# Parallel Tempering MCMC Acceleration Using Reconfigurable Hardware

Grigorios Mingas and Christos-Savvas Bouganis

Department of Electrical & Electronic Engineering,
Imperial College London, Exhibition Road, London SW7 2BT, United Kingdom
{g.mingas10,christos-savvas.bouganis}@imperial.ac.uk

**Abstract.** Markov Chain Monte Carlo (MCMC) is a family of algorithms which is used to draw samples from arbitrary probability distributions in order to estimate - otherwise intractable - integrals. When the distribution is complex, simple MCMC becomes inefficient and advanced variations are employed. This paper proposes a novel FPGA architecture to accelerate Parallel Tempering, a computationally expensive, popular MCMC method, which is designed to sample from multimodal distributions. The proposed architecture can be used to sample from any distribution. Moreover, the work demonstrates that MCMC is robust to reductions in the arithmetic precision used to evaluate the sampling distribution and this robustness is exploited to improve the FPGA's performance. A 1072x speedup compared to software and a 3.84x speedup compared to a GPGPU implementation are achieved when performing Bayesian inference for a mixture model without any compromise on the quality of results, opening the way for the handling of previously intractable problems.

## 1 Introduction

Monte Carlo methods are a wide class of stochastic algorithms that rely on repeated generation of random numbers to solve problems like integration and optimization. By generating a large number of samples from the distribution of a variable of interest, Monte Carlo methods can draw conclusions about this variable and the system under study, without resorting to analytical or other numerical methods which are inefficient for many real applications. Monte Carlo is used in numerous fields ([13] provides an overview) but its core task in most cases is to solve the following problem:

**Problem.** Estimate the expectation of function $f(\mathbf{x})$ under $p(\mathbf{x})$, i.e. compute the following integral:

$$E_p[f(\mathbf{x})] = \int f(\mathbf{x})p(\mathbf{x})\mathbf{dx} \tag{1}$$

where $\mathbf{x}$ is the random variable of interest and $p(\mathbf{x})$ is the probability distribution of $\mathbf{x}$. In Bayesian inference problems (e.g. machine learning, computational biology), $p(\mathbf{x})$ is usually the posterior distribution of the parameters of the model given the data. The function $f(\mathbf{x})$ is the function of interest. For instance, if a moment or tail probability of $p(\mathbf{x})$ is requested, $f(\mathbf{x})$ changes accordingly.

O.C.S. Choy et al. (Eds.): ARC 2012, LNCS 7199, pp. 227–238, 2012.

Monte Carlo methods draw independent samples from $p(\mathbf{x})$ (also known as the target distribution) and approximate the integral using the following sum:

$$\tilde{E_p}[f(\mathbf{x})] = \frac{1}{N} \sum_{i=1}^{N} f(\mathbf{x_i}) \tag{2}$$

where $\mathbf{x_i}$, $i \in \{1, ..., N\}$ are samples taken from $p(\mathbf{x})$.

Sampling independently from $p(\mathbf{x})$ is straightforward only when $p(\mathbf{x})$ has a simple form (e.g. Gaussian, Beta). For non-standard forms of $p(\mathbf{x})$, different approaches like Markov Chain Monte Carlo (MCMC) have to be used. MCMC draws dependent samples from $p(\mathbf{x})$ using a Markov sample chain [13]. The dependent samples can still be used to compute (2). This strategy can theoritically sample from any distribution. The transition kernel of the chain (which defines how each new sample is generated from the previous one) is crucial for the efficiency of the method. It must ensure rapid convergence to $p(\mathbf{x})$ and good mixing (fast movement around the support of $p(\mathbf{x})$). In cases of multimodal target distributions, simple kernels like the Metropolis kernel fail to do that, as they tend to get "stuck" in one of the modes [13].

Advanced MCMC methods have been proposed to sample from these distributions [9]. Here, the focus is on Parallel Tempering (PT) [8], a popular, representative population-based method which is used in Bayesian inference problems as well as in physics and polymer simulations with multimodal targets ([6] is an extensive survey of application areas). PT employs multiple Markov chains to enhance mixing and this makes it computationally intensive. A motivating example is [7], where runtimes of up to 14 days for calibrating a rainfall-runoff model are reported.

In this work, a novel hardware architecture which exploits the characteristics of modern FPGAs to accelerate PT is presented. The architecture is problem-independent; any target distribution can be sampled as long as the respective probability evaluation block is plugged in. Results show that a speedup of 460x compared to software is achieved when performing inference on mixture models. The way this result scales with the size of the problem and the available resources is also investigated. Moreover, the impact of the employed arithmetic precision to MCMC's quality of sampling is addressed for the first time, to the authors' knowledge. It is shown that large reductions in the employed precision of the most area-expensive part of the architecture (probability evaluation) are possible without compromising the accuracy of the estimates, as the error due to precision is lower than the variance of the approximation. This translates to a further speedup of 2.33x compared to the single-precision floating point version of the architecture, making the system 1072x faster than a CPU and up to 3.84x faster than a GPGPU implementation.

The rest of the paper is organized as follows: Section 2 gives a short review of recent research on MCMC acceleration. Section 3 presents PT. Section 4 proposes the hardware architecture and Section 5 describes the class of models used for evaluation. Section 6 presents implementation results and a comparison with existing CPU and GPGPU implementations. Section 7 concludes the paper.

## 2 Related Work

Lately, several studies on accelerating MCMC using hardware have appeared in the literature. The main problem with MCMC acceleration is its inherently sequential nature; generation of the next sample of the chain requires the previous one to be available. Research efforts focus on: 1) Computation of many chains with the same target distribution in parallel. 2) Acceleration of MCMC methods with natural parallelism. 3) Use of speculative methods to avoid stalls. 4) Parallelization of the operations inside every step of MCMC. 5) Separation of the problem into independent sub-problems and execution of separate MCMCs. Only the first three approaches are problem-independent.

In [2], a multi-FPGA architecture for learning Bayesian networks is described which massively parallelizes intra-chain calculations. Also, a small number of parallel chains are used to enhance mixing but no details are given on how much this strategy helps. [14] use FPGAs to accelerate inference in Markov Random Fields by splitting the problem into sub-problems and running many samplers in parallel. These works are not concerned with how to efficiently map MCMC methods on FPGAs, but rather on how to achieve problem-specific acceleration.

In [11], GPGPU implementations of PT and Sequential Monte Carlo (SMC) are presented and the achieved acceleration is encouraging. Section 6 compares their results to this work. [3] use a speculative strategy to accelerate MCMC on a multicore but the gains are limited. [12] demonstrate why inter-chain communications in PT can pose a significant overhead to a Tungsten cluster and propose optimizations to tackle the problem. Here, it is shown that this is not an issue when the proposed FPGA architecture is used.

In contrast to previous MCMC-related research on FPGAs, this work belongs to the second category described above; it is a study on the suitability of FPGAs to accelerate a class of MCMC methods (population-based MCMC) with natural parallelism. It proposes architectural choices and optimizations which simultaneously exploit the nature of these methods and the characteristics of FPGAs (deep pipelining, fast inter-circuit communication, custom precision). Other population-based methods [9,10] can also take advantage of the proposed architecture.

The idea of using custom arithmetic precision when implementing Monte Carlo methods in FPGAs has been investigated in the past [15,17] but this is the first work that examines MCMC in particular. Here, it is proposed that the probability evaluation's targeted precision should be treated in a different way than the precision of the rest of the MCMC operations.

## 3 Parallel Tempering

Parallel Tempering is a population-based method, i.e. it uses a population of Markov chains to improve mixing. These methods are the de-facto approach for sampling from multimodal targets. Each chain $i$ samples from a different distribution $p_i(\mathbf{x})$, $i \in \{1, ..., M\}$. Distribution $p_1(\mathbf{x})$ is equal to the target $p(\mathbf{x})$ and the remaining distributions are smoothened versions of $p(\mathbf{x})$ (i.e. closer to the

uniform distribution). Smoothing is defined by a parameter (the temperature) and distributions become smoother as $i$ increases. If $T_i$ is the temperature of distribution $p_i(\mathbf{x})$ (with $1 = T_1 < T_2 < ... < T_M$), then:

$$p_i(\mathbf{x}) = p(\mathbf{x})^{1/T_i}, i \in \{1, ..., M\} \tag{3}$$

The target distribution is the "coldest" distribution ($T_1 = 1$) and distributions get "hotter" for $i > 1$.

At time $j$, the state of PT comprises the samples $\{\mathbf{x}_1^{(j)}, \mathbf{x}_2^{(j)}, ..., \mathbf{x}_M^{(j)}\}$. PT updates all these samples independently (using separate transition kernels according to (3)). The hot chains move quickly in the space because their distributions are closer to uniform. Periodically, PT proposes sample exchanges between chains. These exchanges push samples from the hot chains to the colder ones and eventually to the coldest chain. The samples help the coldest chain escape from isolated modes, thus enhancing mixing. To compute summary statistics, only samples from the coldest chain are kept. Two kinds of operations are performed:

**Update Operation.** It generates a new sample given the previous one using a kernel. Here, a Metropolis kernel is employed: At time $j$ and for chain $i$, a candidate sample $\mathbf{y}_i$ is drawn from a normal distribution centred around the previous sample $\mathbf{x}_i^{(j)}$. The candidate sample is accepted as the next sample ($\mathbf{x}_i^{(j+1)} = \mathbf{y}_i$) with probability $a(\mathbf{x}_i^{(j)}, \mathbf{y}_i)$:

$$a(\mathbf{x}_i^{(j)}, \mathbf{y}_i) = min\Big(1, \frac{p_i(\mathbf{y}_i)}{p_i(\mathbf{x}_i^{(j)})}\Big) \tag{4}$$

If the candidate is rejected, the previous sample $\mathbf{x}_i^{(j)}$ becomes the next sample ($\mathbf{x}_i^{(j+1)} = \mathbf{x}_i^{(j)}$). A global update comprises the update of all $M$ chains.

**Exchange Operation.** It attempts to exchange samples between two chains. An exchange between chains $q$ and $r$ is accepted with probability $e(\mathbf{x}_q, \mathbf{x}_r)$ (the time step index $j$ is omitted for clarity):

$$e(\mathbf{x}_q, \mathbf{x}_r) = min\Big(1, \frac{p_q(\mathbf{x}_r)p_r(\mathbf{x}_q)}{p_q(\mathbf{x}_q)p_r(\mathbf{x}_r)}\Big) \tag{5}$$

Many strategies have been proposed to exchange samples. Here, it was decided that performing exchanges only between neighbouring chains is the best way to maximize throughput (more in Section 4). A global exchange comprises exchanges between chains $(1, 2), (3, 4), ..., (M-1, M)$ or chains $(2, 3), (4, 5), ..., (M-2, M-1), (M, 1)$ (alternatively) and happens after every global update.

Figure 1 illustrates a simple case of PT. Normally, dozens or hundreds of chains are used, which can lead to long runtimes for difficult-to-compute targets.

## 4 System Architecture

### 4.1 Description

The tempered chains can run independently and only communicate during exchanges. The most computationally demanding task within each chain $i$ is the

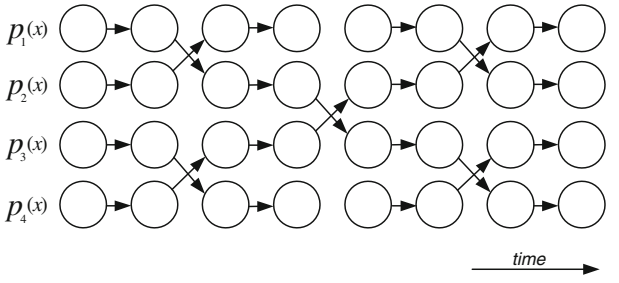


**Fig. 1.** Parallel Tempering updates and exchanges (four tempered chains)

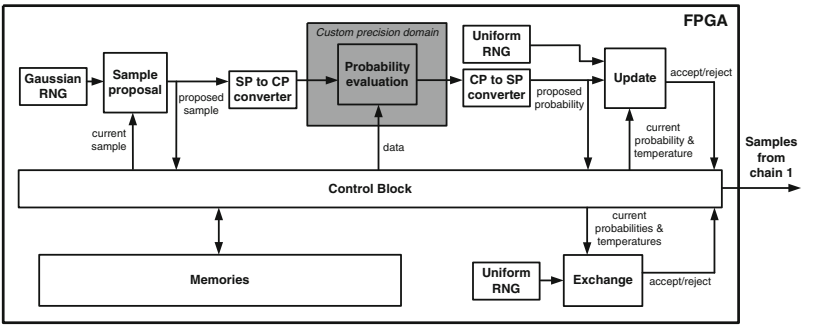


**Fig. 2.** Overview of the architecture

evaluation of the probability $p_i(\mathbf{y}_i)$ of the proposed sample $\mathbf{y}_i$. Due to the form of (3), this computation is the same for every chain (apart from the temperature and the proposed sample). To exploit this, the proposed architecture treats PT as a streaming application: One or more pipelines that implement the probability density are created and then fed with the independent chains. The temperature scaling is applied at a later stage. By keeping the pipelines fully utilized (using the multiple chains), the sampling throughput of PT is maximized.

Depending on the complexity of $p(\mathbf{x})$, there are two possible scenarios for the number of pipelines: 1) There are enough resources in the FPGA to fully parallelize the probability evaluation so that the pipeline can generate one probability per clock cycle. In this case, more pipelines can be instantiated to increase throughput, with the limit being the total resources of the FPGA. 2) There are not enough resources to fully parallelize the probability evaluation, therefore it can generate one probability per $C > 1$ clock cycles. Only one pipeline is used and the effort must be in optimizing the probability evaluation as much as possible to increase throughput. As this is the most realistic scenario given the complexity of real applications (e.g. large number of data in likelihoods), the remaining of this section focuses on this case.

Figure 2 illustrates the architecture of the system. The sample proposal block reads the current sample of every chain and generates a proposed sample, using numbers from a gaussian Random Number Generator (RNG) [16]. The probability evaluation block reads the proposed samples. Every $C$ cycles, one

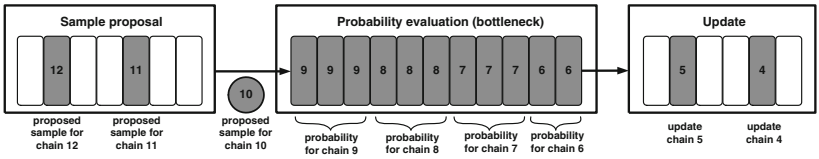


**Fig. 3.** Streaming through the sample proposal, probability evaluation and update pipelines at a specific time instance. Occupied stages are painted grey, unoccupied stages are painted white.

proposed probability is generated, the update block receives it (together with the current probability, the temperature and a uniform random number) and accepts or rejects the proposed sample using (4). This happens until all chains have been updated and then the sequence is repeated.

Similar to the probability evaluation block, all of the other modules are pipelined but they are under-utilized unless the probability evaluation block can generate one sample (or more) per cycle. Figure 3 demonstrates this; the stages of the pipeline occupied by each chain are visible. In this example, one probability value per $C = 3$ cycles is generated.

The exchange step is executed for a pair of chains in parallel to the above update operations. Every $2C$ cycles, the probabilities of the current samples of the chains, the temperatures and a uniform random number are used to implement (5). Exchanges are proposed between neighbouring chains and not randomly. This strategy was chosen to avoid stalls in the probability evaluation pipeline (chains 2, 3 are exchanged while chains 4, 5, etc are updated). Unlike CPU and GPGPU implementations [12,11], no communication between separate processors is necessary for the exchanges, due to the form of the architecture.

The rest of the design includes two uniform random number generators, precision converters to and from custom precision (see 4.2) and small Block RAMs for the storage of temperatures, samples and probabilities. A control block keeps track of which chain is inside which block and synchronizes memory operations.

The architecture is designed so that sampling from any $p(\mathbf{x})$ is possible. Only the appropriate probability evaluation block needs to be plugged in. The implementation of this block can be done with the help of libraries of floating point operators (e.g. FloPoCo [5]). The rest of the architecture is completely generic and does not change. In order to reduce the area consumption of the architecture and the dynamic range of the variables, probabilities are converted to logarithmic scale. This means that division and power operators needed to implement (4) and (5) are replaced by subtraction and multiplication operators which consume less area. More resources could be saved by "slowing down" the under-utilized modules when $C > 1$ but the gains are not significant and reconfiguration of the generic part (based on the value of $C$) would be necessary.

The throughput of the system (in samples per second from the coldest chain) is given by the following equation, assuming there are enough chains to fill the pipeline:

$$Throughput = \frac{clockrate}{CM} \tag{6}$$

where *clockrate* is in Hz, $M$ is the number of chains and $C$ is the number of clock cycles between two consecutive probabilities generated by the probability evaluation block. Equation (6) is valid for any targeted problem and shows that the performance of the system can be maximized if the probability evaluation block takes full advantage of the FPGA's resources in order to minimize $C$. If more than one pipelines are used (first scenario above), $C = 1$ and the throughput in (6) is multiplied by the number of pipelines.

### 4.2 Using Custom Arithmetic Precision

Floating point arithmetic is used throughout the design but there are two precision domains. The first comprises the probability evaluation block. This part can be implemented in any precision to save resources (or increase throughput by parallelizing further) with the price of sampling from an altered distribution. The motivation to experiment with this approach was that in stochastic algorithms, it is possible that changes in precision are "hidden" behind randomness and affect results less than they would in a deterministic algorithm. Previous research has shown that MCMC's results can be robust when substituting the acceptance probability with an unbiased approximation [1]. No work has addressed the effect of limited precision to MCMC, though. Here, it was observed that even a large reduction of the first domain's precision will not affect the sampling quality (details in Section 6). The second precision domain includes the generic part of the system. This part always operates in single-precision floating point (double-precision can also be used). It is important to use high precision in order to preserve the convergence and mixing properties of MCMC and PT (which are mathematically proven for infinite precision). In other words, reducing the precision in the second domain means that sampling from any targeted distribution is not guaranteed and unexpected behaviour might occur, while reducing the precision only in the first domain guarantees correct sampling (from an altered but known distribution).

The use of two precision domains is also motivated by the fact that the generic part of the algorithm consumes a significantly lower amount of FPGA resources than the probability evaluation block for any non-trivial target distribution. Consequently, minimizing the area consumption of the probability evaluation block is the primary concern.

## 5 Bayesian Inference for Mixture Models

### 5.1 Description

Bayesian inference is a method of statistical inference used to draw conclusions about unobserved or unobservable quantities, given observed data [13]. A common setting is the following: Some data are given and a model with unknown parameters is believed to explain the data. The goal is to obtain the posterior distribution of the parameters given the data and calculate expectations under

this posterior (see (1)). The initial belief on the values of the parameters is expressed by a prior distribution. MCMC is the most popular method used to solve this problem.

Mixture models are a powerful family of models used in numerous fields [4]. Multimodal posterior distributions often appear when performing Bayesian inference for these models. Hence, they are a representative case of problems that PT is used to solve.

To test the architecture's performance, a finite gaussian mixture model taken from [9] and [11] is used: A set of independent observations (data) $\mathbf{d} = d_{1:Q}$, where $d_q \in \Re$ for $q \in \{1, ..., Q\}$, is given. Each observation is distributed according to:

$$p(d_q|\mu_{1:k}, \sigma_{1:k}, w_{1:k-1}) = \sum_{i=1}^{k} w_i f(d_q|\mu_i, \sigma_i) \tag{7}$$

Here, $f$ denotes the density of a univariate normal distribution, $k$ is the number of mixture components and $\mu_{1:k}$, $\sigma_{1:k}$ and $w_{1:k-1}$ are the parameters of the model (means, variances and weights of components respectively).

A particular configuration of the model is used to compare this work with the implementations in [11]: $k = 4$, $\sigma_i = \sigma = 0.55$ and $w_i = w = 1/k$ for $i \in \{1, ..., k\}$. The posterior distribution of $\mu = \mu_{1:k}$ needs to be sampled. The prior distribution on $\mu$ is 4-dimensional uniform. The data $\mathbf{d} = d_{1:m}$ (with $m = 100$) are simulated using $\mu = (-3, 0, 3, 6)$. The likelihood function is given by:

$$p(\mathbf{d}|\mu) = \prod_{j=1}^{100} p(d_j|\mu, \sigma_{1:k}, w_{1:k-1}) \tag{8}$$

If $p(\mu)$ is the uniform prior, the posterior distribution for $\mu$ is given by:

$$p(\mu|\mathbf{d}) = p(\mathbf{d}|\mu)p(\mu) \tag{9}$$

The sampling distribution is multimodal (it admits 24 modes). The PT tempering schedule used is $T_i = (\frac{M}{M+1-i})^2$, where $M$ is the number of chains.

### 5.2 Implementation of the Probability Evaluation Block

The target posterior (9) is implemented using pipelined floating point operators. The likelihood's 100 terms (each a 4-component mixture density) can be evaluated in parallel. Nevertheless, there are not enough resources in the targeted FPGA (see 6.1) to fully parallelize it and generate one probability per cycle (second scenario in 4.1). The throughput of the implementation is defined by the number of terms that can be evaluated in parallel (which affects $C$ in (6)). This can vary depending on the operators' implementation and targeted precision. Moreover, if the complexity of the model changes the throughput will change. This is investigated in Section 6.

The 100 data of the problem are stored inside the FPGA and are read by the probability evaluation block. Each term's probability is converted to logarithmic scale so that the product in the likelihood is replaced by a sum. Although a particular configuration of the model is implemented here, different configurations and mixture models can be targeted using the same implementation principles.

**Table 1.** CPU (Xeon E5420) throughput in samples/sec [11] and speedup of FPGA (XC6VLX240T) and two GPGPUs (8800GT & GTX280) [11] vs CPU for different numbers of chains $M$. The likelihood is not parallelized in the GPGPUs.

| Nbr. of chains ($M$) | 8 | 32 | 128 | 512 | 2048 | 8192 | 32768 | 131072 |
|---|---|---|---|---|---|---|---|---|
| CPU thr/put | 8224.8 | 2056.2 | 514.0 | 128.5 | 32.1 | 8.0 | 2.0 | 0.5 |
| LX240T speedup | 92 | 368 | 460 | 460 | 460 | 460 | 460 | 460 |
| 8800GT speedup | 1.1 | 4 | 17 | 60 | 168 | 230 | 268 | 279 |
| GTX280 speedup | 0.9 | 4 | 14 | 51 | 175 | 430 | 527 | 572 |

## 6 Results

### 6.1 Comparison to CPU and GPGPU Performance and Scalability

The architecture was implemented in VHDL and the target FPGA was a Virtex 6 XC6VLX240T, which has a high ratio of DSP-blocks/Slices (DSP blocks are needed for floating point operators). For a single-precision floating point datapath, the achieved frequency was 212 MHz (the critical path was inside the logarithmic operator). The design was tested for different values of tempered chains ($M$). The sampler visited all of the modes, which is the main goal.

The performance of the system was first compared to a CPU implementation of PT for the same target distribution [11]. The CPU code was written in C++ and ran on a Xeon E5420 (2.5 GHz). Table 1 shows the results.

A 460x speedup is achieved compared to software for $M > 40$. For $M \leq 40$ there are not enough chains to fill the pipeline and speedup drops (although it is still significant). This reveals the importance of keeping the pipeline fully occupied. This is easier with more computationally-intensive likelihoods. To illustrate this, Figure 4 shows how the speedup over software scales with the number of mixture components ($k$ in (7)) for different FPGAs of the Virtex-6 family. Here, $M = 128$ and maximum utilization of the available resources is considered. For smaller problems (e.g. $k = 1$) no extra speedup is provided by increasing the size of the FPGA because of stalls in the pipeline. It is also observed that the speedup is not only sustainable for larger problems, but it also increases (for the same FPGA device). This is due to better utilization of resources and resource sharing by the synthesis tool when more components are used (e.g. full utilization of LUTs). Each FPGA's speedup converges to a maximum value defined by its available resources as utilization becomes better.

Comparisons were also made with the GPGPU implementations of PT in [11] (again for the same inference problem). The GPU code was written in CUDA and ran on an Nvidia 8800GT and an Nvidia GTX280 using single-precision. Table 1 shows the GPGPU and FPGA speedups over the CPU. It must be noted that the probability evaluation is not parallelized in the GPGPU implementations (each thread is assigned one chain). All the speedup comes from running many chains in parallel which is an ideal setting for a GPGPU. Nevertheless, in practice, dozens or hundreds of chains are enough for good mixing [9]. Indeed, the authors in [11] note that no improvement in mixing was observed for $M > 128$.

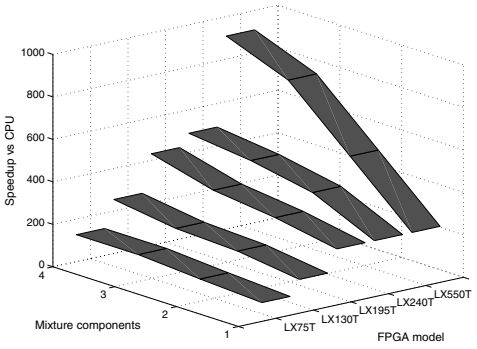


**Fig. 4.** Scaling of speedup over CPU with increasing problem size (number of mixture components) for different FPGAs of the Xilinx Virtex-6 family

A direct comparison with the GPGPUs' performance shows that the FPGA is faster than the 8800GT for all $M$ and faster than the GTX280 for $M \leq 8192$. The maximum GPGPU speedups (279x and 572x respectively) are achieved only in the unrealistic case of thousands of employed chains (when the GPGPUs are fully utilized). No results are given in [11] on what the acceleration would be if fewer chains were used and the likelihood was parallelized.

### 6.2 Performance Evaluation under Custom Precision

Implementing the probability evaluation in reduced precision can provide a further speedup.Nevertheless, samples are taken from an altered distribution. To investigate the impact of this perturbation to the accuracy of sampling, two problems were examined: 1) A fully known mixture of 20 bivariate Gaussians (common test case in MCMC literature [10]) for which it is easy to measure the effect of precision to the output estimates (comparing to the known true values). 2) The more realistic inference problem described above. Here, the true values of estimates are unknown (this is why inference is performed) but it is known that e.g. means should be close to 1.5 [9]. The complexity of this problem allows for a valid throughput comparison to be made between implementations that employ different precisions. The probability evaluations were implemented for the following floating point precision configurations: (8,23), (8,19), (8,15), (8,11), (8,7), (8,5), and (8,3), where the first number represents the exponent bits and the second number represents the mantissa bits. The exponent bits were kept the same as in single precision to ensure adequate range.

For the first problem, Figure 5 illustrates the value of the mean estimator (first dimension) as the number of samples increases, for different precisions. The true value is also visible. The estimator remains accurate (within some variance) until a break-point is reached (precision (8,5)).

For the second problem, the speedup over the single-precision version was measured for all limited precision implementations. Ten runs of $10^6$ samples were performed for each precision. The mean absolute error in the mean estimate over

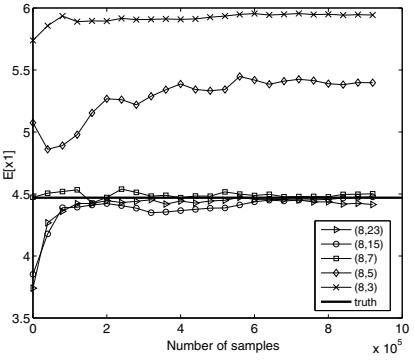


**Fig. 5.** 20-mode Gaussian target: Mean estimate (first dimension) for different precisions as the number of samples increases

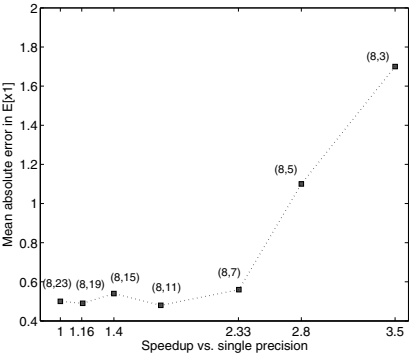


**Fig. 6.** Mixture inference target: Mean absolute error in the mean estimate (first dimension) vs speedup over the single precision implementation

the 10 runs was calculated (taking 1.5 as the true value [9]). Figure 6 plots this error in the first dimension against the achieved speedup for each precision.

It can be observed that an extra speedup of up to 2.33x is provided by reducing precision without losing anything in sampling quality. This makes the system 1072x faster than software. This speedup is 3.84x and 1.87x higher than the maximum speedup of the 8800GT and GTX280 GPGPUs respectively.

These results demonstrate that the probability evaluation inside MCMC is indeed robust to precision perturbations in the sampling distribution. The error due to limited precision is smaller than the error due to the variance in the estimates until a break-point is reached.

## 7 Conclusion

This work presents a generic FPGA architecture for PT, a computationally expensive MCMC method. The characteristics of FPGAs and the nature of PT are exploited to maximize performance. A significant speedup compared to CPU and GPGPU implementations is achieved when doing inference in mixture models. The design is generic and can be used to sample from any target distribution. The work also demonstrates that MCMC is robust to the reduction of the arithmetic precision used to evaluate the sampling density. This implies that the flexibility of FPGAs to use custom precision can translate into significant gains in MCMC throughput, allowing for the tackling of previously intractable problems.

Future work includes the acceleration of other computationally demanding MCMC and SMC methods. A tool which could automate the generation of probability evaluation blocks is also under consideration.

## References


1. Andrieu, C., Roberts, G.O.: The pseudo-marginal approach for efficient Monte Carlo computations. The Annals of Statistics 37(2), 697–725 (2009)
2. Asadi, N.B., Meng, T.H., Wong, W.H.: Reconfigurable computing for learning Bayesian networks. In: Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays, FPGA 2008, pp. 203–211 (2008)
3. Byrd, J., Jarvis, S., Bhalerao, A.: Reducing the run-time of MCMC programs by multithreading on SMP architectures. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–8 (April 2008)
4. Chatzis, S.: A method for training finite mixture models under a fuzzy clustering principle. Fuzzy Sets and Systems 161(23), 3000–3013 (2010)
5. de Dinechin, F., Pasca, B.: Designing Custom Arithmetic Data Paths with FloPoCo. IEEE Design and Test of Computers 28, 18–27 (2011)
6. Earl, D.J., Deem, M.W.: Parallel tempering: Theory, applications, and new perspectives. Phys. Chem. Chem. Phys. 7, 3910–3916 (2005)
7. Fielding, M., Nott, D.J., Liong, S.Y.: Efficient MCMC Schemes for Computationally Expensive Posterior Distributions. Technometrics 53(1), 16–28 (2011)
8. Geyer, C.J.: Markov Chain Monte Carlo Maximum Likelihood. In: Proceedings of the 23rd Symposium on the Interface, Computing Science and Statistics, pp. 156–163 (1991)
9. Jasra, A., Stephens, D.A., Holmes, C.C.: On population-based simulation for static inference. Statistics and Computing, 263–279 (2007)
10. Kou, S.C., Zhou, Q., Wong, W.H.: Equi-energy sampler with applications in statistical inference and statistical mechanics. Ann. Statist. 34(4), 1581–1652 (2006)
11. Lee, A., Yau, C., Giles, M.B., Doucet, A., Holmes, C.C.: On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods. Journal of Computational and Graphical Statistics 19(4), 769–789 (2010)
12. Li, Y., Mascagni, M., Gorin, A.: A decentralized parallel implementation for parallel tempering algorithm. Parallel Comput. 35, 269–283 (2009)
13. Liu, J.S.: Monte Carlo strategies in scientific computing. Springer, Heidelberg (2001)
14. Mansinghka, V.K., Jonas, E.M., Tenenbaum, J.B.: Stochastic Digital Circuits for Probabilistic Inference. Technical Report MIT-CSAIL-TR-2008-069, Massachussets Institute of Technology (2008)
15. Saiprasert, C., Bouganis, C.-S., Constantinides, G.A.: Design of a Financial Application Driven Multivariate Gaussian Random Number Generator for an FPGA. In: Sirisuk, P., Morgan, F., El-Ghazawi, T., Amano, H. (eds.) ARC 2010. LNCS, vol. 5992, pp. 182–193. Springer, Heidelberg (2010)
16. Thomas, D.B., Luk, W., Leong, P.H., Villasenor, J.D.: Gaussian random number generators. ACM Comput. Surv. 39 (November 2007)
17. Tian, X., Bouganis, C.S.: A Run-Time Adaptive FPGA Architecture for Monte Carlo Simulations. In: 2011 International Conference on Field Programmable Logic and Applications (FPL), pp. 116–122 (September 2011)