2013 21st Annual International IEEE Symposium on Field-Programmable Custom Computing Machines

On Optimizing the Arithmetic Precision of MCMC Algorithms

Grigorios Mingas, Farhan Rahman, Christos-Savvas Bouganis
Department of Electrical and Electronic Engineering
Imperial College London
London, UK
Email: {g.mingas10, farhan.rahman09, christos-savvas.bouganis}@imperial.ac.uk

***Abstract*—Markov Chain Monte Carlo (MCMC) is an ubiquitous stochastic method, used to draw random samples from arbitrary probability distributions, such as the ones encountered in Bayesian inference. MCMC often requires forbiddingly long runtimes to give a representative sample in problems with high dimensions and large-scale data. Field-Programmable Gate Arrays (FPGAs) have proven to be a suitable platform for MCMC acceleration due to their ability to support massive parallelism. This paper introduces an automated method, which minimizes the floating point precision of the most computationally intensive part of an FPGA-mapped MCMC sampler, while keeping the precision-related bias in the output within a user-specified tolerance. The method is based on an efficient bias estimator, proposed here, which is able to estimate the bias in the output with only few random samples. The optimization process involves FPGA pre-runs, which estimate the bias and choose the optimized precision. This precision is then used to reconfigure the FPGA for the final, long MCMC run, allowing for higher sampling throughputs. The process requires no user intervention. The method is tested on two Bayesian inference case studies: Mixture models and neural network regression. The achieved speedups over double-precision FPGA designs were 3.5x-5x (including the optimization overhead). Comparisons with a sequential CPU and a GPGPU showed speedups of 223x-446x and 16x-18x respectively.**

I. Introduction

MCMC is a family of Monte Carlo methods designed to draw samples from a given, arbitrary probability distribution. The samples are used to estimate quantities related to the probability distribution. This is a fundamental task in many statistical applications, e.g. in Bayesian inference where the posterior distribution of an unknown quantity, given known data, is sampled [1]. MCMC's universality and efficiency has allowed statisticians to solve extremely complicated Bayesian probabilistic models, previously considered intractable, practically revolutionizing Bayesian statistics. Today, MCMC is used in a range of statistical applications, including machine learning, population genetics, phylogenetics, ecology, computational physics and others [1]–[4].

Despite their success, MCMC samplers are often not as fast as modern demanding applications require. Runtimes can easily reach weeks or months [2]–[4], forcing practitioners to accept more variance in the results or use a simpler model. The high computational load is due to: 1) The massive size and dimensionality of data and the continuously increasing complexity of models in many of today's Bayesian applications. For example, topic models commonly use large text databases for inference [3], while gene-based classification problems have dimensions in the order of thousands [2]. This makes the evaluation of the probability distribution function (necessary in each MCMC step) is extremely expensive. 2) The computational overheads of sophisticated, parallel MCMC methods [1]. 3) The fact that multiple independent MCMC runs are performed in most settings to confirm convergence and approximate the variance of the estimates, thus multiplying runtimes [2].

FPGAs have proven to be a suitable platform for MCMC acceleration, due to their capability to implement massively parallel circuits. [4], [5]. An important advantage of these devices is their flexibility to operate in any custom arithmetic precision format. Instead of implementing operators in double floating-point, which is the default approach in MCMC applications, reduced precisions can be used, making operators cheaper and allowing for more parallelism. At the same time though, more error is introduced in calculations.

This paper proposes the use of custom floating point precision to compute the probability distribution function inside MCMC (the most resource-expensive part of the algorithm), in order to increase the FPGA sampler's throughput, and introduces an automated method to minimize this precision while probabilistically guaranteeing a user-specified error (bias) threshold. The main contributions of this work are:

1) The introduction of an efficient estimator of the bias that appears in the output estimate of MCMC as a result of implementing the probability distribution function in custom precision (Section III). The estimator is able to produce an accurate bias estimate using only few MCMC samples (compared to the ones needed for the output estimate).

2) A precision optimization method for FPGA-mapped MCMC samplers, which exploits the proposed estimator and the reconfigurable property of FPGAs to automatically choose a minimized precision from within a pre-defined set of precision configurations, while satisfying the user's bias tolerance requirements (Section IV). Due to the stochasticity of MCMC, the output estimate is always approximated within some standard deviation. Given a specific output estimate, a target standard deviation, a tolerable bias and a few other user parameters, the method can find the lowest

978-0-7695-4969-9/13 $26.00 © 2013 IEEE
DOI 10.1109/FCCM.2013.31

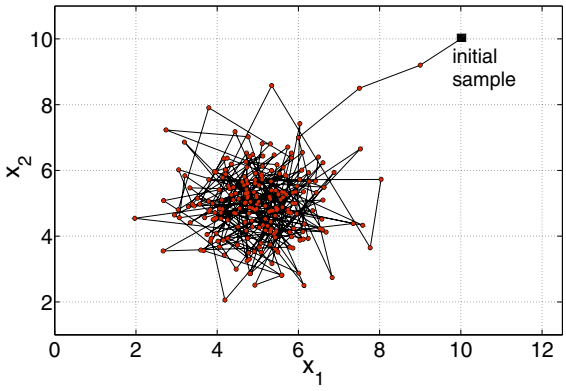

Figure 1. Metropolis MCMC drawing samples from a bivariate Gaussian with mean (5,5). Only few samples are needed for convergence.

precision whose bias is within tolerance (with a certain probability), trading off output accuracy for resource savings (and thus speedup). The process involves pre-runs on the FPGA to estimate the bias of each precision and choose the optimized one for the final, long FPGA run. The pre-runs add only a small overhead due to the efficiency of the bias estimator and the use of a termination criterion. The method is automated, can tailor the precision to user requirements and can be applied to any MCMC algorithm.

The proposed methodology is evaluated using two Bayesian inference problems (Section VI). Results show that the optimized-precision designs are 3.57x-5.02x faster than double-precision FPGA designs (including the optimization overhead), depending on the type of estimate and the user parameters. Performance gains over double-precision, sequential CPU implementations and an existing single-precision GPGPU implementation are 223x-446x and up to 18x respectively.

II. Background

A. Markov Chain Monte Carlo

The typical problem that MCMC aims to solve is the estimation of the integral:

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

where $\mathbf{x}$ is a random vector, $p(\mathbf{x})$ is the probability distribution of $\mathbf{x}$, $f(\mathbf{x})$ is a function of interest (e.g. mean, moment) and $E_p[f(\mathbf{x})]$ denotes the expectation of $f(\mathbf{x})$ when $\mathbf{x}$ is distributed according to $p(\mathbf{x})$. MCMC solves the problem stochastically. It draws samples $\mathbf{x}^{(i)}$, $i \in \{1, ..., N\}$ from $p(\mathbf{x})$ (i.e. the target distribution) and uses them to evaluate:

$$\tilde{I} = \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}^{(i)}) \approx I \quad (2)$$

which is an unbiased estimator of the integral (1). (2) is called the output estimate. The proposed method performs optimization for a specific output estimate, i.e. specific $p(\mathbf{x})$ and $f(\mathbf{x})$, given by the user.

MCMC works sequentially by constructing a Markov chain, with each state of the chain corresponding to a new random sample from $p(\mathbf{x})$. Figure 1 illustrates a simple MCMC algorithm sampling a 2-D Gaussian distribution. The transitions between successive states are visible. The algorithm consist of two parts: 1) The evaluation of the probability of each new sample according to $p(\mathbf{x})$, which is specific to the target distribution, 2) The generic operations, which mainly include proposing samples and accepting/rejecting them according to their probability value [1]. The probability evaluation takes up the bulk of the computation time, especially when complex models and large-scale data are employed. Moreover, advanced MCMC methods, like the ones that use multiple Markov chains, require many $p(\mathbf{x})$ evaluations for each sample [1]. The generic MCMC operations are much less computationally demanding and are outside the scope of this paper.

FPGA-mapped MCMC samplers contain a hardware block for the generic MCMC operations and one or more blocks which implement $p(\mathbf{x})$ (see [4], [6]). Because $p(\mathbf{x})$ is the bottleneck computation, sampling throughput increases when more $p(\mathbf{x})$ blocks are instantiated. This work focuses on how to minimize the cost of implementing these $p(\mathbf{x})$ blocks by automatically optimizing the employed precision.

B. Related work

FPGAs have been used to accelerate computationally expensive MCMC methods in many problems, e.g. phylogenetics [4] and Bayesian network inference [5], with speedups of up to two orders of magnitude over software. Nevertheless, these works focus on specific applications and/or MCMC methods (not the general MCMC case) and none of them has tackled the issue of precision optimization.

For Monte Carlo methods other than MCMC, [7] and [8] have proposed precision optimization techniques based on performing runs in different precisions. [7] use an auxiliary mixed-precision run to estimate and correct the bias in the output. The work presented here does not remove the bias. Based on the assumption that a known standard deviation in the output is targeted, it permits the introduction of bias if it is significantly smaller than this standard deviation (according to user specifications), allowing for more aggressive accuracy/performance trade-offs. Moreover, in [7], the bias is related to the implementation of $f(\mathbf{x})$ in (1) (and not of $p(\mathbf{x})$, as in MCMC) in reduced precision, leading to a different form of bias estimator. In [8], high- and low-precision runs are compared using the Kolmogorov-Smirnoff metric and the precision is adapted so that a threshold is not violated. Placing such a threshold is empirical and does not constrain the bias, in contrast to the method presented here.

The only previous work on optimizing MCMC precision is [6], where population-based MCMC methods are targeted. It is shown that reducing precision in parts of the system has no effect in sampling quality. Nevertheless, this can only be

applied MCMC methods that use multiple parallel chains. The method presented here is not limited by the internal structure of the employed MCMC algorithm.

III. Proposed bias estimator

This section proposes an efficient estimator of the bias in the MCMC output when the probability distribution is evaluated in custom precision. Section IV uses this estimator to develop an automated precision optimization method.

A. Infinite (double) precision output estimator

When $p(\mathbf{x})$ is evaluated in infinite precision, samples are distributed according to the true distribution and the output estimate (2) converges to the true value (1) for $N \to \infty$. In this paper, infinite precision is considered equivalent to double floating-point precision. This is the precision used in the majority of MCMC implementations and is generally considered adequate. For finite N, (2) differs from (1) by some Normal-distributed random number with mean zero and variance $\sigma^2_{\tilde{I}}$, called the variance of the output estimate:

$$\sigma^2_{\tilde{I}} = \frac{\sigma^2_f}{N} \tag{3}$$

where σ^2_f is the variance of the function of interest $f(\mathbf{x})$ under $p(\mathbf{x})$ (constant for fixed $f(\mathbf{x})$ and $p(\mathbf{x})$ but unknown).

B. Custom precision output estimator

When custom floating point precision is used to compute $p(\mathbf{x})$, MCMC samples are distributed according to $p_c(\mathbf{x})$, where $c = (exponent\ bits, mantissa\ bits)$ is the precision configuration used. All custom precisions in this paper are lower than double precision and use 8 exponent bits. The approximated integral is no longer (1), but:

$$I_c = E_{p_c}[f(x)] = \int f(\mathbf{x}) p_c(\mathbf{x}) \mathbf{dx} \tag{4}$$

The value of the custom-precision output estimator is:

$$\tilde{I}_c = \frac{1}{N} \sum_{i=1}^{N} f(\mathbf{x}^{(i)}) \approx I_c \tag{5}$$

where $\mathbf{x}^{(i)}$, $i \in \{1, ..., N\}$ are samples drawn from $p_c(\mathbf{x})$. The variance of this estimator $\sigma^2_{\tilde{I}_c}$ is found in the same way as the variance of the double-precision estimator (3).

C. Bias estimator

The value

$$b_c = I - I_c = \int f(\mathbf{x}) p(\mathbf{x}) \mathbf{dx} - \int f(\mathbf{x}) p_c(\mathbf{x}) \mathbf{dx} \tag{6}$$

is the bias in the estimate due to the use of custom precision. This quantity needs to be approximated to optimize precision. Figure 2 shows a double- and a custom-precision estimator for the same $f(\mathbf{x})$. The bias is visible near the end of the runs. Standard deviations are also visible. The bias is independent of the estimator's standard deviation and cannot be avoided when sampling with reduced precision.

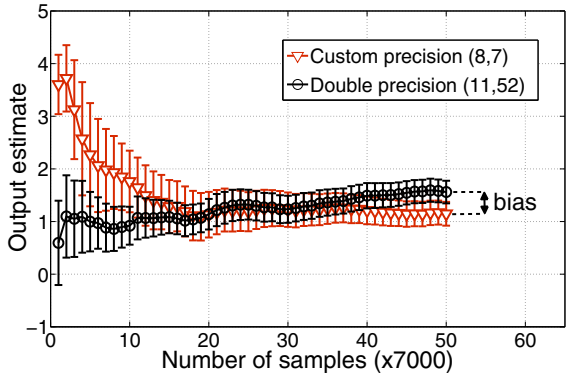

Figure 2. Double- and custom-precision ($c = (8, 7)$) sampling for the same output estimate. The error bars are equal to two times the standard deviation of the estimate ($\pm 2 * \sigma_{\tilde{I}}$).

Estimating b_c based on (6) would require two separate MCMC runs, targeting $p(\mathbf{x})$ and $p_c(\mathbf{x})$, to estimate the two integrals and then subtract them. The variance σ^2_{str} of this straightforward bias estimator would be equal to the sum of the two estimates' variances $\sigma^2_{str} = \sigma^2_{\tilde{I}} + \sigma^2_{\tilde{I}_c}$. Thus, this bias estimator has a larger variance than any of the two output estimators for an equal number of samples. Figure 3 shows the ratio $\frac{\sigma^2_{str}}{\sigma^2_{\tilde{I}_c}}$ for all precisions for one of the examples of Section V. σ^2_{str} is around two times larger than $\sigma^2_{\tilde{I}_c}$. Optimizing the precision with this method would require a long MCMC run for each precision to estimate the bias with sufficient accuracy, making the total runtime larger than the time for a double-precision run (the default choice).

Here, a more efficient bias estimator is proposed. It achieves much smaller variance, allowing for precision optimization using short pre-runs. The bias is rewritten as:

$$\begin{aligned} b_c &= \int f(\mathbf{x}) p(\mathbf{x}) \mathbf{dx} - \int f(\mathbf{x}) p_c(\mathbf{x}) \mathbf{dx} \\ &= \int f(\mathbf{x}) (p(\mathbf{x}) - p_c(\mathbf{x})) \mathbf{dx} = \int f(\mathbf{x}) \tfrac{p(\mathbf{x}) - p_c(\mathbf{x})}{p_c(\mathbf{x})} p_c(\mathbf{x}) \mathbf{dx} \\ &= \int f(\mathbf{x}) (\tfrac{p(\mathbf{x})}{p_c(\mathbf{x})} - 1) p_c(\mathbf{x}) \mathbf{dx} = \int f(\mathbf{x}) (w_c(\mathbf{x}) - 1) p_c(\mathbf{x}) \mathbf{dx} \\ &= \int f_{b_c}(\mathbf{x}) p_c(\mathbf{x}) \mathbf{dx} = E_{p_c}[f_{b_c}(\mathbf{x})] \end{aligned} \tag{7}$$

where $w_c(\mathbf{x}) = \frac{p(\mathbf{x})}{p_c(\mathbf{x})}$ and $f_{b_c}(\mathbf{x}) = f(\mathbf{x})(w_c(\mathbf{x}) - 1)$. The weight $w_c(\mathbf{x}^{(i)})$ is the ratio of the probability of sample $\mathbf{x}^{(i)}$ computed using double and custom precision. The proposed bias estimator approximates the integral in the last line of (7) by taking MCMC samples from $p_c(\mathbf{x})$ (similar to (5)):

$$\tilde{b}_c = \frac{1}{N} \sum_{i=1}^{N} f_{b_c}(\mathbf{x}^{(i)}) \approx b_c \tag{8}$$

only now it is also necessary to compute the probability of each sample in double precision ($p(\mathbf{x}^{(i)})$) to get the weights. Moreover, in Bayesian inference, the distributions $p(\mathbf{x})$ and $p_c(\mathbf{x})$ are usually known only up to a normalizing constant:

$$p(\mathbf{x}) = \frac{p^u(\mathbf{x})}{C_p} \tag{9}$$

$$p_c(\mathbf{x}) = \frac{p^u_c(\mathbf{x})}{C_{p_c}} \tag{10}$$

where only the unnormalized densities $p^u(\mathbf{x})$ and $p_c^u(\mathbf{x})$ can be evaluated. This means that the weights

$$w_c(\mathbf{x}) = \frac{p^u(\mathbf{x})}{p_c^u(\mathbf{x})} \frac{1}{\frac{C_p}{C_{pc}}} = w_c^u(\mathbf{x}) \frac{1}{\frac{C_p}{C_{pc}}} \tag{11}$$

cannot be evaluated exactly. The unknown ratio of constants $\frac{C_p}{C_{pc}}$ is estimated by the mean of the unnormalized weights $w_c^u(\mathbf{x})$ as shown in (12). Since $p(\mathbf{x})$ is a probability density:

$$\begin{aligned} &\int p(\mathbf{x})d\mathbf{x} = 1 \Leftrightarrow \int \frac{p(\mathbf{x})}{p_c(\mathbf{x})} p_c(\mathbf{x})d\mathbf{x} = 1 \Leftrightarrow \\ &\int \frac{\frac{p^u(\mathbf{x})}{C_p}}{\frac{p_c^u(\mathbf{x})}{C_{pc}}} p_c(\mathbf{x})d\mathbf{x} = 1 \Leftrightarrow E_{p_c}[w_c^u(\mathbf{x})] = \frac{C_p}{C_{pc}} \end{aligned} \tag{12}$$

The mean of the weights is computed at the end of the run and used to normalize the weights by dividing each weight with it. This makes the estimator (8) converge to the true bias. The variance of (8) is found in the same way as the previous variances, substituting f_{b_c} for f:

$$\sigma^2_{\tilde{b}_c} = \frac{\sigma^2_{f_{b_c}}}{N} \tag{13}$$

D. Variance: Proposed bias estimator vs. Straightforward bias estimator vs. Custom precision output estimator

Because the weights take values close to one (unless c is very low), the function $f_{b_c}(\mathbf{x}) = f(\mathbf{x})(w_c(\mathbf{x}) - 1)$ takes much smaller values than $f(\mathbf{x})$ and has a smaller variance when $\mathbf{x} \sim p_c(\mathbf{x})$. Figure 4 illustrates this by comparing the histograms of $f(\mathbf{x}) = \mathbf{x}$ (mean estimate) and $f_{b_c}(\mathbf{x}) = \mathbf{x}(w_c(\mathbf{x}) - 1)$ under the same distribution $p_c(\mathbf{x})$ ($c = (8, 13)$), taken from one of the case studies of Section V. Figure 3 shows the ratio $\frac{\sigma^2_{\tilde{b}_c}}{\sigma^2_{\tilde{I}_c}}$ for different precisions c, where $N = 10^6$ for both estimators. $\sigma^2_{\tilde{b}_c}$ is orders of magnitude smaller than $\sigma^2_{\tilde{I}_c}$ for any configuration with more than 9-10 mantissa bits. Therefore, for these precisions, the bias estimator (8) needs much fewer samples than the custom-precision output estimator (5) to achieve the same variance. This property is crucial for the proposed method. It means that, using the proposed bias estimator, it is possible to perform short MCMC pre-runs for a set of candidate precisions to estimate the biases for a given output estimate, without posing a large overhead to the final run (which estimates (4)). Figure 3 also depicts the ratio $\frac{\sigma^2_{str}}{\sigma^2_{\tilde{I}_c}}$ (straightforward estimator), which is always above 1.

IV. Optimization Method

The goal of the optimization method is to choose the most resource-efficient precision configuration c, out of a user-defined set of candidate configurations, so that using the custom-precision output estimator (5) introduces bias b_c within a user-specified range. To do this, short FPGA pre-runs are performed for all candidate precisions and the bias is estimated using the estimator of the previous section.

The optimization process requires some input from the user and then follows four steps: The double- and custom-precision pre-runs (Steps 1 and 2), the precision selection (Step 3) and the final run (Step 4). The method exploits the reconfiguration property of FPGAs to optimize precision. Two different configuration bit-streams are used in the process. Steps 1 and 2 use the same mixed-precision bit-stream, which is able to draw MCMC samples in any of the candidate precisions. Step 4 uses a different, optimized-precision bit-stream, which draws samples only in the optimized precision (found by the previous steps). Step 3 runs in software. Both bit-streams are taken from a library of pre-compiled circuits (Section IV-E). The exact flow of the method is the following:

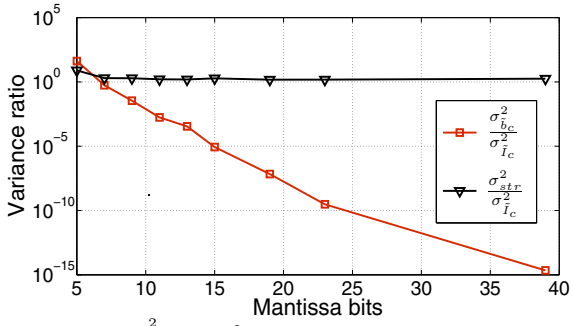

Figure 3. Ratios $\frac{\sigma^2_{\tilde{b}_c}}{\sigma^2_{\tilde{I}_c}}$ and $\frac{\sigma^2_{str}}{\sigma^2_{\tilde{I}_c}}$ for various mantissa bit configurations (exponents bits=8). $p_c(\mathbf{x})$ is a Gaussian mixture distribution (Section V) and $f(\mathbf{x})$ is the mean estimator. $N = 10^6$ samples are used for all estimators. The proposed estimator's variance ($\sigma^2_{\tilde{b}_c}$) is orders of magnitude smaller than that of the output estimator ($\sigma^2_{\tilde{I}_c}$) for most precisions. The straightforward estimator's variance (σ^2_{str}) is always larger than $\sigma^2_{\tilde{I}_c}$.

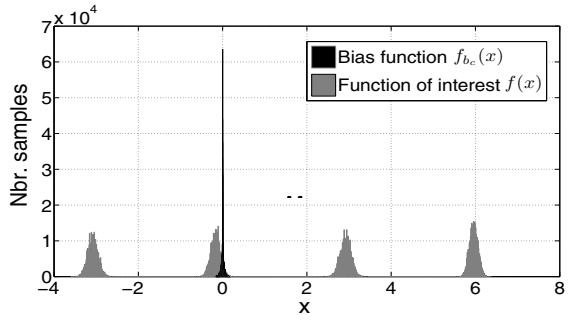

Figure 4. Histograms of $f(\mathbf{x})$ and $f_{b_c}(\mathbf{x})$ for the mean estimator under $p_c(\mathbf{x})$ with $c = (8, 13)$, where $p_c(\mathbf{x})$ is a Gaussian mixture distribution (Section V). The variance of $f(\mathbf{x})$ is significantly larger.

A. User input

Table I summarizes the input parameters. The user defines a target SD_T for the standard deviation $\sigma_{\tilde{I}_c}$ of the output estimate (5), meaning that the final run will continue until this standard deviation is reached. The standard deviation is used instead of the variance because it is easier to interpret. All standard deviations are approximated by performing multiple MCMC runs (typically 50-100) with different random seeds (a run refers to such a set of multiple runs). Apart

Table I
USER PARAMETERS OF THE OPTIMIZATION METHOD

SD_T	Target standard deviation in output estimate.
T_{bias}	Bias threshold as a percentage of SD_T.
S	Set of candidate precisions.
$Term_{\%}$	Time available for Steps 1 and 2 as a percentage of t_{est}.
Pr_{min}	Minimum probability to accept a precision. If the probability that the absolute bias is smaller than $SD_T \cdot T_{bias}$ is higher than Pr_{min}, the precision is acceptable.

from SD_T, the user also gives a threshold T_{bias} for the bias they can tolerate as a percentage of SD_T (e.g. $T_{bias} = 0.5$ for bias tolerance equal to 50% of SD_T). Together, SD_T and T_{bias} determine the bias threshold $SD_T \cdot T_{bias}$, which is the maximum absolute bias that can be tolerated. By changing these parameters, the user trades off accuracy in the output estimate for lower precision. The user also defines the set of candidate precisions (S) and the parameters $Term_{\%}$ and Pr_{min}, which will be explained shortly.

B. Steps 1 and 2: Pre-runs (mixed-precision bit-stream)

The goal of the pre-runs is to estimate the bias for every precision. A mixed-precision bit-stream is loaded on the FPGA (Steps 1, 2 in Figure 5). It contains probability evaluation blocks in all candidate precisions (S) and in double precision, a block which implements the generic operations of the MCMC algorithm in double precision (Section II-A) and a weight evaluation module in double precision. Sampling with precision c is done by plugging the respective probability evaluation block to the generic block.

Step 1 consists of a short double-precision MCMC run, which estimates how much time a double-precision FPGA sampler would need to give an output estimate with standard deviation equal to SD_T. This time is then used to find how long the pre-runs can last without introducing a large overhead to the final run. Step 1 lasts until the sampler converges. Samples are sent to the host PC in DMA batches and the Gelman-Rubin diagnostic (popular in MCMC literature [2]) is used after each batch to detect convergence. After sampling stops, if N_{pre} samples have been taken and the measured standard deviation in the output estimate is SD_{pre}, the samples needed to reach SD_T are $N_T = N_{pre}\frac{SD_{pre}^2}{SD_T^2}$, since standard deviation drops with $\frac{1}{\sqrt{N}}$ (see (3)). A double-precision run would thus need t_{est} seconds to reach SD_T:

$$t_{est} = \frac{N_T}{TH_{dp}} \tag{14}$$

Here, TH_{dp} is the throughput (in samples/sec) of a double-precision sampler utilizing the whole FPGA (known). The available time for Steps 1 and 2 is 5% or 10% of t_{est} (defined by user parameter $Term_{\%}$, e.g. $Term_{\%} = 0.05$).

After the available time for pre-runs is known, Step 2 starts. It estimates the bias $\tilde{b}_c$ introduced in the output estimate by each candidate precision configuration, as well as the standard deviation of the bias $\sigma_{\tilde{b}_c}$. The two quantities are given by (8) and (13) respectively. The custom-precision probability evaluation blocks (unused in Step 1) are sequentially plugged to the generic block in order to get samples from every $p_c(\mathbf{x})$. An equal number of samples is taken for each c. The double-precision block is now used for a different purpose compared to Step 1. It computes all sample probabilities in double precision and sends them to the weight evaluator module to get the weights $w_c(\mathbf{x})$. The samples and weights are sent to the host in DMA batches until $Term_{\%} \cdot t_{est}$ seconds have passed and $\tilde{b}_c$ and $\sigma_{\tilde{b}_c}$ are computed in the host for all precisions.

C. Step 3: Selection (software)

Step 3 chooses the optimized precision based on the user's accuracy requirements. It runs in the host (Figure 5). Each bias estimate is a Normal random variable (Section III-A) with mean $\tilde{b}_c$ and standard deviation $\sigma_{\tilde{b}_c}$. The criterion to accept a configuration c is the probability that the absolute value of its bias b_c is smaller than the user-defined tolerance:

$$\begin{aligned} &p(-SD_T \cdot T_{bias} < b_c < SD_T \cdot T_{bias}) \\ &= N_{CDF}(SD_T \cdot T_{bias}, \tilde{b}_c, \sigma_{\tilde{b}_c}) \\ &-N_{CDF}(-SD_T \cdot T_{bias}, \tilde{b}_c, \sigma_{\tilde{b}_c}) \end{aligned} \tag{15}$$

where $N_{CDF}(x, \mu, \sigma)$ is the value in x of a Normal cumulative density function with mean μ and standard deviation σ. If the probability in (15) is larger than a user-defined value Pr_{min} (e.g. $Pr_{min} = 0.95$), the bias is very likely to be within tolerance and thus the configuration is accepted. Otherwise, the bias is considered too large. The lowest precision to pass this test is chosen as the optimized one. If no precision passes the test, double-precision is chosen. The *optimized* precision might not always be the *optimal* one, as will be demonstrated in Section VI.

D. Step 4: Final runs (optimized-precision bit-stream)

After Step 3, a second configuration bit-stream is selected from the pre-compiled library and loaded. It contains probability evaluation blocks only in the optimized precision and a generic MCMC block (Figure 5 (right)). The throughput of this sampler is higher than Step 2 samplers, since the FPGA resources previously used for the various candidate precisions are now available. The final MCMC run starts and lasts until SD_T is reached. The resulting estimate is biased within tolerance (guaranteed with probability Pr_{min}).

E. Pre-compiled bit-stream library

The method assumes that a library of pre-compiled bit-streams has been generated off-line, containing different versions of the two bit-streams used during optimization. This is a realistic assumption, since the bit-streams are re-usable (i.e. the same $p(\mathbf{x})$ is often sampled in different problems and settings). Because it is difficult to include a mixed-precision bit-stream for every possible combination of candidate precisions S, an alternative is to include only some common combinations in the library. Full utilization of the targeted FPGA's resources is assumed for all bit-streams.

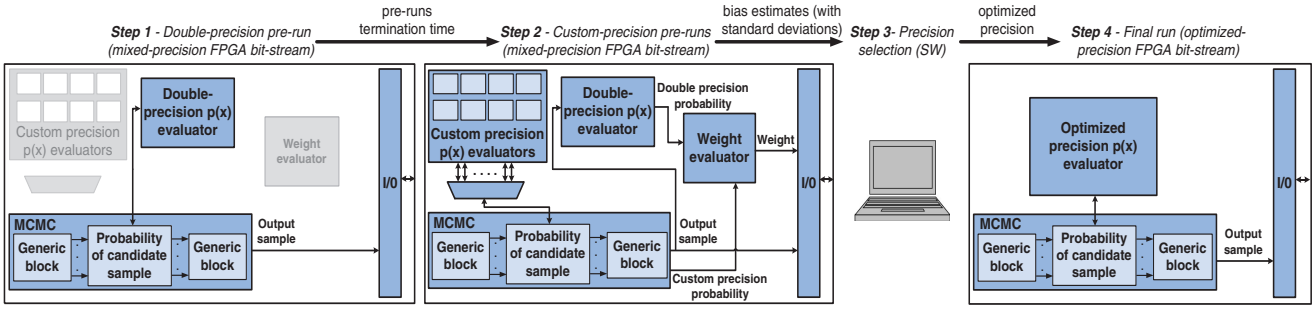

Figure 5. Optimization method flow: Steps 1,2 (mixed-precision FPGA bit-stream), 3 (software) and 4 (optimized-precision FPGA bit-stream) are shown. During Step 1, the double-precision module is used to draw MCMC samples. During Step 2, it evaluates the probabilities of all generated samples, which are used for weight evaluations. Grey-colored blocks (Step 1) are unused. The top part of the figure shows the information passed by each step to the next.

F. Discussion

Taking into account the shape of $p(\mathbf{x})$ and $f(\mathbf{x})$, the accuracy parameters SD_T and T_{bias}, the required certainty that the bias is within tolerance (Pr_{min}) and the tolerable pre-run overhead ($Term_{\%}$), the method can tailor the precision to the targeted problem only, as well as to the specific requirements of the user. It does not attempt to cover every possible output estimate related to the sampled distribution. For example, when a particular $f(\mathbf{x})$ has high values only in areas where precision slightly affects the value of $p(\mathbf{x})$, the output estimate is likely to be robust to decreasing precision and the method exploits this to achieve resource savings.

V. Case studies and MCMC method

Two Bayesian inference problems are used for evaluation. The particular case studies are representative of the distributions normally targeted by MCMC, both in terms of the types of arithmetic operators used, as well as the form of data-parallelism they incorporate. The Parallel Tempering (PT) MCMC algorithm [1] is employed to sample from the distributions. PT runs a population of Markov chains in parallel (here, 128 chains are used) to improve sampling efficiency. PT's inherent parallelism and the data-parallelism mentioned above mean that many $p(\mathbf{x})$ evaluations can happen in parallel. The amount of computations is large enough to fully utilize FPGA resources even for low precisions.

A. Mixture model inference

Mixture models (MMs) are a family of models heavily used in machine learning [2]. Here, a Gaussian MM from [9] is used: A set of 100 independent data $\mathbf{d} = d_{1:100}$ with $d_q \in \Re$ is given. Each datum is distributed according to:

$$p(d_q|\mu_{1:4}, \sigma_{1:4}, w_{1:4}) = \sum_{i=1}^{4} w_i f(d_q|\mu_i, \sigma_i) \tag{16}$$

where f is the density of a univariate normal distribution and $\mu_{1:4}$, $\sigma_{1:4}$ and $w_{1:4}$ are model parameters. Taking $\sigma_i = 0.55$ and $w_i = 1/4$ for $i \in \{1, ..., k\}$, the goal is to sample from the posterior distribution of the means $\mu_{1:4}$ (given the data $\mathbf{d}$), which is unknown. The prior distribution of μ ($p(\mu)$) is uniform. The data are simulated using $\mu = (-3, 0, 3, 6)$. The likelihood of the data is $p(\mathbf{d}|\mu) = \prod_{j=1}^{100} p(d_j|\mu)$. The posterior distribution of μ (sampled by MCMC) is:

$$p(\mu|\mathbf{d}) \propto p(\mathbf{d}|\mu)p(\mu) \tag{17}$$

B. Neural network inference

Neural networks (NNs) are universal function approximators [1]. Bayesian inference is one of the methods used to train NNs. Here, an example from [1] is used: Input data $D_{in} = \mathbf{x}_{1:200}$ with $\mathbf{x}_t = (1, -10 + t * .1)$ and output data $D_{out} = y_{1:200}$ with $y_t = f(\mathbf{x}_t) + \epsilon_t$ are generated, where $\epsilon_t \sim N(0, \sigma^2)$ for $t \in \{1, ..., n\}$ is noise. The function $f(\cdot)$ is unknown and is approximated by a 2-2-1 NN:

$$\tilde{f}(\mathbf{x}_t) = \sum_{j=1}^{j=2} \beta_j \psi(\mathbf{x}_t^{'} \gamma_j) \tag{18}$$

where $\beta_j \in \Re$ and $\gamma_j \in \Re^2$ are the connection weights of the network. The activation function $\psi(z)$ is a $tanh$ function. The model can be viewed as a non-linear regression model, where the aim is to infer the unknown $\beta_{1:2}$ and $\gamma_{1:2}$ and the variance σ^2 of the noise. The prior distributions are described in [1]. The data are simulated using $\gamma_1 = (2, -1)$, $\gamma_2 = (1, 1.5)$, $\beta_1 = 20$, $\beta_2 = 10$, $\sigma = .1$. The log-posterior distribution (sampled by MCMC) is given by (19), where ν, δ, σ_β and σ_γ are all constants [1].

$$\begin{aligned} &logp(\beta_{1:2}, \gamma_{1:2}, \sigma^{-2}|D_{in}, D_{out}) \propto -(100 + \nu - 1)log(\sigma^2) \\ &-\tfrac{1}{2\sigma^2}\{2\delta + \sum_{t=1}^{2}[y_t - \sum_{j=1}^{2} \beta_j \psi(\mathbf{x'}_t \gamma_j)]^2\} \\ &-\sum_{j=1}^{2} \tfrac{\beta_j^2}{2\sigma_\beta^2} - \sum_{j=1}^{2}\sum_{i=1}^{2} \tfrac{\gamma_{ij}^2}{2\sigma_\gamma^2} \end{aligned} \tag{19}$$

VI. Evaluation

A Xilinx ML605 board, which contains a Xilinx Virtex-6 LX240T FPGA was used for evaluation. The card was connected to a host PC containing an Intel i5-650 CPU and running Linux. The RIFFA PCI-Express framework [10] was used for communication with the host. A double buffering

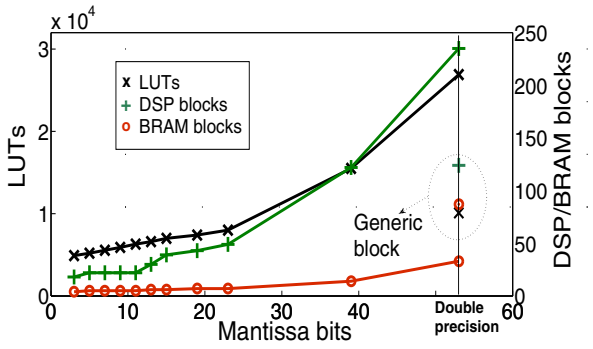

Figure 6. Plotted lines show the resource utilization of a single probability evaluation block which implements the MM posterior for different precisions (exp.bits are 8, except for double precision, where they are 11). The generic MCMC block's resources are also shown (double precision).

architecture with BRAM buffers was implemented on top of RIFFA to avoid halting runs. The sustained FPGA-to-host data transfer throughput was 170 MB/sec (one PCI-Express lane). A C script on the host side handled all software operations and the communication with the FPGA (using RIFFA's software functions). Hardware implementations of the pre- and final-run systems were written in VHDL, using floating point operators generated by FloPoCo [11] and the generic PT MCMC block presented in [12]. All designs ran on a single 125 MHz clock (RIFFA default) and fully utilized the FPGA's resources. All results are post place and route.

Figure 6 shows the resource utilization of a single probability evaluation block for different precisions (MM case study). Large savings are possible by reducing precision, allowing for the instantiation of more parallel blocks. The resource utilization of a generic MCMC block is also shown.

Figures 7 and 8 show Steps 2 and 3 of the optimization process for one of the possible output estimates in the MM case study (2nd moment of μ_1). The target standard deviation was set to $SD_T = 0.02$ and the bias tolerance to $T_{bias} = 0.5$. Six candidate precisions were used ($S = \{(8,23),(8,19),(8,15),(8,13),(8,11),(8,9)\}$). Figure 7 shows the bias estimates for three of these precisions. Both the bias and its standard deviation increase with lower precisions. The figure also shows that Step 2 terminates after around $8.5 \cdot 10^5$ samples to limit the pre-runs' overhead. This number comes from the preceding Step 1 (not shown in the figure), when setting $Term_{\%} = 0.05$. Figure 8 shows the precision selection procedure (Step 3), using $Pr_{min} = 0.95$. The probability that a bias is within tolerance (see (15)) converges to a certain value as more samples are drawn. Few samples suffice to see if this value is acceptable for high-precisions. More samples are needed for low precisions (this is due to the high variance in the bias estimates for low precisions, see Figure 3). The fluctuations of the probability values are due to variance (the values are probabilistic estimates). At termination, the lowest precision which gives tolerable bias with 95% probability is $c = (8,15)$. This is

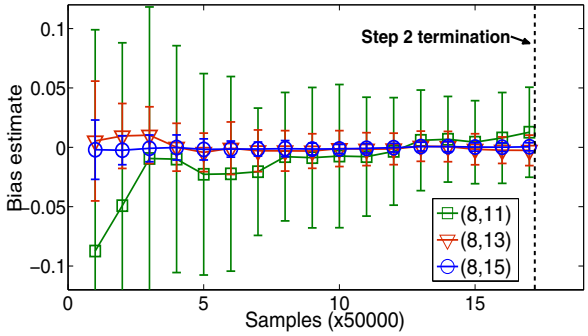

Figure 7. Step 2: Bias estimates for 3 of 6 candidate precisions (MM case study, 2nd moment of μ_1). Error bars represent $\pm 2\sigma_{\tilde{b}_c}$. The vertical line shows when the pre-runs stop to avoid large overheads to the final run.

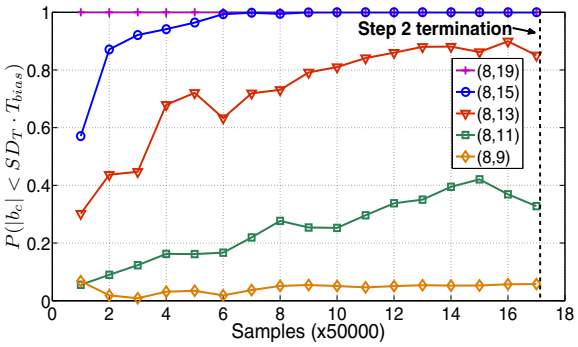

Figure 8. Step 3: Probability that the bias is within tolerance for five precisions. The output estimate is the 2nd moment of μ_1 (MM case study). Parameters are $SD_T = 0.02$, $T_{bias} = 0.5$, $Pr_{min} = 0.95$, $Term_{\%} = 0.05$. The optimized precision is $c = (8,15)$.

chosen as the optimized precision, the respective optimized-precision bit-stream is loaded and the final run starts. The *optimized* precision is not always the *optimal* one in the set S. If the pre-runs terminate too early because of a large target SD_T or small $Term_{\%}$, a sub-optimal precision can be chosen (e.g. if Step 2 terminates after 10^5 samples in Figure 8). This means a sub-optimal sampling throughput in the final run. Nevertheless, the bias threshold is still guaranteed.

Table II lists the optimization and speedup results from applying the method to both case studies, for different functions of interest and parameter combinations. S, $Term_{\%}$ and Pr_{min} are the ones mentioned before. The SD_T values were chosen to represent one to two decimal digits of accuracy in the MCMC estimate and the chosen T_{bias} values give bias thresholds smaller than one SD_T. All final runs lasted until SD_T was reached. Depending on the output estimate and the parameters, the chosen precisions range from $(8,13)$ to $(8,19)$. By targeting a specific output estimate each time and taking into account the various user parameters, the precision is highly optimized for the problem under investigation only.

To evaluate the achieved speedups, Step 1-4 runtimes were measured, added and compared to the double-precision runtimes on the same FPGA. Comparisons to single-precision designs are also included, although double-precision is the

Table II
OPTIMIZATION RESULTS FOR VARIOUS ESTIMATES AND USER PARAMETERS FROM THE MIXTURE MODEL (MM) AND THE NEURAL NETWORK (NN) CASE STUDIES. IN ALL EXAMPLES: $Pr_{min} = 0.95$, $Term_{\%} = 0.05$ AND $S = \{(8,23),(8,19),(8,15),(8,13),(8,11),(8,9)\}$.

Output estimate	SD_T	T_{bias}	Optimized precision	Pre-runs overhead (% of total runtime)	Speedup vs.DP FPGA	Speedup vs.SP FPGA	Speedup vs.SW	Speedup vs. GPGPU (128 chains) [9]
Mean of μ_1 (MM)	$5 \cdot 10^{-2}$	1	**(8,13)**	21.3%	**4.48x**	1.18x	246x	17.5x
	$5 \cdot 10^{-2}$	0.5	**(8,13)**	22.0%	**4.59x**	1.20x	252x	18.0x
	$2 \cdot 10^{-2}$	0.5	**(8,15)**	19.7%	**4.10x**	1.08x	225x	16.0x
2nd moment of μ_1 (MM)	$1 \cdot 10^{-1}$	1	**(8,13)**	24.1%	**4.38x**	1.15x	241x	17.2x
	$1 \cdot 10^{-1}$	0.5	**(8,13)**	21.1%	**4.23x**	1.12x	233x	16.6x
	$5 \cdot 10^{-2}$	0.5	**(8,15)**	22.6%	**4.05x**	1.06x	223x	15.9x
Mean of β_1 (NN)	$2 \cdot 10^{-1}$	0.5	**(8,13)**	18.9%	**4.95x**	1.42x	440x	-
	$5 \cdot 10^{-2}$	0.5	**(8,13)**	21.4%	**5.02x**	1.45x	446x	-
2nd moment of γ_{10} (NN)	$2 \cdot 10^{-1}$	0.5	**(8,15)**	20.1%	**4.76x**	1.38x	423x	-
	$2 \cdot 10^{-2}$	0.25	**(8,19)**	21.6%	**3.57x**	1.03x	317x	-

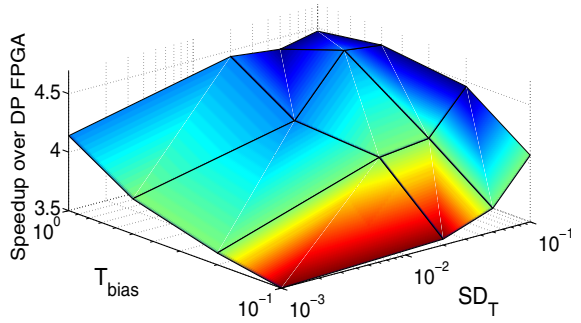

Figure 9. Speedup scaling with SD_T and T_{bias}. The output estimate is the mean of μ_1 (MM case study). $Pr_{min} = 0.95$, $Term_{\%} = 0.05$.

default approach in MCMC (due to the generally unknown robustness of output estimates to reduced precision). The table also includes comparisons to the performance of double-precision C++ implementations on a PC with an Intel i5-650 CPU (for both examples) and a single-precision CUDA implementation on an Nvidia GTX280 GPGPU [9] (only for the MM example). The software implementations did not exploit parallelism. The speedups of the optimized-precision samplers over double-precision and single-precision FPGA samplers (including pre-runs) are 3.57x-5.02x and 1.03x-1.45x respectively. The speedup compared to software ranges from 223x to 446x. The FPGA also out-performs the GPGPU by up to 18 times when 128 parallel tempering chains are employed.

Finally, Figure 9 shows how the speedup over a double-precision FPGA sampler scales with the choice of accuracy parameters SD_T and T_{bias}. The precisions are between $(8,13)$ and $(8,19)$ and 3.50x-4.59x speedups are achieved.

VII. CONCLUSION

This paper presents an automated method that exploits the reconfigurable nature of FPGAs to minimize the arithmetic precision of any FPGA-mapped MCMC sampler, while guaranteeing a user-constrained bias in the output. The method uses an efficient bias estimator, which is proposed here. Results show that significant speedups over double-precision designs are achieved. Future work will focus on extending the methodology to fixed-point arithmetic and comparing it to the method of [7] (applied to MCMC).

REFERENCES

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

[2] *Handbook of Markov Chain Monte Carlo*, 1st ed. Chapman and Hall/CRC, May 2011.

[3] D. Newman, A. Asuncion, P. Smyth, and M. Welling, "Distributed Algorithms for Topic Models," *Journal of Machine Learning Research*, vol. 10, pp. 1801–1828, 2009.

[4] S. Zierke and J. Bakos, "FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods," *BMC Bioinformatics*, vol. 11, no. 1, pp. 184+, 2010.

[5] I. Lebedev, C. Fletcher, S. Cheng, J. Martin, A. Doupnik, D. Burke, M. Lin, and J. Wawrzynek, "Exploring Many-Core Design Templates for FPGAs and ASICs," *International Journal of Reconfigurable Computing, Article ID 439141*, vol. 2012, 2012.

[6] G. Mingas and C.-S. Bouganis, "A Custom Precision Based Architecture for Accelerating Parallel Tempering MCMC on FPGAs without Introducing Sampling Error," in *Proc. FCCM*, 2012, pp. 153 –156.

[7] G. C. T. Chow, A. H. T. Tse, Q. Jin, W. Luk, P. H. Leong, and D. B. Thomas, "A mixed precision Monte Carlo methodology for reconfigurable accelerator systems," in *Proc. FPGA*, 2012, pp. 57–66.

[8] X. Tian and C.-S. Bouganis, "A Run-Time Adaptive FPGA Architecture for Monte Carlo Simulations," in *Proc. FPL*, 2011, pp. 116 –122.

[9] A. Lee, C. Yau, M. B. Giles, A. Doucet, and C. C. Holmes, "On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods," *Journal of Computational and Graphical Statistics*, vol. 19, no. 4, pp. 769–789, 2010.

[10] M. Jacobsen, Y. Freund, and R. Kastner, "RIFFA: A Reusable Integration Framework for FPGA Accelerators," in *Proc. FCCM*, 2012, pp. 216 –219.

[11] F. de Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," *IEEE Design and Test of Computers*, vol. 28, pp. 18–27, 2011.

[12] G. Mingas and C.-S. Bouganis, "Parallel Tempering MCMC Acceleration Using Reconfigurable Hardware," in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7199, pp. 227–238.