling* step in Code 1. These weights (which are the likelihoods of particles fitting the observation) are produced based on the assumption that the importance function of the PF is a Gaussian, i.e. $y_o \sim \mathcal{N}(y_e^i, 1)$. Multiple weight sets are generated, with varying particle number N and $y = mean(\mathbf{y}_e) - y_o$, i.e. $y_e^i - y_o \sim \mathcal{N}(y, 1)$. The parameter y indicates the relative variance in weights. Increasing y means that the relative variance in weights increases too.

First, the resampling quality of the four algorithms is compared using the RMSE given by the equation (3). The RMSE of the SR algorithm in [15] is also shown as a point of reference. Experiments are done for $N = 2^4, 2^5, \ldots, 2^{20}$ and for $y = 1$ to $y = 4$. The results are shown in Figure 6 and 7. The results lead to the following conclusions: ***1)*** the proposed ISR has the same resampling quality as the original Systematic and RSR; ***2)*** RSR and ISR give a lower RMSE than

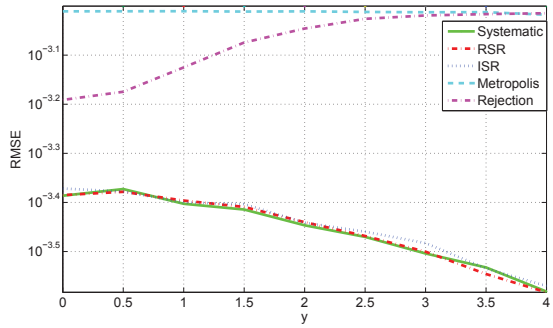

Fig. 6. Root-mean-square error (RMSE) of the resampling algorithms for various relative variances in weights at $N = 2^{10}$.

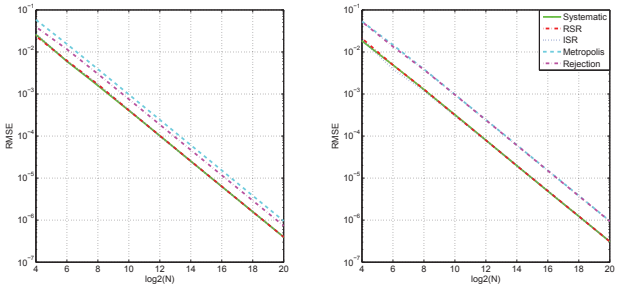

Fig. 7. RMSE of the resampling algorithms for various particle numbers at $y = 1$ **(left)** and $y = 3$ **(right)**.

Metropolis and Rejection resampling, which results from the fact that the Metropolis is a biased sampler and the Rejection is affected by the distribution of uniform random numbers used as the indexes; ***3)*** The RMSE of Rejection is lower than that of Metropolis for small variance in weights, but it converges to the RMSE of Metropolis when y increases.

The resampling quality of the FPGA implementations of ISR and RSR does not change with respect to the degree of parallelism. In contrast, the resampling quality of the parallel implementations of Metropolis and Rejection algorithms can be different compared to the sequential ones, since we use the global memory access strategy described previously. We consider four cases: 1) the sequential implementations; 2) parallel implementations with the Knuth Shuffle RPG; 3) parallel implementations with the simplified RPG; 4) parallel implementations without an RPG. In the last case, each block of Metropolis and Rejection architectures works on a sub-set of weights just like RSR and ISR, and only has access to the corresponding sub-set of weights. In cases 2 and 3 each block has global access to all of the weights as in the sequential one because the RPG is used. We check the RMSE quality of the draws for both algorithms and the average number of proposed indexes (**while** loop iterations in Code 5) until an index is accepted for each weight for Rejection resampling. The average number is taken from 100 independent runs. The results are shown in Figure 8.

The results confirm that there is no quality loss using the memory access strategies with the Knuth Shuffle RPG or our simplified RPG for parallel execution of Metropolis and Rejection resampling compared to the sequential execution. Nevertheless, the absence of RPG has a large impact on the resampling quality as the resampling is only performed inside each block. It becomes even worse when the variance of

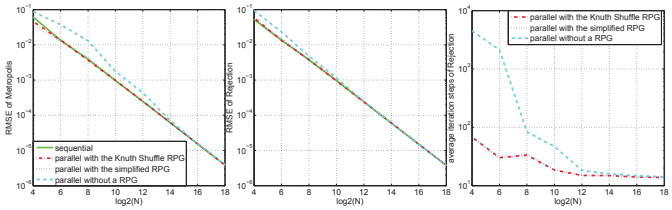

Fig. 8. Test results of different memory access strategies for parallel Metropolis and Rejection implementations with the parallel degree $M = 32$ (which is the maximum degree achieved in our target FPGA device with half-utilization of logic): ***(left)*** RMSE of Metropolis draws at $y = 4$; ***(middle)*** RMSE of Rejection draws at $y = 3$; ***(right)*** the average number of iteration steps for each weight of Rejection at $y = 3$.

weights i.e. y increases or the weights are sorted. The worst case happens when all the non-zero weights are processed by only one resampling block (the results are omitted due to lack of space). Another drawback of not using an RPG for parallel Rejection resampling is that it causes largely increasing execution time as more steps need to run before the acceptance of each weight. In conclusion, the simplified RPG-based memory access strategy permits the parallel algorithms to achieve the same resampling quality as the sequential one. Moreover, the simplified RPG consumes much less resources and has a lower latency compared to the Knuth Shuffle RPG.

B. Resource Utilization and Execution Time

Table I gives the resource utilization (measured in floating point adders and accumulators), and the total number of clock cycles for N weights of the parallel SUM and CUMSUM architectures shown in Figure 4 when using the parallel degree M_1. L_1 and L_2 are the latency of the adder and accumulator respectively.

TABLE I. RESOURCES AND EXECUTION TIME FOR PARALLEL SUM AND CUMSUM ALGORITHMS IN FPGA (PARALLEL DEGREE M_1)

Resources & CLK cycles	SUM	CUMSUM
Adders	$M_1 - 1$	$1.5M_1 + \frac{M_1}{4} log_2 M_1 - 3$
Accumulators	1	1
CLK cycles	$N/M_1 + L_1 log_2 M_1 + L_2$	$N/M_1 + L_1 log_2 M_1 + L_1 + L_2$

Table II shows the resource utilization and total number clock cycles (for a complete resampling operation with N weights) for all four architectures, given the parallel degrees M_1 and M. The memory utilization is also shown here.

Each architecture uses M blocks for offspring evaluation or index generation, while the RSR and ISR also require resources for SUM or CUMSUM implementation. Metropolis and Rejection architectures need extra resources for the simplified RPG which uses a $M log_2 M$ bits Cyclic shifter and also for multiplexers to select data from the memory based on the outputs of the RPG. Note that the $M - 1$ blocks which implement Code 6 can be reused from the offspring evaluation blocks. Regarding memory utilization, ISR architecture needs one more memory to store the cumulative weights compared to the other three architectures. For the RSR, ISR and Metropolis architectures, all the concurrent blocks produce output streams in parallel at the same clock cycle so the outputs from each block can be written to the memory together. Therefore, only

TABLE II. RESOURCES AND EXECUTION TIME OF RSR AND ISR WITH PARALLEL DEGREE (M_1, M), OF METROPOLIS AND REJECTION WITH PARALLEL DEGREE M

		RSR scheme	ISR scheme	Metropolis	Rejection
Resources		SUM (parallel degree M_1) M RSR blocks	CUMSUM (parallel degree M_1) M ISR blocks	M Metropolis blocks $Mlog_2M$ bits Cyclic shifter	M Rejection blocks $Mlog_2M$ bits Cyclic shifter
Weights Memory	Number of Memories	1	2	1	1
	Total Size (bits)	$32 * N$	$32 * (2N)$	$32 * N$	$32 * N$
Output Memory	Number of Memories	1	1	1	M
	Total Size (bits)	$log_2N * N$	$log_2N * N$	$log_2N * N$	$log_2N * N$
CLK cycles		$\frac{N}{M_1} + \frac{N}{M} + L$ $L = 9log_2M_1 + 80$ (-35 if $M = 1$)	$\frac{N}{M_1} + \frac{N}{M} + L$ $L = 9log_2M_1 + 54$	$\frac{BN}{M} + L$ $L = 35 + log_2M$	$\frac{SN}{M} + L$ $L = 35 + log_2M$

one memory unit is needed for these three architectures to store the outputs (offsprings or indexes). However, the Rejection blocks produce output streams at random clock cycles. Therefore, a single memory unit is necessary for each block. In total, M memory units are needed as output memory for parallel Rejection architecture but the total output memory size is the same as that of the other schemes.

Table II also shows the total number of clock cycles needed to complete the resampling operation for all architectures, given the parallel degree M_1 and M and the number of particles. The clock cycles needed by RSR and ISR consist of the cycles needed for SUM or CUMSUM (shown in Table I) and the cycles needed for offspring evaluation which is $N/M + L_{Res}$ where L_{Res} represents the latency of offspring evaluation. L_{Res} for RSR is larger when using $M > 1$ because of the additional latency needed for the computation of u_i. The execution times of Metropolis and Rejection resampling are $BN/M + L$ and $SN/M + L$ cycles respectively for parallel implementations where L is the latency, B in Metropolis is the iteration steps configured by the user and S in Rejection represents the average number of iteration steps performed before acceptance of each weight. Note that the S for Rejection is not fixed and depends on the weight distribution, and we can estimate S with reasonable accuracy after performing the Rejection numerous times. Both B for Metropolis and S for Rejection resampling can be very large as the variance in weights increases, and this makes these two architectures less efficient for large amounts of particles as will be shown shortly.

Table III gives the resource utilization (slices, LUTs, etc.) of a single resampling block for each architecture in the target device. For M parallel resampling blocks which are needed for the respective parallel architectures, the required resources can be estimated by multiplying by M (although in the real implementation the total utilization varies slightly due to synthesis and place and route optimizations). Table III also gives the clock frequency achieved for the non-parallel (i.e. $M = 1$) and parallel ($M = 32$) implementations of each architecture.

TABLE III. RESOURCES (OF ONE RESAMPLING BLOCK) AND CLOCK FREQUENCY ACHIEVED FOR EACH ARCHITECTURE ON A VIRTEX-6 LX240T FPGA

		RSR	ISR	Metropolis	Rejection
Resources of one block	Slices	414	351	430	462
	Slices registers	1545	1490	1696	1647
	LUTs	1063	949	1092	1089
CLK(MHz)	$M = 1$	227	336	289	327
	$M = 32$	193	185	130	156

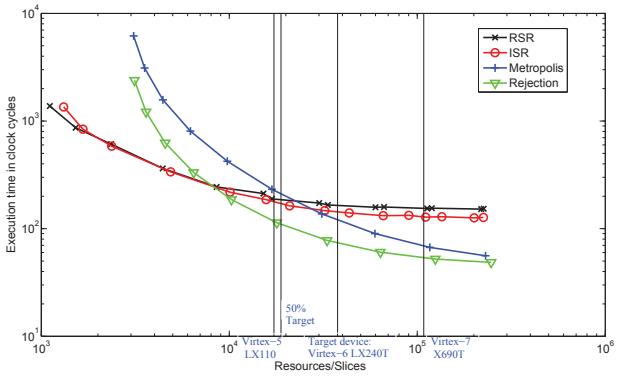

Fig. 9. The execution time in clock cycles of the resampling algorithms against the resources on slices on the target device Virtex-6 LX240T at $N = 2^{10}$ and at $y = 1$.

Figure 9 shows how the execution time (clock cycles) changes with the amount of utilized resources (slices) for each architecture at $N = 2^{10}$ and $y = 1$. The total resources of three representative Xilinx FPGAs are also marked in the figure (vertical lines) to provide a reference. The maximum M_1 and M which can be achieved in different devices are easy to obtain according to this figure. For example, with 50% utilization of slices on the target device, we can achieve (M_1,M)=(32,32), (16,32), (-,32), (-,32) for the four architectures respectively, and with full utilization of the device we can achieve (64,64), (32,64), (-,64), (-,64) respectively. The figure shows that RSR and ISR are faster when we utilize fewer resources but become slower than Metropolis and Rejection as more resources become available and the amount of parallelism increases. This is not surprising given the formulas of Table II as the latency of the SUM and CUMSUM datapaths in RSR and ISR architectures consumes a big part of the time.

To reserve resources for the other stages of particle filter, 50% utilization of the FPGA logic resources is assumed. Given this assumption, Figure 10 shows how the execution times of the four architectures change with N for two cases of variance ($y = 1$ and $y = 3$). It is worth noting that the execution time of RSR and ISR does not change with y when N is fixed. In contrast, the execution time of Metropolis and Rejection changes dramatically because the number of iterations increases considerably when y increases.

Figure 11 shows the speedup of FPGA resamplers versus the respective implementations on an NVIDIA K20 GPU device presented in [15] for the same two cases of y. The speedups for Metropolis resampling and Rejection resampling are in the order of 1.7x-25x and 2.3x-49x respectively for large

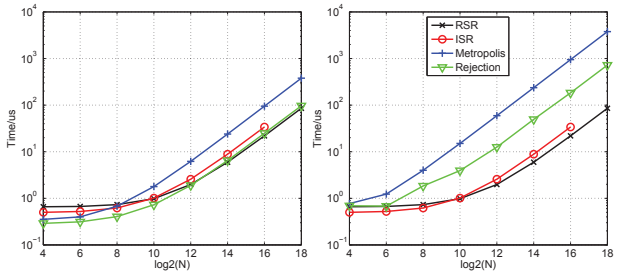

Fig. 10. The execution times against the particle numbers N at $y = 1$ **(left)** and $y = 3$ **(right)** in FPGA with 50% utilization of logic on the target device.

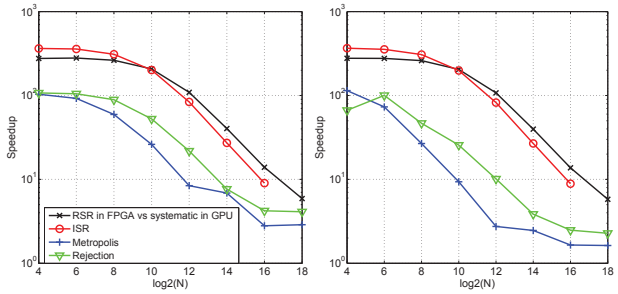

Fig. 11. The Speedups of FPGA implementations on the target device with 50% utilization of logic over the GPU implementations in [15] for the respective resampling algorithms at $y = 1$ **(left)** and $y = 3$ **(right)**.

particle numbers ($N \geq 2^{10}$), while the minimum speedups of RSR and ISR are 5.8x and 8.9x respectively. Another observation is that GPU can provide comparable performance to the FPGA for large numbers of particles, but not for small numbers of particles. This happens because the GPU cannot achieve full thread utilization unless we use a massive amount of particles (see [15]), while the FPGA is able to utilize a high percentage of its resources even for small problems. Therefore FPGAs can give high speedups for $N \leq 2^{10}$ as shown in Figure 11. As a reference, the CPU time of each algorithm can be found in [15].

VI. Discussion and Conclusion

In summary, the execution time of Metropolis and Rejection resampling largely depends on the variance in weights, while that of RSR and ISR is fixed. For FPGA implementations, at small variance in weights, Metropolis and Rejection resampling are preferred to RSR and ISR, and Rejection outperforms Metropolis at large N; at large variance in weights, Metropolis and Rejection resampling should only be considered at small N such as $N \leq 2^{10}$, and RSR or ISR should be preferred at large N. The advantage of ISR compared to RSR is no dependent operations within the offspring evaluation. The proposed ISR should be preferred at small or medium particle numbers (e.g. $N < 2^{12}$) while RSR outperforms ISR at large N (e.g. $N > 2^{14}$). When compared to GPU implementations, all four parallel implementations in FPGAs provide significant speedups. Note that both Metropolis resampling and Rejection resampling cause warp divergence in GPU, so FPGAs are more suitable for these two algorithms.

It should be noted that the use of on-chip FPGA memory is assumed for all the implementations. However it may not be feasible to store all the weights in on-chip memory at very large numbers. For example, the LX240T FPGA has enough on-chip memory to fit at most 2^{16} weights for ISR and 2^{18} weights for the other three architectures, assuming we use single precision arithmetic. With N around one million (2^{20}), off-chip memory needs to be used and the time to transfer weights from/to this memory can limit the FPGA performance if the memory bandwidth is not enough to constantly feed the processing elements. However, employing one million particles is a rare case in contemporary applications of the particle filter, so this work can adapt to most current applications. Future work will focus on applying these designs to particle filter algorithms and related applications.

References

[1] A. Doucet, N. De Freitas, and N. Gordon, "An introduction to sequential monte carlo methods," in *Sequential Monte Carlo methods in practice*. Springer, 2001, pp. 3–14.

[2] J. S. Liu, *Monte Carlo strategies in scientific computing*. Springer, 2001.

[3] T. C. Chau, X. Niu, A. Eele, W. Luk, P. Y. Cheung, and J. Maciejowski, "Heterogeneous reconfigurable system for adaptive particle filters in real-time applications," in *Proc. ARC*, 2013, pp. 1–12.

[4] T. C. Chau, M. Kurek, J. S. Targett, J. Humphrey, G. Skouroupathis, A. Eele *et al.*, "SMCGen: Generating reconfigurable design for sequential monte carlo applications," in *Proc. FCCM*, 2014, pp. 141–148.

[5] G. Hendeby, R. Karlsson, and F. Gustafsson, "Particle filtering: the need for speed," *EURASIP Journal on Advances in Signal Processing*, vol. 2010, no. 22, 2010.

[6] R. Douc and O. Cappe, "Comparison of resampling schemes for particle filtering," in *Proc. ISPA*, 2005, pp. 64–69.

[7] G. Kitagawa, "Monte Carlo filter and smoother for non-gaussian nonlinear state space models," *Journal of computational and graphical statistics*, vol. 5, no. 1, pp. 1–25, 1996.

[8] M. Bolić, P. M. Djurić, and S. Hong, "Resampling algorithms and architectures for distributed particle filters," *Signal Processing, IEEE Transactions on*, vol. 53, no. 7, pp. 2442–2450, 2005.

[9] ——, "Resampling algorithms for particle filters: A computational complexity perspective," *EURASIP Journal on Advances in Signal Processing*, vol. 2004, no. 15, pp. 2267–2277, 2004.

[10] M. Shabany and P. G. Gulak, "An efficient architecture for distributed resampling for high-speed particle filtering," in *Proc. ISCAS*, 2006, pp. 3422–3425.

[11] S. Hong, J. Jiang, and L. Wang, "Improved residual resampling algorithm and hardware implementation for particle filters," in *Proc. WCSP*, 2012, pp. 1–5.

[12] S.-H. Hong, Z.-G. Shi, J.-M. Chen, and K.-S. Chen, "A low-power memory-efficient resampling architecture for particle filters," *Circuits, Systems and Signal Processing*, vol. 29, no. 1, pp. 155–167, 2010.

[13] K. Hwang and W. Sung, "Load balanced resampling for real-time particle filtering on graphics processing units," *Signal Processing, IEEE Transactions on*, vol. 61, no. 2, pp. 411–419, 2013.

[14] P. Gong, Y. Basciftci, and F. Ozguner, "A parallel resampling algorithm for particle filtering on shared-memory architectures," in *Proc. IPDPSW*, May 2012, pp. 1477–1483.

[15] L. M. Murray, A. Lee, and P. E. Jacob, "Parallel resampling in the particle filter," in review. [Online]. Available: http://arxiv.org/abs/1301.4019

[16] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[17] J. D. Hol, T. B. Schon, and F. Gustafsson, "On resampling algorithms for particle filters," in *Nonlinear Statistical Signal Processing Workshop*, 2006, pp. 79–82.

[18] A. Athalye, M. Bolić, S. Hong, and P. M. Djurić, "Generic hardware architectures for sampling and resampling in particle filters," *EURASIP Journal on Advances in Signal Processing*, vol. 2005, no. 17, pp. 2888–2902, 2005.

[19] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," *GPU gems*, vol. 3, no. 39, pp. 851–876, 2007.

[20] J. Butler and T. Sasao, "Hardware index to permutation converter," in *Proc. IPDPSW*, May 2012, pp. 431–436.

[21] D. B. Thomas and W. Luk, "FPGA-optimised uniform random number generators using luts and shift registers," in *Proc. FPL*, 2010, pp. 77–82.