Population-based MCMC on multi-core CPUs, GPUs and FPGAs

Grigorios Mingas, *Student Member, IEEE* and Christos-Savvas Bouganis, *Member, IEEE*

Abstract—Markov Chain Monte Carlo (MCMC) is a method to draw samples from a given probability distribution. Its frequent use for solving probabilistic inference problems, where big-scale data are repeatedly processed, means that MCMC runtimes can be unacceptably large. This paper focuses on population-based MCMC, a popular family of computationally intensive MCMC samplers; we propose novel, highly optimized accelerators in three parallel hardware platforms (multi-core CPUs, GPUs and FPGAs), in order to address the performance limitations of sequential software implementations. For each platform, we jointly exploit the nature of the underlying hardware and the special characteristics of population-based MCMC. We focus particularly on the use of custom arithmetic precision, introducing two novel methods which employ custom precision in the largest part of the algorithm in order to reduce runtime, without causing sampling errors. We apply these methods to all platforms. The FPGA accelerators are up to 114x faster than multi-core CPUs and up to 53x faster than GPUs when doing inference on mixture models.

Index Terms—Field Programmable Gate Array, Graphics Processing Unit, Markov Chain Monte Carlo, Parallel Tempering, Custom Arithmetic Precision

1 Introduction

MCMC is a class of stochastic algorithms which generate random samples from a given probability distribution. They are widely used in statistical applications, ranging from machine learning [1], [2], [3], statistical physics [4], [5] and geostatistics [6] to medical imaging [7], genetics [8], phylogenetics [9], computational biology [10], [11] and stochastic optimization [12], [13]. Sampling from a distribution is fundamental in these applications because we are typically interested in performing the following task:

Monte Carlo Integration: Estimate the expectation of a function $f(\mathbf{x})$ under a probability distribution with density $p(\mathbf{x})$, 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 a random variable of interest. Most problems in the above fields (e.g. parameter inference, prediction, model comparison) can be expressed in the above form.

Because an analytical solution to the integral is usually not available and deterministic numerical methods become inefficient in large dimensions [5], stochastic estimators such as MCMC are often the methods of choice. MCMC estimates (1) by drawing samples from $p(\mathbf{x})$ (the target distribution) and computing the following approximation:

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

where $\mathbf{x}^{(i)}$, $i \in \{1,...,N\}$ are samples taken from $p(\mathbf{x})$. The variance of this estimator reduces linearly with N. MCMC's ability to sample from arbitrary target distributions has made it the mainstream method to perform Bayesian inference for complex probabilistic models.

In this paper, we examine a family of advanced MCMC methods (population-based MCMC [14]) which are designed to address a particular type of complexity in the target distribution, called multi-modality. Multi-modal distributions have multiple separate modes and appear in many problems, e.g. inference on Restricted Boltzmann Machines [1] and mixture models [14], [15], genetics [8], [9] and biological simulations [11].

Population-based MCMC samplers often require large runtimes when applied to modern Bayesian problems of this type, e.g. runtimes of weeks or months are common in [2], [3], [4], [8], [10], [16]. This is due to: 1) The massive size of data sets in Bayesian inference applications. For example, topic models use large text databases [2], while genetic studies use whole genomes of thousands of individuals [8], [9]. In all these problems, it is necessary to process the whole data set to compute the density of the target distribution. Because one density evaluation is needed at each MCMC step, using MCMC with big data is extremely intensive computationally. 2) The fact that population-based MCMC uses a population of Markov chains (typically dozens of chains instead of the single chain used by basic MCMC). 3) The fact that multiple MCMC runs are performed in most settings to confirm convergence and approximate the variance of (2) [7]. 4) The problems of slow convergence (the time until the sampler converges to the "correct" distribution) and slow mixing (how fast the sampler explores the distribution). We often need a large N to get a satisfactory variance in (2).

Because runtimes often become impractical, practitioners are forced to collect fewer samples (which increases

• *The authors are with the Department of Electrical and Electronic Engineering, Imperial College, London, SW7 2AZ, UK. E-mail: g.mingas10@imperial.ac.uk*

variance) or use simpler models and/or fewer data, compromising the effectiveness of the analysis. In order to handle this computational burden, it is insufficient to rely solely on more efficient MCMC algorithms. It is equally important to leverage the power of modern hardware accelerators, such as multi-core Central Processing Units (CPUs), Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs). Even more crucially, understanding of the properties of the underlying hardware is not only necessary during the implementation stage of MCMC; certain computational or algorithmic structures map favorably to existing parallel architectures and this should affect choices during the algorithmic and architectural design stages of population-based MCMC and MCMC methods in general.

Here, we present an endeavour towards this direction, targeting Parallel Tempering (PT) [17], the most popular population-based MCMC method. We present optimized implementations in three hardware platforms; multi-core CPUs, GPUs and FPGAs. As a benchmark, we perform Bayesian inference on a mixture model [15], which is representative of multi-modal problems. In more detail, the contributions of this work are:

1) Three optimized accelerators (for CPUs, GPUs and FPGAs) for population-based MCMC which employ double precision and deliver speedups of up to 16.1x, 165x and 174x respectively over a single-core CPU.

2) Two novel, custom precision methods for population-based MCMC, which allow us to reduce runtimes without affecting sampling quality. The methods either use a weighting scheme to correct errors (Weighted PT - WPT) or use custom precision in parts of the algorithm which do not affect output accuracy (Mixed-Precision PT - MPPT). A theoretical proof of the accuracy of the latter is included.

3) Mappings of these methods to CPUs, GPUs and FPGAs, which result in further speedups of up to 1.4x, 3.2x and 6.5x respectively over the baseline implementations.

4) A precision optimization process for WPT and MPPT on FPGAs and a kernel optimization process for GPU accelerators, which maximize sampling throughput.

5) An investigation of the way the performance of the accelerators scales with the size of the chain population and the size of the data set and results on the power and cost efficiency of each accelerator.

2 Background

2.1 MCMC principles

MCMC draws samples from a distribution $p(\mathbf{x})$ by constructing a Markov chain [5]. Each state of the chain represents a random sample. A transition kernel is used to generate a sample, given the previous one (i.e. update the chain). The kernel comprises two steps: 1) Propose a sample using an easy-to-sample distribution, e.g. a Noraml with mean equal to the previous sample, 2) Compute the probability of the proposed sample according to the target density and accept or reject the sample using some criterion.

The details of each step vary between kernels ([7], [14], [18]), making each kernel suitable for distributions with specific characteristics (e.g. correlations, multi-modality). The goal is to achieve fast convergence and mixing.

2.2 Population-based MCMC and PT

When sampling from multi-modal distributions, elementary kernels (e.g. Metropolis [18]) tend to get "stuck" in one of the modes [6], [14], resulting in slow convergence and mixing. Here, we focus on a family of MCMC methods designed to tackle this problem; population-based MCMC. Population-based MCMC instantiates a population of Markov chains, each of them sampling from a slightly different distribution. Certain types of interactions are introduced between the chains, which help to improve the algorithm's convergence and mixing properties [14].

More specifically, we examine a representative population-based method called Parallel Tempering (PT) [17]. Fig. 1 shows the pseudocode of PT. The chain population consists of m chains and chain j ($j \in \{1,...,m\}$) samples from a distribution with density $p_j(\mathbf{x})$:

$$p_j(\mathbf{x}) = p(\mathbf{x})^{1/T_j}, j \in \{1,...,m\} \tag{3}$$

where T_j (with $1 = T_1 < T_2 < ... < T_m$) is the temperature of chain j. The density $p_1(\mathbf{x})$ is the target density $p(\mathbf{x})$ (since $T_1 = 1$). The remaining densities are "tempered" versions of $p(\mathbf{x})$. Temperatures increase with higher j, which results in gradually smoother densities, i.e. closer to uniform. Practically, this means that "hot" chains move quickly in the state space (they jump more easily between the now smoothed modes), while "cold" chains move slowly but sample from distributions closer to the target $p(\mathbf{x})$.

The algorithm performs N iterations (loop in line 1 of Fig. 1) in order to draw N samples from the target distribution $p_1(\mathbf{x}) = p(\mathbf{x})$. Samples from the tempered (auxiliary) distributions are discarded. Every iteration comprises a Global update (line 2) and a Global exchange (line 9).

During the Global update of iteration i, the current samples of all chains ($\mathbf{x}_1^{(i)},...,\mathbf{x}_m^{(i)}$) are updated using separate Metropolis transition kernels (loop in line 3). Each kernel samples from the distribution of the corresponding chain (see (3)). The proposal and accept/reject steps of the kernel are shown in lines 4 and 5-7. The latter requires computing the probability of the proposed sample using the target density of the chain ($p_j(\mathbf{y}^{(i)})$ for iteration i and chain j).

During Global exchanges, PT attempts interactions between chain pairs; sample exchanges are proposed between all odd pairs of neighboring chains, i.e. $(1,2),...,(m-1,m)$ or all even pairs of neighboring chains, i.e. $(2,3),...,(m-2,m-1)$ (in a rotating order - line 10). These exchanges push samples from the "hot" chains to the "colder" ones and eventually to the first (coldest) chain, helping it escape from isolated modes (thus enhancing mixing).

The choice of the number of chains and their temperatures is important for this enhancement to be significant [11], [14] but it is outside the scope of this paper. The temperature of chain j is set to $(\frac{m}{m+1-j})^2$, which is suitable for the target in Section 2.4 [16]. Only the samples $\mathbf{x}_1^{((B+1):N)}$

```
Inputs:
Densities $p_{1:m}$ (from (3)), initial samples $(\mathbf{x}^{(1)}_{1:m})$,
proposal variances $(\sigma^2_{1:m})$, number of MCMC
samples ($N$), number of burn-in samples ($B$)
Algorithm:
1: for $i = 2,\ldots,N$
2:     Global update:
3:     for $j = 1,\ldots,m$
4:         $\mathbf{y}^{(i)}_j \leftarrow \mathbf{x}^{(i-1)}_j + N(0,\sigma^2_j)$
5:         Do $\mathbf{x}^{(i)}_j \leftarrow \mathbf{y}^{(i)}_j$ with probability:
6:         $a(\mathbf{x}^{(i-1)}_j, \mathbf{y}^{(i)}_j) = min\Big(1, \frac{p_j(\mathbf{y}^{(i)}_j)}{p_j(\mathbf{x}^{(i-1)}_j)}\Big)$
7:         Otherwise do $\mathbf{x}^{(i)}_j \leftarrow \mathbf{x}^{(i-1)}_j$
8:     end for

9:     Global exchange:
10:    Choose even chain pairs $((1,2),(3,4),\ldots)$ or
       odd chain pairs $((2,3),(4,5),\ldots)$ (in turn)
11:    for all chosen chain pairs $(q,r)$
12:        Exchange the samples $\mathbf{x}^{(i)}_q$ and $\mathbf{x}^{(i)}_r$ with
           probability:
13:        $e(\mathbf{x}^{(i)}_q, \mathbf{x}^{(i)}_r) = min\Big(1, \frac{p_q(\mathbf{x}^{(i)}_r)p_r(\mathbf{x}^{(i)}_q)}{p_q(\mathbf{x}^{(i)}_q)p_r(\mathbf{x}^{(i)}_r)}\Big)$
14:    end for
15: end for
Outputs:
Samples from the first chain $\mathbf{x}^{((B+1):N)}_1$
```

Fig. 1. Parallel Tempering algorithm

from chain 1 are kept (B samples are removed as burn-in) and used to get (2) (with $\mathbf{x}^{(i)} = \mathbf{x}^{(i)}_1$).

2.3 Parallelism in the algorithm

There are two forms of parallelism in PT: 1) Inter-chain parallelism during Global updates (loop in line 3 of Fig. 1); all chains can be updated in parallel since they do not interact during this stage. Global exchanges can also be parallelized since pairs of chains communicate but each exchange is independent. Unfortunately, using more than 100-200 chains does not offer any improvement in mixing [11], [16] and thus inter-chain parallelism can be exploited up to a limit. Therefore it is crucial for an accelerator to achieve high performance even for moderate numbers of chains. 2) Intra-chain parallelism during the computation of the probability density of a proposed sample (line 6 in Fig. 1). This parallelism is density-dependent. Due to the large diversity of models and corresponding densities, it is impossible to investigate the effect of parallelising this part in the general case. Nevertheless, it is common in Bayesian problems to use independent and identically distributed (i.i.d.) data. This leads to a density which is equal to the product of the sub-densities of all the data (or the sum if for log-densities). This work only considers this form of density parallelism (Section 2.4), because: 1) It is widely used and representative of many applications (e.g. [3], [16], [19]), 2) Several MCMC algorithms are designed to address this specific form of likelihood [20], 3) The kinds of computations involved in the case study of Section 2.4 (Gaussian evaluations, reductions, exponents, logarithms) are very common, even in non-i.i.d. problems.

TABLE 1
Algorithm and model parameters

Symbol	Description	Range (in Section 5)
m	Number of PT chains	$[8, 32768]$
T	PT temperature set	$(\frac{m}{m+1-j})^2$ for chain j
s	Dimension of sampled state space	4
n	Number of data	$[128, 32768]$
d	Dimension of data	1

2.4 Case study: Bayesian inference on mixture models

Mixture models are a powerful family of probabilistic models used in numerous fields [15]. Multi-modal target distributions often appear when performing inference on these models. Hence, they are representative of the problems on which PT is applied. A Gaussian mixture model taken from [16] is used here. This model obeys the i.i.d. principle described in the previous section. It allows us to easily scale the data-related computational load, i.e. the available parallelism in intra-chain computations.

A set of i.i.d. data $D_{1:n}$, where $D_l \in \Re$ for $l \in \{1,...,n\}$, is given. Each observation is distributed according to:

$$p(D_l|\mu_{1:k},\sigma_{1:k},a_{1:k-1}) = \sum_{i=1}^{k} a_i f(D_l|\mu_i,\sigma_i) \qquad (4)$$

Here, f denotes the density of a univariate Gaussian distribution, k is the number of mixture components and $\mu_{1:k}$, $\sigma_{1:k}$ and $a_{1:k-1}$ are the parameters of the model (means, variances and weights of components respectively). We use $k = 4$, $\sigma_i = \sigma = 0.55$ and $a_i = a = 1/k$ for $i \in \{1,...,k\}$ and $\mu_{1:4}$ is the unknown parameter, as in [16]. The prior distribution on $\mu_{1:4}$ is a four-dimensional uniform. The data $D_{1:n}$ are simulated using $\mu_{1:4} = (-3,0,3,6)$. Due to the i.i.d. assumption, the likelihood is a product of sub-densities:

$$p(D_{1:n}|\mu_{1:4}) = \prod_{j=1}^{n} p(D_j|\mu_{1:4},\sigma_{1:4},a_{1:3}) \qquad (5)$$

If $p(\mu_{1:4})$ is the prior, the posterior density of $\mu_{1:4}$ (which is the target density and admits 24 modes) is given by:

$$p(\mu_{1:4}|D_{1:n}) = p(D_{1:n}|\mu_{1:4})p(\mu_{1:4}) \qquad (6)$$

The main parameters of PT and the parameters of the mixture model target distribution are shown in Table 1.

3 Related Work

CPU-based work on accelerating PT has focused on lifting the inter-processor communication bottleneck caused by chain interactions. In [12], this is done via a decentralized exchange method, implemented on a CPU cluster. The method scales better than centralized code. In [21], a scheme to allocate chains to CPU cores in order to minimize CPU idle time is proposed. In Section 4.1.1 we show how these overheads can be avoided in FPGAs.

Most GPU and FPGA-based works have been limited to exploiting inter-chain parallelism in a straightforward manner. In [16], a GPU implementation of PT achieves one to two orders of magnitude acceleration over a CPU by mapping each chain to a separate thread. In [22], a GPU is used to accelerate DEMCMC, another population-based MCMC method, achieving a 100x speedup over a CPU.

[4] and [3] use small numbers of parallel PT chains in FPGA implementations but the focus of these works is on exploiting the form of the target density to maximize performance. [23] and [24] present FPGA architectures for PT, which exploit custom arithmetic precision and outperform a GPU by up to 1.5x. [25] propose a precision optimization method for FPGAs, applicable to any MCMC algorithm which delivers a 4x speedup over double precision.

In contrast to the above works which focus on specific platforms, we study the suitability of all three platforms for accelerating PT. This is also the first work that jointly examines how each platform's performance scales with the number of chains and the size of the data. Previous works investigate only one or none of the above. In addition, we present, for the first time in MCMC literature, a power and cost efficiency assessment of PT in various platforms.

Section 4.1.3 builds upon [16] by introducing intra-chain parallelism and other enhancements (including optimal GPU kernel configuration) in order to maximize the performance of the GPU implementation. Unlike [16], we also show how speedups scale with data size.

Additionally, this paper focuses on the use of custom precision as a novel means to accelerate MCMC computations. One of the methods presented here to achieve this (MPPT) is based on the idea proposed in [23] but the present paper also: 1) Introduces an enhanced FPGA architecture for MPPT, 2) Applies the MPPT method to the CPU and GPU, 3) Proposes an entirely novel method (WPT) which is also based on custom precision but uses importance weights to correct errors. WPT is mapped to CPUs, GPUs and FPGAs, 4) Includes results on FPGA precision optimization for WPT/MPPT, GPU kernel optimization and the scaling of performance with data size, 5) Presents highly optimized CPU and GPU implementations for all methods, in contrast to the naive implementations of [23],

Custom precision in MCMC has also been employed in [24] and [25]. In [24], custom precision is used throughout the MCMC method, resulting in biased sampling. No formal way is provided to quantify the bias (and choose a precision accordingly) before sampling starts. [25] follow the same approach but supplement it with a precision optimization method for FPGAs; pre-runs are used to quantify the bias of all candidate precisions and the minimum precision which satisfies a user-defined bias threshold is chosen. In contrast, the custom precision methods presented here: 1) Are tailored for PT and exploit specific characteristics of the algorithm, 2) Guarantee unbiased sampling (despite the use of reduced precision) by trading MCMC mixing for speedup instead of MCMC accuracy for speedup.

4 Accelerating PT

The PT code in Fig. 1 was first implemented in C++ using double precision arithmetic and without exploiting parallelism. We use this sequential implementation as reference. Section 4.1 describes the baseline accelerators (which use double precision) in each platform. Section 4.2 proposes two custom precision methods to improve PT's efficiency. Section 4.3 maps these methods to the three platforms.

4.1 Baseline accelerators

4.1.1 FPGA

Here, we propose a baseline hardware architecture for PT in double floating point precision. Fig. 2 shows the block diagram of the architecture. There are four computational blocks (Sample Proposal, Probability Evaluation, Accept/Reject and Exchange), three memories (Sample, Probability and Data) and three random number generators.

The implementation is based on extensively pipelining all computational blocks. Pipelining is possible because PT chains perform the same computations on different data during Global updates and exchanges. The system can be thought of as a long pipeline which works iteratively, performing the same steps for each Global update/exchange.

One MCMC iteration (outer loop in Fig. 1) includes the following: The current samples of all parallel chains ($\mathbf{x}_j$ for all $j \in \{1,...,m\}$) are read from Sample memory and forwarded to the Sample Proposal block. The Sample Proposal block proposes candidate samples $\mathbf{y}_j$ by adding Gaussian random numbers to the current samples. The candidates are then moved to the Probability Evaluation block (shown in Fig. 3), which computes the candidate probabilities $p_j(\mathbf{y}_j)$. This is done by calculating n probability sub-densities (n is the number of data), finding their logarithms and summing them to get the total log-density. For big n, the whole computation takes a large number of cycles and becomes the bottleneck of the system. To reduce this bottleneck, we instantiate as many parallel pipelines inside this block as allowed by the chip size. Moreover, since each sub-density requires that a datum be read from the Data memory, the Data memory is designed to provide data to all pipelines in parallel at the same cycle (each memory address stores multiple data). An adder tree performs the reduction. The Accept/Reject block receives the candidate probabilities $p_j(\mathbf{y}_j)$ and reads the previous probabilities of each chain ($p_j(\mathbf{x}_j)$) from Probability memory. It also computes the temperature $T_j = (\frac{m}{m+1-j})^2$. All these values (along with a uniform random number) are used to find the Metropolis ratio and accept or reject each candidate sample.

The above steps comprise the update operation. The updated samples also pass through the Exchange block before they are written back to Sample memory. Unlike

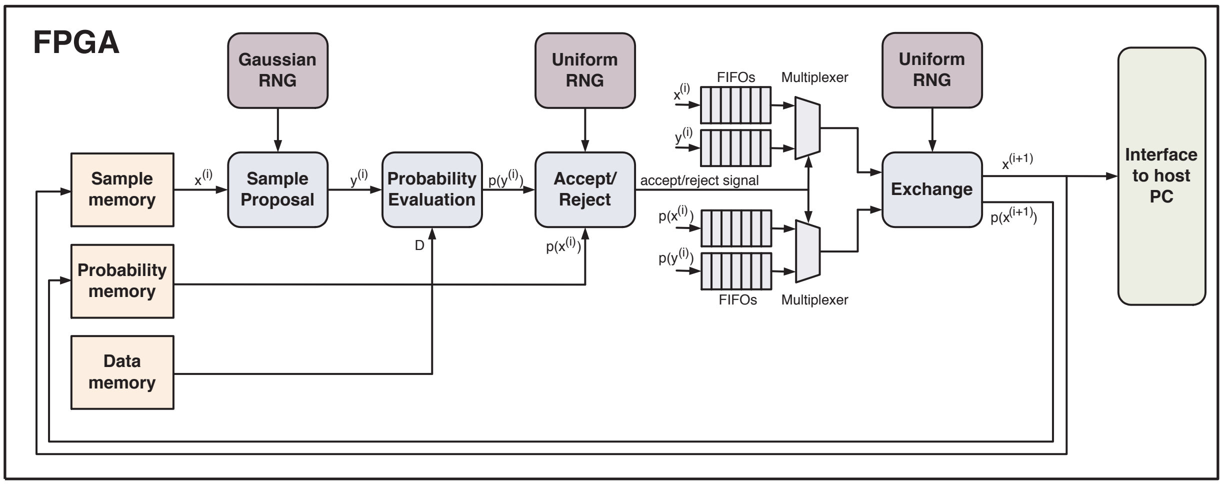

Fig. 2. Baseline FPGA architecture

CPU and GPU implementations (which are presented in the following sections), we do not wait for the Global update to finish before starting the Global exchange. As soon as a chain is updated, it moves to the Exchange block while the next chains are processed by the Update block.

Each exchange is performed between a pair of chains. Therefore, the block has to wait for two successive chains to be updated and then attempt the exchange. Because we use a neighbor exchange scheme, the two potentially exchanged samples conveniently reach the block successively. The temperatures and the updated sample probabilities are used to accept or reject the exchange. Finally, the new samples and probabilities are written back to memories.

The system's performance depends on how fast the Probability Evaluation block processes incoming samples. With P probability sub-density pipelines, the block can process a candidate sample every $\lceil \frac{n}{P} \rceil$ cycles (n is the data size). Fig. 3 demonstrates the utilization of pipelines by PT chains both inside and outside the block when $n = 16$, $P = 4$ and $\lceil \frac{n}{P} \rceil = 4$. A sample reaches the block every four cycles. At the same time, $n = 16$ data are sent to the block (one quadruple per cycle). The Data memory is designed to "match" the consumption rate of the block. The sizes of all system memories are shown in Table 2.

Here, we assume that $n > P$, which is the case for all non-trivial data sets. The throughput of the architecture (MCMC iterations it processes per second) is:

$$TP_{baseline} = \frac{f_{clk}}{m\lceil \frac{n}{P} \rceil} \quad (MCMC\ iterations\ /sec) \tag{7}$$

where f_{clk} is the clock rate of the device in Hz and m is the number of chains. This throughput is equal to the throughput of the Probability Evaluation block. Each MCMC iteration results in one MCMC sample from the first chain, which is the output of the algorithm. (7) assumes that there are enough chains to keep the pipeline busy at all times. When m is small (i.e. $m\lceil \frac{n}{P} \rceil$ is less than the latency of the full system pipeline), some pipeline stages remain empty, leading to performance degradation (see Section 5).

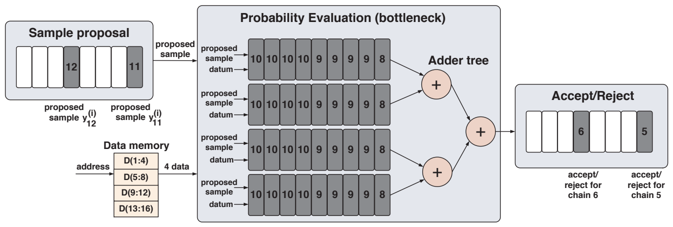
Sample proposal
proposed sample
proposed sample $y^{(i)}_{12}$
proposed sample $y^{(i)}_{11}$
Data memory
address
D(1:4)
D(5:8)
D(9:12)
D(13:16)
4 data
Probability Evaluation (bottleneck)
proposed sample datum
Adder tree
Accept/Reject
accept/reject for chain 6
accept/reject for chain 5

Fig. 3. Chain streaming through the Sample Proposal, Probability Evaluation and Update pipelines. Occupied stages are grey, unoccupied white. Numbers represent the chains that occupy each stage.

TABLE 2
Baseline architecture memories.

Memory	Description	Depth	Width (bits)
Sample memory	Stores current samples of all PT chains	m	$64s$
Probability memory	Stores probabilities of current samples of all chains	m	64
Data memory	Stores the data set used for inference	$\lceil \frac{n}{P} \rceil$	$64P$

4.1.2 Multi-core CPU

In order to exploit PT's parallelism on a multi-core CPU, we embed pragmas and Intel Cilk keywords [26] in the sequential C++ code. We also make use of all Intel Compiler optimizations [27] (including the ones related to the CPU architecture). More specifically: 1) We transform the Global update loop into a **cilk_for** loop and optimize the granularity of the parallelisation by ordering the compiler to group loop iterations into groups of a certain number of iterations each. Depending on the number and type of CPU cores and the amount of work per iteration, a specific granularity maximizes performance. 2) We use **simd reduction** to parallelize the reduction operation (necessary to sum the sub-densities and evaluate the total probability density). A parameter is also used here to specify the granularity of parallelization [26]. 3) We vectorize sub-density evaluations.

4.1.3 GPU

Our GPU design is based on the state of the art CUDA code of [16]. In [16] the main computational work of the implementation is split into two kernels, the global update and the global exchange kernels. There are also kernels for random number generation and initialization. All of the remaining work is done on the CPU. The global update kernel updates all chains once. It exploits chain parallelism, assigning the work of every PT chain to a separate thread. The exchange kernel performs exchanges between neighboring chains and these are also parallelized.

Here, we present an PT implementation which uses an enhanced global update kernel (which is the kernel that takes most of the runtime - more than 90% in all scenarios) and keeps all of the remaining components the same (for more details on these components see [16]). The changes we introduce to the global update kernel to increase thread utilization and maximize performance are the following:

1) We parallelise intra-chain computations by assigning the calculation of the sub-densities (or groups of them) of each chain to separate threads, in contrast to [16] where all sub-densities of a chain were assigned to the same thread. This makes the comparison to other platforms fair.

2) The global update kernel processes m chains, each of which contains n sub-density evaluations. These mn tasks can be allocated to CUDA blocks and threads in many combinations. We range the number of blocks from 1 to m and within each block we allocate 1 to n tasks to each thread. We do not use combinations which allocate the work of a chain into separate blocks. For each (m,n)setting, we choose the combination of blocks and tasks per thread which maximizes the kernel's throughput. For example, when we use few PT chains there is not enough parallelism in the inter-chain level. It is then beneficial to assign each sub-density (task) of each chain to a separate thread to introduce as much intra-chain parallelism as possible. On the other hand, when the number of chains reaches a few thousands, it is preferable to assign more tasks to each thread, because 1) there is now enough inter-chain parallelism to saturate the device, 2) inter-chain parallelism does not require a reduction, unlike intra-chain parallelism. The above optimization is described in detail in Section 5.5 and leads to increased GPU utilization compared to [16] (also considering that we use much larger data sets than [16]). We have not attempted to further distribute computations by invoking multiple CUDA kernels simultaneously because device utilization is high even with one kernel (demonstrated in Section 5) and also because this functionality is not supported by the GPGPU-Sim tool.

2) We unroll reduction operations inside the density computation of each chain. After the independent sub-densities are computed by the threads, we need to sum them to get the likelihood. This requires communication between threads through the shared memory. Although this reduction can be easily done using a reduction tree, this forces half of the threads to be inactive in the first tree stage, 75% of the threads to be inactive in the second stage, etc. In our implementation, we apply a technique which completely unrolls the reduction calculations and minimizes thread imbalance. This technique is proposed in [28] and is reported 21x faster than the tree approach.

3) The implementation of [16] stores all the data in the GPU's constant memory (typically limited to a few dozens of KBs). This is possible because the data sizes used are small (100 data). Here, we store data in global GPU memory, which is a realistic strategy given the data sizes in real applications. During execution, we move the data, in chunks, to the shared memory of all blocks. All the chains of a block can use the data to compute part of the log-density before the next chunk is read, increasing the compute-to-memory ratio of the kernel by a factor equal to the number of chains per block (ranging from 2 to 32).

4.2 Custom precision methods for PT

4.2.1 Custom arithmetic precision in MCMC

The baseline accelerators use double precision throughout the system. This is the case in the overwhelming majority of implementations in the literature, since this precision is considered enough for real problems. Here we use reduced precision in certain parts of PT to lower the area utilization of arithmetic operators in FPGAs (and thus do more operators in parallel) and achieve lower runtimes for computations in CPUs and GPUs. Although reduced precision results in larger arithmetic errors in calculations, we show that these errors can be avoided or corrected in a PT setting. We customize precision by changing the number of mantissa bits only. Precision configurations are described by the pair $(mantissa\ bits,\ exponent\ bits)$.

In both custom-precision methods we present, we reduce precision only when we evaluate the probability density. All other parts of PT work in double precision. The reason is twofold: 1) Altering the precision of all MCMC steps can result in unexpected behavior (non-convergence or convergence to an unknown distribution) [29]. In contrast, by using custom precision only when computing $p(\mathbf{x})$, we preserve the properties of MCMC, i.e. we converge to an altered but known distribution (a custom-precision approximation of $p(\mathbf{x})$). Here, we show that even this approximation can be avoided using "smart" custom precision schemes. 2) The probability evaluation is the bottleneck computation for all accelerators.

4.2.2 Weighted PT method

When the probability evaluation is implemented in custom precision, PT samples from an approximation of $p(\mathbf{x})$. We call the approximate density $\tilde{p}(\mathbf{x})$. When samples from $\tilde{p}(\mathbf{x})$ are used to estimate (1), the estimate will be biased.

To avoid this bias, we propose a novel method called Weighted PT (WPT). The main idea of WPT is to use custom precision in the density evaluation of all chains and assign importance weights to the samples generated by the first chain to correct the first chain's bias. This transforms the algorithm to an Importance Sampling (IS) method [5].

More specifically, the density of chain j is $p_j(\mathbf{x}) = p(\mathbf{x})^{1/T_j}$. For each chain, we compute $p(\mathbf{x})$ in custom precision and then apply the temperature T_j and do all other operations in double precision. Weights are assigned to the samples of the first chain based on the fact that the desired target density is $p_1(\mathbf{x})$ ($= p(\mathbf{x})$), while samples are actually taken from $\tilde{p}_1(\mathbf{x})$ ($=\tilde{p}(\mathbf{x})$), the custom-precision version of $p_1(\mathbf{x})$. Thus $\tilde{p}_1(\mathbf{x})$ functions as the importance sampling distribution in IS; each sample $\mathbf{x}_1^{(i)}$ from the first chain at time i, where $i \in \{1,...,N\}$, is assigned the weight:

$$w_i = \frac{p_1(\mathbf{x}_1^{(i)})}{\tilde{p}_1(\mathbf{x}_1^{(i)})} \tag{8}$$

Integral (1) is then estimated by the IS sum:

$$\tilde{E}_{p-IS}[f(\mathbf{x})] = \frac{1}{N}\sum_{i=1}^{N} w_i f(\mathbf{x}_1^{(i)}) \tag{9}$$

The PT steps remain the same as in Fig. 1, with the only differences being the use of $\tilde{p}(\mathbf{x})$ instead of $p(\mathbf{x})$ everywhere and the computation of the weights.

Generally, the density $\tilde{p}_1(\mathbf{x})$ is a close approximation to $p_1(\mathbf{x})$ and therefore it can be considered a good (efficient) importance sampling distribution [5]. This efficiency drops only for very low precisions (see Section 5). IS also requires that the importance sampling distribution has a wider support than the target distribution [5]. To guarantee this, we saturate the computation of $\tilde{p}_1(\mathbf{x})$, i.e. we transform zero values to the minimum value representable by the employed precision configuration. This guarantees that the support of $\tilde{p}_1(\mathbf{x})$ is larger than that of $p_1(\mathbf{x})$. Once meeting this requirement, the samples and weights generated by WPT can be used to estimate (1) with zero bias compared to the double precision (baseline) sampler.

4.2.3 Mixed-precision PT method

PT has the distinctive property of running many Markov chains but keeping samples from the first chain only. Therefore, only the first chain's target distribution affects the output estimate (2). Auxiliary chains only contribute to the acceleration of the first chain's mixing. This means that they can sample from any distribution with the only possible effect being a change in the mixing speed.

We propose a method called Mixed-Precision PT (MPPT), which exploits this fact to increase performance. It uses double precision only when updating the first PT chain, which means that the first chain samples from the "correct" distribution $p_1(\mathbf{x}) = p(\mathbf{x})$. The auxiliary chains (indexes $j \in \{2,...,m\}$) sample from tempered versions of the custom precision approximation $\tilde{p}(\mathbf{x})$:

$$\tilde{p}_j(\mathbf{x}) = \tilde{p}(\mathbf{x})^{1/T_j} \tag{10}$$

In order to guarantee the correctness of the first chain's target distribution, we also change the way the algorithm exchanges samples. Exchanges between chains $q > 1$ and $r > 1$ at MCMC iteration i are accepted with probability:

$$e(\mathbf{x}_q^{(i)}, \mathbf{x}_r^{(i)}) = min\left(1, \frac{\tilde{p}_q(\mathbf{x}_r^{(i)})\tilde{p}_r(\mathbf{x}_q^{(i)})}{\tilde{p}_q(\mathbf{x}_q^{(i)})\tilde{p}_r(\mathbf{x}_r^{(i)})}\right) \tag{11}$$

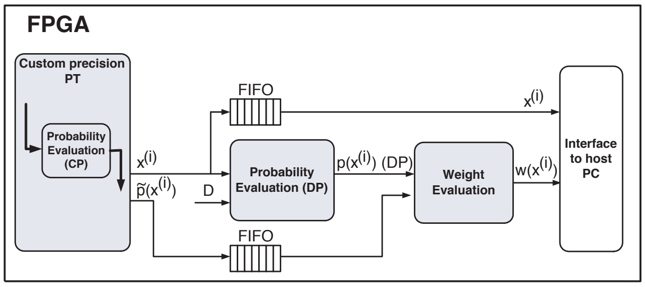

Fig. 4. FPGA architecture for WPT: DP stands for double precision, CP stands for custom precision

Exchanges that include the first (coldest) chain and chain r (here, r is always 2) are accepted with probability:

$$e(\mathbf{x}_1^{(i)}, \mathbf{x}_r^{(i)}) = min\left(1, \frac{p_1(\mathbf{x}_r^{(i)})\tilde{p}_r(\mathbf{x}_1^{(i)})}{p_1(\mathbf{x}_1^{(i)})\tilde{p}_r(\mathbf{x}_r^{(i)})}\right) \tag{12}$$

Notice the use of both double and custom precision densities in (12). The above equations replace the one in line 13 of Fig. 1. By altering the exchange operations which include the chain 1, we guarantee that the detailed balance condition [5] holds for all exchange moves (see Appendix for a proof). This is necessary to guarantee that the distribution of chain 1 remains the same as in double precision samplers. Thus MPPT induces no loss in sampling accuracy.

In [30], Gaussian process approximations of the distributions of some PT chains were used, while retaining the target distribution of the first chain. MPPT uses a custom precision-based approximation rather than a probabilistic approximation. Moreover, in contrast to our work, [30] do not investigate the effect of the approximation on mixing.

The price for not using double precision for auxiliary chains is that mixing can be negatively affected, since sample exchanges between the first and second chains are sometimes less likely to succeed. The trade-off between precision and mixing is explored in Section 5.5.

4.3 Custom precision accelerators

4.3.1 FPGA

WPT architecture: The FPGA architecture for WPT is shown in Fig. 4, comprising: 1) A custom precision PT system (which is the system of Fig. 2 with the Probability Evaluation block in custom precision), 2) A block which computes the probabilities of all samples of the first chain in double precision ($p_1(\mathbf{x}_1^{(i)})$ for all i) , 3) A Weight Evaluation block, which uses the probabilities in custom and double precision to find the weights in (8).

The double precision Probability Evaluation block processes samples from the first chain only. The custom precision Probability Evaluation block processes samples from all m chains. We use a dual port Data memory with one port assigned to each block to feed the blocks with at different rates.

The accelerator's throughput depends on the number of custom precision pipelines in the Probability Evaluation block (inside Custom precision PT), again symbolized by

P, and is given again by (7). Compared to the baseline architecture, the use of reduced precision leads to a net gain in P (and subsequently in throughput), despite WPT's resource overhead due to the extra blocks of Fig. 4.

MPPT architecture: Fig. 5 illustrates the FPGA architecture for MPPT. There are two Probability Evaluation blocks. The double precision block is responsible for: 1) Computing $p_1(\mathbf{x}_1^{(i)})$ for all first chain updates, 2) Computing $p_1(\mathbf{x}_r^{(i)})$ (where $r = 2$) in (12) for all exchanges between chains 1 and 2. The custom precision block is responsible for: 1) Computing $p_j(\mathbf{x}_j^{(i)})$ for $j \in \{2,...,m\}$ (auxiliary chains), 2) Computing $\tilde{p}_r(\mathbf{x}_1)$ (where $r = 2$) in (12) for exchanges between chains 1 and 2. Fig. 5 provides a simplified view of the pipelines inside the two blocks. An extra Sample Proposal block is needed to feed the double precision block. A dual port Data memory is used as in WPT. As for previous architectures, throughput is given by (7). The throughput increases with smaller precisions because we can use larger P (as in WPT). Although mixing decreases with lower precisions, the MPPT accelerator achieves a net gain in performance compared to the baseline accelerator (see Section 5).

4.3.2 Multi-core CPU

It is straightforward to apply the two custom precision methods to the CPU. CPUs have inherent support only for single and double precision floating point arithmetic, so our only choice for the precision of the weight computation in WPT and the auxiliary chains in MPPT is single precision.

4.3.3 GPU

Applying the two custom precision methods to the GPU requires the use of extra kernels compared to the baseline implementation of Section 4.1.3. Only single precision can be used as reduced precision. For WPT, the global update kernel runs exclusively in single precision and, after it terminates, a second kernel is invoked to evaluate the density of the first chain in double precision. The weight is computed in software. For MPPT, the global update comprises two kernel invocations, which correspond to the custom and double precision pipelines of the FPGA architecture. The first kernel (single precision) computes 1) $p_j(\mathbf{x}_j^{(i)})$ for $j \in \{2,...,m\}$ for all auxiliary chain updates and 2) $\tilde{p}_r(\mathbf{x}_1)$ (where $r = 2$) in (12) for the exchange between chains 1 and 2. The second kernel (double precision) computes 1) $p_1(\mathbf{x}_1^{(i)})$ for the first chain update and 2) $p_1(\mathbf{x}_r^{(i)})$ (where $r = 2$) in (12) for all the exchange between chains 1 and 2. The second kernel only processes two density evaluations and therefore underutilizes the GPU, which becomes inefficient for small numbers of PT chains (see Section 5.4). In both methods, the global exchange kernel runs in double precision.

5 Investigation and Results

5.1 Evaluation platforms

We evaluate the performance of the samplers using the following devices: For the multi-core CPU, we use a pair of Intel Xeon E5-2660 v2 processors with 10 cores each (placed on separate sockets). We use 16 GBs of RAM and Intel's C++ compiler version 2015.1.

For the GPU platform, we use Nvidia's GTX285 and GTX480 devices. Actual runs were performed only for GTX480 (hosted by an Intel Core 2 Q9550 CPU with 8 GBs of RAM). The GTX285 measurements came from the GPGPU-Sim simulator [31] (version 3.2.2). This simulator can estimate the runtime of a CUDA kernel (an accuracy of 97-98% is reported in [31]). CUDA 1.3 (for GTX285) and 2.0 (for GTX480) were used for compilation.

For the FPGA platform, we present results for Xilinx Virtex 6 (LX240T) and Virtex 7 (VX1140T) devices. Actual runs were performed only for LX240T (placed on an ML605 board and hosted by an Intel Core i7-2600 CPU with 4 GBs of RAM). The FPGA was clocked at 200 Mhz and communicated with the host PC through a PCI-Express bus, using the RIFFA framework [32]. Performance estimates for VX1140T come from combining post-place and route resource utilization and (7). The Xilinx Power Estimator was used to get power results.

The sequential reference implementation on C++ ran on the multi-core CPU device with one CPU core activated and with all compiler and Cilk optimizations deactivated.

5.2 Performance metrics

The performance criterion to compare PT accelerators is the mixing (exploration) achieved by the accelerator per second of runtime, given a fixed selection of PT tuning parameters (number of chains, temperatures) and a particular target distribution (implying a fixed data set size). This criterion depends on two factors: 1) How many MCMC samples can be generated per second, i.e. the raw throughput, 2) How fast the MCMC samples explore the distribution, i.e. how much "exploration" is achieved with a given amount of samples. Here, since the PT algorithm and all its tuning parameters are fixed, the second factor depends only on precision (precision affects exploration, Sections 4.2.2-3).

For double precision (baseline) accelerators, only the first factor affects performance. It is enough to find the Raw Speedup ($Speedup_{raw}$) of each accelerator against the reference sequential CPU sampler to make comparisons, (speedup refers to samples/sec). Nevertheless, for custom precision accelerators, the second factor has to be taken into account too. Specifically, the performance of WPT is affected by: 1) Different first chain mixing speed compared to the baseline sampler (due to custom precision), i.e. the difference in mixing when sampling from $\tilde{p}_1$ instead of p_1. 2) The importance sampling mechanism; the more the importance distribution $\tilde{p}_1$ deviates from the IS target distribution p_1, the more the importance weights take extreme values. This makes the IS estimator in (9) less efficient. The sampling efficiency of MPPT is affected by the use of custom precision for the auxiliary chains ($\tilde{p}_{2:m}$), which changes the mixing speed of the first chain (compared to the baseline sampler). Due to these reasons, one MCMC sample from WPT or MPPT has a different "exploration value" compared to one sample from baseline samplers.

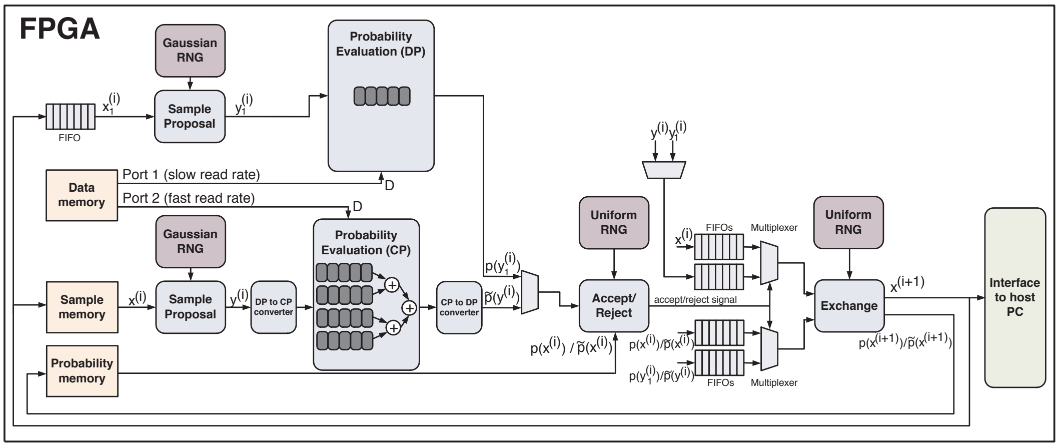

Fig. 5. FPGA architecture for MPPT: The double precision (DP) Probability Evaluation block has one pipeline and the custom precision (CP) Probability Evaluation block has multiple pipelines.

To capture the second factor in comparisons, we use two metrics: 1) The Effective Sample Size due to MCMC autocorrelation (ESS_{mcmc}) [7], a typical metric of mixing speed in MCMC. ESS_{mcmc} gives an estimate of how many independent (or "effective") samples the dependent MCMC samples are equivalent to. ESS_{mcmc} can be estimated using the MCMC samples' autocorrelations [7]. In both WPT and MPPT, ESS_{mcmc} quantifies the effect that custom precision (in chains 1 to m or 2 to m respectively) has on the mixing of the first chain. 2) The Effective Sample Size due to importance sampling (ESS_{is}) [5], which gives an estimate of how many independent samples from the IS target distribution the samples from the IS importance distribution are equivalent to (when used to find (9)). This is a standard metric in IS literature and it can be estimated using the weights of the samples [5]. In WPT, ESS_{is} quantifies the loss in efficiency due to the use of IS weights.

Combining these metrics with Raw speedup, we propose a performance criterion, the Effective Speedup:

$$Speedup_{eff} = Speedup_{raw} \cdot \frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}} \cdot \frac{ESS_{is}^{(CP)}}{ESS_{is}^{(DP)}} \qquad (13)$$

Here, the ratio $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}}$ represents the loss/gain in the mixing efficiency of the chain 1 when going from double (DP) to custom (CP) precision, i.e. the relative ESS_{mcmc}. The ratio $\frac{ESS_{is}^{(CP)}}{ESS_{is}^{(DP)}}$ represents the loss/gain in efficiency due to IS when going from DP to CP, i.e. the relative ESS_{is}. For the baseline samplers $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}} = 1$ and $\frac{ESS_{is}^{(CP)}}{ESS_{is}^{(DP)}} = 1$ (since we do not use custom precision). Therefore, $Speedup_{eff} = Speedup_{raw}$. For WPT, $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}} \neq 1$ and $\frac{ESS_{is}^{(CP)}}{ESS_{is}^{(DP)}} \neq 1$. For MPPT, $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}} \neq 1$ and $\frac{ESS_{is}^{(CP)}}{ESS_{is}^{(DP)}} = 1$.

$Speedup_{eff}$ measures the performance gain (in effective samples/sec) of an implementation compared to the reference implementation (for identical tuning parameters and target distribution). For WPT and MPPT, $Speedup_{eff}$ changes with precision. In the FPGA case, precision can be optimized to maximize $Speedup_{eff}$ (see Section 5.5). In the CPU and GPU case only the single precision configuration can be used so no optimization takes place, although $Speedup_{eff}$ is still affected by the change in precision.

TABLE 3
Resource utilization of the various processing blocks

Block name	LUTs	FFs	DSPs
Sample proposal	6688	5286	48
Accept/reject	9123	7422	43
Exchange	6277	5123	62
Other functions, control and I/O	16965	12592	0
WPT overhead	26329	29012	221
MPPT overhead	29549	33087	230
Probability evaluation - 1 pipeline (DP)	25581	28408	218
Probability evaluation - 1 pipeline (24,11)	8267	8146	58
Probability evaluation - 1 pipeline (16,11)	7098	7355	38
Virtex-6 LX240T total resources	150720	301440	768

5.3 FPGA resource utilization

For FPGA architectures, $Speedup_{raw}$ (and $Speedup_{eff}$) depends on the number of pipelines in the Probability Evaluation block (P), since bigger P leads to higher throughput in (7). All FPGA implementations presented here use the maximum number of pipelines that can fit in the targeted device (given the device's LUTs, registers and DSPs). In all cases, DSP blocks are the limiting resource. Table 3 shows the post place and route resource utilization of the various blocks of the FPGA architectures when mapped to the LX240T device. The total resources of LX240T are also shown. The overheads of MPPT and WPT comprise one double precision probability evaluation pipeline and the extra modules described in Section 4.3.1. MPPT needs slightly more resources than WPT due to the extra (slower) Sample Proposal block. The table also shows the resources needed for a single pipeline in different precisions.

5.4 Performance evaluation

Here, we vary the number of chains and data and use $Speedup_{eff}$ to compare accelerators. We target the distribution of Section 2.4. We also compare with the sampler of [16], where each thread is assigned one PT chain without exploiting intra-chain parallelism. Although investigating performance scaling with data size is specific to the i.i.d. data assumption, this does not restrict the applicability of the inter-chain parallelization techniques and custom precision methods we propose. The precision optimization described in Section 5.5 has been applied to all custom precision FPGA implementations of this section. Also, each GPU sampler uses kernels with the optimal combination of CUDA blocks and data per thread (given m and n), as will be demonstrated in Section 5.5. Finally, CPU implementations use the optimal granularities for chain and reduction parallelization.

Scaling the number of chains: In Fig. 6, we set the number of data to $n = 128$ and vary the number of chains (m). The mean $Speedup_{eff}$ from 30 independent runs is shown for all samplers. Error bars represent one standard deviation. The deviations are small compared to the mean values in all cases, showing that the speedups are not due to random behaviour in the runs. The baseline multi-core CPU (with 20 cores) achieves a peak speedup of 13.8x. This is not reached with fewer than 2048 chains due to inter-core communication overhead during exchanges. The WPT and MPPT CPU samplers reach a speedup of 18.4x. This improvement is due to the use of single precision for the majority of density evaluations.

The baseline GPUs achieve significantly higher peak $Speedup_{eff}$. To reach peak performance (165x for GTX480, 78x for GTX285), $m = 32768$ chains are used. For m close to a few hundred, speedups are in the range 15x-50x. This slow scaling is due to lack of enough parallelism to fully utilize the GPU. These speedups are up to 4.4x higher than those of the state-of-the-art implementation of [16]. [16] achieves the same performance as our implementation only for $m = 32768$. WPT and MPPT samplers achieve similar $Speedup_{eff}$ in the GPU. They outperform baseline implementations by up to 3.2x (GTX285) and 2.4x (GTX480). For fewer than 512 chains, baseline samplers are faster than WPT/MPPT by up to 1.3x. This is due to the fact that two update kernels (in double and single precision) are called in WPT/MPPT (Section 4.3.3). The kernels' runtime is stable or increases slightly when they process less than 100-200 chains (due to GPU under-utilization). This means that, e.g. calling a double precision kernel to process only the first chain of MPPT takes almost the same time as the baseline (double precision) kernel takes to process 32 or 128 chains. On top of that, MPPT involves the cost of calling a second (single precision) kernel for the auxiliary chains. This results in lower performance than the baseline GPU until enough chains are used, at which point the benefits from processing the auxiliary chains in single precision outweighs the cost of running two kernels. For all GPU samplers, the main bottleneck resource which limits the peak speedup is the number of registers in each streaming processor. When we manually doubled the number of registers per processor in the GPGPU-Sim model of GTX285, the peak $Speedup_{eff}$

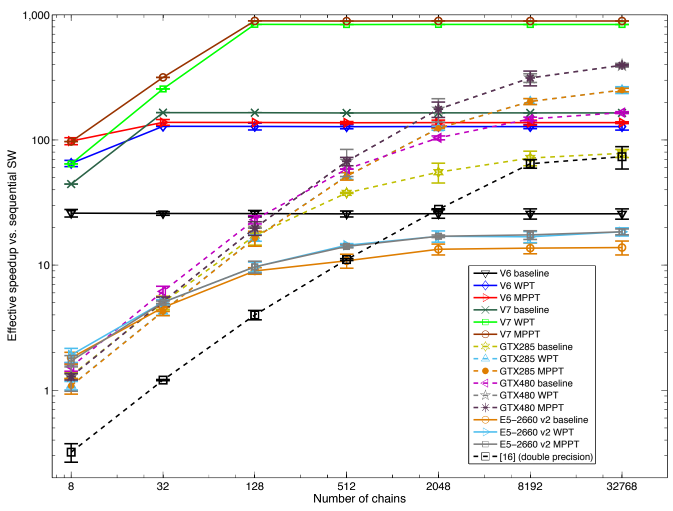

Fig. 6. Scaling of $Speedup_{eff}$ with number of chains m. Baseline and custom precision accelerators are included. $n = 128$ were used. Error bars represent standard deviation from 30 runs. V7 measurements are projections and have no error bars.

of the baseline sampler increased by 1.7x. Increasing the processing capabilities of the processors (e.g. by adding more floating point units) had minimal effect on peak performance.

On the FPGA platform, the baseline architecture on the LX240T and VX1140T is 26x and 44x-165x faster than the reference respectively. WPT increases these speedups to 65x-128x and 66x-854x respectively and MPPT to 98x-138x and 76x-997x respectively. These speedups are due to increased P, resulting from reduced precisions. Peak performance is reached with only 8-32 chains for the baseline and 32-128 chains for the custom precision architectures. These speedups come without compromising sampling quality.

Comparing across platforms, it is clear that the big Virtex 7 FPGA with the baseline sampler is 24x-36x and 0.9x-28x faster than the baseline CPU and GPU (on the GTX480) implementations respectively. These speedups increase to 57x-92x and 2.2x-76x when WPT/MPPT are used in all platforms. This shows that the two custom precision methods are more suitable for mapping on the FPGA because they can exploit the fully custom precision of the platform, while CPUs and GPUs can only use single or double precision. Similarly, the small Virtex 6 FPGA is slower than the small GTX285 GPU for $m > 128$ when the baseline sampler is used but only for $m > 2048$ when the WPT/MPPT methods are used. Nevertheless, GTX286's peak performance is higher than LX240T's by 1.8x-3.1x (depending on the method). All FPGA architectures reach their peak performance much earlier than GPUs, since their flexibility allows them to utilize resources efficiently and exploit modest amounts of parallelism. For $m < 512$, the Virtex 6 FPGA is faster than both GPUs when using MPPT/WPT. For $m < 128$, even the baseline Virtex 6 is faster than all GPU samplers. In PT literature, the values of m typically range from less than 10 to a few dozens

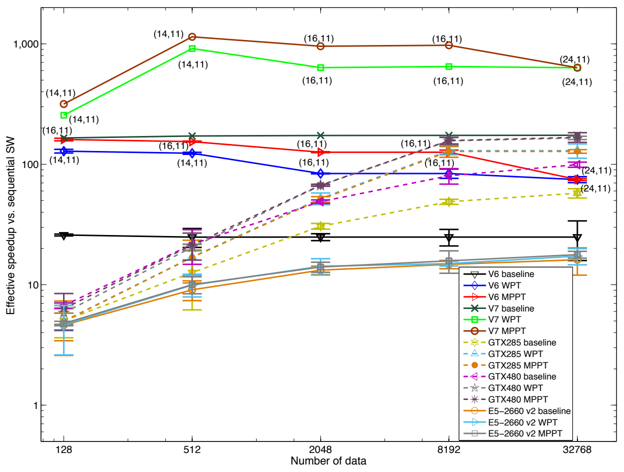

Fig. 7. Scaling of $Speedup_{eff}$ with number of data n. Baseline and custom precision accelerators are presented. $m = 32$ chains are used. Error bars represent standard deviation from 30 runs. V7 measurements are projections and have no error bars.

(and rarely 100-200). Thousands of chains do not improve mixing (e.g. see [16]). This makes the advantage of FPGAs at small m significant. Note also that for $m < 32$ CPU samplers are up to 1.8x faster than GPU samplers due to limited amounts of parallelism. Finally, MPPT outperforms WPT by up to 10% in FPGAs. This is due to the fact that WPT's efficiency is affected by both ESS_{mcmc} and ESS_{is} in (13), the latter contributing to performance deterioration.

Scaling the number of data: Fig. 7 shows the mean $Speedup_{eff}$ from 30 independent runs (along with standard deviations) when the number of chains is set to a realistic $m = 32$ and the number of data (n) varies. Next to each point corresponding to WPT and MPPT on FPGAs, the selected custom precision is shown (see Section 5.5).

The peak $Speedup_{eff}$ of the baseline CPU is 16.1x, which is slightly higher than in Fig. 6. The speedups of the WPT and MPPT methods on the CPU are only marginally higher than the baseline sampler. $Speedup_{eff}$ for the baseline GTX285 and GTX480 ranges from 5x to 99x. These are lower than Fig. 6 for large m because the GPUs can exploit inter-chain parallelism more efficiently than intra-chain parallelism (intra-chain reductions involve thread communication). Using $m = 32$ does not provide enough inter-chain parallelism to fully exploit all GPU resources. The WPT and MPPT methods offer an extra speedup of up to 2.2x over the baseline but are slightly slower for $n = 128$ (in the GTX480 case). FPGA samplers reach peak performances similar to Fig 6 (174x-1143x for VX1140T and 25x-160x for LX240T), showing that the FPGA exploits intra- and inter-chain parallelism equally well. The main reason is the efficient implementation of reductions; the adder tree which receives the values coming out of the sub-density pipelines is designed to match the number of the pipelines and does not waste resources.

The figure also reveals the scaling disadvantage of the custom precision methods on FPGAs. When n increases, more sub-densities are summed during density evaluation, leading to more arithmetic error and rougher density approximations. This negatively affects the mixing/efficiency of WPT/MPPT (Section 5.5). Higher precisions need to be used in the custom precision part to compensate for this efficiency decline. This reduces $Speedup_{eff}$, since pipelines cost more FPGA resources and thus fewer pipelines are instantiated. For example, MPPT on VX1140T for $n = 32768$ needs 24 mantissa bits to achieve optimal $Speedup_{eff} = 633x$. This is smaller compared to the $Speedup_{eff} = 1143x$ achieved for $n = 512$ (requiring 14 mantissa bits). This problem does not appear in CPUs and GPUs, since only single precision is used. Due to the above issue, WPT/MPPT on GTX280 outperforms WPT/MPPT on LX240T for $n \geq 8192$. WPT/MPPT on VX1140T is still faster than WPT/MPPT on GTX480 by 3.7x-54x.

Power and cost efficiency: Apart from $Speedup_{eff}$, power efficiency is also important for many applications, especially in HPC (where power consumption is key). Also, since it is impossible to find devices with equal size across platforms (in part due to lack of data from FPGA vendors), power efficiency is a fairer way to compare across platforms, since performance can be normalized (e.g. per unit of energy). Performance per dollar is another useful metric, although purchase costs are much smaller than energy costs in HPC. Table 4 compares accelerators in two (m,n) settings, using three metrics: Effective samples per Joule (i.e. Performance per Watt), Effective Samples per Joule·sec and Effective Samples per \$·sec. FPGA accelerators can generate up to 193x and 362x more Effective samples per Joule than CPUs and GPUs respectively for small m and n (in these cases CPU/GPU resources remain largely under-utilized). These numbers reduce to 135x and 19x for large m and n. The FPGA's performance per Joule is up to 10210x and 27647x higher than the CPU's and GPU's respectively for small m and n and up to 8889x and 69x for large m and n. The above numbers reveal that the FPGA is able to extract significantly more performance per unit of energy. The GPU's performance per dollar is generally the highest (up to 120x and 18x higher than the CPU's and the FPGA's), except for the case of small m and n. This is due to the high cost of the CPU and FPGA devices.

Enabled parallelism vs. theoretical model: Table 5 shows how much parallelism each baseline implementation is able to "extract" from the algorithm, compared to the theoretical maximum given by Amdahl's law for the case $(m,n) = (32768,128)$. The theoretical maximum is 8703x because the non-parallelizable parts of the algorithm take 0.0115% of the runtime in the sequential, reference CPU implementation. The GPU and FPGA enable similar amounts of parallelism but when these are normalized over power the FPGA is 4.7x more efficient. The enabled parallelism can increase if larger devices are used.

5.5 Precision optimization for FPGAs and kernel optimization for GPUs

Precision optimiazation for FPGAs: The $Speedup_{eff}$ of custom precision FPGA samplers cam be maximized by

TABLE 4
Power efficiency of the proposed accelerators

	Accelerator (device)	ES/J	$ES/(J\cdot sec)$	$ES/(\$\cdot sec)$
(m,n)=(8,128)	Baseline (E5-2660v2)	1.5	$4.5\cdot10^{-1}$	$1.0\cdot10^{-1}$
	MPPT (E5-2660v2)	1.5	$4.6\cdot10^{-1}$	$1.0\cdot10^{-1}$
	Baseline (GTX480)	$9.7\cdot10^{-1}$	$2.4\cdot10^{-1}$	$4.8\cdot10^{-1}$
	MPPT (GTX480)	$8.0\cdot10^{-1}$	$1.7\cdot10^{-1}$	$4.0\cdot10^{-1}$
	Baseline (VX1140T)	$1.3\cdot10^{2}$	$9.1\cdot10^{2}$	$4.2\cdot10^{-1}$
	MPPT (VX1140T)	$\mathbf{2.9\cdot10^{2}}$	$\mathbf{4.7\cdot10^{3}}$	$\mathbf{9.3\cdot10^{-1}}$
(m,n)=(32768,128)	Baseline (E5-2660v2)	$2.8\cdot10^{-3}$	$1.6\cdot10^{-6}$	$1.9\cdot10^{-4}$
	MPPT (E5-2660v2)	$3.7\cdot10^{-3}$	$2.7\cdot10^{-6}$	$2.5\cdot10^{-4}$
	Baseline (GTX480)	$2.5\cdot10^{-2}$	$1.7\cdot10^{-4}$	$1.2\cdot10^{-3}$
	MPPT (GTX480)	$6.1\cdot10^{-2}$	$9.9\cdot10^{-4}$	$\mathbf{3.0\cdot10^{-2}}$
	Baseline (VX1140T)	$1.2\cdot10^{-1}$	$8.2\cdot10^{-4}$	$3.9\cdot10^{-4}$
	MPPT (VX1140T)	$\mathbf{6.6\cdot10^{-1}}$	$\mathbf{2.4\cdot10^{-2}}$	$2.1\cdot10^{-3}$
(m,n)=(32,32768)	Baseline (E5-2660v2)	$1.3\cdot10^{-2}$	$3.7\cdot10^{-5}$	$9.3\cdot10^{-4}$
	MPPT (E5-2660v2)	$1.4\cdot10^{-2}$	$4.2\cdot10^{-5}$	$9.9\cdot10^{-4}$
	Baseline (GTX480)	$6.3\cdot10^{-2}$	$1.1\cdot10^{-3}$	$3.2\cdot10^{-2}$
	MPPT (GTX480)	$1.0\cdot10^{-1}$	$2.9\cdot10^{-3}$	$\mathbf{5.3\cdot10^{-2}}$
	Baseline (VX1140T)	$5.3\cdot10^{-1}$	$1.5\cdot10^{-2}$	$1.7\cdot10^{-3}$
	MPPT (VX1140T)	**1.9**	$\mathbf{2.0\cdot10^{-1}}$	$6.2\cdot10^{-3}$

TABLE 5
Enabled parallelism for $(m,n) = (32768,128)$

Device	E5-2660v2	GTX480	VX1140T
% of maximum th. parallelism	$1.6\cdot10^{-1}$	1.90	1.89
% of maximum th. parallelism / Watt	$8.4\cdot10^{-4}$	$7.6\cdot10^{-3}$	$3.6\cdot10^{-2}$

optimizing their precision configuration, i.e. the number of mantissa bits. This optimization has to be done for each combination of (m,n) and FPGA device. It comprises pre-runs in all candidate precisions. From these pre-runs we collect samples and/or weights and use them to evaluate the ESS ratios in 13 and then get $Speedup_{eff}$. The two ESS metrics are easy to compute [5], [7]. Here, we demonstrate the optimization process for one parameter combination $(m,n) = (128,128)$ when targeting the LX240T FPGA.

For WPT, Fig. 8 shows how the three terms in the right-hand side of (13) ($Speedup_{raw}$, $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}}$ and $\frac{ESS_{is}^{(CP)}}{ESS_{is}^{(DP)}}$) change with precision. The mixing-related $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}}$ remains close to one as we move to fewer mantissa bits because $p_1(\mathbf{x})$ and $\tilde{p}_1(\mathbf{x})$ are very similar. For 6 mantissa bits the ratio starts growing, which means that WPT mixes faster than the baseline sampler. This is because $\tilde{p}_1(\mathbf{x})$ becomes a coarsely quantized version of $p_1(\mathbf{x})$, which is easy to sample from.

The IS-related $\frac{ESS_{is}^{(CP)}}{ESS_{is}^{(DP)}}$ is also very close to one for most precisions, signifying that no loss of efficiency is caused by the IS scheme. For 10 or fewer mantissa bits, the ratio decreases. This is because parts of the support of $p_1(\mathbf{x})$ take much lower probability density values under $\tilde{p}_1(\mathbf{x})$ or vice versa, especially in the distribution's tails. This results in large weights being generated, making IS inefficient.

$Speedup_{raw}$ increases up to 5.3x when precision drops, due to the reduced area cost per pipeline which allows us to increase P and thus the throughput in (7).

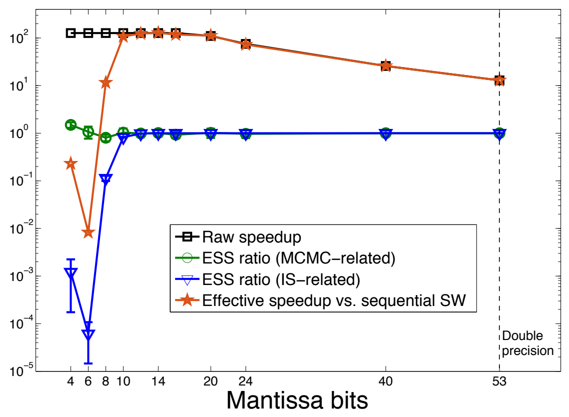

Fig. 8. WPT on FPGA: The effect of precision on the factors of (13). $(m,n) = (128,128)$. FPGA: LX240T. The optimal number of mantissa bits is 14. Error bars represent standard deviation from 30 runs.

Multiplying the three above terms gives $Speedup_{eff}$, also shown in Fig. 8. We get the maximum $Speedup_{eff}$=128x for configuration (14,11), which is the optimal precision.

Precision optimization for MPPT is similar but without the IS-related effect. The auxiliary chains' precision affects the mixing of chain 1, i.e. the term $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}}$ in (13). Fig. 9 shows that $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}}$ varies but remains close to one as precision drops until we reach 8 bits. This stability is due to the fact that exchange moves between chains 1↔2 (which help chain 1 escape from local modes) can succeed even if the density of chain 2 is calculated in low precision. Although samples of the second (and all auxiliary) chains are not "correctly" distributed around the mode centers, all modes still exist and chains traverse from mode to mode due to tempering. This suffices to occasionally supply the chain 1 with a samples that help it escape from a mode. This behaviour is confirmed by the constant percentage of successful exchange moves as precision drops. When very low precisions, we observe shifts in the modes positions, leading $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}}$ to collapse. The maximum $Speedup_{eff}$=138x is achieved by configuration (14,11).

In the examined case, the optimized $Speedup_{eff}$ of both WPT and MPPT is 4.5x larger than the $Speedup_{eff}$ of the baseline FPGA sampler with no cost in sampling quality.

For CPUs and GPUs, WPT and MPPT do not require precision optimization because the only available reduced precision is single precision. Of course, single precision has some effect on the ESS ratios of (13). This has been incorporated into all Section 5 results.

Kernel optimization for GPUs: As mentioned in Section 4.1.3, the combination of blocks and tasks (sub-densities) per thread in the CUDA kernels of GPU samplers can be optimized to minimize runtime. This optimization is shown in Fig. 10 for $(m,n) = (8192,128)$ on the GTX285. Values come from pre-runs. Some combinations are impossible because they lead more threads per block than CUDA

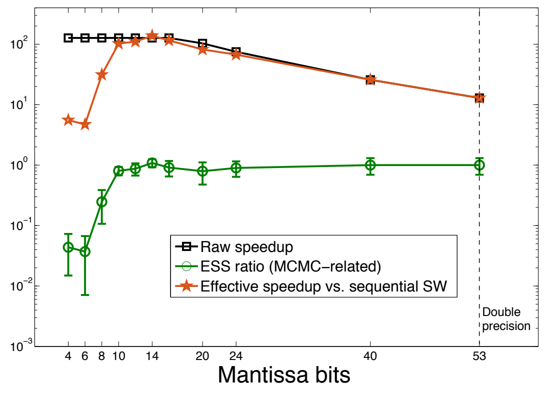

Fig. 9. MPPT on FPGA: The effect of precision on the factors of (13). $(m,n) = (128,128)$. FPGA: LX240T. The optimal number of mantissa bits is 14. Error bars represent standard deviation from 30 runs.

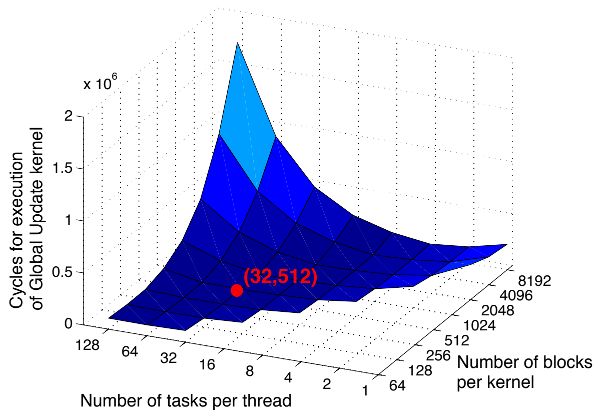

Fig. 10. GPU kernel optimization: Cycles of global update kernel when the number of blocks and the number of tasks (data) per thread change. $(m,n) = (8192,128)$. GPU: GTX285. Cycles are minimized with 512 blocks and 32 tasks per thread.

allows. It is clear from the graph that assigning sufficient tasks per block is necessary to achieve good performance and that processing several tasks per thread is beneficial. This optimization has been applied to all the GPU samplers.

6 Discussion

Given the limits on how much MCMC's efficiency can be improved and the loss in accuracy incurred by approximate methods, the use of parallel hardware is necessary when faced with difficult problems in MCMC-based inference.

Multi-core CPUs are by far the easiest to program when used by practitioners, but their limited parallelism per chip means that it cannot tackle demanding analyses (e.g. PT). CPUs should be preferred for MCMC methods which are non-parallel and/or dominated by conditional operations, e.g. the Gibbs and Reversible-Jump samplers [7].

GPUs are a good match for population-based MCMC, since the Global update is a Single-Instruction-Multiple-Data (SIMD) computation. Unfortunately, most of the GPU's resources remain unused for the realistic scenario of using up to 100-200 chains, unless the likelihood is also SIMD. GPUs provide massive off-chip memory bandwidth (up to 300 GB/sec), which can be handy for big-data likelihoods, provided that data access is non-random.

FPGAs, although hard to program, deliver high speedups for population-based MCMC. They are not limited to exploiting embarrassingly inter-chain computations and reach their peak performance with few chains. Off-chip memory bandwidth in FPGAs is typically limited to under 50 GB/sec. This can be a disadvantage for memory-bound likelihood computations. Nevertheless, FPGAs enjoy massive on-chip memory bandwidth (20-40 TB/sec [33]) due to large amounts of built-in memory. GPU on-chip memory bandwidths are limited to 8 TB/sec and 1.5 TB/sec [33]. If data is kept inside the device or a data reuse scheme can be devised (e.g. [33]), this benefits the FPGA.

Moreover, it is clear from the results that custom precision can benefit population-based MCMC without compromising sampling quality, provided that it is used in accordance with the algorithm's properties. Both custom precision methods proposed here do this. They are particularly suitable for mapping on FPGAs, where they deliver up to a 6.5x speedup over the double precision sampler. The gains on CPUs and GPUs are smaller. The main disadvantage of WPT/MPPT is the way their performance scales with data size. The strengths and weaknesses of the two methods when mapped on FPGAs are:

WPT: This method is easier to implement; the main processing block is the same as in the baseline architecture (with reduced precision) and the extra blocks are not complex. WPT is slightly slower than MPPT in most situations. Also, since WPT is an IS method, the distribution is not actually sampled and cannot be visualized.

MPPT: This method is faster than WPT by around 10% in most cases and generates actual samples from the distribution. However, it is more complex to implement, since the main block has to be modified significantly to handle the use of two different precisions in updates.

WPT and MPPT can be applied to other sampling scenarios. WPT can be used in any MCMC method in combination with thinning, a technique which retains only some of the samples to decrease correlation. A custom precision block could process all samples, while a double precision block could compute weights for thinned samples (smaller workload). Both WPT and MPPT can be applied to other population-based MCMC methods [14], [22].

The paper focuses on optimally mapping PT to hardware and exploiting precision. The way performance changes when using different likelihoods is outside the scope of this work. Having said that, the speedups demonstrated for the mixture model are expected to hold for other models with i.i.d. data, since the same form of parallelism can be exploited. Performance scaling is also expected to be similar to Figures 6 and 7 when using i.i.d. likelihoods (which is the case in a wide variety of real problems, see Section 2.3). For non-i.i.d. likelihoods, probability evaluation might require computations which are not SIMD, e.g. [9], [13], so speedups (and the way they scale) depend largely on the problem. Nevertheless, the trend shown in Figure 6 (FPGAs reach peak performance earlier than GPUs) will still hold as long as chains can be streamed through the likelihood datapath. Moreover, the CPU/GPU/FPGA design is largely unaffected by the target distribution; only the module/kernel which computes the probability density needs to change. The rest of the system is generic.

In summary, multi-core CPUs are preferable for specific MCMC variants and GPUs are suitable for population-based and other parallel MCMC methods, especially if the density computation is SIMD. The FPGA should be preferred for SIMD and non-SIMD MCMC methods which have communication-bound operations and/or are robust to precision reduction and when energy consumption is critical. For problems which are extremely intensive computationally, a possible solution is the use of multiple GPUs and FPGAs and the exploitation of heterogeneous computing techniques to optimally allocate tasks to devices.

REFERENCES

[1] R. Salakhutdinov, A. Mnih, and G. Hinton, "Restricted Boltzmann Machines for Collaborative Filtering," in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07. New York, NY, USA: ACM, 2007, pp. 791–798.

[2] A. U. Asuncion, P. Smyth, and M. Welling, "Asynchronous distributed estimation of topic models for document analysis," *Statistical Methodology*, vol. 8, no. 1, pp. 3 – 17, 2011.

[3] N. B. Asadi, T. H. Meng, and W. H. Wong, "Reconfigurable computing for learning Bayesian networks," in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2008, pp. 203–211.

[4] F. Belletti et al, "Janus: An FPGA-Based System for High-Performance Scientific Computing," *Computing in Science Engineering*, vol. 11, no. 1, pp. 48–58, 2009.

[5] J. S. Liu, *MC strategies in scientific computing*. Springer, 2001.

[6] J. Yan, M. Cowles, S. Wang, and M. Armstrong, "Parallelizing MCMC for Bayesian spatiotemporal geostatistical models," *Statistics and Computing*, vol. 17, no. 4, pp. 323–335, 2007.

[7] S. Brooks, A. Gelman, G. L. Jones, and X.-L. Meng, *Handbook of Markov Chain Monte Carlo*, 1st ed. Chapman and Hall/CRC, 2011.

[8] L. Bottolo et al., "GUESS-ing Polygenic Associations with Multiple Phenotypes Using a GPU-Based Evolutionary Stochastic Search Algorithm," *PLoS Genet*, vol. 9, no. 8, p. e1003657, 2013.

[9] S. Zierke and J. Bakos, "FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods," *BMC Bioinformatics*, vol. 11, no. 1, pp. 1–12, 2010.

[10] J. Gross, W. Janke, and M. Bachmann, "Massively parallelized replica-exchange simulations of polymers on GPUs," *Computer Physics Communications*, vol. 182, no. 8, pp. 1638–1644, 2011.

[11] D. J. Earl and M. W. Deem, "Parallel tempering: Theory, applications, and new perspectives," *Phys. Chem. Chem. Phys.*, vol. 7, pp. 3910–3916, 2005.

[12] Y. Li, M. Mascagni, and A. Gorin, "A decentralized parallel implementation for parallel tempering algorithm," *Parallel Comput.*, vol. 35, pp. 269–283, May 2009.

[13] L. Bottolo and S. Richardson, "Evolutionary stochastic search for Bayesian model exploration," *Bayesian Analysis*, vol. 5, no. 3, pp. 583–618, 09 2010.

[14] A. Jasra, D. A. Stephens, and C. C. Holmes, "On population-based simulation for static inference." *Statistics and Computing*, pp. 263–279, 2007.

[15] G. McLachlan and D. Peel, *Finite mixture models*. Wiley, 2004.

[16] 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.

[17] C. J. Geyer, "Markov Chain Monte Carlo Maximum Likelihood," in *Computing Science and Statistics, Proceedings of the 23rd Symposium on the Interface*, 1991, pp. 156–163.

[18] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," *The Journal of Chemical Physics*, vol. 21, no. 6, 1953.

[19] M. Suchard, Q. Wang, C. Chan, J. Frelinger, A. Cron, and M. West, "Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures," *Journal of Computational and Graphical Statistics*, vol. 19, no. 2, p. 419438, 2010.

[20] D. Maclaurin and R. P. Adams, "Firefly Monte Carlo: Exact MCMC with Subsets of Data," *ArXiv e-prints*, Mar. 2014.

[21] D. J. Earl and M. W. Deem, "Optimal Allocation of Replicas to Processors in Parallel Tempering Simulations," *The Journal of Physical Chemistry B*, vol. 108, no. 21, pp. 6844–6849, 2004.

[22] W. Zhu, A. Yaseen, and Y. Li, "DEMCMC-GPU: An Efficient Multi-Objective Optimization Method with GPU Acceleration on the Fermi Architecture," *New Generation Computing*, vol. 29, no. 2, pp. 163–184, 2011.

[23] G. Mingas and C. Bouganis, "A Custom Precision Based Architecture for Accelerating Parallel Tempering MCMC on FPGAs without Introducing Sampling Error," in *FCCM, 2012 IEEE 20th Annual International Symposium on*, 2012, pp. 153–156.

[24] G. Mingas and C.-S. Bouganis, "Parallel tempering mcmc acceleration using reconfigurable hardware," in *ARC*, ser. Lecture Notes in Computer Science. Springer, 2012, vol. 7199, pp. 227–238.

[25] G. Mingas, F. Rahman, and C.-S. Bouganis, "On Optimizing the Arithmetic Precision of MCMC Algorithms," *FCCM, Annual IEEE Symposium on*, vol. 0, pp. 181–188, 2013.

[26] Intel, "Intel Cilk Plus," http://software.intel.com/, 2011.

[27] ——, "Intel C and C++ Compilers," http://software.intel.com, 2011.

[28] M. Harris *et al.*, "Optimizing parallel reduction in CUDA," *NVIDIA Developer Technology*, vol. 2, p. 45, 2007.

[29] L. Breyer, G. O. Roberts, and J. S. Rosenthal, "A note on geometric ergodicity and floating-point roundoff error," *Statistics & Probability Letters*, vol. 53, no. 2, pp. 123–127, June 2001.

[30] M. Fielding, D. J. Nott, and S.-Y. Liong, "Efficient MCMC Schemes for Computationally Expensive Posterior Distributions," *Technometrics*, vol. 53, no. 1, pp. 16–28, 2011.

[31] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, 2009, pp. 163–174.

[32] M. Jacobsen, Y. Freund, and R. Kastner, "RIFFA: A Reusable Integration Framework for FPGA Accelerators," in *FCCM, 2012 IEEE 20th Annual International Symposium on*, 2012, pp. 216–219.

[33] A. Rafique, N. Kapre, and G. Constantinides, "Enhancing performance of Tall-Skinny QR factorization using FPGAs," in *FPL, 2012 22nd International Conference on*, Aug 2012, pp. 443–450.

PLACE PHOTO HERE

Grigorios Mingas is a PhD candidate and Research Assistant in the Department of Electrical and Electronic Engineering of Imperial College London, UK. He received the M.Eng. degree in Electrical and Computer Engineering in 2010 from the Aristotle University of Thessaloniki, Greece. His main research interests are reconfigurable and high-performance computing for MCMC/SMC, linear algebra and SLAM acceleration.

PLACE PHOTO HERE

Christos-Savvas Bouganis is a Senior Lecturer in the Department of Electrical and Electronic Engineering of Imperial College London, UK. He is an editorial board member of IET Computers and Digital Techniques and Journal of Systems Architecture. His research interests include the theory and practice of reconfigurable computing and design automation, mainly targeting digital signal processing algorithms.