Imperial College of Science, Technology and Medicine
Department of Electrical and Electronic Engineering

Acceleration of MCMC-based Algorithms Using Reconfigurable Logic

Shuanglong Liu

Supervised by Christos-Savvas Bouganis

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering of Imperial College London
and the Diploma of Imperial College London, April 2017

Abstract

Monte Carlo (MC) methods such as Markov chain Monte Carlo (MCMC) and sequential Monte Carlo (SMC) have emerged as popular tools to sample from high dimensional probability distributions. Because these algorithms can draw samples effectively from arbitrary distributions in Bayesian inference problems, they have been widely used in a range of statistical applications. However, they are often too time consuming due to the prohibitive costly likelihood evaluations, thus they cannot be practically applied to complex models with large-scale datasets. Currently, the lack of sufficiently fast MCMC methods limits their applicability in many modern applications such as genetics and machine learning, and this situation is bound to get worse given the increasing adoption of big data in many fields. The objective of this dissertation is to develop, design and build efficient hardware architectures for MCMC-based algorithms on Field Programmable Gate Arrays (FPGAs), and thereby bring them closer to practical applications.

The contributions of this work include: 1) Novel parallel FPGA architectures of the state-of-the-art resampling algorithms for SMC methods. The proposed architectures allow for parallel implementations and thus improve the processing speed. 2) A novel mixed precision MCMC algorithm, along with a tailored FPGA architecture. The proposed design allows for more parallelism and achieves low latency for a given set of hardware resources, while still guaranteeing unbiased estimates. 3) A new variant of subsampling MCMC method based on unequal probability sampling, along with a highly optimized FPGA architecture. The proposed method significantly reduces off-chip memory access and achieves high accuracy in estimates for a given time budget. This work has resulted in the development of hardware accelerators of MCMC and SMC for very large-scale Bayesian tasks by applying the above techniques. Notable speed improvements compared to the respective state-of-the-art CPU and GPU implementations have been achieved in this work.

Acknowledgements

I would like to thank my supervisor, Dr. Christos-Savvas Bouganis, without whom the thesis would not be completed. He has always been there supporting me over these years, and giving me valuable advice in exploring different aspects of the problem in the work. The most thing I admire and appreciate is that he always encouraged me to do high quality research and contribute to the real-world problems. All of these experience will be of great help for my future research.

I would like also to thank Grigorios Mingas, who helped me a lot in the first year of my research and gave me many suggestions to both my hardware design and my writing for publications. All the people in CAS, who helped me improve my presentation, provided valuable feedback of my work.

I also thank to my family for their support in my life. My grandparents who raised me up and taught me the good things. My parents who saved money to send me to colleges and supported me for every decision I made. My brother who always gave me advices when I met problems in my life and research. Also all my friends in both China and UK, for bringing the happiness to my life.

Finally I would like to thank the financial support of this research: the President's PhD Scholarship, and it is gratefully acknowledged.

Declaration of Originality

I herewith certify that the work presented in this thesis is my own. All material in the thesis which is not my own work has been properly referenced.

Copyright Declaration

Contents

List of Tables

List of Figures

List of Acronyms

CA-MCMC ... communication-aware Markov chain Monte Carlo algorithm

CF-MCMC .... custom-precision firefly Markov chain Monte Carlo algorithm

CLK ..... clock

CMC ..... Consensus Monte Carlo

CP-MCMC .... custom-precision Markov chain Monte Carlo algorithm

CPU ..... central processing unit

CUMSUM ..... cumulative sum

DP-MCMC .... double-precision Markov chain Monte Carlo algorithm

ESS ..... effective sample size

FlyMC ..... firefly Monte Carlo algorithm

FPGA ..... field programmable gate array

GPF ..... Gaussian particle filtering

GPU ..... graphics processing unit

HT ..... Horvitz-Thompson

i.i.d. ..... independent and identically distributed

IMHA ..... Independent Metropolis Hastings Algorithm

IS ..... importance sampling

ISR ..... improved systematic resampling

M-H ..... Metropolis-Hastings algorithm

MC ..... Monte Carlo

MCMC ..... Markov chain Monte Carlo

PF ..... particle filter

PPS ..... Probability Proportional-to-Size

RMSE ..... root-mean-square error

RNG ..... random number generator

RPG ..... random permutation generator

RS-MCMC .... random sampling based Markov chain Monte Carlo algorithm

RSR ..... residual systematic resampling

SI ..... simple random sampling

SIS ..... sequential importance sampling

SMC ..... sequential Monte Carlo

SP ..... stream multi-processor

SR ..... systematic resampling

SUM ..... sum algorithm

Chapter 1

Introduction

1.1 Motivation

Bayesian Inference has become increasingly popular in modern machine learning such as Bayesian networks, due to its ability to represent uncertainty in parameter estimates and analyse data of complex structures using flexible models [3, 4]. The computations associated with most common Bayesian tasks, e.g. estimation, prediction and model comparison, boil down to integrations. In some situations, it is possible to perform such integrations exactly either by simple Riemann integration or splines [71]. Unfortunately, most real-world problems are rarely amenable to these exact inference especially when modelling large data sets, and numerical integration is limited by large dimensions [77]. Therefore most of the interest in Bayesian methods have been focused on better methods of approximate inference in the form of Monte Carlo estimates or variational approximations [3]. The Monte Carlo methods tackle the problem of integrations as expectations, and thus estimate the intractable integrals through sampling. Therefore the key problem in approximate Bayesian inference is the sampling from any arbitrary probability distribution.

In most applications, generating independent random samples from the target probability distribution in the Bayesian model is not feasible, due to its high-dimension and multi-modality. In practice, it is often the case that either the generated samples have to be dependent (e.g. from carefully designed Markov chains), or the samples are generated from some standard distributions such as Normal dis-

tribution and then the samples are weighted by the target distribution (e.g. Importance Sampling and Rejection Sampling). In other words, these two techniques of generating random samples are essential to Monte Carlo integration procedure.

Markov chain Monte Carlo (MCMC) is a class of methods that generate dependent samples by evolving a Markov chain designed to have a stationary distribution as the target distribution. MCMC sampling has been the main tool used to draw samples in Bayesian inference problems since 1990s, because of its ability to sample from posterior distributions in Bayesian modelling regardless of dimension or complexity. However, MCMC methods are often far too computationally intensive to be of any practical use [67], and their runtimes can easily reach weeks or months [1, 16, 60]. The mainstream of current research on MCMC methods has been focused on proposing variants of algorithms that scale to large datasets to reduce their runtimes.

Sequential Monte Carlo (SMC) methods are based on the framework of Sequential Importance Sampling (SIS), in which one builds up the trial sampling distribution sequentially and computes the importance weights recursively [53]. SMC methods have been particularly popular to generate samples from posterior distributions in dynamic systems, where a sequence of target distributions with increasing dimension happens. They are applicable to a very large class of models especially the non-linear problems where the interest is in tracking and/or detection of dynamic signals [25]. However, SMC methods may be slower than that Bayesian applications require in some real-time problems due to large number of samples which are necessary to guarantee the accuracy in the estimation of the states of the systems. This situation would be changed by parallel computing, since most parts of SMC procedures are ready to be executed in a parallel computing platform, as will be shown later.

1.2 Challenges and Contributions

Although SMC methods are powerful and effective in the application of non-linear and/or non-Gaussian state space models, they are often too computationally intensive in their application to complex models. The most commonly used SMC algorithms consist of three basic steps: generation of new samples, computation of sample weights and resampling. Sample generation and weight

computation are the most computationally intensive steps, but they are straightforward for parallel implementation in order to increase the speed. Resampling is not computationally intensive, but it requires a collective operation (such as the sum or cumulative sum) among the generated samples. Thus it affects the speed of the whole algorithm. When these algorithms are implemented in parallel computing platforms, the resampling becomes a bottleneck due to the necessity for exchanging a large number of samples through the processing elements [13]. The main challenges for speed increase of the SMC algorithms in parallel computing devices include exploiting parallel resampling algorithms and architectures.

MCMC methods are time-consuming mainly due to the likelihood computations as they necessarily need to access all of the data at each iteration in order to estimate how well the data are explained by the sampled parameters. This makes MCMC prohibitively slow to converge, especially when the distribution is high-dimensional and multi-modal. Assuming independent and identically distributed (*i.i.d.*) data, the likelihood function complexity is $O(N)$, where N is the size of the data set. The evaluation of the likelihood functions has become the dominant computational bottleneck when large datasets are targeted. Therefore, accelerating the likelihood computations is the most crucial task in order to allow the application of MCMC to complex models with large-scale data sets.

Besides the likelihood computations over the whole dataset, memory issue has also become a bottleneck in the acceleration of MCMC algorithms, as the dataset to make inference has been largely increased and growing fast. The data transfer between the processing elements and external memories can limit the system's performance if the memory bandwidth is not enough to constantly feed the processing elements. Thus it imposes a big challenge on the applicability of MCMC in many modern applications, as MCMC needs to access the data memory across iterations and the latency imposed by the data transfers between memory and processing elements limits the overall performance.

Due to the increasing speed demands of MCMC and SMC applications, a number of previous works have proposed the use of multi-core computational devices such as Central Processing Units (CPUs) and Graphics Processing Units (GPUs) to accelerate these algorithms by parallel computation. In this work, the focus is on the acceleration of the above methods using the reconfigurable device: Field-Programmable Gate Arrays (FPGAs). FPGA technology has shown to be a promising candidate for

accelerating many algorithms due to its highly-parallel bit-oriented architecture [66]. By appropriately decomposing the problem into different blocks that can be executed in parallel and mapping them into an FPGA, a considerable amount of acceleration can be achieved compared to CPU implementation [12]. Another important advantage of FPGAs is their flexibility to operate in any custom arithmetic precision format. Instead of implementing operators in double floating-point precision, which is the default approach in MCMC applications, reduced arithmetic precision can be used, making operators utilising fewer resources and allowing for more parallelism for a given device. Besides, the fully customizable architecture in FPGAs can largely take advantage of the characteristics of the specific MCMC-based algorithms, in order to further improve the speed.

This thesis aims to speed up the SMC and MCMC algorithms in FPGAs by proposing new designs of algorithms and novel customized architectures, in order to largely utilize the advantages and characteristics of FPGAs. The runtime of the SMC methods is reduced by parallel implementation of the algorithm with optimized parallel resampling architectures, to fully utilize the computational resources in FPGAs. The MCMC sampling speed is increased by implementing likelihood-related arithmetic operators in custom precision arithmetic in FPGA, but without introducing any bias in the estimates of the integrals. As such, more parallel operators can be instantiated for a given resource budget, and thus improving performance. The thesis also investigates how to adapt the data subsampling based MCMC algorithms to be implemented in FPGAs for applications with large datasets, in order to reduce the external memory access latency. This is achieved by proposing unequal probability sampling to select the subset based on the data contribution to the estimated likelihood. The results presented in the thesis have shown that significant speedups can be achieved with the proposed FPGA implementations against the respective state-of-the-art implementations in CPUs and GPUs.

This thesis contains a number of contributions on the development of the hardware accelerators of the MCMC-based algorithms. They are summarized in the following list (the detailed discussions of the contributions are given in each Chapter):

- The introduction of novel parallel architectures which map four resampling algorithms to an FPGA, taking advantages of the inherent parallelism of SMC algorithms. It is the first work that presents parallel FPGA architectures for the state-of-the-art resampling algorithms, and it

showed significant speedups compared to that of CPU and GPU implementations.

- A custom-precision firefly MCMC algorithm which guarantees unbiased sampling under custom precision arithmetic, leading to significant performance gains; An optimised FPGA-based architecture of this algorithm is proposed, which capitalises on the nature of FPGA devices to support custom arithmetic precision.

- A novel methodology for the construction of tight lower bound functions of the target probability distribution function based on the selection of the rounding mode of the FPGA arithmetic operators in combination with verification tools for modelling numerical behaviour (i.e. Gappa++), in order to maximise the performance of the proposed custom-precision MCMC algorithm.

- A methodology for selecting the custom arithmetic precision of the custom-precision MCMC system that would maximise its performance based on the system's performance model and the estimates of the parameters from pre-runs.

- A communication-aware MCMC algorithm based on unequal probability sampling, that takes into account the performance characteristics of the underlying memory hierarchy. The proposed algorithm reduces the data transfer overheads among memories, compared to the regular MCMC and other subsampling-based algorithms, leading to faster execution times and higher accuracy in the estimates for a given time budget.

- An optimized hardware architecture tailored for FPGA implementation that maps the communication aware MCMC algorithm on FPGA and efficiently utilises high bandwidth on-chip memory blocks on FPGAs.

1.3 Organization of the Thesis

This thesis is organized in six chapters. In this chapter, the motivation and contributions of this work are presented.

In Chapter 2, a brief background on the theory of Bayesian Inference is described. It contains the basic principles of MCMC and SMC methods to follow the remaining chapters. Besides, the main challenges and recent developments on both algorithms focusing on how to improve the performance and increase the speed of these methods are described and concluded in details. Then we summary that how we tackle the speed problem of these two algorithms, especially for big data applications. Finally, a separate section is devoted to the related work where a complete literature review is provided, together with the previous works on GPUs and FPGAs which accelerated the MCMC-based algorithms.

Chapter 3 investigates ways to accelerate the SMC methods by proposing novel parallel resampling algorithms and architectures. An optimized version of Systematic Resampling (SR) is proposed, while other three state-of-the-art resampling algorithms are presented. The parallel architectures for each algorithm are proposed to be ready to implement in FPGAs. The speedups of the four architectures are compared to the respective GPU implementations and discussions on the results are provided.

An unbiased MCMC FPGA-based accelerator under custom precision regimes is proposed in Chapter 4. This novel mixed precision MCMC algorithm simulates from the exact target distribution in contrast to existing approximate MCMC samplers, while the large majority of likelihood computations are performed in reduced precision. Two Bayesian logistic regression case studies of varying complexity are used to evaluate the performance of the proposed hardware architecture. The results show significant speedups compared to existing FPGA- and CPU-based works that utilise double floating point arithmetic, without any bias on the sampling-based estimates.

In Chapter 5, we propose a communication-aware MCMC framework that takes into account the underlying performance of the memory subsystem during the sampling process. The framework is based on a novel subsampling algorithm that utilises an unbiased likelihood estimator based on Probability Proportional-to-Size (PPS) sampling, allowing information on the performance of the memory system to be taken into account during the sampling stage. The proposed system in FPGA addresses exactly the memory-bound problem in the MCMC construction, opening the way for applying the MCMC algorithm to large scale datasets.

Finally, the current state of our work and potential future works are summarized in Chapter 6.

1.4 Published Work

This thesis is based on the following previously published work:

- S. Liu, G. Mingas and C.-S. Bouganis. "Parallel Resampling for Particle Filters on FPGAs". *IEEE International Conference on Field Programmable Technology (FPT), December, 2014.*
- S. Liu, G. Mingas and C.-S. Bouganis. "An Exact MCMC Accelerator Under Custom Precision Regimes". *IEEE International Conference on Field Programmable Technology (FPT), December, 2015.*
- S. Liu, G. Mingas and C.-S. Bouganis. "An Unbiased MCMC FPGA-based Accelerator in the Land of Custom Precision Arithmetic". *IEEE Transactions on Computers, 2017.*
- S. Liu and C.-S. Bouganis. "Communication-Aware MCMC Method for Big Data Applications on FPGAs". *IEEE 25st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017.*

Chapter 2

Background Theory and Related Work

2.1 Background

In the era of abundant data, many real-world data analysis problems involve estimating unknown quantities or parameters from some given observations, which needs tools for modelling, searching and understanding large datasets. Examples include air traffic management using radar measurements [80, 18, 43], digital signal extraction [25, 53], machine learning [71, 85, 5, 6], statics physics [53, 11] and genetics [16, 87], just to name a few. In most of these applications, prior knowledge about the unknown parameters being modelled is available [25]. This knowledge allows us to explain the observed dataset by Bayesian models, which infer the posterior probability of the unknown variables given the observed data. Bayesian modelling can represent the uncertainty in the model using the information of prior distributions.

The Bayesian Theorem [30] follows:

$$p(\theta|\mathbf{y}) = \frac{p(\mathbf{y}|\theta)p(\theta)}{p(\mathbf{y})} \tag{2.1}$$

where $\mathbf{y}$ denotes the data of observed values of a system, and θ denotes some unobserved parameter(s) for the model of the system. There are four key quantities in the Bayes formula which have their respective meanings: $p(\theta)$ is the **prior probability** of the parameter θ while $p(\mathbf{y}|\theta)$ is the **likelihood**

function of $\mathbf{y}$; the **posterior probability distribution** of the unknown parameters θ given $\mathbf{y}$ is $p(\theta|\mathbf{y})$ and it contains all of the information combining prior knowledge and observations, and the $p(\mathbf{y})$ is called as **normalized constant** or **marginal likelihood**.

With $p(\theta|\mathbf{y})$ the Bayesian rule can be applied to make model comparison and prediction. However, computing the Marginal Likelihoods:

$$p(\mathbf{y}) = \int p(\mathbf{y}, \theta) d\theta = \int p(\mathbf{y}|\theta) p(\theta) d\theta \tag{2.2}$$

can be computationally intractable for two reasons: 1) it can be a very high dimensional integral; 2) the likelihood term can be complicated. Historically, the need to evaluate integrals was a major stumbling block for the take up of Bayesian methods. Hence, drawing samples to estimate related quantities from a given probability distribution is a fundamental task in Bayesian inference as well as in many other statistical applications [30, 53]. Nevertheless, MCMC is a method to generate random samples from any posterior probability distribution function in Bayesian Inference, which doesn't need to compute the normalizing constant $p(\mathbf{y})$ as we will show later. This is the first reason why Monte Carlo sampling such as MCMC is popular in Bayesian Inference problems.

Using the samples generated from the posterior probability distribution (2.1) in Baysian model, we can perform a series of inference tasks, e.g. estimate the unknown parameters of the model for the given dataset, make predictions of the coming data using the model parameters and compare different models based on the observed dataset. All these quantities or tasks can be computed from the samples by evaluating the following integral:

$$\mathbb{E}_{p(\theta|\mathbf{y})}(f) = \int f(\theta) p(\theta|\mathbf{y}) d\theta \tag{2.3}$$

where $f(\theta)$ is a function which depends on the task of interest. The above integral is thus the expectation of $f(\theta)$ under the Bayesian posterior distribution. For example, to estimate the mean of the parameters we set $f(\theta) = \theta$. To predict the future observations, we set $f(\theta) = p(\mathbf{y}'|\theta)$ where $\mathbf{y}'$ is the coming observations. These integrals can be easily estimated when enough samples from the posterior distribution $p(\theta|\mathbf{y})$ are available (which will be explained in details later), otherwise they

can be intractable in real applications since both the parameters and the data can have high dimensions, making it impossible or impractical to evaluate the integrals analytically or using numerical methods [53]. This is the second reason why we need Monte Carlo sampling (MCMC or SMC) in these problems.

An example application: classifying handwritten digits

Here we use a real application to clearly illustrate how the above Bayesian modelling framework is applied in practice. MNIST database is a set of handwritten digit images containing a training set of 60,000 examples, and a test set of 10,000 examples. Let $\mathbf{x} = \{x_1, ..., x_N\}$ be the training digits of N images and each x can be seen as a vector. Correspondingly, let $\mathbf{y} = \{y_1, ..., y_N\}$ be their classes. The goal is to infer a Bayesian model which can predict the class of the digit in the test dataset or the future digits, taking into account the above training data. Assume that a logistic regression model[1] is used to solve a two-class classification problem such as classifying MNIST 7s and 9s [58]. The probability of its class ($y \in \{-1, 1\}$) for a input data x (one image) given the parameter values (θ) of the logistic model, i.e. $p(y|x, \theta)$ is equal to the logistic regression likelihood:

$$L(\theta) = p(y|x, \theta) = \frac{1}{1 + exp\{-y \cdot \theta^T x\}} \tag{2.4}$$

The probability of the total training data set $\{\mathbf{x}, \mathbf{y}\}$ given the parameters of the model θ is:

$$p(\mathbf{y}|\mathbf{x}, \theta) = \prod_{i=1}^{N} p(y_i|x_i, \theta) = \prod_{i=1}^{N} L_i(\theta) \tag{2.5}$$

The aim of Bayesian inference is to estimate the unknown parameters of the model, in order to predict future data or make model comparison. That is to say, at first we need to compute the posterior probability distribution $p(\theta|\mathbf{y}, \mathbf{x})$ of the unknown parameters θ, given the training data $\{\mathbf{x}, \mathbf{y}\}$:

$$p(\theta|\mathbf{y}, \mathbf{x}) = \frac{p(\mathbf{y}|\mathbf{x}, \theta)p(\theta)}{p(\mathbf{y})} = \frac{p(\theta) \prod_{i=1}^{N} L_i(\theta)}{p(\mathbf{y})} \tag{2.6}$$

[1] There are many alternative models available in Bayesian Inference [3].

Using the above posterior distribution, we then are able to predict its class $\tilde{y}$ of the future data $\tilde{x}$:

$$p(\tilde{y}|\tilde{x}) = \int p(\tilde{y}|\tilde{x}, \theta)p(\theta|\mathbf{y}, \mathbf{x})d\theta \tag{2.7}$$

Note that the prediction in Bayesian inference is not based on a point estimate of the unknown parameters, but rather on the whole posterior probability distribution. Besides, other tasks (e.g. model comparison, finding moments of the posterior) can be performed in a similar way.

The key and necessary task in Bayesian problems is how to compute the above integrals, which is far from straightforward and therefore it requires sampling from the posterior probability distribution to compute the integral using Monte Carlo integration. The following sections in this chapter will focus on how MCMC or SMC algorithms tackle the sampling problem as the basis of Bayesian Inference, in order to estimate these integrals.

2.2 Markov chain Monte Carlo (MCMC)

2.2.1 Basic Principles

MCMC methods can be used to sample from any given probability distribution such as the complex posterior distributions in the Bayesian problems. This is achieved by relaxing the requirement that the samples should be independent. The output of an MCMC algorithm is a sequence of samples from the correctly normalised distribution of the target distribution $p(\theta)$. These samples then can be used to estimate any function of interest in respect of θ, e.g. its mean, variance as mentioned above.

Assuming we need to draw samples from a desired distribution $p(\theta)$, a Markov chain generates a correlated sequence of states: $\theta_0 \rightarrow \theta_1 \rightarrow \theta_2 \rightarrow \theta_3 \rightarrow \theta_4 \rightarrow \theta_5 \cdots$. Each step in the sequence is drawn from a transition operator $T(\theta \rightarrow \theta')$, which gives the probability of moving from state θ to state θ'. According to the Markov property, the transition probabilities depend only on the current state θ. A basic requirement for T is that given a sample from $p(\theta)$, the marginal distribution over the

next state in the chain is also the target distribution of interest p:

$$p(\theta') = \sum_{\theta} T(\theta \to \theta')p(\theta) \text{ for all } \theta'. \tag{2.8}$$

This requires **irreducibility**: the ability to reach any θ where $p(\theta) > 0$ in a finite number of steps, and **aperiodicity**: no states are only accessible at certain regularly spaced times [64].

The samples generated from MCMC are typically used to estimate the expectation of the functions $f(\theta)$ we have mentioned in the above section, with respect to the Bayesian posterior or any probability function $p(\theta)$. We rewrite the equation (2.3) as following:

$$I(f) = \int f(\theta)p(\theta)d\theta \tag{2.9}$$

By collecting N_s samples from MCMC, the above integral $I(f)$ can be approximated by tractable sums that converge (as the number of samples N_s tends to infinity) to $I(f)$. The following central limit theorem holds for suitable test functions f [24]:

$$\tilde{I}(f) = \frac{1}{N_s}\sum_{n=1}^{N_s} f(\theta_n) \longrightarrow \text{Normal}(I(f), \sigma^2_{lim}(f)) \tag{2.10}$$

i.e. the sum is an asymptotically unbiased estimator of the integral $I(f)$ [71].

2.2.2 MCMC Algorithms

The simplest way to construct a Markov chain with stationary distribution $p(\theta)$ is the Metropolis-Hastings algorithm (M-H) as shown in Algorithm 1, where $q(\theta)$ is the proposal distribution.

In each iteration, a proposed move of the chain is considered, by using the proposal distribution (line 2) to generate the new state, i.e. sample θ'. Then the acceptance ratio a (line 3) is computed by evaluating the target probabilities and proposal distributions of the proposed sample and the previous sample. When the Bayesian posterior probability distribution in Equation (2.1) is targeted, the normalizing constant is cancelled out when computing the ratio a. Therefore, only the likelihood

Algorithm 1: Metropolis-Hastings MCMC

Input: initial setting θ_0, number of samples N_s;
Output: parameter samples $\theta_i, i = 1, ..., N_s$;

1: **for** $i = 1$ **to** N_s **do**
2: Propose $\theta' \sim q(\theta'|\theta_{i-1})$.
3: Compute $a = \dfrac{p(\theta')q(\theta_{i-1}|\theta')}{p(\theta_{i-1})q(\theta'|\theta_{i-1})}$;
4: Set $\theta_i = \theta'$ with probability $min(1, a)$, otherwise $\theta_i = \theta_{i-1}$.
5: **end for**

function and prior distribution need to be evaluated in MCMC construction. Finally the proposed sample is accepted or rejected with the probability based on the acceptance ratio (line 4).

When a symmetric proposal distribution which satisfies $q(\theta_i|\theta') = q(\theta'|\theta_i)$, such as the random walk proposal, i.e. Gaussian distribution, is used in M-H, the algorithm is named as Metropolis algorithm, which is shown in Algorithm 2. The benefit of Metropolis algorithm compared to M-H is that the ratio a reduces to that of the target probabilities (line 3).

Algorithm 2: Metropolis MCMC

Input: initial setting θ_0, number of samples N_s;
Output: parameter samples $\theta_i, i = 1, ..., N_s$;

1: **for** $i = 1$ **to** N_s **do**
2: Propose $\theta' \sim \theta_{i-1}$+Normal(0, $\sigma^2 I$);
3: Compute $a = \dfrac{p(\theta')}{p(\theta_{i-1})}$;
4: $u \sim$ Uniform(0,1);
5: **if** $u \leq a$ **then**
6: $\theta_i = \theta'$;
7: **else**
8: $\theta_i = \theta_{i-1}$;
9: **end if**
10: **end for**

Metropolis and M-H algorithms are the most fundamental algorithms in MCMC literature. The choice of the proposal $q(\cdot)$ in M-H is fairly arbitrary, but it should be easy to simulate and evaluate in practice. There is often a trade-off: we would like "large" jumps (updates), so that the chain explores the state space, but large jumps usually have low acceptance probability as the posterior density can be highly peaked. As a rule of thumb, it often sets the spread of $q(\cdot)$ to be as large as possible without leading to very small acceptance rates, say < 0.1 [40]. For example, when taking $q(\theta'|\theta) = \theta + N(0, \sigma^2)$ in

Metropolis MCMC, the step size σ is often chosen to obtain the acceptance rate a around 0.23.

2.2.3 MCMC Output Analysis

MCMC generates samples from the probability distribution $p(\theta)$ by sequentially constructing a Markov chain that satisfies (2.10). In practice it is often advisable to discard some initial states of the chain (throwing away a number of iterations at the beginning of an MCMC run is often called "burn-in"), in order to reduce the initialisation bias. Although MCMC generates statistically consistent samples from the target distribution, the samples are correlated due to the use of a Markov chain. This dependency leads to an increase in asymptotic variance σ^2_{lim} of the MCMC estimate in (2.10), compared to the case where independent samples of the target distribution are used. This loss in efficiency can be quantified by the Effective Sample Size (ESS) [40] in (2.11):

$$ESS = N_s/(1 + 2\sum_{j=1}^{k} \rho(j)) \tag{2.11}$$

where N_s is the number of post burn-in MCMC samples and $\sum_{j=1}^{k} \rho(j)$ is the sum of the first k monotone sample autocorrelations. The ESS estimates the "effective" number of samples, which is always lower than N_s. Thus the adopted performance metric for MCMC samplers is ESS/sec, which combines raw sampling speed (runtime) and ESS [40].

Although the MCMC estimates are asymptotically unbiased, in practice the bias can not be avoided due to running MCMC for a finite number of steps. Besides, the bias is also introduced for some other reasons: 1) subsampling of the data which uses only a fraction of the whole data to provide a faster estimation of the likelihood; 2) reduced precision in the evaluation of the likelihood function. Both approximations aim to increase the speed of MCMC execution, by allowing a small bias in the stationary distribution of the Markov chain. The ESS metric is often used to compare MCMC algorithms that simulate from the exact posterior distributions, and it only considers the variance in the output. However, when bias is introduced in the algorithm due to the need for faster execution, both the bias and variance are fundamental for understanding and comparing the performance of the estimator.

The bias and variance of the estimator $\tilde{I}$ are defined as

$$\text{Bias}[\tilde{I}] = \mathbb{E}[\tilde{I} - I] \tag{2.12}$$

$$\text{Var}[\tilde{I}] = \mathbb{E}[(\tilde{I} - \mathbb{E}[\tilde{I}])^2] \tag{2.13}$$

The metric that is commonly used in literature to compare different MCMC algorithms is the error in the estimate of (2.10). This error can be quantified by the risk in the estimate, which is defined as the mean squared error in the estimate of (2.10), i.e. $R = \mathbb{E}[(\tilde{I} - I)^2]$, where the expectation is taken over multiple simulations of the Markov chain [48]. This risk can be decomposed as the sum of squared bias and variance:

$$\begin{aligned} \mathbb{E}[(\tilde{I} - I)^2] &= \mathbb{E}[(\tilde{I} - \mathbb{E}[\tilde{I}] + \mathbb{E}[\tilde{I}] - I)^2] \\ &= \mathbb{E}[(\tilde{I} - \mathbb{E}[\tilde{I}])^2] + (\mathbb{E}[\tilde{I}] - I)^2 \\ &= \text{Var}[\tilde{I}] + \text{Bias}[\tilde{I}]^2 \end{aligned} \tag{2.14}$$

The objective of MCMC in practice that runs for a finite number of samples is to obtain estimates with lower risk in a given time. The approximate MCMC algorithms often allow a small bias in the estimates. By doing so, a larger number of samples can be collected in the same amount of computational time and therefore reduce the variance in the estimate. The design of high performance MCMC algorithms for big data often comes down to the bias-variance trade-off, and can be studied using the risk of the estimator.

2.3 Sequential Monte Carlo (SMC)

2.3.1 Principles and Algorithms

Sequential Monte Carlo (SMC) is a valid alternative sampling method to MCMC, which is often used to compute the posterior distributions in dynamic models with time-varying parameters, while MCMC methods are used in static (or steady-state) models which are time-invariant. The key idea

of SMC is to generate a set of samples in the full space and then update them with corresponding weights. The basis of SMC is *Importance Sampling* (IS) and *Sequential IS* (SIS), but SMC operates in parallel inherently in comparison to SIS. SMC methods are very flexible, easy to implement, parallelisable and applicable in very general settings [25]. Over the last few years, SMC has developed in many application fields such as simulating macromolecules, statistical missing data problem, nonlinear filtering in the state-space model [53]. Several closely related algorithms, under the names of bootstrap filters and particle filters, have appeared in different research fields.

SMC methods are density estimation algorithms which are commonly used to infer the hidden state sequence of a state-space model, given a set of observations. Assuming we observe y_t at each time t, the basic problem is to estimate the posterior distribution of "hidden" trajectories $p(x_{0:k}|y_{1:k})$. The key idea of SMC is to use a set of weighted samples (particles) $\{x^i_{0:k}, w^i_k\}_{i=1}^{N_s}$ to represent the posterior $p(x_{0:k}|y_{1:k})$. That is to say, the posterior is approximated using N_s particles $x^i_{0:k}, i = 1, \cdots, N_s$, with the corresponding importance weights w^i_k at time k ($1 \leqslant k \leqslant t$):

$$p(x_{0:k}|y_{1:k}) := \sum_{i=1}^{N_s} w^i_k \delta(x_{0:k} - x^i_{0:k}) \tag{2.15}$$

where δ function is the Dirac delta function.

SMC methods inherit the idea of IS that samples from the proposal distribution (also named as importance density) $q(x_{0:k}|y_{1:k})$ instead of the target distribution which is intractable to sample. Accordingly, the unnormalized weights are defined as $w^i_k = \dfrac{p(x^i_{0:k}|y_{1:k})}{q(x^i_{0:k}|y_{1:k})}$, and the corresponding normalized weight is $\tilde{w}^i_k = w^i_k / \sum w^i_k$. The purpose of SIS is to update the particle x^i_{k-1} to obtain the next state x^i_k recursively when obtaining a new observation y_k. Suppose the proposal distribution can be decomposed into

$$q(x_{0:k}|y_{1:k}) = q(x_k|x_{0:k-1}, y_{1:k}) q(x_{0:k-1}|y_{1:k-1}) \tag{2.16}$$

then the weights of the particles can be computed recursively. Namely, the particles are drawn according to $x^i_k \sim q(x_k|x_{0:k-1}, y_{1:k})$. According to $p(x^i_{0:k}|y_{1:k}) = p(y_k|x_k)p(x_k|x_{k-1})p(x_{0:k-1}|y_{1:k-1})$

and (2.16), the weights are updated via

$$w_k^i = w_{k-1}^i \frac{p(y_k|x_k^i)p(x_k^i|x_{k-1}^i)}{q(x_k^i|x_{0:k-1}^i, y_{1:k})} \tag{2.17}$$

If the proposal distribution is only dependent on the previous particles x_{k-1} and current observations y_k, the weights can be rewritten as

$$w_k^i = w_{k-1}^i \frac{p(y_k|x_k^i)p(x_k^i|x_{k-1}^i)}{q(x_k^i|x_{k-1}^i, y_k)} \tag{2.18}$$

In this condition, we only need to store the particles $\{x_k^i\}_{i=1}^{N_s}$ without the need for previous particles $\{x_{0:k-1}^i\}_{i=1}^{N_s}$ and observations $y_{1:k-1}$.

The critical disadvantage of SIS is weight degeneracy: the variance of the importance weights is increasing over time. Therefore, after several iterations only one normalized importance weight tends to be 1 while others tend to be zero which is negligible. The direct consequence is that the vast majority of particles have little significance because of their too small weights. When some degeneracy criterion is fulfilled, we need to use some *resampling* techniques, also named as *selection*, *rejuvenation* in some literature. Resampling removes particles of small weights and copies the particles of high weights according to their respective weights, in order to get a new set of particles. The details on how to implement resampling for SMC methods will be discussed in Chapter 3.

The general framework of SMC (or particle filter) algorithm is described in Algorithm 3. It mainly consists of three steps at each time instant: 1) In the Sampling step, the particles are drawn from the proposal distribution; 2) In the Importance step, the weight of each particle is updated to obtain the unnormalized weight set; 3) In the Resampling step, firstly the weights are normalized, then the particles are copied or replaced according to their normalized weight to obtain a new set of particles with the same number of total particles, and the new particles are weighted equally. Note that the Sampling and Importance steps are independent operations among the particles, and they can be implemented in parallel for each particle. However, the resampling step needs a collective operation among the particles, which imposes some constraints on the step's parallelization.

Algorithm 3: Sequential Monte Carlo Framework

Input: initial or prior distribution $p(x_0)$, number of particles N_s, observations $y_{1:t}$;

1: **// Initialization**
2: **for** $i = 1$ **to** N_s **do**
3: Sample x_0^i from prior distribution $x_0^i \sim p(x_0)$;
4: **end for**
5: **for** $k = 1$ **to** t **do**
6: **for** $i = 1$ **to** N_s **do**
7: **// Sampling Step**
8: Sample $\tilde{x}_k^i \sim q(x_k|x_{0:k-1}, y_{1:k})$;
9: **// Importance Step**
10: $w_k^i = w_{k-1}^i \dfrac{p(y_k|x_k^i)p(x_k^i|x_{k-1}^i)}{q(x_k^i|x_{k-1}^i, y_k)}$.
11: **end for**
12: **// Resampling Step**
13: Normalize the weights;
14: Multiply/Discard particles $\{\tilde{x}_{0:k}^i\}_{i=1}^{N_s}$ according to their high/low importance weights w_k^i, to obtain the new set of N_s particles $\{x_{0:k}^i\}_{i=1}^{N_s}$;
15: Set $w_k^i = 1/N_s$ for $i = 1, ..., N_s$.
16: **end for**

2.3.2 SMC for Nonlinear Filtering

The SMC methods are commonly used to solve nonlinear filtering problems in the dynamic system such as the state-space model. When used for filtering, SMC methods are also named as particle filters. Dynamic models are defined by a pair of equations: 1) the observation equation $y_k = g(x_k, v_k)$, which gives the value of observation under the current unobserved signal (hidden states) with noise v_k; 2) the state/system equation $x_k = f(x_{k-1}, u_k)$, which can be represented by a Markov process. The filtering problem consists of estimating the hidden states in dynamical systems when partial observations are made, and random perturbations (v_k and u_k in the equations) are present in the observations as well as in the dynamical system. The objective is to compute posterior distribution of the states of the model, given some noisy and partial observations.

Considering the nonlinear filtering problems in the dynamic systems, using the observation and state equations of the system, we can prove that $w_k^i = w_{k-1}^i \dfrac{g(y_k|x_k^i)f(x_k^i|x_{k-1}^i)}{q(x_k^i|x_{k-1}^i, y_k)}$. For sake of simplicity, we often use the prior distribution as the proposal (importance sampling) distribution, i.e. $q(x_0) = p(x_0)$ and $q(x_k|x_{k-1}, y_k) = f(x_k|x_{k-1})$, then the recursion is simply $w_k^i = w_{k-1}^i g(y_k|x_k^i)$. The procedures

of particle filter/SMC are shown below in Algorithm 4.

Algorithm 4: SMC or Particle Filter (PF) for nonlinear filtering

Input: prior distribution $p(x_0)$, number of particles N_s, observations $y_{1:t}$;

1: Sample x_0^i from prior distribution $x_0^i \sim p(x_0)$ and set $k = 1$;
2: **// Sequential importance sampling Step**
3: **for** $i = 1$ **to** N_s **do**
4: Sample $\tilde{x}_k^i \sim f(x_k|x_{k-1})$;
5: Evaluate the importance weight $w_k^i = g(y_k|x_k^i)$.
6: **end for**
7: **// Selection Step**
8: Resample with replacement N_s particles $\{x_{0:k}^i\}_{i=1}^{N_s}$ from the set $\{\tilde{x}_{0:k}^i\}_{i=1}^{N_s}$ according to the normalised importance weights;
9: Set $k := k + 1$; go to Step 2.

2.3.3 Resampling

Resampling is a key stage in SMC methods to prevent the weight degeneracy problem and improve the estimation of states by concentrating particles into domains of higher posterior probability [25]. Resampling normally consists of two stages: computing the sum of the weights and weights normalization. Whereas most of the steps of SMC like the generation of particles and weights can be implemented in parallel, the calculation of the sum of weights requires a collective operation among weights. The main algorithms for resampling include 1) Multinomial resampling; 2) Stratified resampling; 3) Systematic resampling; 4) Metropolis resampling; 5) Rejection resampling. These methods have been presented and compared by Lawrence M. Murray et al. (2013) with an implementation on a GPU device [65]. Another problem brought by resampling is *Sample Impoverishment*, because the particles with low weights generate less or even no descendants while descendants of that with high weights increase more and more. The diversity of the particles after resampling weakens, thus it is insufficient to approximate the posterior density. This becomes more serious when the noise in the state equation of the system is too small, which results in the worst scenario where the new set of particles is actually descendants of only one i.e. the most robust particle. There are several methods (Resampling-Move, Kernel Smoothing etc.) to overcome impoverishment and one of the most simple and direct way is to utilise a large number of particles, but this often leads to large amounts of

computations.

2.4 Recent Progress

2.4.1 The trend in SMC

In the previous section the basic particle filter with some general resampling techniques is introduced. Recently there are numerous work in particle filtering focusing on the improvements of various methods. They include the design of importance functions, resampling strategies with decreased computational complexity, methods to overcome the sample impoverishment problem and methods to improve the particle filter performance.

First, the sample impoverishment problem can be addressed via Regularized particle filter [26] or by introducing Markov Chain Monte Carlo (MCMC) moves within a particle filter [28, 26]. Regularised resampling improves the sample diversity by using a continuous kernel approximation instead of a discrete one. In [76], the resampling step is replaced by Markov Chain Monte Carlo moves using the Independent Metropolis Hastings Algorithm (IMHA), in order to generate completely new particles. The computational complexities for the generic, regularized and IMHA particle filters are also studied in [76], and it is shown that the IMHA resampling method resulted in the least computational complexity.

Besides, there are a number of methods to improve the performance of the particle filter. Auxiliary particle filters [26, 68] modify the resampling weights to incorporate measurement information, which comes down to using an importance function that takes measurement data into account. An alternative method is the SMC Filtering with the Resample-Move algorithm. It is introduced in [27] which adds an MCMC move to each particle after resampling to improve the sample diversity.

Furthermore, there are some works which introduce the simplified resampling algorithms to increase the speed of the particle filter with a small penalty in the performance [42, 86]. Another type of filters named as Gaussian particle filtering (GPF [49]) completely removes the resampling step. GPFs operate by approximating the desired densities as Gaussian distributions. Hence only the mean and the

variance of the densities are propagated recursively in time. At each time, the particles are generated from a Gaussian distribution, then the mean and covariance of the filtering are updated using the importance weights of the particles. Therefore there is no need to perform the resampling step.

Another major research direction of the current particle filter (PF) is the real-time PF and distributed PF, which aim to reduce the computational complexity of the filter and meet the requirement of the system with time constraints. The adaptive Real-time PF was proposed by C. Kwok etc. in [50], by dividing sample sets between all available observations and then representing the state as a mixture of sample sets. The distributed PF with two resampling schemes has been proposed by Bolić etc. in [15], along with the FPGA architectures.

2.4.2 MCMC for Big Data

Currently, the dataset to make inference in Bayesian statistics has been largely increased and growing fast not just in size but in complexity of the structures [3]. Running MCMC methods on the big datasets is often far too computationally intensive to be of any practical use [9], as MCMC algorithms require at each iteration to sweep over the whole dataset. Besides, the datasets become "big data" at a much lower dimension than in many frequentist settings due to the repeated computation of the expensive likelihood function in MCMC simulations. There are two main challenges in the MCMC methods to be used in the big data applications. Firstly, the likelihood computations over the whole dataset at each iteration can be very time-consuming for large datasets; Secondly, MCMC methods require keeping the whole dataset in memory and the latency of accessing data points from the memory can have a large impact on the system's performance.

Lots of effort has been recently spent on proposing variants of MCMC algorithms that scale to large datasets. The simplest method involves parallelizing the likelihood to speed up computations. In this method, the data are partitioned with each assigned to a processor or core. At each iteration, each core computes the likelihood for its partition and then passes the result to a central processor to obtain the sum of the log likelihood. As long as there is no significant communication overhead between the processes, the speed of the whole algorithm will be increased while still sampling from the true

posterior distribution.

Recent developments in Markov chain Monte Carlo (MCMC) have been focused on taking advantage of approximations to the target density. These approaches can be broadly classified into two groups: Consensus Monte Carlo (CMC) and subsampling-based algorithms. The CMC approaches (also named as Divide-and-conquer approaches, see [74]) divide the initial dataset into batches, run MCMC on each batch separately and then combine the results to obtain an approximation of the posterior. For these algorithms, the strategy to efficiently combine the batch posterior approximations is difficult to obtain and it has no theoretical guarantees for convergence [9]. The subsampling approaches use subsets of data to provide a faster estimation of the likelihood in which only a fraction of the whole dataset is employed to estimate the likelihood. Thus they often lead to biased estimates. Other recent work uses a lower bound on the local likelihood factor to simulate from the exact posterior distribution while evaluating only a subset of the data at each iteration. However, the construction of these functions and the quality of the bound depends on the target distribution. The details of these algorithms will be discussed in the Section: Related Work.

2.4.3 Specialised Hardware Accelerations

Monte Carlo based algorithms such as MCMC and SMC are usually very computationally intensive tasks, and their runtimes can easily reach weeks or months in the general-purpose microprocessor, i.e., Central Processing Unit (CPU), especially when using large-scale data sets. Parallel computational devices such as multi-core CPUs, the massively parallel processors of Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) have been proposed recently to accelerate these algorithms.

Multi-core CPUs integrates two or more independent actual processing units in one chip while the multiple cores can run multiple operations or tasks at the same time, increasing the overall speed for algorithms which are amenable to parallel computing. Besides, running tasks on multi-core CPU is easier to program compared to other competing platforms. For this reason, multi-core processors have been widely used across many application domains. However, regarding the MCMC algorithms,

a multi-core CPU cannot provide enough parallelism like GPUs or FPGAs to satisfy the speed requirements, especially for real time applications. Also, they cannot fully exploit the properties and characteristics of specific algorithms, e.g. reducing the precision used for likelihood evaluation of MCMC algorithms, which is a normal technique used in Monte Carlo simulations by a trade-off between the accuracy and speed.

GPUs are massively parallel processors in one chip, and their highly parallel structure makes them more efficient than general-purpose CPUs for algorithms where the processing of large blocks of data is done in parallel. The works for Bayesian researchers on GPU-accelerated Bayesian learning have been rapidly rising in recent years. However, the programming languages used for GPUs such as Nvidia's CUDA need to be optimized carefully in order to maximize the system's performance. The optimization process includes the configuration of threads and blocks, balancing the stream multiprocessors (SP) and avoiding bank conflicts etc. These optimizations can be difficult for non-experts compared to the parallel programming in multi-core CPUs.

FPGAs have proven to be a very promising and competing platform for MCMC and SMC accelerations, due to their capability to implement massively parallel computational units to speed up the likelihood evaluations [63]. Contemporary FPGAs have massive resources of reconfigurable logic gates and RAM blocks to implement complex digital computations. The parallelism and reconfigubility properties make FPGAs very suitable platform to map variants of MC-based algorithms and compare their respective performance improvement. By appropriately dividing the algorithm into parallel tasks that can be executed at the same time and mapping them into an FPGA, a considerable amount of acceleration can be achieved. An important advantage of acceleration of MCMC algorithms on the FPGA devices is their flexibility to operate in any custom arithmetic precision, instead of implementing operators in double floating-point. Utilizing reduced precisions in Monte Carlo algorithms has a positive impact on both memory- and computation- bound problems.

When running MCMC methods in GPUs and FPGAs platforms, relaxing the requirement for high precision allows the MCMC algorithms to execute faster and with less energy, compared to double floating-point design in multi-core CPUs. By utilising low precision (custom floating point) datapaths, it consumes fewer resources and leads to a higher degree of parallelism compared to full precision

(double floating point) datapaths for a fixed area resource. As such, recent works that target GPUs and FPGAs have been investigating the utilisation of custom arithmetic precision for the estimation of the likelihood. However, departing from double precision arithmetic for likelihood evaluation leads to biased estimates with respect to systems that employ double precision arithmetic throughout the computations, because of the approximations in each likelihood term. When exact estimates are required, the utilisation of high precision data-paths with lower performance is unavoidable.

2.5 The Approach of this Thesis

Distributed and parallel implementations of the Particle Filter or SMC algorithms are needed in order to achieve minimum execution time to allow these algorithms practicable for modern SMC applications. Therefore, in this thesis parallel resampling algorithms and architectures are studied and proposed to allow parallel implementation of SMC algorithms in FPGAs. This is achieved by two ways: 1) parallel architectures for two traditional resampling algorithms (Residual Systematic Resampling and Systematic Resampling) are proposed using the adder tree and recursive doubling methods to compute the sum and cumulative sum of the weights of particles respectively; 2) two Monte Carlo resampling (Metropolis and Rejection) algorithms are presented while their corresponding parallel architectures are proposed.

In order to allow faster execution of MCMC methods, multi-core implementations of the likelihood computation in MCMC construction have been studied recently. This thesis builds upon the approximation models of the likelihood terms, which builds a lower bound on the real likelihood terms based on their respective reduced precision values and then simulates MCMC while evaluating the likelihood mostly on reduced precisions. In such a way, more parallelism of the likelihood computation can be achieved for a given resource in the computational devices such as FPGAs. This method can guarantee unbiased estimates if we make sure that the reduced precision values are lower bounds on the real values (double floating point values). In addition, MCMC methods can be accelerated by allowing a small bias in the estimates and utilizing a fraction of the data points to evaluate the likelihood at each iteration, i.e. data subsampling based MCMC algorithms. This thesis proposes the use

of unequal probability sampling in the data subsampling based MCMC algorithms, in order to expose the performance of the memory sub-system to the MCMC algorithm and thus guide the sampling process, leading to a reduction in the access to memories which have large latency.

The main objective of this thesis is to develop hardware architectures that allow for high speed SMC and MCMC algorithms. It focuses on how to modify the existing algorithms to be mapped on FPGAs with the proposed and tailored FPGA architectures. Although the modified algorithms and proposed architectures are devoted to FPGAs, the ideas of how to utilize parallelization can be applicable for other parallel computing platforms such as GPUs, which will be discussed in Section 6.2.

2.6 Related Work

In this section we review the existing works in literature which are closely related to the work presented in this thesis. These works include SMC acceleration using parallel hardware such as GPUs and FPGAs, MCMC algorithmic adaptations and current MCMC methods for large dataset. The previous work on SMC accelerations is particularly focused on the parallel and/or optimized resampling algorithms and architectures in hardware. The literature on MCMC to tackle the time-consuming likelihood computations includes the data subsampling based MCMC algorithms and MCMC accelerations in FPGAs and GPUs using custom precision techniques and other optimization methods. At the end of each section, we provide comparisons and comments on how the work in the thesis differs and outperforms previous work in the reviewed literature.

2.6.1 Resampling for SMC methods in GPUs and FPGAs

There is exhaustive literature on parallelization of particle filters (SMC methods) and resampling using GPUs and FPGAs. A large number of works on resampling is trying to simplify the resampling step by trading the performance for speed improvement, instead of improving the resource utilization in FPGAs or GPUs to increase the speed. Besides, a full comparison on how to parallelize these resampling algorithms has never been done.

FPGA-based implementations of parallel particle filters are presented in [15, 14, 75] and more recent work is done in [41, 42]. [15] proposes a new form of systematic resampling (which is named as residual systematic resampling, RSR) algorithm and ways in which this algorithm can be parallelized. The authors also propose two efficient distributed implementations of the particle filters on FPGA. The proposed RSR algorithm can be easily pipelined in hardware implementation and applied in the two proposed distributed particle filter architectures. However, a sum of the weights of the particles needs to be performed for the RSR algorithm and this operation can become the computation bottleneck in the proposed architectures. Both [14] and [42] introduce the simplified partial resampling algorithms which perform resampling only in part of the particles by using a simple threshold-based scheme. In the partial resampling proposed in [14], the particles are grouped into two separate classes: one composed of particles with moderate weights and another with dominating and negligible weights. The particles with moderate weights are not resampled and unchanged, whereas the negligible and dominating particles are resampled. In the simplified resampling algorithm proposed in [42], the weights of each particle are compared with a threshold T. The particles with weights less than T are discarded and replaced by the particles with weights greater than T. Both resampling algorithms reduce the execution time because a part of the particles is not resampled at all, but these algorithms negatively affect the resampling quality and the overall performance of the filter. [75] proposed fully pipelined distributed implementation of particle filter using the RSR algorithm proposed in [15] and they use some additional memory block to handle the resampling routing between the distributed processing elements. [41] proposed an improved residual resampling algorithm for non-normalized weights. However, this algorithm assumes that the sum of the weights is known in advance.

GPU-based implementations of resampling have been presented in [45, 35, 65]. Both [45] and [35] employ the traditional systematic resampling (SR) but using different parallel implementations of the cumulative sum of the weights. [35] use a forward adder tree to calculate the sum of the weights; then a backward adder tree is used to construct the cumulative sum of the weights. The authors also provide information on the relative time spent in the different steps of particle filter, in the GPU and CPU implementation respectively. The results show that the major part of the time is spent on resampling in the GPU implementation, whereas the particle update is a dominant step in the CPU implementation. [45] propose a load balanced particle replication algorithm for systematic resampling, which gives

almost constant execution speed in a GPU device. All these works we have mentioned are based on the traditional systematic resampling and therefore they need to calculate the sum or the cumulative sum of the weights, which consumes lots of computational resources in FPGAs and GPUs. In contrast, [65] proposes two novel resampling algorithms (Metropolis and Rejection resampling) which are more readily parallelised in hardware. The authors showed that these alternative approaches perform significantly faster on the GPU than the commonly used systematic resampling algorithm, due to their avoidance of collective operations across all weights, which better suits the GPU architecture. However, the main disadvantage for these two algorithms when implemented on a GPU is that they cause warp divergence [65][46] due to branch statements, or different trip counts of loops. Besides, the execution times of these two algorithms are sensitive to the variance in weights. That is to say, Rejection resampling may suffer from frequent rejections and Metropolis resampling from possibly large convergence times.

The work we present in Chapter 3 of the thesis is the first to propose parallel FPGA architectures for Systematic, Metropolis and Rejection resampling. We also propose an improved FPGA implementation of the Residual Systematic Resampling (RSR). It is the first to include a comparison of the main state-of-the-art resampling algorithms when implemented on FPGAs, highlighting their advantages and demonstrating how each scales with the number of particles and the variance in weights of particles. Finally, it also provides a comparison on their execution times on FPGAs, GPUs and CPUs.

2.6.2 MCMC in Algorithm Adaptation

For problems with large data-set, the evaluation of the likelihoods dominates the computational cost of MCMC algorithms. The increased computational time lies in the complete scan of the data at each iteration through likelihood evaluations. There are several major methods to overcome computational problems brought by the big data. A detailed review of current MCMC methods used for large datasets can be found in [9].

Speeding up likelihood computations

The simplest method is to speed up the required computations by parallelising the likelihood evalua-

tions exploiting the data independence property. [21] implemented Bayesian models draw on MCMC simulation using parallel computations by CUDA implementations in GPUs which reduced significantly the runtime processing of MCMC simulations. Other approaches can be broadly classified into two groups: Consensus Monte Carlo (CMC) and subsampling-based algorithms. The CMC approaches [74] divide the initial dataset into batches, run MCMC on each batch separately and then combine the results to obtain an approximation of the target posterior distribution. [44] propose to combine the batch posterior approximations using Gaussian approximations or importance sampling. However, the strategy to efficiently combine the batch posterior approximations is difficult to obtain and it has no theoretical guarantees for convergence [9].

Many recent works are based on subsampling methods. These approaches use subsets of data to provide a faster estimation of the likelihood in which only a fraction of the whole data set is employed to estimate the full likelihood. [48] introduce an approximate Metropolis-Hasting rule with controlled bias that allows accepting or rejecting samples with high confidence using only a fraction of the data. [8] propose an adaptive subsampling technique which is an alternative approximate implementation to [48] that only requires evaluating the likelihood of a random subset of the data. This algorithm is a more robust approach compared to [48] and can provide estimates under a user-controlled error. However, both algorithms in [48] and [8] are approximate, and they rely on a bound for the difference between the log-likelihood contributions at the proposed and current sample, and that of the control variates [70]. [69] propose a subsampling of the data based on the contribution on each likelihood term, which by a bias-correction can be turned into an unbiased estimator of the likelihood function. This algorithm needs to build a surrogate of the true likelihood, using either a Gaussian process or a spline approximation. As such, it requires computing the surrogate likelihood for all data before running the subsampling step, thus introducing another costly requirement. [58] present an auxiliary variable MCMC algorithm that also queries the likelihoods of a small subset of the data but achieves exact posterior distribution. The fundamental assumption of this approach is that each product term in the likelihood can be approximated from below by a function easier to compute. The drawback of this method lies in the construction of these functions and the quality of the bound depends on the target distribution. Furthermore, the authors have demonstrated that an acceleration is only achieved when the approximation is tight enough.

Data transfer problem in MCMC for big data

Most methods such as parallelizing likelihood and data subsampling address the problem of speeding up the likelihood computations. However the existing work doesn't consider the properties of memory hierarchies in modern computational systems, but treats the memory as one monolithic storage space. As such, in current design of MCMC applications for big data with high performance, normally the memory issue other than the likelihood computation becomes the dominant bottleneck, and often impacts significantly the system's performance of the whole algorithm.

A subsampling approach that addresses the memory issue is proposed in [31], which applied random projection techniques to Bayesian regression of large dataset. The proposed algorithm used the fixed projected dataset, which is built by a single multiplication by a random matrix, to construct MCMC simulations. However, the authors in [10] show that the accuracy guarantee of [31] is too weak for some important Bayesian inference tasks, and usually the complete random sampling does not work well in MCMC methods [67].

The thesis proposes a communication-aware MCMC framework that takes into account the underlying performance of the memory subsystem during the sampling process in Chapter 5, leading to faster execution times. The proposed algorithm addresses exactly the memory-bound problem in the MCMC construction, opening the way for applying the MCMC algorithm to large scale datasets.

2.6.3 MCMC methods in GPUs and FPGAs

Previous works on accelerating MCMC methods on GPUs are very limited. The GPU implementations are mainly focused on parallelizing the likelihood computations [21]. Most FPGA-based Monte Carlo designs exploit the reduced precision data-paths to allow for more parallelism for the likelihood computations. Previous works on FPGAs using mixed precision can be found in [83, 20, 61, 63, 62]. [83] propose an FPGA-based architecture for Monte Carlo simulations that monitors and configures the precision used in the system during run-time. In [83] high- and low- precision simulations are compared using the Kolmogorov-Smirnoff metric and the precision is adapted so that the distance between the distribution of the samples from high precision module and low precision module is

smaller than a threshold. However, placing such a threshold is empirical and does not constrain the bias. [20] proposes a mixed precision methodology for Monte Carlo simulation in reconfigurable accelerator systems. They use an auxiliary mixed precision run to correct the bias in the output estimates. However, their method requires knowledge of the function of interest during the design of the system, and as such the generated samples cannot be used for other estimates. The authors in [61] propose the use of custom precision arithmetic for population-based MCMC methods where multiple parallel chains are used to improve the mixing properties of the chain. However, [61] can only be applied to the MCMC algorithms which use multiple parallel chains. [63] propose a framework for identifying an optimum custom precision number representation in the MCMC architecture, targeting a specific bias-variance ratio at the output. Furthermore, [62] propose an optimised FPGA architecture for Population-based MCMC. Besides the introduced error and bias correction in the outputs of the system proposed previously, the biggest problem for all previous works using custom or mixed precision on FPGAs is that it's hard to guarantee which precision to be utilised that would lead to sufficiently accurate result defined by the user.

In this work we also focus on accelerating the likelihood evaluation part of MCMC by utilizing the custom precision technique in FPGA design, but we arrive at a method in which the samples are generated from true posterior distribution. Here we apply the underlying idea in [58] to the custom precision support on FPGAs, in order to reduce the computation time as well as achieve an unbiased estimator. Instead of using approximate functions for the likelihood terms as in [58], we use custom precision approximations and we show how we can guarantee that these approximations are a lower bound to the true likelihood term (which is a requirement for [58] to generate accurate samples from the posterior distribution). To the best of our knowledge, this is the first work to produce and guarantee an unbiased MCMC estimation using mixed precision design on FPGAs. Also it is the first work to present the combination of the ideas on custom precision design and data subsampling in MCMC for big data applications, which results in an unbiased MCMC accelerator.

2.7 Summary

This Chapter provides the basic concepts of the Bayesian Inference, introduces the SMC and MCMC algorithms and principles and highlights their current research directions especially the hardware accelerations.

From the literature review, the parallelization method is only proposed for the systematic resampling algorithm, while the two Monte Carlo resampling algorithms are only implemented in CPU and GPU. FPGA-based architectures for these two algorithms have never been done. Besides, a full comparison of these state-of-the-art resampling algorithms in CPUs, GPUs and FPGAs has never been done. Although there is a significant amount of work on MCMC acceleration recently, they are limited because of a few reasons: 1) the approximate MCMC algorithms such as data subsampling methods suffer from the biased estimates or another costly computations for bias correction; 2) research that uses FPGAs suffers from the memory bandwidth problem and also the biased estimates when using custom precision technique.

The following chapters focus on the acceleration of these algorithms using FPGAs. We show how these algorithms can be adapted and more suitable for FPGA implementations compared to the existing work on CPUs and GPUs. Novel algorithms are also proposed to fully utilize the resources in FPGA to minimum the execution time.

Chapter 3

Parallel Resampling Algorithms and Architectures for SMC Methods on FPGAs

3.1 Introduction

Particle Filters (PFs), also known as Sequential Monte Carlo (SMC) methods, are density estimation algorithms which are commonly used to infer the hidden state sequence of a state-space model, given a set of observations. They can efficiently handle non-linearity and/or non-Gaussianity in the model and they exhibit great robustness and accuracy. They have thus been widely used in target tracking, digital signal extraction, air traffic management and robot localization [25, 53, 19, 18], among other applications. Although powerful, PFs are also computationally intensive, which becomes a major issue in its application to complex models, especially with real-time constraints [25].

PFs use a set of N particles (i.e. samples) to estimate the density of the state at each time step t. The most common PF algorithm (bootstrap filter) is shown in Algorithm 5. For each time t and for each particle i, the following steps are performed. In the *sampling* step, each particle's state $\mathbf{x}_t^i$ (which is a vector) is propagated to the next time step using the transition equation of the model. In the *importance computation* step, the likelihood of each particle given the observation $\mathbf{y}_t$ at the present time step is evaluated. This is the weight of the particle. Based on the values of the weights, a new set of particles is generated in the *resampling* step. In this step, particles with large weights are replicated

several times while those with small weights are discarded. Finally, the resampled particles are used to estimate the new state.

Algorithm 5: Particle Filter Algorithm

for $t = 1, \ldots, T$
  for each $i \in \{1, \ldots, N\}$

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

The *sampling* and *importance computation* steps are independent operations for each particle, so they are inherently parallel and straightforward to implement in parallel devices such as GPUs and FPGAs. Resampling however requires a collective operation (either a sum or a cumulative sum of all the weights), which makes it the most challenging step to parallelize. Moreover, resampling is crucial for the stability of the PF because it prevents the filter from weight degeneracy and improves the estimation of states by concentrating particles into domains of higher posterior probability [25]. However, parallelizability of the filter is affected by the *resampling* step, and the major part of time in the GPU implementation of PF is spent on resampling, where the time spent on the other three steps become almost negligible when the number of particles increases considerably [35]. Resampling gives a serial time complexity of $O(Nlog_2N)$ for the multinomial resampling compared to $O(N)$ for the stratified and systematic resampling [39], where N is the number of particles. [35] analysed the time spent on each part of PF in the GPU implementation, which is shown in Figure 3.1. The results demonstrated that the major part of the time is spent on resampling in the GPU, especially when the number of particles (N) increased. Therefore, resampling is a crucial and computationally expensive part in PF [25]. It becomes a bottleneck in parallel hardware implementations of PF and there is much to gain in the speedup if this step can be accelerated.

This Chapter focuses on four state-of-the-art resampling algorithms: Residual Systematic Resampling (RSR), Improved Systematic Resampling (ISR), Metropolis Resampling and Rejection Resampling based on recent literature. Novel parallel architectures are proposed for each algorithm. RSR and

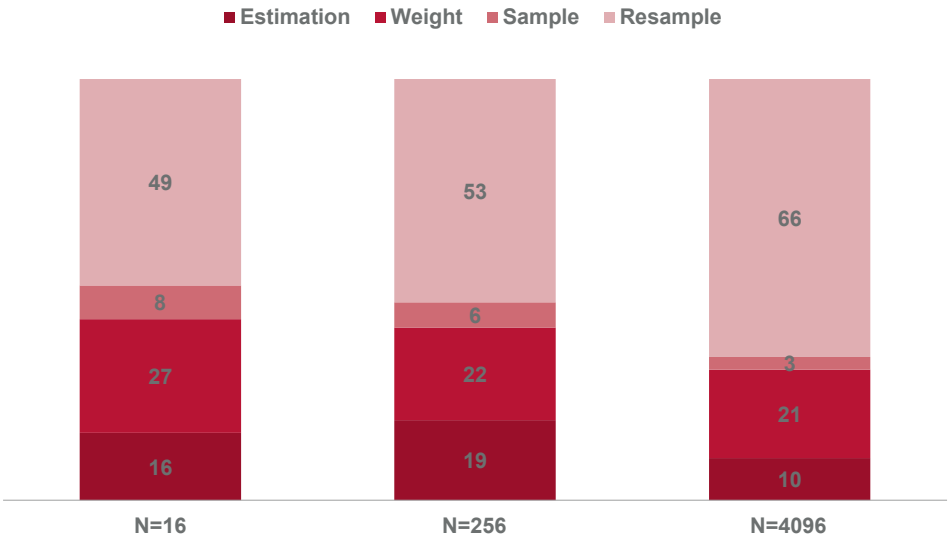


Figure 3.1: *Time percentage spent on each step in PF by Gustaf Hendeby's work in GPU implementation in 2010 [35].*

ISR use a parallel SUM and CUMSUM (cumulative sum or pre-fix sum) operation respectively followed by parallel and pipelined offspring evaluators. On the other hand, Metropolis and Rejection architectures do not require collective operations, but the parallel blocks of them must have global access to all the weights. Therefore, memory access strategies which implement a simplified Random Permutation Generator (RPG) are proposed for the parallel Metropolis and Rejection architectures.

The main contributions of this work are:

- The introduction of novel parallel architectures which map four resampling algorithms to an FPGA. This is the first work that presents parallel FPGA architectures for the state-of-the-art resampling algorithms and compares their execution time with that of GPU implementations;
- Memory access strategies for parallel Metropolis and Rejection architectures. An optimized RPG circuit which uses a cyclic shifter is proposed to randomly forward weights from the memory to the parallel processing blocks and guarantee that all blocks have global access to the weights memory;
- A modified version of SR, which we call Improved Systematic Resampling (ISR), is introduced to save resources and achieve further speedup in hardware. Moreover, the advantages of each algorithm for parallel implementation are summarized.

3.2 Resampling Principles

The *resampling* step in PF aims to regenerate the particle population by removing particles with small weights and replicating particles with large weights. It can be considered as a randomised algorithm which inputs the number of particles (N) and the weights of all particles ($w_i, i = 1, \ldots, N$). The weights can be normalized or non-normalized as the normalization can be performed either in the step of *importance computation* or *resampling*. Here, the weights are assumed to be non-normalized. The output of the algorithm is the number of offsprings of each old particle ($o_i, i = 1, \ldots, N$), i.e. how many times particle i is replicated. Alternatively, the output can be the index of the ancestor of each particle of the new particle population ($a_i, i = 1, \ldots, N$). In Table 3.1, the two different types of outputs of resampling for the particles with normalized weights are shown from a simple example. As we shall see, each resampling algorithm naturally takes one form or the other, and it is easy to convert between the two forms.

Table 3.1: *Two types of the outputs after resampling: (a) index of the replicated particles (b) replication factors or offsprings of the particles*

Particles	Weights	(a) index	(b) offsprings
1	$w_1 = 1/2$	1	2
2	$w_2 = 1/3$	1	1
3	$w_3 = 1/12$	2	1
4	$w_4 = 1/12$	3	0

The principle of resampling is to make sure the offspring vector satisfy the following two conditions (3.1) and (3.2) [23]:

$$\sum_{i=1}^{N} o_i = N \tag{3.1}$$

$$\mathbb{E}(o_i) = N w_i / sum(\mathbf{w}) \tag{3.2}$$

The first equation ensures that the total number of the new resampled particles keeps unchanged. The second equation shows that the expected value of the number of replications of a particle should be proportionate to the value of its weight. The resampling quality is often quantified by how much the algorithm's result deviates from the expected value of equation (3.2). This is given by the relative root-mean-square error (RMSE) (3.3), computed from the offspring vector and weight vector [47].

$$RMSE = sqrt(\frac{1}{N}\sum_{i=1}^{N}(\frac{o_i}{N} - \frac{w_i}{sum(\mathbf{w})})^2) \tag{3.3}$$

3.3 Resampling Algorithms

This section presents the four resampling algorithms, followed by a summary and comparison of these algorithms at the end. The improved systematic resampling (ISR) is proposed here to achieve further speedup in hardware implementation.

3.3.1 Residual Systematic Resampling (RSR)

The standard resampling algorithms (e.g. Multinomial, Stratified, Systematic resampling) are based on multinomial selection of o_i, which is equivalent to selecting with replacement N particles $\tilde{x}^j$ from the original set of particles $\{x^i\}$ where $P(\tilde{x}^j = x^i) = w_i, i, j = 1, \ldots, N$. Among these methods, Systematic Resampling (SR) is favourable over the others considering resampling quality and computational complexity [39] and thus it is the method of choice for most implementations, including those in FPGAs and GPUs. The original SR proposed in [39] has non-deterministic runtime which depends on the distribution of the weights, as it needs to precompute a cumulative sum of the weights and do a binary search. [15] proposed residual systematic resampling (RSR) as an alternative form in order to introduce deterministic runtime, and its pseudocode is given in Algorithm 6. In RSR, the number of offsprings of a specific particle is determined in the **for** loop by truncating the product of the number of particles and the normalized weight using uniform random numbers. The random number is updated at each iteration as shown in line 6 of the pseudocode. As a result, the algorithm has a deterministic processing time. However the data dependency inside the resampling loop limits its potential parallelization.

Algorithm 6: Residual Systematic Resampling

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

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

3.3.2 Improved Systematic Resampling (ISR)

Recently, [65] presented another form of systematic resampling, which delivers the cumulative offspring vector i.e. O_j as shown in Algorithm 7. This algorithm first calculates the cumulative sum of the weights, then truncates the product of the number of particles and the cumulative weight. As a result, it has no data dependency inside the resampling loop. As SR replicates the particle i, $o_j = \lfloor Nw_j \rfloor + 0/1$ times for any values of u in $[0, 1)$, the expected number of replications is consistent with (3.2). In this subsection, an improved SR algorithm (ISR), shown also in Algorithm 7, is introduced to simplify the calculation of the systematic outputs using $u = 0$. This proposed modification removes the need to generate a uniform random number for each execution of SR, which will translate in resource savings in the FPGA, as will be shown in the next section. However, by setting $u = 0$, the resampling quality is slightly affected, which will be discussed in the Section 3.5.

Algorithm 7: Improved Systematic Resampling

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

1: $\mathbf{cw} = cumsum(\mathbf{w})$;
2: $sum = cw_N$,$O_0 = 0$;
3: **for** $j = 1$ **to** N **do**
4: $\quad O_j = \lfloor Ncw_j/sum \rfloor$; //$O_j = \lfloor Ncw_j/sum + u \rfloor$ in [65]
5: $\quad o_j = O_j - O_{j-1}$;
6: **end for**

3.3.3 Metropolis and Rejection Resampling

[65] proposes two novel resampling algorithms which are more readily parallelized in hardware. The pseudocode of the two algorithms is presented in Algorithm 8 and Algorithm 9. Neither algorithm requires a collective sum or cumsum operation. Metropolis resampling is based on the well-known Metropolis algorithm and it requires the ratio between two weights at each step of the inner loop. After B steps for each weight (outer loop), the algorithm is assumed to have converged to the correct particle distribution implied by (3.2) (for more details see [65]). The selection of B is a trade-off between speed and accuracy, with smaller B achieving faster execution time but a larger bias is introduced in the resampling results. [65] provides guidance as to the selection of B by assuming the upper bound $sup\ w$ on non-normalized weights and the expected weight value are known. One can always use large B (resulting in increased execution time) to improve the resampling quality. However, this algorithm still produces a biased sample as B must be finite.

Algorithm 8: Metropolis Resampling

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

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

Compared to Metropolis resampling, Rejection resampling is easier to configure and unbiased. Rejection resampling is based on the rejection sampling algorithm. The idea of this method is to propose ancestor indexes until an index is accepted based on the ratio in line 4. As the $sup\ w$ operated in the $while$ conditional is a constant in each *resampling* step, one speed improvement can be achieved by setting $sup\ w = 1$ in the *importance computation* step without introducing any extra calculations. Hence no division needs to be performed in the *resampling* step. Compared to Metropolis resampling,

Rejection resampling is easier to configure and unbiased but it has a non-deterministic runtime, since the number of **while** loop iterations in line 4 are unknown.

Algorithm 9: Rejection Resampling

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

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

3.3.4 Algorithm Summary

Both RSR and ISR require a collective operation over the weights, specifically sum and cumulative sum, which makes them less readily parallelised in hardware. ISR also needs an additional memory space to store the cumulative weights. Another disadvantage of these two algorithms is that the collective operation can exhibit numerical instability for large N or large weight variance [65]. This is more observable when using single-precision arithmetic instead of double-precision arithmetic. On the other hand, Metropolis and Rejection resampling can be parallelized more easily, due to the lack of collective operations (the outer loop iteration of both algorithms are completely independent). However, Metropolis results in increased complexity and a biased result, while Rejection's runtime is non-deterministic. FPGAs are more suitable than GPUs to implement Metropolis and Rejection algorithms, due to the GPU divergence problem mentioned above. Note that the output of the four algorithms takes one of the two forms described in Section 3.2. Even though the form of the output will affect the overall PF's architecture, this is out of the scope of this Chapter which focuses solely on the resampling stage.

3.4 Proposed Resampling Architectures

In this Section, we propose optimized FPGA architectures for all four algorithms described in the previous section. First, the parallel architecture of RSR is presented based on [7], and novel parallel architectures are proposed for the other three algorithms. All architectures parallelize resampling by splitting the particles into M sub-sets of N/M particles each and assigning them to M parallel processing blocks. The form of the processing block differs between architectures. Parallel architectures to implement sum and cumulative sum on FPGAs are also introduced for RSR and ISR algorithms respectively. Furthermore, in order to satisfy that the multiple resampling blocks for Metropolis and Rejection architectures have global access to the weights memory at each iteration, an optimized Random Permutation Generator (RPG) is proposed to connect the weights memory and resampling blocks.

3.4.1 Parallel Architectures for RSR and ISR

To achieve parallel architectures for the RSR algorithm, we first propose parallel sum computations for streaming weight sets as inputs. Following this step and based on [7], we propose the method to remove the data dependency inside the RSR resampling loop by utilizing the intermediate results from the previous parallel computation of sum. To parallelize ISR algorithm, we first propose parallel architecture to compute the cumulative sum of the streaming weight sets, then the parallel computation of the cumulative offsprings is straightforward as it has no data dependency.

The RSR and ISR architectures are shown in Figure 3.2 and 3.3 respectively. The total N weights are split into M sub-sets and assigned to M parallel processing blocks to process N/M weight by each block. The M weight sets are stored in one memory unit where each memory address stores M weights (one from each set). This allows us to read M weights in the same cycle. Firstly RSR calculates the sum of the weights while ISR calculates the cumulative sum of weights and stores them in the cumulative weights memory. After the collective operations, M weights are assigned to the parallel RSR blocks and M cumulative weights to ISR blocks at each time for offspring evaluation. Following the evaluation, the offspring results are stored in the respective output memory.

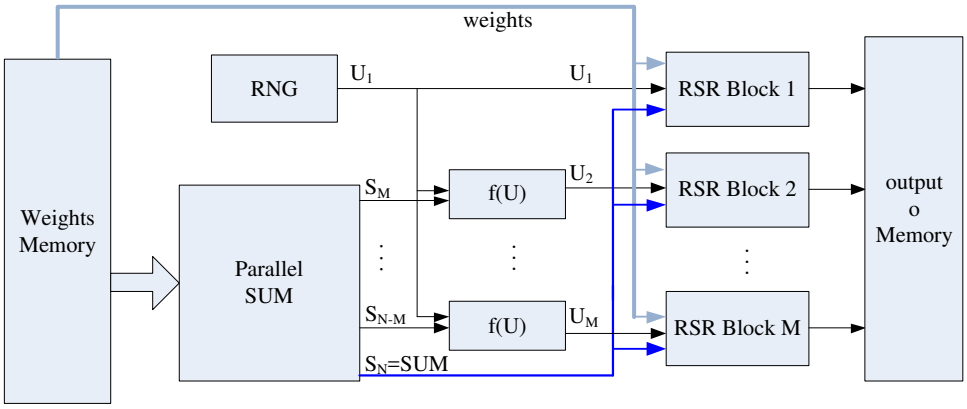


Figure 3.2: *Parallel architecture for RSR resampling*

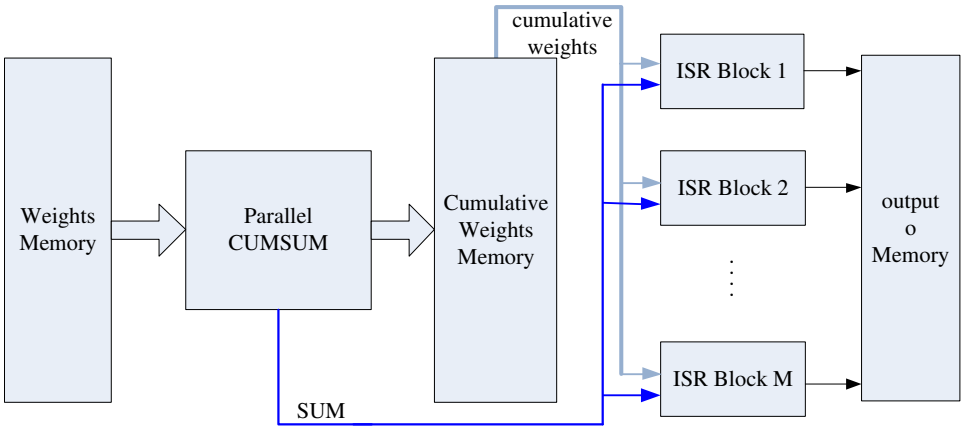


Figure 3.3: *Parallel architecture for the proposed ISR resampling*

RSR and ISR algorithms need a first step that computes the sum or cumulative sum of weights respectively. Although the sequential computation is straightforward, its parallelization in hardware is challenging due to the output data dependency. Parallel sum (SUM) and cumulative sum (CUMSUM) algorithms are described in [34] and shown in Figure 3.4. The sum algorithm is an adder tree. With respect to CUMSUM, recursive doubling is a naive parallel scan and needs many data exchange operations. The three-step recursive doubling algorithm is more suitable for a large number of inputs, as it consumes fewer resources (i.e. adders) compared to the recursive doubling method.

The proposed FPGA implementations for the two collective operations are shown in Figure 3.5. Please note here we need to compute the sum or cumulative sum for streaming weight sets and this is the reason that an accumulator is placed in the end of the block in Figure 3.5. In Figure 3.4, only

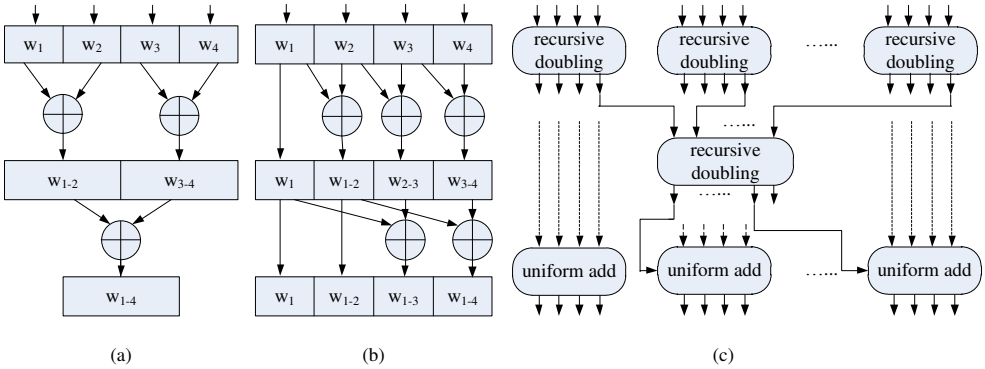


Figure 3.4: *The parallel sum and cumsum algorithms in the form of block diagrams ($w_{i-j} = \sum_i^j w$): **(a)** Tree SUM; **(b)** Recursive doubling CUMSUM for a small number of input; **(c)** Three-step recursive doubling CUMSUM for a large number of inputs.*

one weight set as parallel input is assumed. Both architectures in Figure 3.5 are based on using a large amount of parallel pipelined adders and feeding them with a new set of weights at each cycle. For both architectures, an accumulator is placed after the main datapath in order to accumulate the values of each weight set. Note that the SUM and CUMSUM can have different parallel degrees as the offspring evaluation of RSR and ISR. We use M_1 to represent the parallel degree of both SUM and CUMSUM. For CUMSUM, $M_1 - 1$ uniform adders are also necessary to produce the M_1 outputs (cumulative sums) for each weight set, which is illustrated in Figure 3.5. The execution time (in clock cycles) for SUM and CUMSUM can be reduced from $N + L_{SUM}$ and $N + L_{CSUM}$ cycles for a sequential implementation to $N/M_1 + L_{SUM}$ and $N/M_1 + L_{CSUM}$ cycles respectively, where L_{SUM} and L_{CSUM} are the latency of the SUM and CUMSUM datapaths respectively.

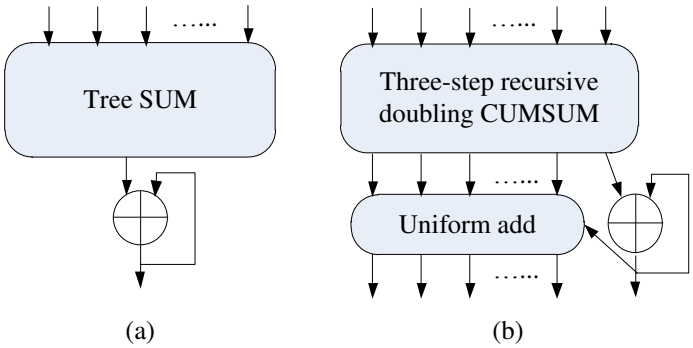


Figure 3.5: *Block diagram of the parallel sum and cumsum algorithms for streaming weight sets as inputs on FPGAs: **(a)** Parallel SUM; **(b)** Parallel CUMSUM.*

Following the collective operations, both RSR and ISR have to evaluate the number of offsprings of each particle (**for** loop in Algorithm 6 and 7). These operations can also be parallelized. In contrast to ISR, RSR offspring evaluation cannot be straightforwardly implemented in parallel. [7] proposed a way to parallelize RSR by calculating the initial random number used for each block in advance. Assuming M offspring evaluation blocks are implemented for RSR and each block processes N/M weights independently, the random number used for each block is generated using the algorithm in Algorithm 10 which is represented as $U_i = f(U)$ in Figure 3.2. The algorithm needs to compute the sum of the weights processed by each block, which is achieved from the pipelined SUM algorithm in Figure 3.5.

The proposed ISR resampling algorithm is more readily parallelized compared to RSR, due to lack of data dependency between its loop iterations. The drawback is that one additional memory is needed to store the cumulative sum of the weights. Another difference between ISR and RSR is the absence of random number generator (RNG) in the proposed ISR architecture as shown in Figure 3.3.

Algorithm 10: Parallel computation of random numbers used for parallel RSR resampling architecture

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

3.4.2 Parallel Architectures for Metropolis and Rejection with Memory Access Strategies

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

1) novel parallel architectures for Metropolis and Rejection resampling; 2) memory access strategies using the proposed Random Permutation Generator (RPG).

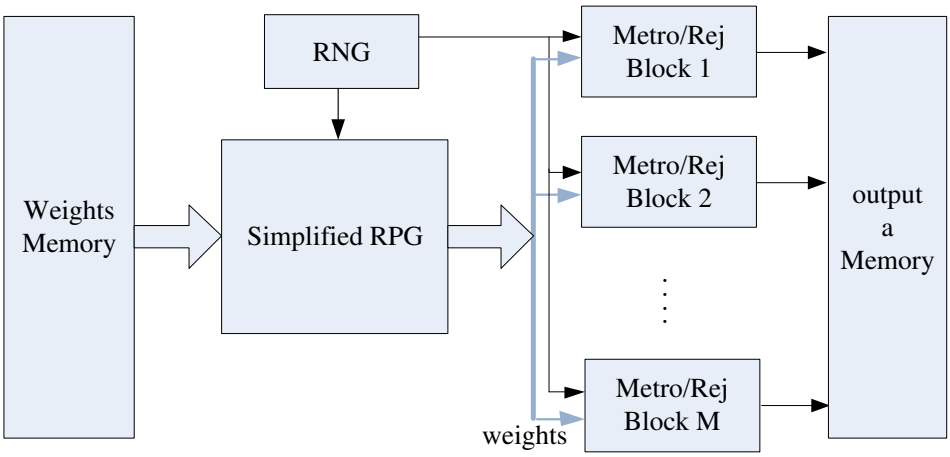


Figure 3.6: *Parallel architecture for Metropolis and Rejection resampling*

The parallel architectures for Metropolis and Rejection resampling based on the proposed memory access strategy are presented in Figure 3.6. The weights are stored in the same way as described for RSR and ISR. Here, an RPG circuit is introduced to handle the communication between the weights memory and the Metropolis or Rejection resampling blocks. This RPG circuit has to guarantee that each block has random and global access to all the weights, i.e. every block can choose any weight with equal probability (uniform sampling). At each cycle, M weights are read from the memory and they pass through this circuit which, by using the outputs of an RPG, randomly allocates the M weights to the parallel blocks. Then the indexes of the ancestors of the resampled particles are stored in the respective output memory.

The commonly used algorithm for an M-element RPG is the Knuth Shuffle which is shown in Algorithm 11, and its FPGA implementation is presented in [17]. The Knuth Shuffle RPG proceeds through $M-1$ steps and each step has operators including RNG, remainder and swap functions.

As the purpose of the RPG here is to randomly distribute the weights to each block and complete permutations implementation is not necessary, we propose a simplified circuit with only one step using a cyclic shifter and $Mlog_2M$ bits RNG to implement an RPG in FPGA. The optimized RPG algorithm to be implemented in FPGA is described in Algorithm 12. It treats the permutation from 0

Algorithm 11: Knuth Shuffle RPG

1: Initialization: $\mathbf{Q} = \{1, \ldots, M\}$;
2: **for** $n = M$ **to** 2 **do**
3: $j \sim U\{1, \ldots, M\}$;
4: $i = j\%n + 1$;
5: swap(Q_n, Q_i);
6: **end for**

to $M - 1$ as one value, and then shifts this binary value by some random number of bits. Finally, the value after cyclic shifting can be regarded as one random permutation of M-element.

Algorithm 12: simplified RPG in FPGAs

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

This simplified RPG takes advantage of the hardware implementation in which all numbers are represented in binary, and thus leads to a reduction in resources and time compared to the Knuth Shuffle RPG. It still satisfies the condition that every element is chosen with equal probability at each place i.e. $p(Q_i = j) = 1/M$, for each $i = 0, \ldots, M - 1$ and $j = 0, \ldots, M - 1$. The disadvantage of this algorithm when compared to the Knuth Shuffle RPG is that the number of total permutations is reduced from $M!$ to $Mlog_2M$. Nevertheless, this is shown to affect the resampling quality of parallel Metropolis and Rejection algorithms only minimally (see Section 3.5.1). Therefore, this optimized RPG is proposed to randomly allocate the weights read from the memory to each parallel resampling block of Metropolis and Rejection architectures as shown in Figure 3.6.

3.5 Evaluation and Experiments

The four proposed architectures are implemented on a Xilinx Virtex-6 LX240T FPGA. All arithmetic operators are taken from the Xilinx Coregen library in single floating point precision. Uniform random numbers are generated using the cores described in [81]. As mentioned before, two parameters (M_1, M) are used for the parallel RSR and ISR architectures, where M_1 represents the parallel degree of SUM and CUMSUM, and M represents the parallel degree of the offspring evaluation. Accord-

ingly, only one parallel degree M for the Metropolis and Rejection architectures is considered since no sum or cumulative sum is needed for these two algorithms.

3.5.1 Resampling Quality

We first assess the resampling quality of the proposed architectures, i.e. how close the resampled particle set is to the ideal given by (3.2). We use the same simulated weights as the ones used in [65], which is the standard way to assess the resampling quality, and this allows us to easily compare to the results of the GPU implementations in [65]. The simulated weights are generated based on the following equation:

$$w_i = \frac{1}{\sqrt{2\pi}} exp(-\frac{1}{2}(y_e^i - y_o)^2) \tag{3.4}$$

y_o represents the time varying observation value and y_e^i represents the estimated values based on the particles $\mathbf{x}^i$ in the *sampling* step in Algorithm 5. These weights (which are the likelihoods of particles fitting the observation) are produced based on the assumption that the importance function of the PF is a Gaussian, i.e. $y_o \sim \mathcal{N}(y_e^i, 1)$. Multiple weight sets are generated, with varying particle number N and $y = mean(\mathbf{y}_e) - y_o$, i.e. $y_e^i - y_o \sim \mathcal{N}(y, 1)$. The parameter y indicates the relative variance in weights. Increasing y means that the relative variance in weights increases too.

First, the resampling quality of the four algorithms is compared using the RMSE given by the equation (3.3). The RMSE of the SR algorithm in [65] is also shown as a point of reference. Experiments are done for $N = 2^4, 2^5, \ldots, 2^{20}$ and for $y = 0$ to $y = 4$. The results are shown in Figure 3.7 and 3.8. The results lead to the following conclusions: ***1)*** the proposed ISR has the same resampling quality as the original Systematic and RSR; ***2)*** RSR and ISR give a lower RMSE than Metropolis and Rejection resampling, which results from the fact that the Metropolis is a biased sampler and the Rejection is affected by the distribution of uniform random numbers used as the indexes; ***3)*** The RMSE of Rejection is lower than that of Metropolis for small variance in weights, but it converges to the RMSE of Metropolis when y increases.

The resampling quality of the FPGA implementations of ISR and RSR does not change with respect to the degree of parallelism. In contrast, the resampling quality of the parallel implementations of

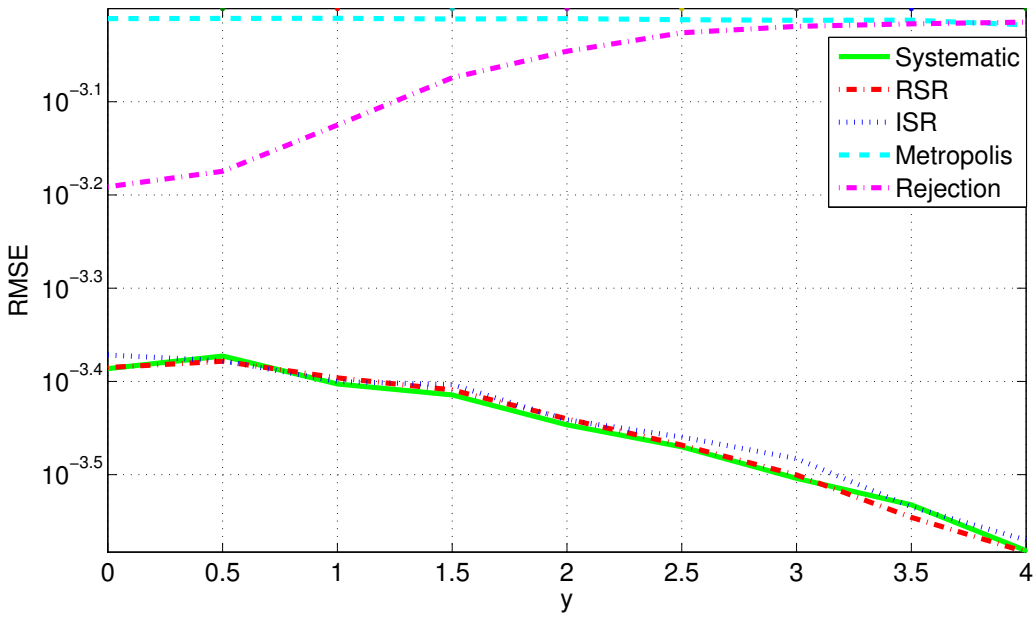


Figure 3.7: *Root-mean-square error (RMSE) of different resampling algorithms for various weight sets having different variances (which are indicated by y) at $N = 2^{10}$.*

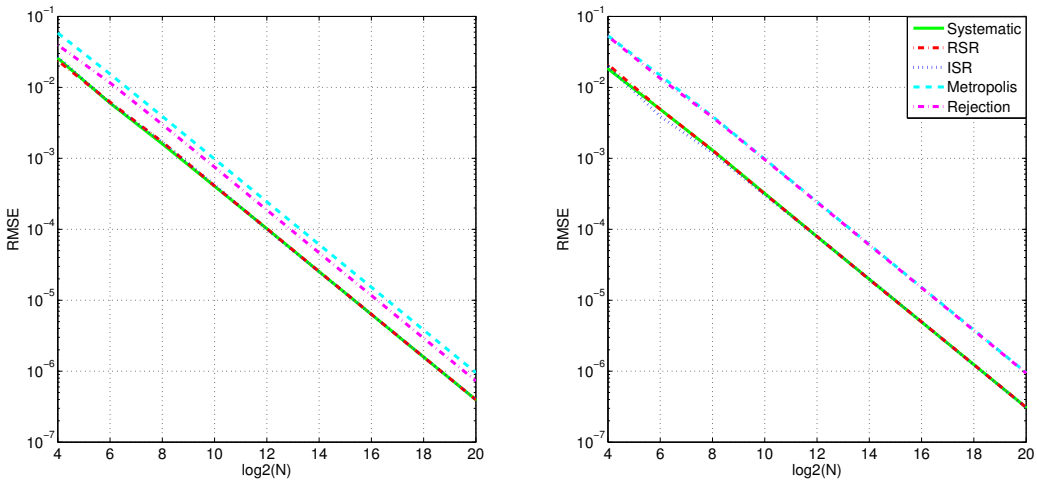


Figure 3.8: *RMSE of resampling algorithms for various particle numbers at $y = 1$ **(left)** and $y = 3$ **(right)**.*

Metropolis and Rejection algorithms can be different compared to the sequential ones, since we use the global memory access strategy described previously. We consider four cases: 1) the sequential implementations; 2) parallel implementations with the Knuth Shuffle RPG; 3) parallel implementations with the simplified RPG; 4) parallel implementations without an RPG. In the last case, each block of Metropolis and Rejection architectures works on a sub-set of weights just like RSR and ISR, and only has access to the corresponding sub-set of weights. In cases 2 and 3 each block has global access to all of the weights as in the sequential one because the RPG is used. We check the RMSE quality of the draws for both algorithms and the average number of proposed indexes (**while** loop

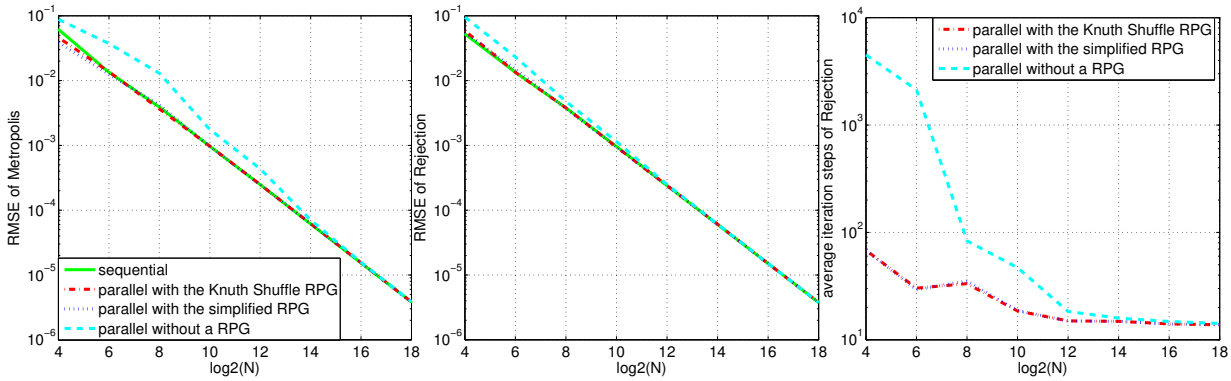


Figure 3.9: *Test results of different memory access strategies for parallel Metropolis and Rejection implementations with the parallel degree $M = 32$ (which is the maximum degree achieved in our target FPGA device with half-utilization of logic):* ***(left)*** *RMSE of Metropolis draws at $y = 4$;* ***(middle)*** *RMSE of Rejection draws at $y = 3$;* ***(right)*** *the average number of iteration steps for each weight of Rejection at $y = 3$.*

iterations in Algorithm 9) until an index is accepted for each weight for Rejection resampling. The average number is taken from 100 independent runs. The results are shown in Figure 3.9.

The results confirm that there is no quality loss using the memory access strategies with the Knuth Shuffle RPG or our simplified RPG for parallel execution of Metropolis and Rejection resampling compared to the sequential execution. Nevertheless, the absence of RPG has a large impact on the resampling quality as the resampling is only performed inside each block. It becomes even worse when the variance of weights i.e. y increases or the weights are sorted. The worst case happens when all the non-zero weights are processed by only one resampling block. Another drawback of not using an RPG for parallel Rejection resampling is that it causes largely increasing execution time as more steps need to run before the acceptance of each weight. In conclusion, the simplified RPG-based memory access strategy permits the parallel algorithms to achieve the same resampling quality as the sequential one. Moreover, the simplified RPG consumes much less resources and has a lower latency compared to the Knuth Shuffle RPG.

3.5.2 Resource Utilization and Execution Time

Table 3.2 gives the resource utilization (measured in floating point adders and accumulators), and the total number of clock cycles for N weights of the parallel SUM and CUMSUM architectures shown in Figure 3.5 when using the parallel degree M_1. L_1 and L_2 are the latency of the adder and accumulator

respectively. All results are post place and route.

Table 3.2: *Resources and Execution time for parallel SUM and CUMSUM algorithms in FPGA (parallel degree M_1)*

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

Table 3.3 shows the resource utilization and total clock cycles (for a complete resampling operation with N weights) for all four architectures, given the parallel degree M_1 and M. The memory utilization is also shown here.

Each architecture uses M blocks for offspring evaluation or index generation, while the RSR and ISR also require resources for SUM or CUMSUM implementation. Metropolis and Rejection architectures need extra resources for the simplified RPG which uses a $M log_2 M$ bits Cyclic shifter and also for multiplexers to select data from the memory based on the outputs of the RPG. Note that the $M-1$ blocks which implement Algorithm 10 can be reused from the offspring evaluation blocks. Regarding memory utilization, ISR architecture needs one more memory to store the cumulative weights compared to the other three architectures. For the RSR, ISR and Metropolis architectures, all the concurrent blocks produce output streams in parallel at the same clock cycle so the outputs from each block can be written to the memory together. Therefore, only one memory unit is needed for these three architectures to store the outputs (offsprings or indexes). However, the Rejection blocks produce output streams at random clock cycles. Therefore, a single memory unit is necessary for each block. In total, M memory units are needed as output memory for parallel Rejection architecture but the total output memory size is the same as that of the other schemes.

Table 3.3 also shows the total number of clock cycles needed to complete the resampling operation for all architectures, given the parallel degree (M_1,M) and the number of particles. The clock cycles needed by RSR and ISR consist of the cycles needed for SUM or CUMSUM (shown in Table 3.2) and the cycles needed for offspring evaluation which is $N/M + L_{Res}$ where L_{Res} represents the latency of offspring evaluation. L_{Res} for RSR is larger when using $M > 1$ because of the additional latency needed for the computation of u_i. The execution times of Metropolis and Rejection resampling are

Table 3.3: *Resources and Execution time of RSR and ISR with parallel degree (M_1, M), of Metropolis and rejection with parallel degree M*

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

$BN/M + L$ and $SN/M + L$ cycles respectively for parallel implementations where L is the latency, B in Metropolis is the iteration steps configured by the user and S in Rejection represents the average number of iteration steps performed before acceptance of each weight. Note that the S for Rejection is not fixed and depends on the weight distribution, and we can estimate S with reasonable accuracy after performing the Rejection numerous times. Both B for Metropolis and S for Rejection resampling can be very large as the variance in weights increases, and this makes these two architectures less efficient for large amounts of particles as will be shown shortly.

Table 3.4 gives the resource utilization (slices, LUTs, etc.) of a single resampling block for each architecture in the target device. For M parallel resampling blocks which are needed for the respective parallel architectures, the required resources can be estimated by multiplying by M (although in the real implementation the total utilization varies slightly due to synthesis and place and route optimizations). Table 3.4 also gives the clock frequency achieved for the non-parallel (i.e. $M = 1$) and parallel ($M = 32$) implementations of each architecture.

Table 3.4: *Resources (of one resampling block) and Clock frequency achieved for each architecture on a Virtex-6 LX240T FPGA*

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

Figure 3.10 shows how the execution time (clock cycles) changes with the amount of utilized re-

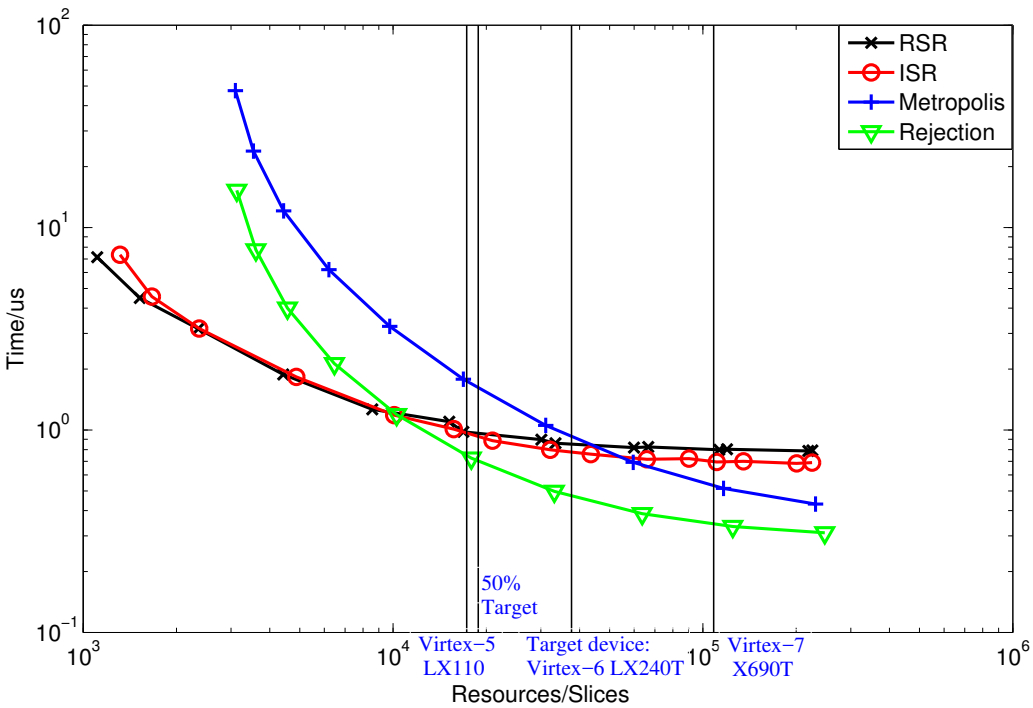


Figure 3.10: *The execution times for $N = 2^{10}$ against the resources of slices on the target device Virtex-6 LX240T of the algorithms at $y = 1$.*

sources (slices) for each architecture at $N = 2^{10}$ and $y = 1$. The total resources of three representative Xilinx FPGAs are also marked in the figure (vertical lines) to provide a reference. The maximum M_1 and M which can be achieved in different devices are easy to obtain according to this figure. For example, with 50% utilization of slices on the target device, we can achieve (M_1,M)=(32,32), (16,32), (-,32), (-,32) for the four architectures respectively, and with full utilization of the device we can achieve (64,64), (32,64), (-,64), (-,64) respectively. The figure shows that RSR and ISR are faster when we utilize fewer resources but become slower than Metropolis and Rejection as more resources become available and the amount of parallelism increases. This is not surprising given the formulas of Table 3.3 as the latency of the SUM and CUMSUM datapaths in RSR and ISR architectures consumes a big proportion of the total time.

To reserve resources for the other stages of particle filter, 50% utilization of the FPGA logic resources is assumed. Given this assumption, Figure 3.11 shows how the execution times of the four architectures change with N for two cases of variance ($y = 1$ and $y = 3$). It is worth noting that the execution time of RSR and ISR does not change with y when N is fixed. In contrast, the execution time of Metropolis and Rejection changes dramatically because the number of iterations increases considerably when y increases.

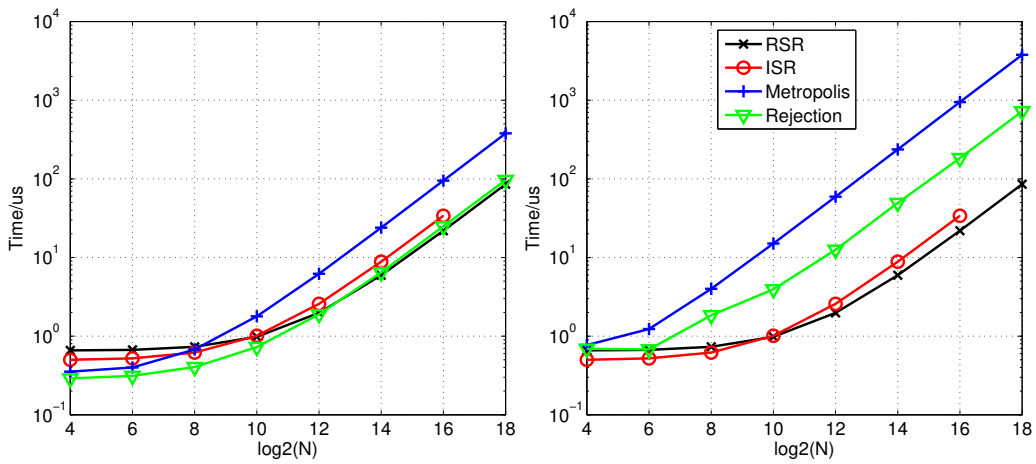


Figure 3.11: *The execution times against the particle numbers N at $y = 1$ **(left)** and $y = 3$ **(right)** in FPGA with less than 50% utilization of the target device.*

Figure 3.12 shows the speedup of FPGA resamplers versus the respective implementations on an NVIDIA K20 GPU device presented in [65] for the same two cases of y. The speedups for Metropolis resampling and Rejection resampling are in the order of 1.7x-25x and 2.3x-49x respectively for large particle numbers ($N \geq 2^{10}$), while the minimum speedups of RSR and ISR are 5.8x and 8.9x respectively. Another observation is that GPU can provide comparable performance to the FPGA for large numbers of particles, but not for small numbers of particles. This happens because the GPU cannot achieve full thread utilization unless we use a massive amount of particles (see [65]), while the FPGA is able to utilize a high percentage of its resources even for small problems. Therefore FPGAs can give high speedups for $N \leq 2^{10}$ as shown in Figure 3.12. As a reference, the CPU time of each algorithm can be found in [65].

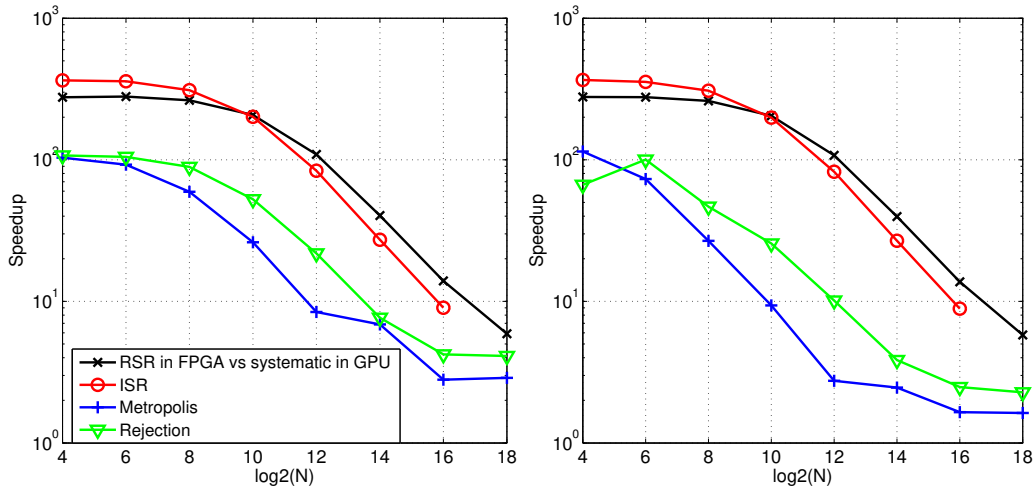


Figure 3.12: *The Speedup of FPGA implementation of the target device with 50% utilization over the GPU implementation in [65] at $y = 1$ **(left)** and $y = 3$ **(right)**.*

3.6 Discussion and Conclusion

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

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

Finally, discussions and considerations on how to utilize the proposed parallel resampling architectures in the distributed particle filter system for real applications are provided in Chapter 6.

Chapter 4

An Unbiased MCMC FPGA-based Accelerator Under Custom Precision Arithmetic

SMC and MCMC methods are the two main tools to sample from high dimensional probability distributions in Bayesian inference [2]. Unlike SMC, MCMC methods are often used in static models with time-invariant parameters. The main computation in the MCMC algorithms lies in the posterior probabilities of the model parameters for a given set of observations. In Chapter 4 and Chapter 5 of the thesis, we focus on the optimization of MCMC algorithms in algorithmic modifications and architectural optimizations. Here we propose a mixed precision MCMC algorithm which is suitable for FPGA implementation. With the proposed architecture, significant speedups compared to existing FPGA- and CPU-based works that utilise double floating point arithmetic can be achieved, while still guaranteeing asymptotically unbiased estimates.

4.1 Introduction

Bayesian methods play a central role in modern Machine Learning mainly due to their ability to capture uncertainty in parameter estimation [4]. A key step in Bayesian inference is the sampling from

an arbitrary probability distribution [53, 71, 32, 85]. Markov chain Monte Carlo (MCMC; Chapter 6 of [71]) method is one of the most popular and successful tools to draw samples effectively from arbitrary probability distributions in Bayesian inference problems. For this reason, it has been widely used in a range of statistical applications, including computational physics, population genetics and statistical classifications [53, 32, 62].

The MCMC algorithms allow sampling from a large class of distributions and scale well with the dimensionality of the sample space. They are often used to tackle the problem of sampling from a probability distribution known up to a normalizing constant, with the purpose of using the generated samples to estimate otherwise intractable integrals (this task is known as Monte Carlo integration). For the estimation of the above integrals, the MCMC algorithms need to estimate how well the data are explained by the sampled parameters (i.e. likelihood function estimation), which becomes the dominant computational bottleneck when large datasets are targeted. Thus, speeding up the likelihood computation has attracted the focus in academia and industry in order to allow the application of MCMC to models with large-scale dataset. Currently, the lack of sufficiently fast MCMC methods limits their applicability in many modern applications like genetics and machine learning, and this situation is bound to get worse given the increasing adoption of big data in many fields of industry and research.

This challenge has motivated approximate MCMC approaches [37, 38, 33, 82] that are based on approximations of the target distribution. A summary and review of current MCMC methods used for large datasets can be found in [9]. Most of them tend to use subsampling based approaches to provide a faster estimation of the likelihood for only a subset of the whole data [48, 8, 58, 69]. Other recent works focus on the computation engine, and investigate the calculation of the likelihood function based on custom precision arithmetic in order to achieve low latency and allow for more parallelism for a given set of hardware resources. However, both approaches lead to biased estimates that exhibit large variance due to the approximations of the target distribution. Even though a controlled biased estimate can be accepted in certain applications [63], there is a large number of applications where unbiased estimates of the given parameters of the sampling distribution are required, and MCMC algorithms are expected to perform exact inference in these problems [20].

In this Chapter we are focusing on problems that are compute-bound, having the evaluation of the likelihood function as the limiting performance factor. Towards addressing this problem, a novel MCMC construction is proposed, the custom-precision firefly MCMC (CF-MCMC), that samples from the exact posterior distribution even though it operates under a custom precision regime, and its implementation in an FPGA device. The key idea behind this work, that enables custom precision arithmetic in the computation of the likelihood function, is the introduction of an extra parameter in the problem parameter space that models the mode of the computations, i.e. the arithmetic precision, in the calculation of the likelihood function. This chapter shows that by properly sampling the new augmented space, unbiased estimates of the parameter of the distribution are computed even though part of the computations are performed under custom precision arithmetic. Furthermore, we extended the work of the custom precision design in FPGA by (1) investigating and comparing alternative custom precision likelihood construction approximates targeting improved performance (i.e. effective samples per second) and (2) proposing a method to maximize the performance of the algorithm by selecting the optimal arithmetic precision based on performing short MCMC pre-runs on a set of candidate precisions. A summary of the main contributions of this work are as follows:

- A novel mixed precision MCMC algorithm which leads to unbiased estimates, taking into account the unique custom precision capabilities of FPGAs;
- A novel architecture which maps the algorithm to an FPGA. The architecture includes the necessary data structures and sampling mechanisms to accommodate the use of the auxiliary binary variables. With the proposed data structure for storing the auxiliary variables, both high and low precision data paths can be fully pipelined, and thus further improving the throughput;
- A novel methodology for the construction of tight lower bound functions of the target probability distribution function based on the selection of the rounding mode of the FPGA arithmetic operators in combination with verification tools for modelling numerical behaviour (i.e. Gappa++) in order to maximise the performance of the proposed algorithm;
- A methodology for selecting the custom arithmetic precision of the system that would maximise its performance based on the system's performance model and the estimates of the parameters from pre-runs.

4.2 MCMC Basics

In scientific computing, we often need to compute some integrals in a very high dimensional space such as:

$$I(f) = \int f(\theta)p(\theta)d\theta \tag{4.1}$$

The probability distribution $p(\theta)$, i.e. the target density, can be a distribution from statistical physics or a conditional distribution arising in data modelling - for example, the posterior probability of a model's parameters given some observed data, where $f(\theta)$ is the function of interest.

In the field of statistics, these integrals are vital for calculating the expectation or expected values of distributions. However many functions and distributions cannot be integrated analytically especially for higher-dimensional integrals. For most probabilistic models of practical interest, these expectations cannot be evaluated by exact methods. In these cases, a general and powerful framework, i.e. the Markov chain Monte Carlo (MCMC) method, is employed, which can be used to generate samples from any given probability distribution. Using the generated samples, the integral $I(f)$ can be approximated by tractable sums that converge (as the number of samples N_s tends to infinity) to $I(f)$. The following central limit theorem holds for suitable test functions f under weak assumptions [24]:

$$\tilde{I}(f) = \frac{1}{N_s}\sum_{n=1}^{N_s} f(\theta_n) \longrightarrow \text{Normal}(I(f), \sigma^2_{lim}(f)) \tag{4.2}$$

i.e. the sum is an asymptotically unbiased estimator of the integral $I(f)$ [71].

MCMC generates samples from the probability distribution $p(\theta)$ by sequentially constructing a Markov chain that satisfies (4.2). In practice it is often advisable to discard some initial states of the chain (throwing away a number of iterations at the beginning of an MCMC run is often called "burn-in"), in order to reduce the initialisation bias. In this work, the parameters of interest are denoted by θ of D-dimensions, and it is assumed that N data points $\{x_n\}_{n=1}^N$ (with each component x_n as a vector) have been observed. An MCMC sampler makes transitions from a given θ to a new θ' such that the posterior distribution $p(\theta \mid \{x_n\}_{n=1}^N)$ remains invariant. Consider the most commonly used MCMC algorithm (Metropolis MCMC; Chapter 7.3 of [71]) in Algorithm 13. In each iteration, a proposed

move of the chain is considered, by using a proposal such as a Gaussian random walk (line 2) to generate the new θ' that is accepted or rejected with the probability based on the ratio of the posterior probabilities (i.e. how well the new value explains the data) (line 4-9). The main computation load lies in the evaluation of the full posterior probability at every iteration in line 3. Using the Bayesian theorem and assuming that the data $\{x_n\}_{n=1}^{N}$ are *i.i.d.* (it is often assumed in real applications) and θ has the prior $p(\theta)$, the posterior distribution breaks down into a product of the likelihood of each data point i.e. $p(x_n \mid \theta)$ as:

$$p(\theta \mid \{x_n\}_{n=1}^{N}) \propto p(\theta) \prod_{n=1}^{N} p(x_n \mid \theta) \tag{4.3}$$

For notational convenience, we write the nth likelihood term as

$$L_n(\theta) = p(x_n \mid \theta) \tag{4.4}$$

Algorithm 13: Metropolis MCMC

Input: initial setting θ_0, number of samples N_s;
Output: parameter samples $\theta_i, i = 1, ..., N_s$;

1: **for** $i = 1$ **to** N_s **do**
2: Propose $\theta' \sim \theta_{i-1}$+Normal(0, $s^2 I_D$); // a random walk proposal with step size s.
3: Compute $a = \dfrac{p(\theta' \mid \{x_n\}_{n=1}^{N})}{p(\theta_{i-1} \mid \{x_n\}_{n=1}^{N})}$;
4: $u \sim$ Uniform(0,1);
5: **if** $u \leq a$ **then**
6: $\theta_i = \theta'$;
7: **else**
8: $\theta_i = \theta_{i-1}$;
9: **end if**
10: **end for**

Although MCMC generates statistically consistent samples from the target distribution, the samples are correlated due to the use of a Markov chain. This dependence leads to an increase in asymptotic variance σ_{lim}^2 of the MCMC estimate in (4.2), compared to the case where independent samples of the target distribution are used. This loss in efficiency can be quantified by the Effective Sample Size (ESS) [40] in (4.5):

$$ESS = N_s/(1 + 2\sum_{j=1}^{k} \rho(j)) \tag{4.5}$$

where N_s is the number of post burn-in MCMC samples and $\sum_{j=1}^{k} \rho(j)$ is the sum of the first k monotone sample autocorrelations. The ESS estimates the "effective" number of samples, which is always lower than N_s. Thus the adopted performance metric for MCMC samplers is ESS/sec, which combines raw sampling speed (runtime) and ESS [40].

4.3 Mixed Precision MCMC Methodology

4.3.1 Custom-Precision Firefly MCMC (CF-MCMC)

On each iteration of MCMC, the likelihood term for each data point must be evaluated to obtain the target density, which is the most computation expensive part of the algorithm. [58] proposed Firefly Monte Carlo (FlyMC), which introduces an auxiliary variable for each observation which determines whether it should be included in the exact evaluation of the posterior distribution or not. A lower bound function for each likelihood term caters for the observations that are not included in the evaluation of the posterior, and an extra sampling step is included in the algorithm in order to sample the above indication parameter. As such, FlyMC generates samples from the exact target posterior rather than from an approximation distribution. Nevertheless, the drawback of FlyMC is that useful lower bounds can be difficult to obtain for many problems. Moreover, [58] have shown that the algorithm's performance depends on the tightness of the bound; it only achieves significant gains when computational light and tight bounds are applicable. The idea of using lower bounds to reduce the cost of MCMC has been exploited previously in [59]; [9] (Section 4.3) propose construction of the lower bound that avoids specifying a resampling fraction, but it requires the integrals of the exponents of the lower bound functions to be tractable.

This work is based on the same principle as the FlyMC, but the introduced auxiliary parameter is utilised to indicate whether or not the likelihood computation for each data point is performed under double precision or custom precision regime. Thus, instead of requiring the derivation and use of approximate functions for the likelihood terms, the work utilises custom precision approximations and utilize precision-related tools to guarantee that these approximations are indeed a lower bound

to the true likelihood term (which is a requirement for [58] to generate samples from the posterior distribution), removing the need to manually design the approximation function as in FlyMC. Thus, the proposed framework produces lower bounds automatically regardless of the class of problem.

In the rest of this chapter, $LD_n(\theta)$ and $LC_n(\theta)$ denote the double precision likelihood term and the custom precision lower bound of the likelihood of the nth data point, respectively. For each data point x_n, a binary auxiliary variable $z_n \in \{0, 1\}$ is introduced, indicating the type of the likelihood term computation i.e. double or custom precision. Assuming that $LC_n(\theta)$ has been constructed such as it is always less than the double precision likelihood $LD_n(\theta)$, i.e. $LC_n(\theta) \leq LD_n(\theta)$ (such construction is shown later on in the Chapter), then each z_n is modelled to have the following Bernoulli distribution conditioned on the relative difference between these two precision values:

$$z_n \sim Bernoulli(1 - LC_n(\theta)/LD_n(\theta)) \tag{4.6}$$

The augmented posterior distribution is shown below:

$$p(\theta, \{z_n\}_{n=1}^N \mid \{x_n\}_{n=1}^N) \propto p(\theta) \prod_{n=1}^N p(x_n \mid \theta) p(z_n \mid x_n, \theta) \tag{4.7}$$

As in other auxiliary variable methods, this augmentation does not damage the target distribution in (4.3):

$$\begin{aligned} &\sum_{z_1} \cdots \sum_{z_N} p(\theta) \prod_{n=1}^N p(x_n \mid \theta) p(z_n \mid x_n, \theta) \\ &= p(\theta) \prod_{n=1}^N p(x_n \mid \theta) \sum_{z_n} p(z_n \mid x_n, \theta) \\ &= p(\theta) \prod_{n=1}^N p(x_n \mid \theta) \end{aligned} \tag{4.8}$$

Therefore, the marginal distribution over θ in (4.7) is still the correct posterior distribution given in Equation (4.3).

Consider each product of the joint distribution:

$$
\begin{aligned}
& p(x_n \mid \theta) p(z_n \mid x_n, \theta) \\
& = LD_n(\theta)[\frac{LD_n(\theta) - LC_n(\theta)}{LD_n(\theta)}]^{z_n}[\frac{LC_n(\theta)}{LD_n(\theta)}]^{1-z_n} \\
& = \begin{cases} LD_n(\theta) - LC_n(\theta) & \text{if } z_n = 1 \\ LC_n(\theta) & \text{if } z_n = 0 \end{cases} .
\end{aligned} \tag{4.9}
$$

For simplicity, we call the data points with their $z_n = 1$ as "bright data" and those data points with their $z_n = 0$ as "dark data" to follow FlyMC terminology. Please note that the double precision likelihood $LD_n(\theta)$ now only appears in those bright data. At any given iteration, we only compute their likelihoods in reduced precision for the dark data. Therefore, the full likelihood is now given by:

$$
L(\theta) = \prod_{i}^{z_i=1} (LD_i(\theta) - LC_i(\theta)) * \prod_{j}^{z_j=0} LC_j(\theta) \tag{4.10}
$$

This algorithm can be seen as shifting the computational burden from evaluating $LD_n(\theta)$ to evaluating $LC_n(\theta)$ plus a step to sample this new parameter. The computational gains are coming from evaluating some likelihoods in custom precision instead of utilising double precision in all likelihood evaluations.

For the rest of the thesis, the above proposed algorithm is called custom-precision firefly MCMC (CF-MCMC) algorithm and its steps are shown in Algorithm 14. The overhead of introducing a sampling stage of the auxiliary variable z_n, has a small penalty in the performance of the algorithm as this resampling is performed only for a random fixed-size subset of the data [58]. This results from the fact that at every iteration most of the binary variables are kept unchanged. The sampling step for z_n is shown in lines 8-11 and 14-19 of Algorithm 14, which is performed immediately after the computation of the likelihood. Since the likelihoods of the bright data points have already been evaluated in the MCMC step of line 7, the implementation of the algorithm can reuse these values and resample all the instances that correspond to "bright data" points without any extra computational cost. As only few z_n variables that $z_n = 0$ change in each iteration (assuming a tight lower bound), the resampling of the dark points' variables is performed at a fixed rate (1/ResampleFraction as shown

in line 15), to avoid computing the full precision likelihoods for all the dark data in each iteration[2]. The above partial resampling leads to a chain with slower mixing rate. However, as indicated in [58], the approach works well in practice as the bottleneck for mixing is usually in the space of θ. For a given budget in likelihood evaluations, allowing more steps in the θ space is intuitively likely to reduce initialization bias faster than resampling all variables at each iteration.

Algorithm 14: CF-MCMC Algorithm

Input: initial setting θ_0 and $\{z_n\}_{n=1}^{N}$, N_s;
Output: parameter samples $\theta_i, i = 1, ..., N_s$;

1: **for** $i = 1$ **to** N_s **do**
2: Propose $\theta' \sim \theta_{i-1}$+Normal(0, $s^2 I_D$);
3: $L(\theta') = 1$; // likelihood initialization
4: **for** $n = 1$ **to** N **do**
5: $u_1 \sim$ Uniform(0,1);
6: **if** $z_n = 1$ **then**
7: // likelihood computation for bright data
$L(\theta') = L(\theta') * (LD_n(\theta') - LC_n(\theta'))$;
8: // z_n sampling
9: **if** $1 - LC_n(\theta')/LD_n(\theta') \leq u_1$ **then**
10: $z_n = 0$;
11: **end if**
12: **else**
13: // likelihood computation for dark data
$L(\theta') = L(\theta') * LC_n(\theta')$;
14: // partial sampling of z_n for dark data
15: **if** $n\%ResampleFraction = RandInteger(1, ResampleFraction)$ **then**
16: **if** $1 - LC_n(\theta')/LD_n(\theta') > u_1$ **then**
17: $z_n = 1$;
18: **end if**
19: **end if**
20: **end if**
21: **end for**
22: Compute $a = \dfrac{L(\theta')}{L(\theta_{i-1})}$;
23: $u_2 \sim$ Uniform(0,1);
24: **if** $u_2 \leq a$ **then**
25: $\theta_i = \theta'$;
26: **else**
27: $\theta_i = \theta_{i-1}$;
28: **end if**
29: **end for**

[2]Setting the value of ResampleFraction is discussed in Section 4.4.

4.3.2 Lower Bound Function Construction

In order for the samples to come from the original posterior distribution when the augmented posterior distribution is utilised, the custom precision likelihood $LC_n(\theta)$ is required to be a lower bound on the full precision likelihood $LD_n(\theta)$, i.e. $LC_n(\theta) \leq LD_n(\theta)$. [58] uses specific expressions (distribution classes) for the lower bound. In order to achieve this in a custom precision setting, we firstly proposed to use the tool Gappa++ [52] which determines and verifies numerical behaviour, and particularly rounding error in computations with floating point operations. The tool manipulates logical formulas stating the enclosures of expressions in some intervals. In particular, Gappa++ allows bounding computational errors due to floating point arithmetic. It works effectively and fast across a range of function constructions and especially for the linear functions. For most problems it takes less than a minute to obtain the precision-related error bound [52].

Let's denote the maximum absolute error bound between two floating point precision constructions of $p(x_n \mid \theta)$, one under double precision arithmetic (i.e. $LD_n(\theta)$) and one under a custom precision $p(x_n \mid \theta)_c$, that is provided by the Gappa++ tool as ε, where $\varepsilon \geq 0$ (i.e. $|LD_n(\theta) - p(x_n \mid \theta)_c| < \varepsilon$). Then, $LC_n(\theta)$, is defined as:

$$LC_n(\theta) = p(x_n \mid \theta)_c - \varepsilon \tag{4.11}$$

which ensures that $LC_n(\theta) \leq LD_n(\theta)$, i.e. that $LC_n(\theta)$ is a lower bound of $LD_n(\theta)$.

The tightness of the lower bound construction is important to the performance of the CF-MCMC algorithm because it impacts the number of bright data points at each iteration, which essentially determines the execution time of the CF-MCMC algorithm. In the first method proposed to construct the lower bound, Gappa++ was used solely in order to obtain the lower bounds for the likelihood function. However, the tightness of the bounds provided by Gappa++ (i.e. ε) depend on the actual operations involved in the function under investigation. [52].

For this reason, we propose an alternative lower bound function construction in order to provide a tighter custom precision bounds, further boosting the performance of the algorithm. The second approach capitalises on the fact that in FPGA designs the user can tune the rounding modes of the floating point operators. As such, by appropriately tuning the rounding mode of the operators under

a custom precision implementation, the user can guarantee a lower bound by construction. To take advantage of this, we separate the parts of the likelihoods for which lower bound guarantees are obtained by construction (for example in the case where there is an addition of two positive quantities) and to parts for which this is not possible and their error bounds are estimated through Gappa++. The proposed lower bound function design allows utilization of the Gappa++ tool in combination with the rounding mode configuration of the arithmetic operators on FPGAs, producing tighter lower bounds for a given custom precision, with respect to the existing methodology.

Given the logistic regression likelihood function in Equation (4.12) as an example, the first proposal of the lower bound function (4.13) is based on the error bound ε_1 of the whole function which is provided by Gappa++.

$$L_n(\theta) = \frac{1}{1 + exp\{\theta^T x_n\}} \tag{4.12}$$

$$LC_n(\theta) = \frac{1}{1 + exp\{\theta^T x_n\}} - \varepsilon_1 \tag{4.13}$$

Regarding the second proposal, we firstly use Gappa++ to obtain the rounding error ε_2 of the dot product operation inside the exponent operation. Then we add ε_2 to the custom precision dot product values. Secondly, we set specific rounding modes (round up or round down) for the other operators that are monotonic, in order to guarantee the final result is a lower bound. This alternative lower bound function can be shown as the following equation:

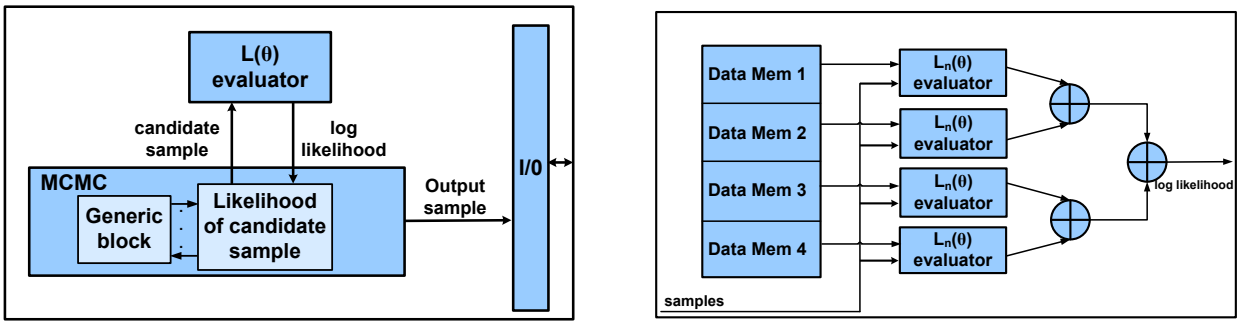
$$\begin{aligned} LC_n(\theta) =& div(1, add(1, exp(\theta^T x_n + \varepsilon_2, \textbf{RoundUp}), \\ & \textbf{RoundUp}), \textbf{RoundDown})) \end{aligned} \tag{4.14}$$

where the rounding modes of the division, addition and exponent operations are set to round down, round up and round up respectively.

4.4 FPGA Implementation

4.4.1 Proposed Hardware Architecture

FPGA devices have been considered by researchers and practitioners for MCMC acceleration because of their ability to implement many processing elements for the likelihood calculation, as well as due to their flexibility to implement any custom arithmetic precision regime. Assuming that the data points can fit in the on-chip memory blocks (an assumption that will be lifted later on), an FPGA system that implements an MCMC sampler is given in Figure 4.1a, where high memory bandwidth that matches the computational capabilities of the processing elements (for likelihood evaluation) is provided through the on-chip memories.



(a) *The overall architecture* **(b)** *The parallel likelihood architecture*

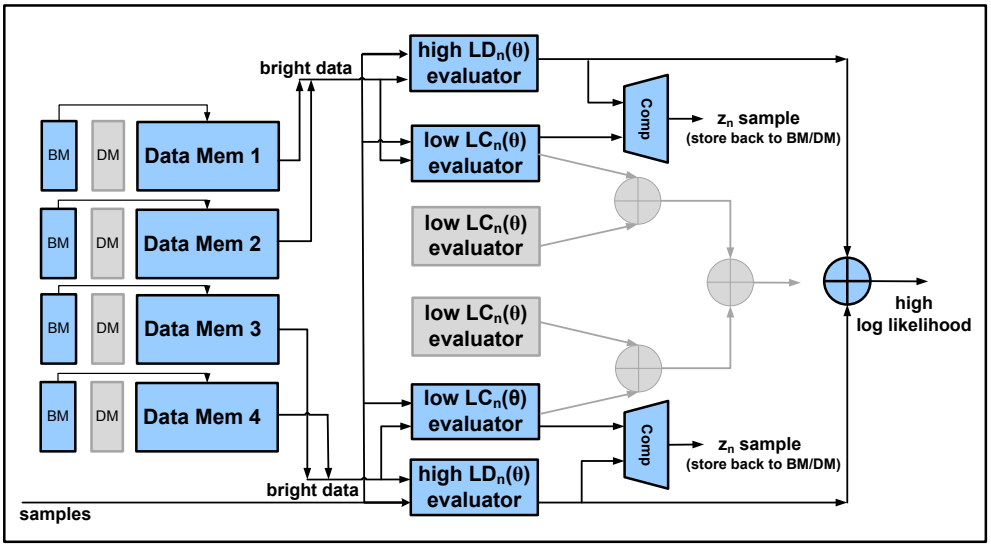
Figure 4.1: *(a) The overall architecture of the FPGA-mapped MCMC sampler which mainly contains the generic and likelihood $L(\theta)$ evaluator block. (b) The architecture of double-precision floating point likelihood $L(\theta)$ evaluator design with the conventional parallel implementation at $P = 4$.*

The FPGA-mapped MCMC sampler (not considering the off-chip memory access) generally contains two blocks as shown in Figure 4.1a: a hardware block for the generic MCMC operations (i.e. propose new sample, accept/reject ratio calculation) and a block to compute the full likelihood $L(\theta)$ in the logarithmic domain in order to avoid numerical instability in the evaluation of the likelihood [78]. The generic block in Figure 4.1a corresponds to the operations in line 2, 5 and line 23 to 28 of Algorithm 14. Because likelihood evaluations dominate the computational cost, the performance of the MCMC sampler can be improved by implementing many parallel likelihood evaluation blocks. When the likelihood evaluation can be decomposed into sub-components due to i.i.d assumption of the data (which is also assumed in the thesis), FPGA implementations typically use a likelihood evaluation

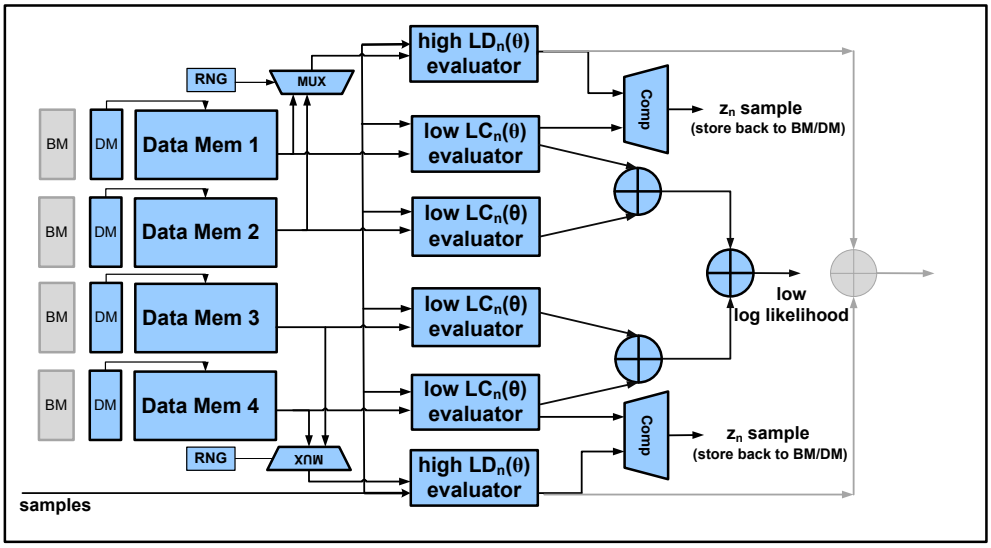
block which consists of parallel likelihood modules [55]. Let's denote the number of modules by P, and an example of the conventional double-precision floating point design at $P = 4$ is given in Figure 4.1b. Accordingly the data memory is partitioned into P blocks, thus each evaluation block processes one block of data. Finally the total sum (i.e. the full log likelihood) is computed by combining the outputs of the P blocks. For compute-bound tasks (as the one considered in this work), the goal is to maximize the number of parallel blocks within the available resources in the FPGA device, minimizing as such the execution time of a single MCMC iteration. This motivates the idea in this Chapter to implement low precision data paths in order to save computational resources and increase the sampling throughput. Generally the low precisions can be any arithmetic precisions smaller than double floating point, such as single floating point or fixed point. In this work, we use the floating point arithmetic with different significand bits as the low precisions and this will be shown later.

The generic architecture designed for the CF-MCMC algorithm which utilizes multiple high-precision datapaths is presented here and it is depicted in Figure 4.2. We denote each parallel degree of the high and low precision datapaths as P_H and P_L respectively ($P_H < P_L$). Accordingly, the data are stored in P_L memories and each data memory is attached with a set of BM and DM memories to store the indexes of the bright and dark data points respectively. Rather than storing the binary value of each z_n, we store the indexes of the bright and dark binary variables separately in the two independent memories (BM and DM). Also, for each memory, the system keeps track of the total number of bright and dark points.

For each iteration, the system needs to perform the likelihood computation and z_n sampling for the bright and dark points. Thus two steps (as shown in Figure 4.2) are performed in sequence to accept/reject the proposed sample. First, in Step 1, the system accesses P_H BM memories in parallel to use the bright data index to access the bright data in the corresponding data memories, which are then passed to the high-precision data paths to evaluate the sum of the log likelihood of the data points with $z_n = 1$. At the same time, these bright data are also passed to the first P_H low-precision data paths to compare the difference between these two values $LD_n(\theta)$ and $LC_n(\theta)$ in order to update z_n required for the next iteration. Therefore, the other $(P_L - P_H)$ low-precision datapaths remain idle (appear light gray in the figure) during this step. The above process continues till all the bright data points are processed. Then, in Step 2, the P_L DM memories are read in parallel to access the dark data points



Step 1 – Bright data likelihood computation and z_n sampling



Step 2 – Dark data likelihood computation and partial z_n sampling

Figure 4.2: *The architecture of CF-MCMC algorithm using mixed precision design at $P_H = 2$ and $P_L = 4$. The full likelihood is computed and the binary variables are updated by two steps: 1) Bright data likelihood computation in parallel using 2 blocks and z_n sampling; 2) Dark data likelihood computation in parallel using 4 blocks and partial z_n sampling. The light gray blocks indicate they remain idle at the corresponding step.*

in parallel, which are passed through the P_L low-precision data paths. Accordingly, at every cycle, P_H out of the P_L dark data will be chosen randomly to go through the P_H high-precision data paths for updating the corresponding z_n. Therefore only P_H/P_L of the dark variables are resampled, and all the data paths are fully utilised in this second step. Since the dark data can be resampled at a fixed fraction rate as mentioned previously, the system samples the dark data at the fraction rate P_H/P_L based on the degree of parallelism that has been achieved for the high and low precision paths, in order to maintain all data paths busy and maximize utilization.

As many applications target sets of data that do not fit in the on-chip memory of FPGAs, external memories are utilised to store the data. In such situation, the system segments the data set into smaller subsets, and operates on them in a sequential order until all the subsets are processed. Standard techniques can be applied such as double buffering, in order to match the computational and memory bandwidth capabilities of the system. The overall processing on the FPGA device remains the same, but for every iteration the system needs to transfer data from the external memory. As a result, compared to the on-chip memory which can be directly addressed and accessed by the datapaths, the communication between the FPGA and off-chip memory can limit the FPGA performance if the memory bandwidth is not enough to constantly feed the processing elements.

4.4.2 Intelligent Data Distribution

In order to maximise the performance of the system, the bright data and dark data points need to be equally distributed in the memories, otherwise there may be a deviation in the utilization of the datapaths when executed in parallel. This work proposes a methodology based on proportional allocation for redistributing the data to the memories in an intelligent way in order to maximize the performance of the system. The goal is to rebalance the dark data points across the available memories, in order to reduce the overall latency of each iteration of the MCMC algorithm.

Assuming the case where all data can fit in the on-chip memories, the system can introduce a rebalancing step after each iteration in order to minimize the latency of each iteration by maximizing the utilisation of the processing elements.

At every point in time, the system keeps track of the number of total bright and dark points stored in each memory block. After each iteration, the system checks the percentage of the bright or dark points stored in each memory block against the average number of bright and dark points respectively in the system. Then, a reshuffle of the data points is initiated in order to result in memory blocks that have equal proportion of dark points.

To briefly demonstrate the benefit of rebalancing the dark data points among memories, assume that we have two memories and each memory contains M_1 and M_2 dark data ($M_1 > M_2$) at one iteration. Let C_{PE} denote the input throughput of the evaluation block in cycles. The execution time (in clock cycles) to process these data points without data distribution is $T_1 = max(M_1 C_{PE}, M_2 C_{PE})$, where if a distribution step is introduced to the system, the execution time T_2 is given by

$$T_2 = \frac{M_1 - M_2}{2} C_M + \frac{M_1 + M_2}{2} C_{PE}$$

where C_M models the clock cycles needed to transfer a data point from one memory to another, and $\frac{M_1 - M_2}{2} C_M$ models the time taken for data distribution. Assuming that one data point can be read/stored in the memory in every cycle (i.e. $C_M = 1$), it is easy to show that $T_1 \geq T_2$ always holds if $C_{PE} \geq 1$. The above implies that it always pays off to rebalance the dark data points, when it takes more than one clock cycle to consume a data point and the data points can be transferred from one memory block to another in one clock cycle. The above model is used to decide whether a redistribution of the data would improve the performance of the system.

In the case where off-chip memories are used to store the data, the above redistribution of the data takes place when the data are transferred from the external memory to the on-chip memories on FPGA, removing any time penalty imposed by the distribution of the data as a separate process as in the case where the data are stored in on-chip memories.

4.4.3 Performance Model

In this section, an analytical performance model of the system is derived in order to reason on how the selected custom precision impacts the execution time of the system. The total processing time (in

cycles) of the MCMC method for generating N_s samples consists of the time spent for performing the MCMC sampling T_{MCMC}, assuming the data are already on chip, and the time required to transfer the data on-chip/off-chip, $T_{transfer}$, which is shown in Equation (4.15).

$$T_{total} = T_{MCMC} + T_{transfer} \tag{4.15}$$

In the case of a double-precision MCMC design (i.e. baseline), a sample is generated every N/P_{DP} clock cycles. Thus, the total time spent for generating N_s samples is:

$$T_{DP-MCMC} = N_s * N/P_{DP} \tag{4.16}$$

where N is the number of the data points and P_{DP} is the parallelism of double-precision MCMC design. In comparison, the time needed for the CF-MCMC architecture in total is given by:

$$T_{CF-MCMC} = N_s * (N\alpha/P_H + N(1-\alpha)/P_L) \tag{4.17}$$

where α is the proportion of bright data, and P_H, P_L are the parallelism of the high- and low-precision paths respectively.

In the case where no off-chip memory is used, $T_{transfer}$ can be omitted otherwise $T_{transfer}$ can be modelled as:

$$T_{transfer} = \frac{N_s * (N * D * sizeof(data) + N * \lceil logN \rceil)}{bandwidth} \tag{4.18}$$

where D is the dimension of the each data point and $\lceil logN \rceil$ is the bit-width of the index for the $z(n)$ variables.

Please note that the above execution time T_{total} only refers to the raw execution time (i.e. time needed to generate N_s samples) of the corresponding MCMC sampler. As mentioned in Section 4.2, the sampling efficiency metric which is used to compare the performance of different MCMC algorithms and their implementations also needs to include the effect of sample dependency using the ESS metric given in (4.5). The rate of effective samples per clock cycle can be derived by dividing ESS by the execution time i.e. ESS/T_{total}, which can also be seen as the effective throughput of the

MCMC system.

4.5 Custom Precision Tuning

Even though the above construction guarantees unbiased estimates for any adopted custom precision, the selected custom precision has impact on the performance of the system and needs to be tuned. In order to achieve the maximum performance in terms of effective throughput, the designer needs to consider the impact of custom precision selection to the parallelisation factor achieved, as well as the percentage of bright data points during the execution of the algorithm. As the precision is reduced, more parallelism can be obtained for a set of resources, and thus reducing the total execution time of the system; however the percentage of the bright data points increases accordingly, as the gap between the lower bound function and the target likelihood function increases, which in turn introduces additional runtime as more high-precision computations need to be performed. Here, the work focuses on exploiting the optimal precision selection, in order to maximise the performance of the system.

In this work, a static analysis selection method is proposed by modelling the processing time and resources versus the utilised custom precision. Please note that the ESS cannot be modelled during static analysis, so this work aims to maximise samples per cycle that are generated by the system (i.e. raw speed). The proposed methodology consists of two steps:

1) Resources v.s. Precision

The first step is to compute how the resource utilisation varies with the custom precision of the system. This step only requires the pre-synthesis of the floating point IPs under different precisions on FPGA in order to estimate the total resource utilisation of the likelihood function evaluation block under different custom precision regimes. Then, for a give target FPGA device the maximum achievable parallel degree, i.e. P, can be obtained for each custom precision candidate.

2) Bright Data v.s. Precision

The second step needs to consider the percentage of the bright data points during the execution of the

algorithm in order to estimate the optimal configuration of the system. An estimate of the number of bright data points M, is given by:

$$M = \sum_{n=1}^{N} \int p(\theta \mid \{x_n\}_{n=1}^{N}) \frac{LD_n(\theta) - LC_n(\theta)}{LD_n(\theta)} d\theta \tag{4.19}$$

However, in order to estimate the above quantity, we need to draw samples from the actual target distribution $p(\theta \mid \{x_n\}_{n=1}^{N})$, which is the reason for the design of such system. Following [63], M can be estimated by short MCMC pre-runs. As the $\frac{LD_n(\theta) - LC_n(\theta)}{LD_n(\theta)}$ factor in equation (4.19) takes small values, the variance of the estimation of M will drop down fast as the number of samples increases. Here we propose that M is estimated using short FPGA-mapped pre-runs, taking advantage at the same time of the parallelism offered by FPGA devices across the different runs, as have been demonstrated in [63].

Utilising information from these two steps, i.e. information on the resource requirements for likelihood function evaluation under different custom precisions and an estimate of the number of bright points M, the performance model introduced before is utilised in order to provide estimates of the theoretical raw speedup leading to the selection of the optimal custom precision of the system. Please note that the ESS effect is only known at runtime and cannot be captured by the above static analysis based model. However, as the obtained results indicate, the above model can provide an informative prediction of the performance of the system.

4.6 Performance Evaluation

4.6.1 Case Studies

Logistic regression is used in many fields, including medical and social sciences. Here we consider two Bayesian problems with different dimensionality and data size that utilize a logistic regression model. Both 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 problem size they incorporate.

Synthetic Problem

Initially, the performance of the proposed system is evaluated by performing logistic regression on a synthetic data set, a two-class classification problem in two dimensions (and one bias dimension). As such, the ground truth of the parameters is known and it is used for the evaluation of the obtained estimates. Here the linear model in (4.20) is used: a set of 500 independent data $\mathbf{x} = x_{1:500}$ is generated randomly; the data set $y_{1:500} \in \{-1, 1\}$ is simulated using the parameters $\beta = (-10, 5, 10)$. The logistic regression likelihood of each data point is given by (4.21), where $\theta = (\beta_0, \beta_1, \beta_2)$ are the parameters, and the bias parameter is absorbed into θ by including 1 as an entry in x_n.

$$y = sign(\beta_0 + \beta_1 x_1 + \beta_2 x_2) \tag{4.20}$$

$$L_n(\theta) = \frac{1}{1 + exp\{-y_n \theta^T x_n\}} \tag{4.21}$$

MNIST Classification

The second case study focuses on a real problem, which is the logistic regression task described in [85]. The task is to classify handwritten digits (7s and 9s) in the large MNIST database, which has been widely used for training and testing in the field of machine learning [51]. The first 12 principal components (and one bias) are used as features. A set of 2000 data points are chosen from the total 12,214 data so the MCMC algorithm queries 2000 likelihood terms per iteration in this experiment. Each likelihood takes the form shown in (4.21) where x_n is the set of features for the nth data point. As opposed to the first case study, the parameter dimension increases to 12 plus a bias, where the total number of likelihood terms increases from 500 to 2000.

4.6.2 Hardware Implementation Details

The architecture in Figure 4.2 is implemented on a Xilinx Virtex-6 LX240T FPGA. The arithmetic operators of the generic MCMC block are implemented in double-precision floating point. The high

and low likelihood evaluation blocks which are fully pipelined are implemented under double precision floating point arithmetic and reduced custom precision arithmetic respectively, using floating point operators generated by FloPoCo [22]. All designs run on a single 150 MHz clock and fully utilized the available FPGA's resources. All results are post place and route.

4.6.3 Quality of MCMC Samples

We first assess the quality of the generated samples of three algorithms: the proposed custom-precision firefly MCMC (CF-MCMC), a double-precision implementation MCMC (DP-MCMC), which is used as a reference design, and a custom-precision MCMC (CP-MCMC), which uses the reduced precision for the computation of all likelihood terms i.e. existing approach in digital hardware design community. For convenience, the notation sAeB is used in this Chapter to denote a floating point representation, where A is the number of significant bits and B is the number of exponent bits.

Figure 4.3 shows the distributions of the predictive mean of parameter β_1 for the synthetic problem using the above three Monte Carlo simulations. In each MCMC simulation, $N = 30,000$ sample points are generated, and each of three algorithms are repeated for $3,000$ times with different random seeds. Both CP-MCMC and CF-MCMC algorithms utilised a custom precision of s8e8 (i.e. 8 significant bits and 8 bits for the exponent). As the results indicate, for CP-MCMC simulations where the reduced precision data-paths are used for all the likelihood terms and models the existing hardware design approaches, the mean value of the parameter has a significant bias and also a larger variance compared to the DP-MCMC results, supporting the results obtained in [20, 63]. However, the CF-MCMC sample distributions have the same variance and mean as DP-MCMC samples, which demonstrates that the proposed algorithm removes any bias introduced in the results due to low-precision computations. Please note that even though the actual value of the estimated parameter is 5, the data that have been generated by the model support a parameter value of 5.12 (the mean estimate of DP-MCMC and CF-MCMC algorithms).

Figure 4.4 depicts the bias and the variance in estimating the predictive mean on MNIST dataset, for a range of reduced custom precisions (with the number of significant bits varying from 5 to 23

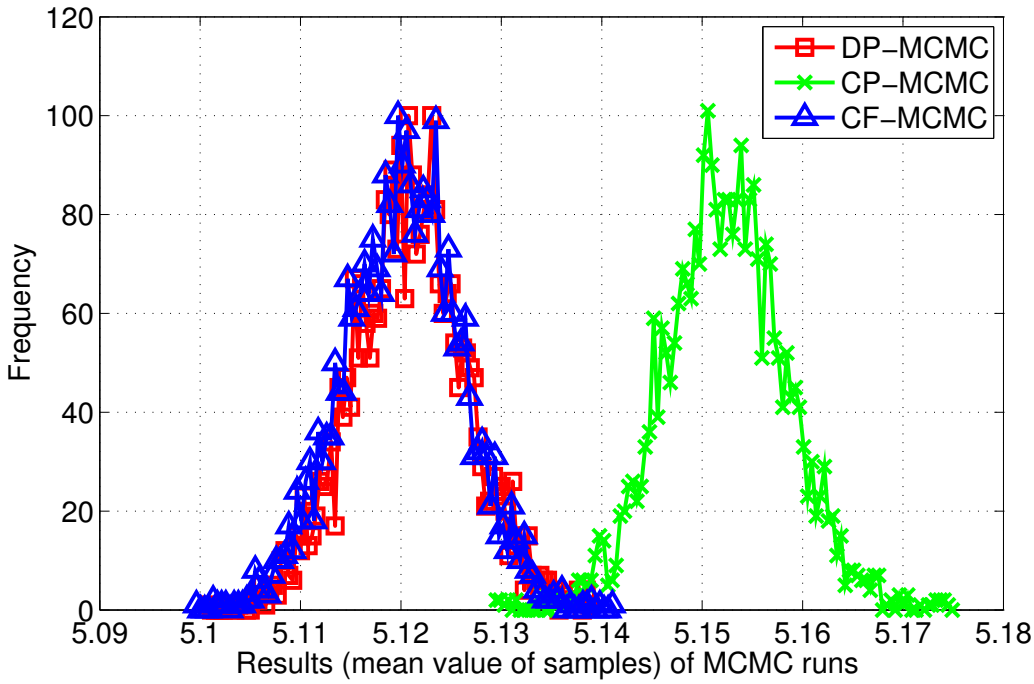


Figure 4.3: *Distribution of the mean value of the first parameter samples in the synthetic problem with 3,000 runs of DP-MCMC (at precision s52e11), CP-MCMC and CF-MCMC (both at precision s8e8).*

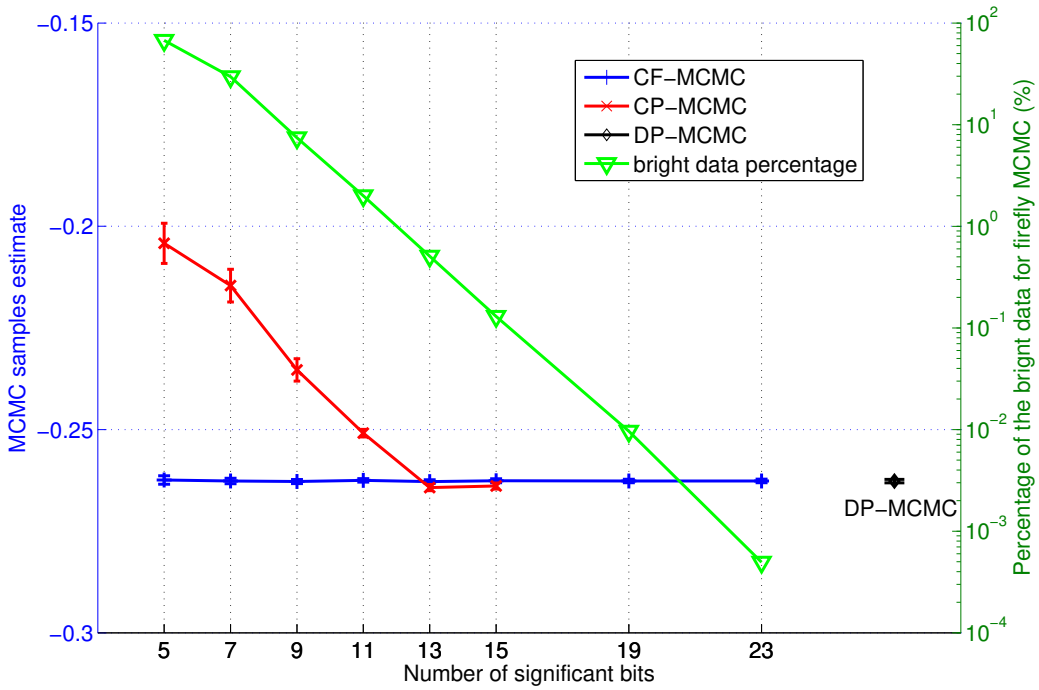


Figure 4.4: *The bias and its variance estimates of the parameter for MNIST problem, where DP-MCMC runs in double precision (s52e11), CP-MCMC and CF-MCMC run in the precision with the number of significant bits from 5 to 23. The green line indicates the average percentage of the bright data by 10,000 iterations in each of the 100 CF-MCMC runs.*

and a fixed exponent bits number of 8) for CP-MCMC and CF-MCMC algorithms with comparison to DP-MCMC simulations (where double precision with the significant bits 52 and the exponent bits 11 used). For each algorithm, 10,000 sample points are generated, and each MCMC simulation is repeated for 100 times. As shown in the figure, for every design point, both bias and standard deviation increase as the utilised custom precision uses fewer bits in the case of CP-MCMC. However, for the proposed algorithm, CF-MCMC, the obtained estimates have the same mean value and deviation as in the case of DP-MCMC algorithm regardless of how much the precision is reduced (apart when the number of significant bit is reduced to 5, but still there is no bias in the estimate). Figure 4.4 also shows the proportion of bright data point of the data set for CF-MCMC under different reduced precisions. The proportion of bright data is only 0.0005% at single-precision s23e8, but increase to 67% at a very small precision s5e8, indicating the the lower bound function becomes less tight as the precision decreases.

In summary, the two case studies indicate that the proposed system can produce unbiased estimates even under custom precision regimes without any noticeable difference in the variance of the estimate compared to an implementation that utilises double-precision floating point arithmetic throughout the system, except in the case where the precision is reduced significantly (i.e. 5 bits for the significant). Furthermore, it is observed that the current techniques that utilise custom precision in the likelihood evaluation lead to biased estimates, as it is expected.

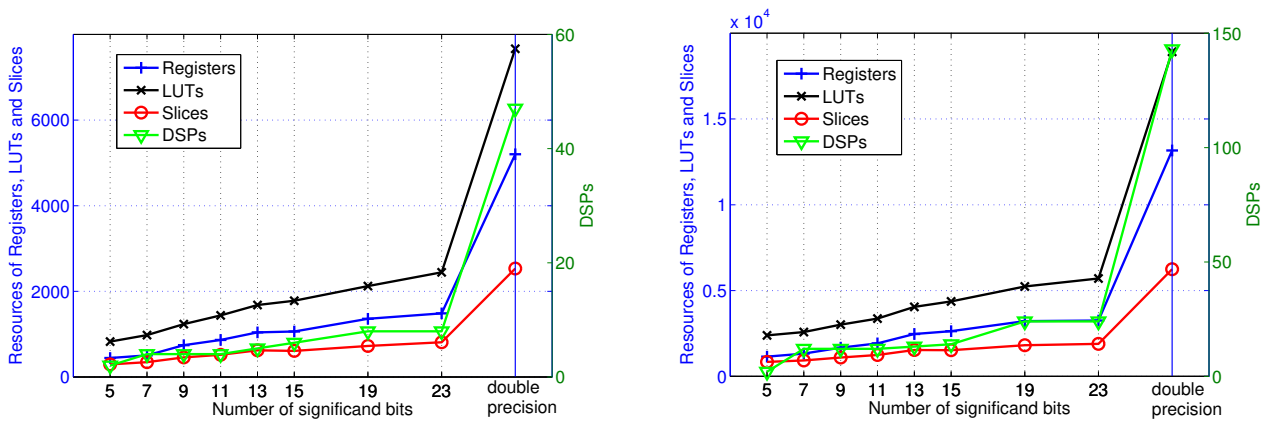
4.6.4 Resource Utilization

The proposed architecture CF-MCMC shown in Figure 4.2, and the double-precision architecture (DP-MCMC) shown in Figure 4.1b have been implemented in the target device, where the dataset are stored in the on-chip BRAMs. Table 4.1 gives the resource utilization (Registers, LUTs, Slices etc.) of the generic MCMC block which uses double precision floating point arithmetic operators, and also the resources required by one log-likelihood evaluation block at double-precision for both case studies. Here we also show the total memory size needed by both architectures to store the data and the auxiliary variables.

Table 4.1: *Resources of the generic block and one log-likelihood evaluation block using double floating point arithmetic operators, and memory size for CF-MCMC on Xilinx Virtex-6 LX240T.*

Resources (double-precision)		Slices Registers	LUTs	Slices	DSP48E1s	Memory Size
Generic block		2853	4837	1722	11	-
Log Likelihood Evaluation block	Synthetic Problem	5203	7666	2533	47	9.2 KB
	MNIST Problem	13168	18919	6242	143	0.2 MB

Figures 4.5a and 4.5b show the resource utilization of a single likelihood evaluation block of the synthetic example and MNIST problem under different reduced precisions respectively. The double-precision block's resources in Table 4.1 are also plotted in this figure. The figures imply that a factor of parallelism between 4-5 can be extracted when part of the computations can be mapped to reduced precision likelihood evaluation blocks utilising a precision between 8-18 significant bits. The above figure provides an expectation of the maximum gain in the performance that can be delivered by the proposed system compared to a double precision floating point implementation.



(a) *Synthetic Problem* **(b)** *MNIST Problem*

Figure 4.5: *The resource utilization of a single likelihood evaluation block of (a) the two-dimension (plus a bias) problem and (b) the MNIST problem with custom precision where the significant bits range from 5 to 23 and a fixed exponent bits at 8, and also double precision.*

4.6.5 Effective Sampling Throughput

In this section, the Effective Sampling Throughout speed-up is investigated for the proposed architecture. As the aim is to achieve the maximum throughput, full utilization of the FPGA logic resources is assumed for both architectures (CF-MCMC and DP-MCMC). Therefore, a maximum number of

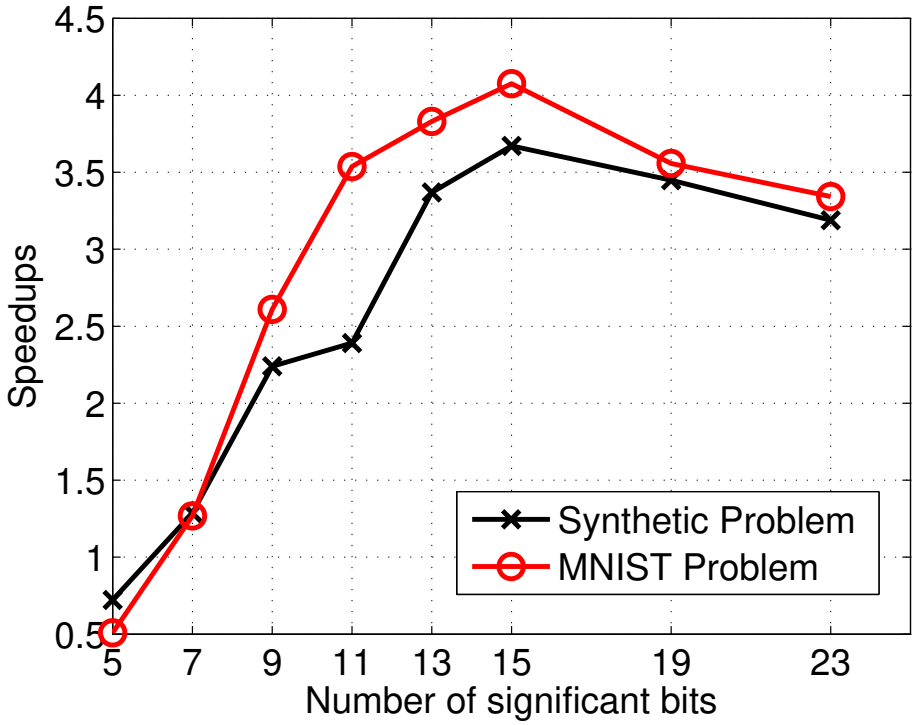


Figure 4.6: *The speedups in terms of the effective sampling throughput of CF-MCMC architecture over DP-MCMC on the target device for the two case studies.*

parallel log-likelihood evaluation blocks is obtained for each sampler, by selecting the optimal configuration of high and custom precision likelihood calculation blocks, P_H and P_L, using the proposed performance model. The effective sampling throughput is measured by ESS/T_{total} as described in Section 4.4.3. We compare the speedups in the throughput for our proposed CF-MCMC accelerator over DP-MCMC design, and the results for both problems are shown in Figure 4.6 for a range of values of the number of significant bits utilised in the CF-MCMC system. For the precision shown in the figure, the optimal configurations (P_H and P_L) that fully utilise the targeted FPGA device are: $(P_H, P_L) = \{(8, 22), (8, 20), (4, 36), (1, 46), (1, 38), (1, 36), (1, 34), (1, 30)\}$ for the synthetic problem, and $(P_H, P_L) = \{(2, 22), (2, 20), (2, 16), (1, 14), (1, 12), (1, 12), (1, 10), (1, 10)\}$ for the MNIST problem.

As the figure shows, the speedups obtained for the two problems with the evaluated precisions are in the order of 0.72x-3.67x and 0.51x-4.07x, for the synthetic and the MNIST problem respectively. The results show that by reducing the utilised precision in CF-MCMC, a higher degree of parallelism is possible. However, at the same time, the proportion of the bright data increases, which in turn introduces extra latency as the computation for bright data likelihood is executed in a lower parallel

degree. Furthermore, there is a reduction in the effective sample size (ESS) as the precision is reduced. This is evident by the obtained results, where a peak in effective sampling throughput speedup is observed at a specific precision configuration. Furthermore, it is observed that the proposed system can be outperformed by the double precision implementation when few significant bits are utilised (less than 6), indicating the need to have in place a performance model that predicts the throughput of the system. The optimal precisions for both problems are (s, e) = (15, 8), with the corresponding speedup of 3.67x and 4.07x.

Another metric to compare the performance of MCMC algorithms is the mean squared error (or risk) in the estimate of (4.2), i.e. $R = (I - \tilde{I})^2$, where the expectation is taken over multiple simulations of the Markov chain [48]. The risk can be decomposed as the sum of squared bias and variance, and the objective of MCMC in practice is to obtain estimates with lower risk. Figure 4.7 shows how the logarithm of the risk in estimating the mean of the parameter for MNIST, decreases as a function of the execution time. The experiment configuration is the same as in [48]: we first estimate the true mean using a long run of regular MCMC; then we compute multiple estimates of the mean from the algorithms under investigation and obtain the risk in these estimates. In our test, the average risk is based on 100 runs for each algorithm. The figure demonstrates that the proposed CF-MCMC algorithm largely reduces the risk compared to that of DP-MCMC, by reducing the variance faster within the same time period.

4.6.6 Theoretical Performance Model Evaluation

In this subsection, the accuracy of the theoretical performance model proposed in Section 4.5 is evaluated, and it is shown how it can be utilised in order to maximise the performance of the proposed system. For this investigation, the MNIST case study is utilised.

The evaluation of the framework is performed under three metrics: 1) the Theoretical speedup, which is computed by the theoretical performance model $T_{CF-MCMC} = M/P_H + (N - M)/P_L$ using the parallelism P_H and P_L we achieved in the target hardware and the bright data point M in Equation (4.19); 2) the actual Raw speedup, which is computed using the actual runtime by executing the

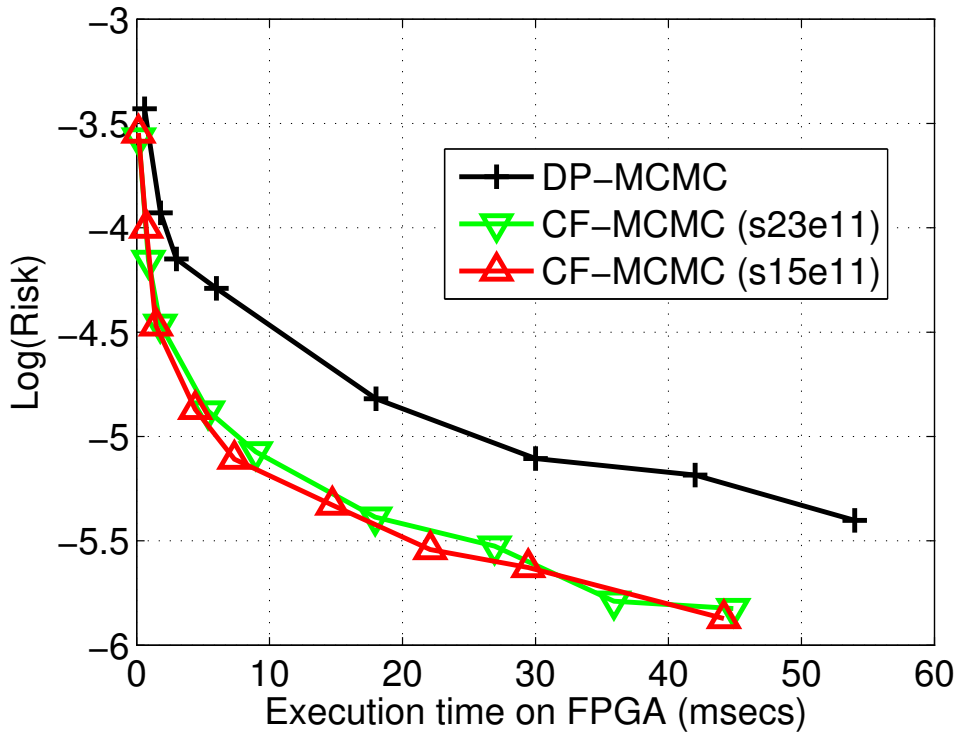
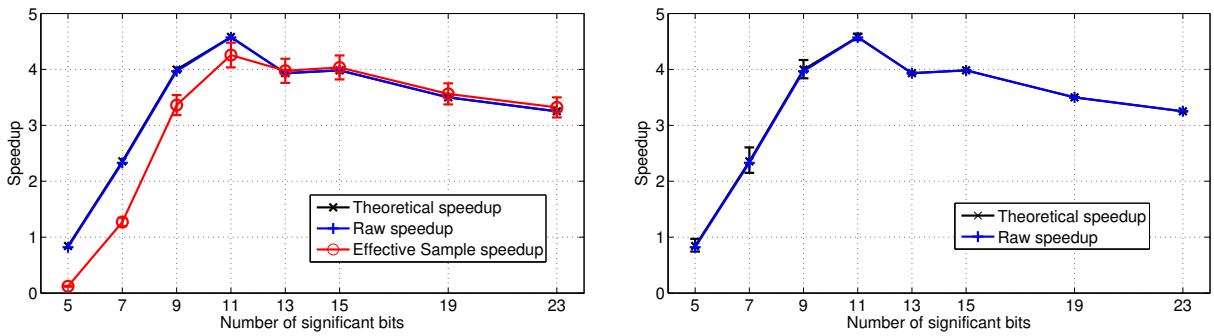


Figure 4.7: *The Risk in the estimate of the mean of the parameter.*

configuration in the FPGA device at each precision; 3) the Effective Sample speedup, which is computed through ESS/T_{total} i.e. the effective sampling throughput as described in the above subsection, and it includes the actual runtime when executed in the FPGA device and also the ESS effect in the generated samples (this speedup is used in the rest of the article unless explicitly stated).

Figure 4.8a shows the Theoretical, Raw, and Effective speedups for a range of precisions between the proposed architecture CF-MCMC and DP-MCMC. In total 100 runs were conducted in order to capture the possible variations in ESS. The results confirm that the derived performance model captures well the performance of the system under all the precisions, where the ESS effect, even though it is not captured by the performance model, does not have significant impact on the model's performance prediction accuracy. The confidence interval bars denote the variation in the Effective Sample speedup performance along different runs of the system for 95% confidence interval.

As has been described before, the performance model utilises the estimation of the number of bright points based on short pre-runs (e.g. 1000 samples). Given that this estimate is not exactly the same as the converged parameter values and thus it only provides an approximation of the number of bright points in the system, it is necessary to investigate the sensitivity of the predicted speedups obtained



(a) *Theoretical, Raw, and Effective speed-ups.* **(b)** *Theoretical and Raw speed-ups.*

Figure 4.8: *The theoretical performance model is evaluated by (a) comparisons of the Theoretical speed (estimated time), actual Raw speedup (execution time in FPGA), and the Effective Sample speedup (execution time in FPGA and including the ESS effects); and (b) the range (max and min values as denoted by the bars) of the theoretical speedup assuming a maximum deviation between the actual and predicted number of bright points of up to* 15%.

by the performance model with respect to the variation of the actual number of bright points from the predicted one. In this investigation, the number of bright points M is set to take values in an interval $[M^*(1-p) \quad M^*(1+p)]$, where M^* denotes the estimate of (4.19) based on short pre-runs, and then compute the theoretical speedups according to the number of bright data in this interval. The results for $p = 15\%$ are shown in Figure 4.8b.

The results indicate that at high precisions, the theoretical speedups have little variation with respect to the number of bright points. However, the above variations do not have an impact on the choice for the optimal custom precision that should be employed by the system, and thus the provided theoretical performance model and optimal precision selection method are still valid even in the case where the estimates of (4.19) have up to 15% deviation from the real values.

4.6.7 Lower Bound Construction Comparison

In this subsection, the construction of the lower bound functions proposed in Section 4.3.2 is investigated in order to assess how the lower bound constructions impact the speedups of the CF-MCMC algorithm. The MNIST case study is used for this investigation. Three different constructions are compared. The first construction is the method that utilises only Gappa++ in order to estimate the error bound (Gappa++). The second method is to estimate the error bound for part of the function through Gappa++ and utilises specific rounding modes in order to ensure a lower bound construction

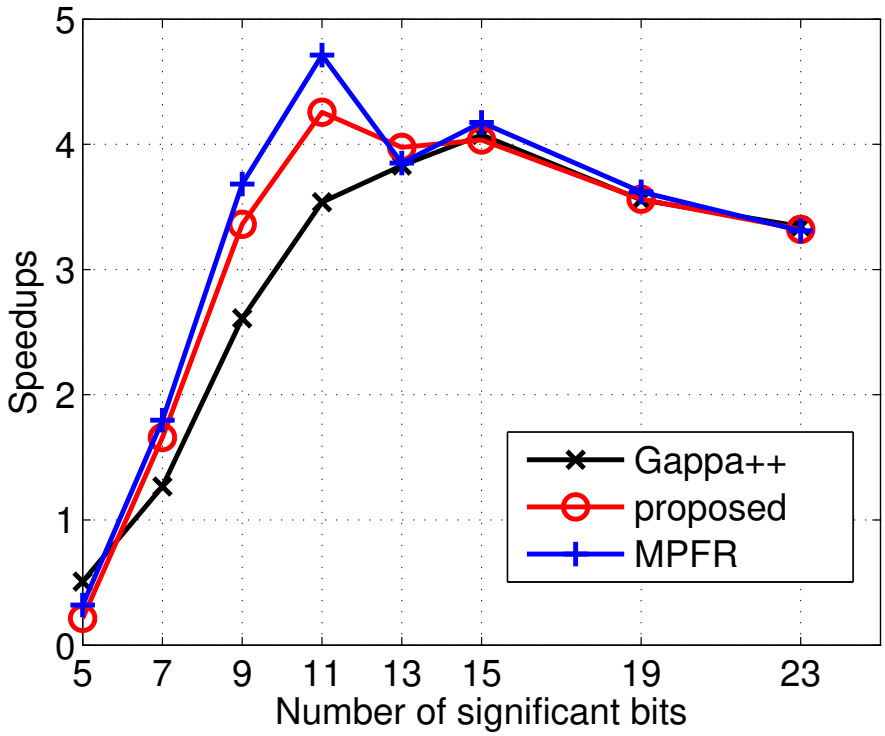


Figure 4.9: *The speed-ups (Effective Sample speed-up) of CF-MCMC architecture over DP-MCMC implementations on the target device for the MNIST problem, under the three lower bound function constructions.*

(proposed). The third method utilises the round mode for part of the function (similar as before), but now the final error bound for the whole function is estimated through simulations using the MPFR library [29] (MPFR). The third method does not provide any theoretical guarantees for lower bound, but it can be seen as a reference of the maximum speedups of the CF-MCMC algorithm that could be achieved. The results in terms of Effective Sample Speedup performance for the MNIST problem are shown in Figure 4.9, while all constructions have no obvious influence on the bias and variance of the generated samples.

As shown in the figure, the proposed construction, which is based on the rounding mode, results in systems that outperform systems that are based on the construction of the lower bound function based on our previous work (Gappa++). Furthermore, the proposed approach which guarantees a lower bound construction gives similar performance results to the method that is based on simulations (MPFR) and no guarantees in the construction can be provided. Nevertheless, all constructions lead to similar speedups when high precisions are utilised.

The above observation can be further generalised beyond the specific case study. Let us rewrite the

performance model provided in Section 4.4.3. Assuming $P_H = 1$, the execution time to generate one sample: $T_{sample} = N\alpha + N(1-\alpha)/P_L$ can be rewritten as: $T_{sample} = N*(1/P_L + (1-1/P_L)\alpha)$. For a given utilised precision, the number of parallel low precision units P_L is fixed on a target device. If the proportion of the bright data points (i.e. α, which depends on the lower bound proposal) satisfies $(1-1/P_L)\alpha << 1/P_L$ i.e. $\alpha << 1/(P_L - 1)$, then the execution time T_{sample} will be almost equal to N/P_L which doesn't depend on the quality of the lower bound construction. As such, all constructions would provide similar speedup when high custom precision evaluation blocks are utilised.

For example, for the MNIST problem when the precision bits are at 5, 7 and 9, $P_L = [29\ 26\ 22]$ (for $P_H = 1$) and $1/(P_L - 1) = [0.0357\ 0.04\ 0.048]$ while $\alpha = [0.10\ 0.025\ 0.0066]$. When more than 11 significant bits are used, $\alpha << 1/(P_L - 1)$ and thus all the constructions provide similar speedups.

Looking further into the obtained results, the last two methods (i.e. the proposed method and the MPFR method), provide the best performance when $P_H = 1$ instead of $P_H = 2$, which is the case for the first method (Gappa++) for precisions 7 and 9. This is due to the fact that the proposed method and MPFR provide tighter bounds than the first method (Gappa++), and as a result the proportion of bright data decreases at these two precisions points, leading to configurations that utilise fewer high precision evaluation units.

4.6.8 Comparison to a Multi-core CPU Implementation

The proposed system was also evaluated against an optimised version of the standard (i.e. double precision) MCMC algorithm that was running on a multicore system using the MNIST case study. The selected system was running CentOS 7 64-bit and utilised an Intel i7-3770 processor with 8 cores and 8 GBs of RAM, where the compiler is the gcc version 4.8.3. The CPU-based system was developed using OpenMP in order to utilise all the available cores in the system (i.e. 8), as well as using -O3 optimisations. The program takes full advantage of the available cores (100% utilisation), as the likelihood calculations are distributed evenly across the available cores (assumption of i.i.d data).

The obtained speedup results (i.e. Effective Speed-up) are shown in Table 4.2. The DP-MCMC

implementation achieves a speedup around 44.3x against the CPU, where the proposed CF-MCMC achieves speedups in the range between 22.5x to 180.5x. depending on the utilised custom-precision. The above results demonstrate the speedup gains of the proposed system.

Table 4.2: *Speed-ups of DP-MCMC and CF-MCMC FPGA samplers against an 8-core CPU sampler.*

FPGA designs	DP-MCMC	CF-MCMC at various precisions (number of significant bits)							
		5	7	9	11	13	15	19	23
Speedup vs. 8-core CPU	44.3x	22.5x	56.1x	115.6x	156.7x	169.6x	180.5x	157.5x	148.0x

4.6.9 Comparison to an FPGA Implemented FlyMC Algorithm

Finally, a comparison against the FlyMC algorithm proposed in [58] is performed in this section, as this is the closest work to ours. FlyMC algorithm can provide considerable speed ups when it is compared against a regular MCMC algorithm implementation in software, as is acknowledged by the authors [58]. For this to be the case, a lower bound function needs to be constructed as well as tuning for each data point needs to be performed through MAP in order to ensure tight bounds at the data points. The authors in [58] call this version of the algorithm MAP-tuned MCMC. However, such lower bound functions can be difficult to be obtained for many problems [58], and MAP tuning for each data point is required prior to the execution of the system imposing overheads to the overall execution time of the algorithm. The authors also investigate an alternative implementation of FlyMCMC that skips the MAP tuning for each data point and call this algorithm Untuned FlyMC. However, their obtained results show that the actual speed ups compared to regular MCMC are less impressive and sometimes it can lead to longer execution times compared to regular MCMC.

The important point of departure between this work and [58], is that the lower bound functions are custom precision versions of the target distribution and are automatically calculated by our system. As such, our proposed algorithm can be applied to any problem but as a trade-off it does not give the impressive speed ups claimed in [58] (i.e. for the MNIST problem, the authors claim 22x speedup for the MAP-tuned FlyMC compared to their regular MCMC implementation in a CPU).

In order to investigate the performance of the FlyMC algorithm in an FPGA, and how it compares against an implementation of a regular MCMC and the proposed CF-MCMC algorithm, the untuned

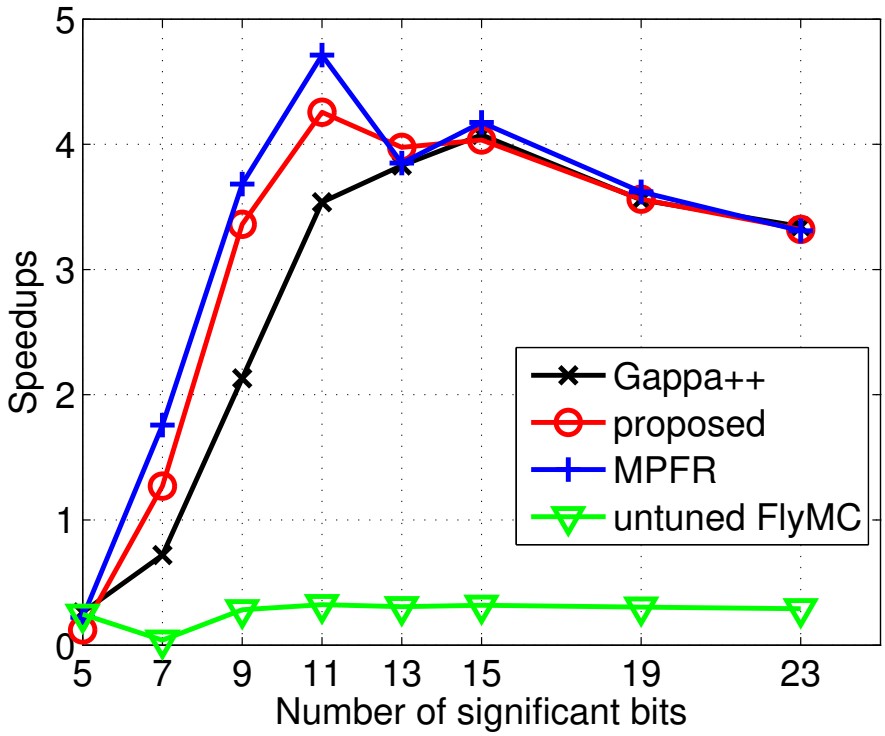


Figure 4.10: *Achieved speed-ups in terms of the resulting sampling efficiency of the original FlyMC algorithm in [58] and our proposed algorithms over the double-precision MCMC implementation (DP-MCMC) on the target device for the MNIST problem.*

FlyMC was mapped in an FPGA. To further explore the custom precision supported by the FPGA device, the lower bound function in the FlyMC architecture was implemented under different precision regimes (i.e. 23 significant bits correspond to a single precision floating point implementation of the lower bound function suggested by [58] for the MNIST problem). The obtained results are shown in Figure 4.10. As the results indicate, the FlyMC's performance (in terms of effective samples per second) is inferior to the regular MCMC FPGA implementation following the patters that was observed in the corresponding CPU implementations [58]. Furthermore, the use of custom precision in the implementation of the lower bound function leads to similar performance systems. In both cases, the underlying reason for leading to these performance points is the loose lower bound function used in the system. In any case, the proposed algorithm outperforms the FPGA implementation of the FlyMC.

4.7 Conclusions

In this Chapter, the CF-MCMC algorithm and its mapping to an FPGA device was presented. The proposed algorithm exploits the custom precision support of FPGAs in order to accelerate computational bounded MCMC problems, by utilising low precision calculations of the likelihood function in an intelligent way. The key contribution of this work is that by introducing a set of auxiliary variables, the proposed CF-MCMC accelerator guarantees the generation of unbiased estimates which is important for applications that cannot tolerate any bias in the estimates. Experimental results show that notable speedups over double-precision designs can be achieved with our proposed architecture in both software and hardware implementations.

The focus of this chapter is to optimize the likelihood computation in FPGA implementation. However, it doesn't consider the data transfer overhead in applications with large dataset. The following chapter will investigate the potential of using FPGA to overcome the memory issue for MCMC algorithms based on data subsampling.

Chapter 5

Communication-Aware MCMC Method for Big Data Applications on FPGAs

5.1 Introduction

Over recent years, Bayesian methods have become increasingly popular due to their ability to analyse data of complex structures using flexible models. Modern Bayesian inference problems utilise large dataset and the current trend is for these datasets to grow fast [3]. The availability of large datasets allow the construction of complex models, leading to computationally expensive likelihood functions in the MCMC methods. As such, the application of MCMC algorithms to modern problems start becoming prohibited and many researchers and practitioners work on the acceleration of MCMC.

Two main research directions can be found in the literature for MCMC acceleration. The first direction focuses on the acceleration of the likelihood computation through approximation models, and/or parallelising its evaluation based on the assumption of *i.i.d* data [58]. The former approach has the difficulty of selecting a suitable approximation model that has the desired properties of fast evaluation and at the same time obeys certain assumptions on the quality of the approximation [58]. The latter approach has been explored by various works using multi-core CPU and GPU devices, and eventually the system's performance is limited by the available memory bandwidth [56].

The second direction focuses on reducing the number of data points that need to be processed in every MCMC iteration, and effectively addresses the memory bandwidth problem. Most of the methods in this category propose a sampling scheme to generate an unbiased estimate of the likelihood function based only on a small sample of the dataset [48, 8, 69]. However, the existing works treat the memory as one monolithic storage space, and are oblivious on the performance characteristics of the various memory technologies of modern memory system hierarchies. As such, the actual latency of accessing a data point from the memory is not taken into account in the MCMC construction, and all the "accesses" are considered to have the same cost (i.e. latency, power).

The work proposed in this Chapter belongs to the second set of work, and it proposes a sampling-based algorithm that aims to reduce the memory accesses, but it exposes the performance of the memory sub-system to the MCMC algorithm in order to guide the sampling process, constructing as such a communication-aware MCMC algorithm. The key idea is the use of Probability Proportional-to-Size (PPS) sampling, where the inclusion probability of each data point is proportional to its approximate contribution to the likelihood function, allowing the system to reason during run-time on how often a specific data point will be accessed, and as such it can decide on its suitable storage location across the memory hierarchy.

The main contributions of this work are:

1) A communication-aware MCMC algorithm based on PPS sampling is proposed, that takes into account the performance characteristics of the underlying memory hierarchy. The proposed algorithm reduces the data transfer overheads among memories, compared to the regular MCMC and other subsampling-based algorithms, leading to faster execution times;

2) An optimized hardware architecture tailored for FPGA implementation that implements the proposed algorithm and efficiently utilises the on-chip memory blocks;

3) Evaluation of the proposed architecture in the Xilinx ZedBoard containing a Z-7020 device using the logistic regression model on MNIST database. The proposed design achieves a speedup of 3.37x over a highly optimised regular MCMC design in FPGA and obtain much lower risk in the estimates.

It should be noted, that FPGAs are particular suited for the proposed algorithm due to the customi-

sation of the use of the on-chip memory blocks, as well as due to the available high on-chip memory bandwidth that allows the full utilisation of of multiple processing elements for the likelihood evaluation.

5.2 Communication-Aware MCMC Method

This Chapter proposes a novel subsampling-based MCMC method which utilises an intelligent way to sample the dataset in order to evaluate the likelihood partially. The main focus of the work is on the approximation of the regular Metropolis test step in line 3 to line 9 in Algorithm 13 of Chapter 4. As the code illustrates, this is equivalent to compare two values: the reformulated random number u_0 and the average difference μ in the log-likelihoods of θ' and θ_{i-1} (the computations are performed in the log-domain).

$$u_0 = \frac{1}{N}\log(u), \text{where } u \sim \text{Uniform[0,1]} \tag{5.1}$$

$$\mu = \frac{1}{N}\sum_{i=1}^{N} l_i, \text{where } l_i = \log p(x_i \mid \theta') - \log p(x_i \mid \theta_{i-1}) \tag{5.2}$$

If $\mu > u_0$, the proposed sample θ' is accepted, otherwise it is rejected. In this work, the above Metropolis step is approximated similar to [48], and it is casted as a hypothesis test. Given the random sample $\{l_{i_1}, ..., l_{i_n}\}$ drawn from the population $\{l_1, ..., l_N\}$, a statistical hypothesis test is developed to decide whether the population mean is greater or less than u_0 with a user defined confidence. The standard deviation s of the sample mean $\bar{l}$, together with $\bar{l}$, is used to compute the test statistic $t = (\bar{l} - u_0)/s$ which follows a standard Student-t distribution with $n - 1$ degrees of freedom. Then $\delta = 1 - \Phi_{n-1}(|t|)$ is computed, where $\Phi_{n-1}(|t|)$ is the cdf of the standard Student-t distribution, and it is compared with a fix threshold (user defined) ϵ, in order to determine the level of confidence for a decision to be taken.

The parameter ϵ is introduced in the subsampling-based MCMC algorithms to control the bias in the estimates. When the ϵ is set to 0, there is no bias and the variance in the estimate can be brought down to zero if we can draw an infinite number of samples. However, for a given amount of computational time, it is better to allow a small bias in the estimate if it makes it cheap to generate a sample and thus

reduce the variance quickly. The optimal setting of ϵ is to minimize the risk (Equation 2.14 in Section 2.2.3 of Chapter 2) for a given time budget.

In [48], the authors propose to use equal probability sampling, i.e. simple random sampling (SI), and the results show that it doesn't work well as a large sampling fraction is needed for the algorithm to make a decision. On contrary, Probability Proportional-to-Size sampling (PPS) is an unequal sampling method where each sample is selected with a probability (which is often called "inclusion probability") that is proportional to its contribution towards the quantity under estimation in (5.2). Compared to SI, PPS can largely reduce the variance in the estimate of the mean, leading to a MCMC chain with more efficient draws for a given time budget compared to a regular MCMC on the full dataset. Moreover, in unequal sampling, the data points that have been assigned high probabilities will have higher probability to be chosen across iterations than the samples with low probabilities. The proposed work exploits the above property of PPS, in order to reason on the actual storage of the data points during the execution of the algorithm. Thus, the data points with high inclusion probabilities are kept in the on-chip memories where data points with low inclusion probabilities are left in the slower access off-chip memory. Please note that the inclusion probability of a data point depends on the values of the current sample, creating the need to dynamically reallocating data point across the memory hierarchy during the execution of the algorithm. However, due to the smoothness of the likelihood function, and the small distance between the current and proposed samples, the inclusion probabilities do not have to be evaluated in every iteration. In more details, to design a sampling scheme with unequal probabilities, we first construct sampling weights $\omega_i = |l_i|$. Then for a given target size m of the subset $\mathcal{S}$, i.e., $|\mathcal{S}| = m$, each unit is selected with inclusion probability $\pi_i = c\omega_i$ where c is a positive constant satisfying $\sum_{i=1}^{N} \pi_i = m$. The population mean and its variance can be estimated using the Horvitz-Thompson (HT) estimator[3]:

$$\bar{l} = \frac{1}{N} \sum_{i \in \mathcal{S}} l_i / \pi_i \tag{5.3}$$

$$var(\bar{l}) = \frac{1}{N^2} \sum_{i \in \mathcal{S}} (1 - \pi_i)(l_i/\pi_i)^2 \tag{5.4}$$

Given the above estimates, the algorithm follows the same flow as in SI, where t is computed and

[3]Please note that the HT estimator has these expressions under a design of Poisson sampling [84].

compared to ϵ in order to determine if a decision can be made.

The proposed communication-aware MCMC (CA-MCMC) algorithm is shown in Algorithm 15. The PPS sampling design is based on the Poisson sampling proposed in [84]. The advantage of Poisson sampling is its simple implementation, and on the simplicity in estimating the variance of the HT-estimator as shown in (5.4). There are some key points in the proposed algorithm: Firstly, this method doesn't need to update π_is at every iteration. This would require access on the whole dataset, leading to the same cost as in the regular Metropolis step. The algorithm only updates the probabilities when it cannot use the subset to make a decision (defined by the parameter $flag\_regular\_mcmc$ to 1 in line 22). If a decision cannot be taken, a regular Metropolis step is performed. This is shown in line 5 to 9, where the on-chip dataset is also built. Secondly, an adaptive method is proposed to determine the size of the subset M_target which is initialised at m in line 1. The variable $conseq\_done$ is used to record the progression of the algorithm, and it is used to guide the selection of the sample size M_adjust. This is shown in line 24 to 29.

The advantage of the CA-MCMC algorithm compared to the random sampling based MCMC (RS-MCMC) algorithm proposed in [48] is that it reduces the variance in the estimator of the likelihood using PPS sampling. Thus the average size of the subset is reduced. Moreover, as the algorithm is now based on an unequal sampling technique, reasoning on the "optimum" storage location of the data points can be performed. The proposed algorithm can store the data points with high inclusion probabilities in the on-chip memories, with the potential to reduce considerably the data transfer times from the off-chip memory, pushing further the memory-bound problem and allowing more samples to be drawn in a given time budget.

5.3 Hardware Mapping

5.3.1 IP Architectures and FPGA system Integration

In this section, an FPGA-based architecture is proposed for the implementation of the proposed CA-MCMC sampler, together with two other architectures for the implementation of the regular (tradi-

Algorithm 15: Communication-Aware MCMC Algorithm

Input: initial setting θ_0, number of samples N_s, parameters: ϵ, m, M_adjust, $conseq\_thres$;
Output: samples of parameter $\theta_i, i = 1, ..., N_s$;

1: Initialization: $i = 1, flag\_regular\_mcmc = 1$,
$M\_target = m, conseq\_done = 0$;
2: **while** $i < N_s$ **do**
3: Propose $\theta' \sim \theta_{i-1}$+Normal(0, $s^2 I_D$);
4: $u \sim$ Uniform(0,1);
5: **if** $flag\_regular\_mcmc == 1$ **then**
6: Compute the log-likelihood term l in (5.2) for the whole data;
7: Update π_is and build the on-chip dataset;
8: Perform the regular Metropolis step;
9: $i = i + 1; flag\_regular\_mcmc = 0$;
10: **else**
11: Draw a subset $\mathcal{S}$ with $|\mathcal{S}| = M\_target$ from Poisson Sampling, and access the data either from on-chip dataset or the main memory; // where we reduce the off-chip access.
12: Estimate $\bar{l}$ and its variance using HT estimator;
13: Compute $\delta = 1 - \Phi_{n-1}(|t|)$;
14: **if** $\delta \leq \epsilon$ **then**
15: **if** $u_0 \leq \bar{l}$ **then**
16: $\theta_i = \theta'$; //accept
17: **else**
18: $\theta_i = \theta_{i-1}$; //reject
19: **end if**
20: $i++; flag\_regular\_mcmc = 0; conseq\_done++$;
21: **else**
22: $flag\_regular\_mcmc = 1; conseq\_done--$;
23: **end if**
24: **if** $conseq\_done \geq conseq\_thres$ **then**
25: $M\_target = M\_target - M\_adjust$;
26: **end if**
27: **if** $conseq\_done \leq (-1 * conseq\_thres)$ **then**
28: $M\_target = M\_target + M\_adjust$;
29: **end if**
30: **end if**
31: **end while**

tional) MCMC shown in Algorithm 13 and the RS-MCMC algorithm proposed in [48].

The high-level view of the proposed system is shown in Figure 5.1. It consists of the MCMC IP (which represents one of the above three algorithms), an AXI bus interface, an ARM Cortex processor and a DMA controller. The ARM processor is responsible to initialise the system, and to read the generated MCMC samples from DDR for further processing. The DMA controller is used by both the ARM processor and the MCMC IP to access the off-chip DDR memory on the board.

A run of the MCMC algorithm proceeds as follows: initially, the implemented MCMC IP of each algorithm is initialised and started by the processor. The IP generates the required number of samples using the dataset which is stored in the off-chip memory (DDR). The generated samples are written in the off-chip memory, and at the end of the processing, the ARM core reads the data for further evaluation.

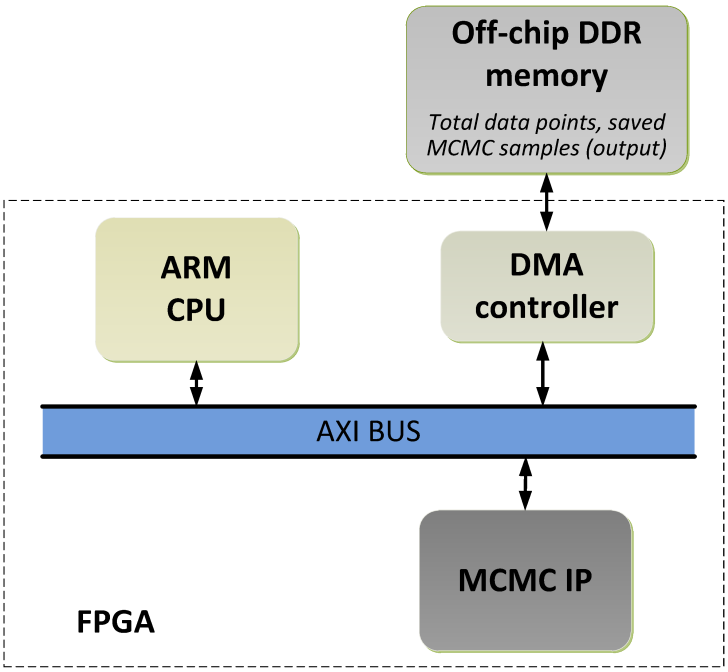


Figure 5.1: *The architecture of the FPGA system integrated with MCMC IP.*

The architectures of the MCMC IPs and the likelihood datapaths are optimized aiming to speed up the data transfers between the off-chip and FPGA device. For each MCMC IP, we built a on-chip memory with the size of M data points using the available BRAMs in the FPGA to reduce the data transfer times from off-chip memory to the IPs. Figure 5.2 shows the block diagram of the architectures of

the three algorithms. Assuming the total data size is N with each component x_n as a vector of D-dimension (consisting of D single precision floating point numbers), DDR stores the whole dataset and the on-chip memory can store M data points. For the regular MCMC, the on-chip memory performs as temporary storage. When computing the likelihood, we first transfer a subset of the data in off-chip to the on-chip memory and constantly process these data points. This is performed repeatedly until we processed all the data points in off-chip memory. For the random sampling based MCMC (RS-MCMC), the on-chip memory performs as a FIFO. In order to draw a subset from the whole dataset, in every cycle a data point is drawn randomly without replacement from off-chip to on-chip memory. At the same time, the likelihood datapath can process the data points from the on-chip memory without stalling. As the data points are randomly chosen, the DDR memory needs to be accessed in every data point acquisition, which is a disadvantage of the RS-MCMC algorithm. Please note that the sampling stage is done in parallel with the computation of the likelihoods (the subset does not depend on the result of the likelihood evaluation), and the existence of the on-chip memory guarantees that the datapaths can be fully pipelined and utilised.

In the case of the proposed CA-MCMC architecture, the on-chip memory performs as a FIFO but also as working memory. In each iteration, when we update π_is, we store a subset of data from the dataset on the on-chip memory using Poisson sampling. As such, the data points with high probability to be selected in the next subset are already in the on-chip memory (which is determined by the "On-chip flag" shown in Figure 5.2(c)). As the data in the on-chip memory maintain similar inclusion probability values for several iterations, the off-chip access is largely reduced.

To further improve the performance of the systems, the CA-MCMC and RS-MCMC architectures have been parameterised to access a mini-batch of data points B (contiguously stored in memory) in each read operation from the off-chip memory instead of reading one data point each time. The benefit of increasing the mini-batch size B is the reduced latency of accessing B data point from the off-chip memory, but this is in the expense of introducing a more coarse sampling of the dataset. This will be further discussed in Section 5.4.

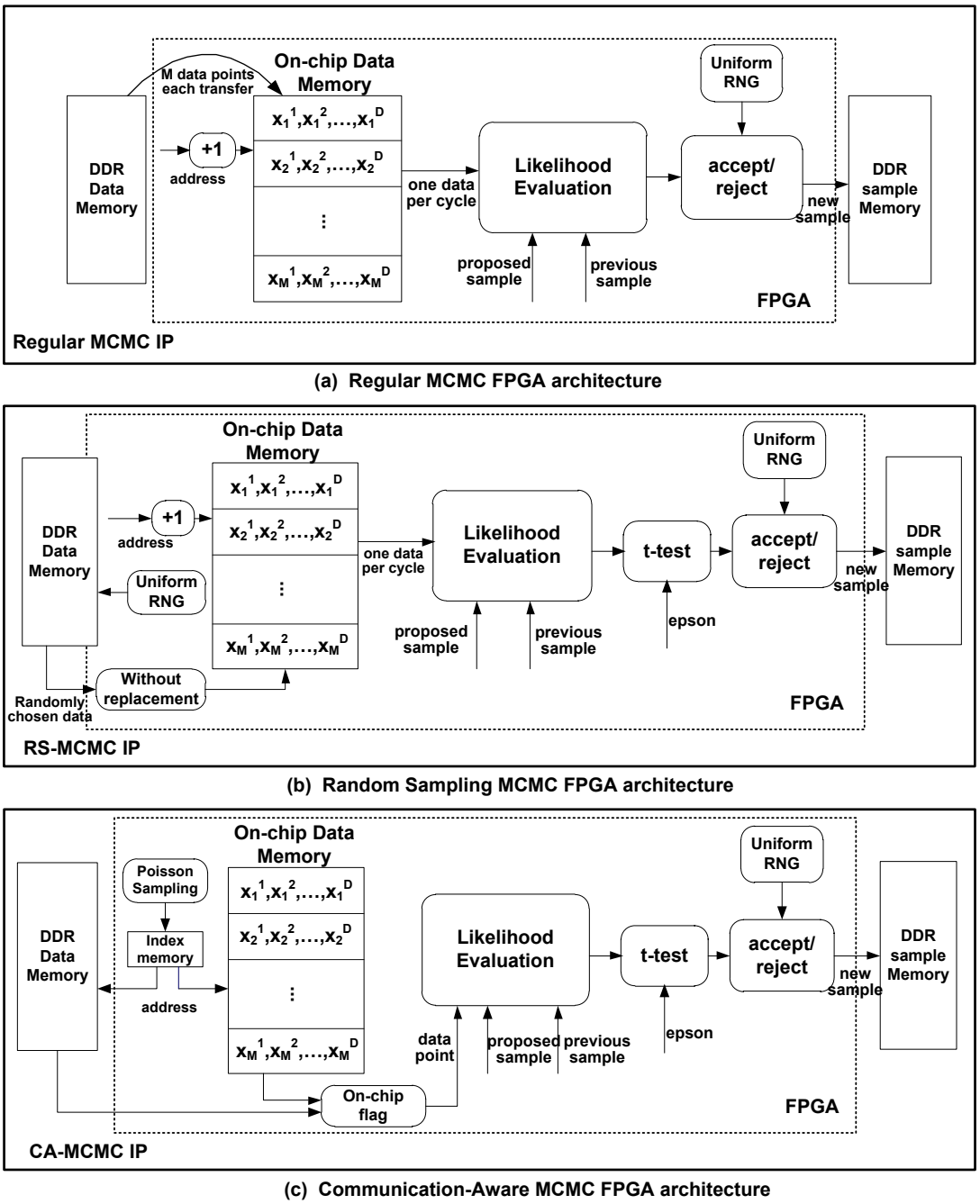


Figure 5.2: *FPGA architectures of the three implemented IPs: (a) regular MCMC; (b) RS-MCMC; (c) CA-MCMC.*

Table 5.1: *System and Problem parameters.*

N	Total Number of data points.
D	Dimensions of each data point.
N_s	Number of samples to be generated for each MCMC IP.
ϵ	the fixed threshold to control the bias of the RS-MCMC and CA-MCMC IPs.
M	the number of the datapoints which can be stored in the on-chip BRAMs of the three IPs when mapped on an FPGA.
B	the size of the mini-batch to be transferred from the DDR to FPGA each time for the RS-MCMC and CA-MCMC algorithms when mapped on an FPGA.

5.3.2 Performance Model

As the work is focused on pushing the memory bandwidth bound, the derived performance models capture the required memory accesses to the external memory. As the latency of the on-chip memory to the IP can be designed to be one clock cycle per data point (D floating numbers), as we have shown in Figure 5.2, and since all datapaths are fully pipelined, the overall performance of the system is dictated by the transfer of data points from the off-chip memory to the IP. Table 5.1 provides a summary of the problem and system parameters.

Let N be the total number of data points, where each point is a D-dimensional vector whose elements are under single precision floating point representation. Assuming that the algorithms perform N_s iterations, then the regular MCMC architecture which requires access to all the data points in each iteration, needs N_{MCMC} external memory access in total, which is given by:

$$N_{MCMC} = N_s N D \tag{5.5}$$

In the case of the RS-MCMC architecture, the number of data used for the likelihood evaluation is reduced. Due to random sampling, each sampled data-point needs to be transferred from the off-chip memory to the IP. Assuming that on average each iteration requires M_{RS} data points, the expected off-chip memory access is given by:

$$N_{RS-MCMC} = N_s M_{RS} D \tag{5.6}$$

In the case of the proposed CA-MCMC architecture, most of the data points used for the likelihood

evaluation will consist of the data points with high inclusion probabilities. Let's assume that in average M_{CA} data points used in each iteration (this is different from M_{RS} as the two architectures utilise different sampling techniques), and on average a percentage, α, of these points will be available in the on-chip memory. Please note that α will increase as the size of the on-chip memory (M) increases. Then the off-chip memory access times will be $(1-\alpha)M_{CA}$. Assuming the rate to perform the regular Metropolis step and update the inclusion probabilities π_is is on average β, the total number of off-chip memory accesses for N_s iterations is given by:

$$N_{CA-MCMC} = \beta N_s ND + (1-\beta)N_s M_{CA} * (1-\alpha) \tag{5.7}$$

5.4 Evaluation and Experiments

5.4.1 Case Study

As the case study to assess the performance of the proposed system, the logistic regression problem in Chapter 4 which is widely used in the MCMC literature [48, 8, 56, 57, 54] is considered. Logistic regression is used in many fields, including medical and social sciences. In this case study, the target distribution is the posterior for a logistic regression model trained on the MNIST dataset for classifying digits 7 vs 9. The logistic regression likelihood is given by:

$$p(x_n|\theta) = \frac{1}{1 + exp\{-y_n \theta^T x_n\}} \tag{5.8}$$

where $x_n \in \mathbb{R}^D$ is the set of features for the nth data point and $y_n \in \{-1, 1\}$ is its class. The dataset in the study consists of 10,000 data points chosen from the total 12,214 data points in the database, and the first 12 principal components (and one bias) from PCA are used as features, i.e., $D = 12$.

5.4.2 Parameter Selection

In the following sections, an investigation is performed to assess the impact of the various parameters to the overall performance of the algorithms. In the proposed implementation, the initial values for the parameters are as following: $m = 500$, $M\_adjust = 100$, $conseq\_thres = 10$. Please note that no actual tuning has been performed in the selection of these parameters, and their values were mainly guided to be similar to the parameters used in RS-MCMC [48]. The performance of the proposed algorithm under different values of ϵ, and the size of mini-batch size B is investigated in the following sections.

5.4.3 Resource Utilization

The three MCMC IPs proposed in Section 5.3 were implemented using Xilinx Vivado HLS, and single precision floating point number representation was used for all datapaths. The overall system was mapped on a Xilinx ZC702 ZedBoard with the system clock (which is also used in the IPs) set to 100 MHz. All the reported results are measured results from the board except if it is stated otherwise.

Table 5.2 shows the results of the FPGA resource utilisation for the three MCMC samplers, with the same architecture parameter $M = 1000$ which is limited by the number of BRAMs available in the ZedBoard. The regular and RS-MCMC use fewer BRAMs than the CA-MCMC, as the two algorithms do not substantially benefit from storing a subset of the data on-chip [4]. In the case of the proposed CA-MCMC architecture, extra on-chip memory is needed to store the inclusion probabilities, one-bit flag for each data to remark if the data is on-chip or not, and the address of the on-chip data in off-chip memory, as shown in Figure 5.2. Please note, as the dimensionality D of the data increases, the above overheads diminish as their storage requirements do not scale with the dimensionality of the data. Also, the logic resources such as LUTs and DSPs for RS-MCMC and CA-MCMC are very close and both utilise more logic resources than the regular MCMC architecture. This is expected as both subsampling-based architectures need to perform extra computations for calculating the mean and variance of the likelihood estimator in order to make a decision at each iteration. It should be

[4] A small gain can be realised in these two algorithms by having a small proportion of the data on chip. However as the size of the dataset increases, this gain diminishes, which is not the case for the proposed CA-MCMC.

noted that all systems fit easily in this relative small FPGA device, indicating the potential use of FPGA devices in this application domain.

Table 5.2: *Resource utilisation of the three MCMC FPGA-mapped samplers with the architecture parameter* $M = 1000$.

IP Block	LUTs	FFs	DSPs	BRAMs
Regular	25639 (48%)	15623 (14%)	105 (47%)	24 (9%)
RS-MCMC	34718 (65%)	20240 (19%)	138 (62%)	36 (13%)
CA-MCMC	33618 (63%)	20756 (19%)	151 (68%)	86 (30%)
Total	53200	106400	220	280

5.4.4 Obtained Risk and Speedup

Figure 5.3 shows how the Risk in estimating the mean parameter of the MNIST decreases as a function of the execution time for the different architectures, having fixed the design parameters as follows: $B = 1$ and $\epsilon = 0.1$. The experiment configuration is the same as in [48]: the true value of the mean is estimated using a long run of the regular MCMC algorithm, and then this values is used as a reference point. Multiple estimates are then computed from the three architectures, and the Risk is calculated. In the provided results, the risk is calculated over 200 runs of each algorithm. The figure demonstrates that the proposed architecture largely reduces the risk compared to that of the regular and RS- MCMC, by drawing more samples and reducing the variance faster within the same time period. The obtained speed-ups for the CA-MCMC and RS-MCCM over the regular MCMC are 1.58x and 1.19x respectively. The speed-up of the proposed system CA-MCMC compared to the single precision optimised CPU implementation of the regular MCMC executed in an Intel i7-3770 3.4 GHz CPU (single thread) is 13.5x.

The performance of the proposed algorithm was also investigated under different values of ϵ, which determines the required confidence in the estimation of the likelihood for taking a decision in the Metropolis step. The obtained results are shown in Figure 5.4. In all the tested cases, the proposed algorithm outperform the regular MCMC architecture, with an optimal setting for ϵ to be 0.1. As ϵ is increased, the algorithm needs fewer data to sample in order to make a decision, and thus more samples are generated per unit of time, leading to a reduction in the variance of the estimate. However

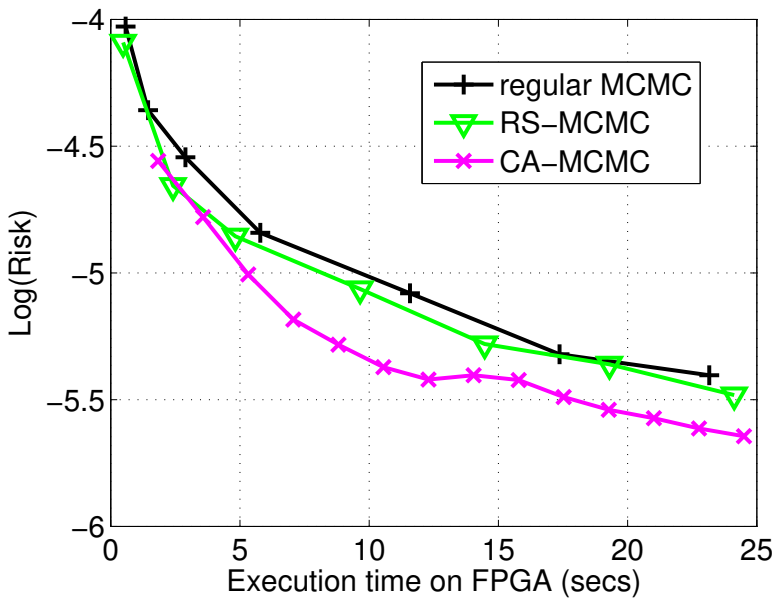


Figure 5.3: *The Risks in the estimate of the mean of the parameter with the design parameters:* $B = 1$ *and* $\epsilon = 0.1$.

the bias in the estimates also increases. It should be noted, that as more samples are generated, the risk will be dominated by the bias, leading to a monotonic relationship between ϵ and risk. Nevertheless, for the given execution times the best performance is obtained for $\epsilon = 0.1$.

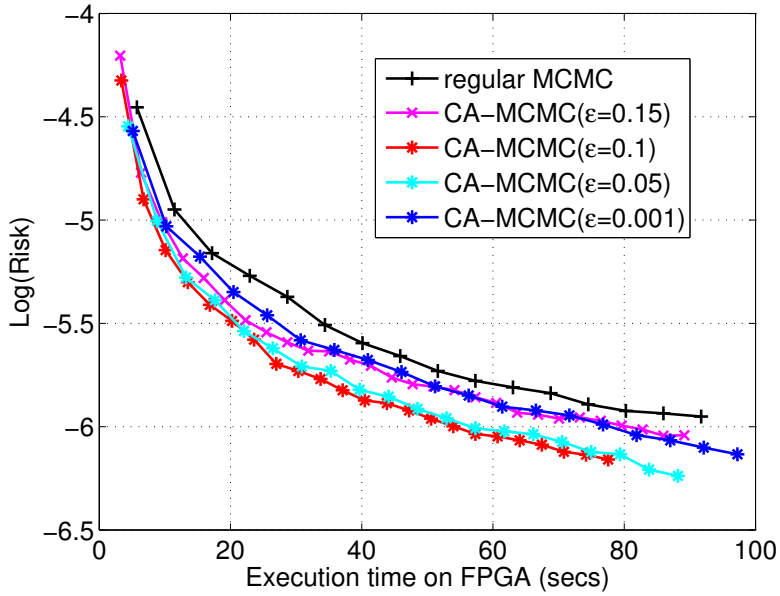


Figure 5.4: *The Risks of CA-MCMC in the estimate for different settings of* ϵ *with the design parameter* $B = 1$.

Moreover, we investigated the impact of the mini-batch size B to the performance of the algorithm having fixed ϵ to 0.1. The speedups of the two architectures normalised over the performance of the regular MCMC [5] for different values of B are shown in Table 5.3. The obtained results show that

[5]Please note that the performance of the regular MCMC architecture is not affected by B, as it requires all the data to be streamed to the FPGA.

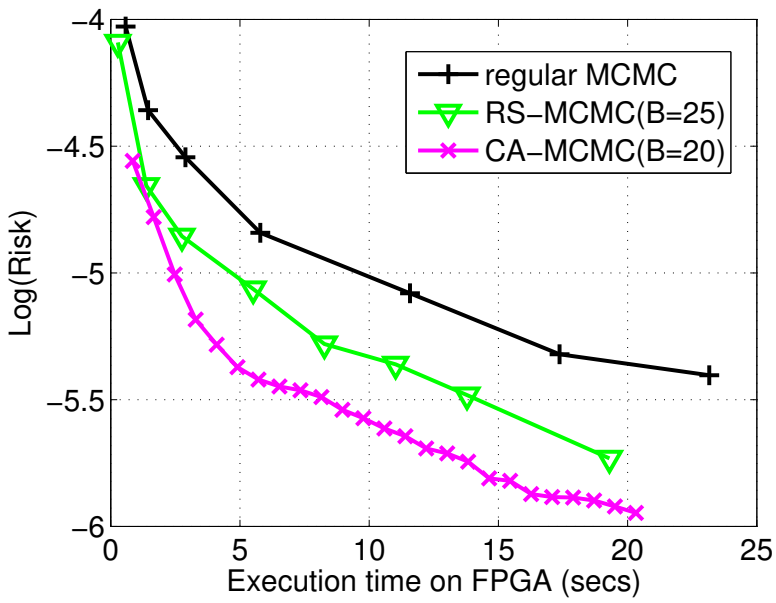


Figure 5.5: *The Risks in the estimate of the mean of the parameter with the optimal design parameters.*

the optimal speedups for RS-MCMC and CA-MCMC are 2.10x at $B = 25$ and 3.37x at $B = 20$ respectively. Figure 5.5 captures how the Risk in the estimate reduces as a function of wall-clock time for these two optimally tuned designs.

Table 5.3: *Speedups of RS- and CA- MCMC for different values of B, normalised over regular MCMC.*

B	1	5	10	20	25	50
RS-MCMC	1.19x	1.48x	1.72x	1.84x	2.10x	1.52x
CA-MCMC	1.58x	2.39x	3.21x	3.37x	3.35x	2.70x

Finally, the performance model introduced in Section 5.3.2 is utilised to make predictions of the speedups of CA-MCMC architecture for different FPGA devices that utilise larger on-chip memory storage space. The predicted speed-ups are shown in Figure 5.6, assuming the default design $B = 1$. As more on-chip BRAMs are available, the data reuse percentage α in (5.7) is increased and thus the performance of the proposed architecture improves. However, as M increases, the execution time of the regular and RS- MCMC has little improvement as we have shown in (5.5) and (5.6), and effectively diminished for large datasets.

It should be noted that storing 10% of the whole dataset in on-chip BRAMs seems a very optimistic scenario in real big data applications. Nevertheless, when the on-chip data size reduces to be very small, the main contribution on the performance improvement comes from that the size of the subset

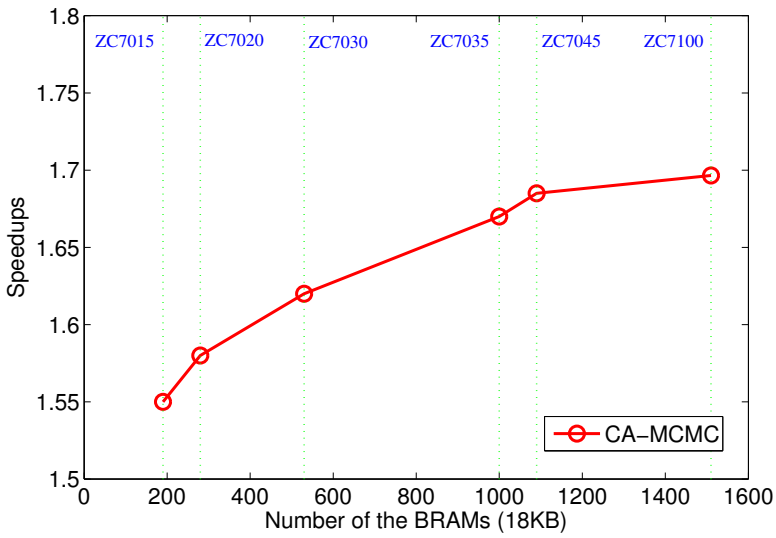


Figure 5.6: *Scaling of Speedups of CA-MCMC compared to the regular one when varying the device (number of BRAM blocks), at default design $B = 1$. The percentage of data that fit in BRAMs on each device are 5%, 10%, 20%, 35%, 40%, 50% respectively.*

used in CA-MCMC algorithm is smaller that that of regular and random subsampling MCMC. For example, from Figure 5.6, when the available BRAMs in the FPGA device (ZC7015) only stores 5% of the whole data, it shows a speedup of 1.55x. That is to say, even considering to store on-chip a smaller percentage of the dataset, the reduced subset size of CA-MCMC algorithm still gives performance improvement compared to regular and random subsampling MCMC algorithms.

5.5 Conclusions

This Chapter presents a novel communication-aware and subsampling-based MCMC framework, to push further the memory bandwidth bottleneck in current MCMC applications for big data. The key contribution of this work is that by introducing the PPS sampling to draw subset, the proposed design can largely reduce the off-chip memory access across iterations, therefore a lower risk in the estimate can be achieved for a given time budget. Experimental results show that notable speedups and reduced risks over other highly optimized FPGA designs are achieved with the proposed architecture. Even though the presented results are problem specific, the methodology can be applied to other problems with potential gains as it couples the “contribution” of the data with its storage location.

Chapter 6

Conclusion

The previous Chapters propose novel resampling algorithms, parallel resampling architectures, precision optimization for MCMC on FPGAs to speed up likelihood computation and the communication-aware MCMC method to reduce the memory access cost. This chapter starts with the summary of the current achievements of the thesis, discussions on some important issues posed in previous chapters followed by the directions of the future work.

6.1 Summary and Discussion of the Achievements

Bayesian methods have attracted lots of attention in modern deep learning systems and they have been widely used by practitioners and researchers mainly due to their ability to capture uncertainty in the models and systems. Examples include deep generative models [72] or deep belief networks (DBNs) [36] and dropout neural networks [79]. Accounting for uncertainty is central to deep learning system to guarantee the artificial intelligence safety. By conditioning on the data, the Bayesian methods not only perform point estimation, but also convey the uncertainties associated with the estimates. However, at the same time, with the availability of large data sets and the constant need to develop more complex models that better capture the targeted problem, significant computational challenges have been presented in Bayesian methods such as SMC and MCMC. Currently the approaches based on multi-core CPUs, GPUs, and FPGAs, have become the main trend aiming to accelerate these

methods. In this thesis, we show how the FPGA devices can be used to accelerate MCMC and SMC methods by fully utilizing the computational resources, word-length optimizations and high on-chip memory bandwidth. Notable speedups compared to the respective CPU and GPU design have been achieved in this work.

Chapter 3 focused on how to parallelize the resampling step of SMC method. We proposed parallel architectures for four state-of-the-art resampling algorithms (systematic, residual systematic, Metropolis and Rejection resampling). We also proposed the memory access strategies for Metropolis and Rejection resampling architectures to guarantee the same reampling quality as non-parallel implementations. Speedups of 10x to 49x over the respective GPU implementations were achieved using the proposed design on FPGAs.

The main limitation of the systematic and residual systematic resampling algorithms is that they need to compute the sum or the cumulative sum of the weights, which consumes lots of computational resources such as adders in FPGAs. For large number of particles/weights, it limits the maximum degree of parallelism we can achieve in FPGA device and therefore the speedups compared to GPU reduced when the number of particles increased a lot. For this reason, it is beneficial to store the weights and perform sum or cumulative sum using fixed point representation instead of the floating point numbers. Nevertheless, the Metropolis and Rejection resampling don't need to perform the collective operations. However, their execution time largely depends on the variance of the weights, which may cause slow convergence and thus long execution time. Rejection resampling also has non-deterministic runtime and this will have impact on the implementation of the whole SMC or particle filter system.

When considering how to use these parallel resampling architectures for the particle filter system, it should be noted that the output of the two resampling categories characterizes different forms, and this can lead to alternative architectures when implementing the distributed particle filter system. In [13], two architectures for distributed particle filters were presented: one is based on distributed resampling with proportional allocation (RPA) and the other is based on resampling with non-proportional allocation (RNA). Both architectures can benefit from each of the propose parallel resampling architectures in this work. Nevertheless, the RNA architecture is more suitable for FPGA implementation when

using systematic or residual systematic resampling, as it has less memory requirement compared to the RPA architecture.

In Chapter 4, the focus was transferred to MCMC methods. A novel FPGA-based MCMC construction is proposed that utilises the custom precision support of FPGA devices in order to accelerate the computations, guaranteeing at the same time asymptotically unbiased estimates. Key to this approach is the extension of the parameter space by an extra parameter that indicates the required precision in the computation of the likelihood of a data point. Compared to existing FPGA- and CPU-based work which utilises double floating point arithmetic, significant speedups have been achieved.

The main difference of this design compared to previous work on mixed precision MCMC is that we use the custom precision likelihood as a lower bound on the actual likelihood values. The lower bound requirement was satisfied by using formal verification tools Gappa++. However, it's not clear how tight the bound provided by Gappa++. Therefore we also proposed the utilization of the rounding mode configuration of the operators. Unfortunately, the rounding mode cannot be used for all types of operation. Nevertheless, combining Gappa++ and the configuration of rounding mode builds a lower bound tight enough to give significant performance improvement compared to previous work.

The main limitation of this work is that the results are achieved in the situation where the data set can be are stored in the on-chip memories in FPGA. However, when the utilized MNIST database comes larger or a real problem with big data is targeted, external memories such DDR and disks need to use for these problems. In this case, every time before we process the data, we need to bring part of the data sets in external memory to the on-chip memories, then this part of data will be processed first and this is repeated until the total data sets are processed. As a result, the communication time between off-chip memory and FPGA device can exceed the processing time, thus it becomes the dominant bottleneck in terms of the performance of the whole system. This motivates the work in Chapter 5 that aims to tackle the memory bound problem in MCMC construction.

Finally, Chapter 5 proposed a communication-aware MCMC framework that takes into account the underlying performance of the memory subsystem during the sampling process, leading to faster execution times. The framework is based on a novel subsampling algorithm that utilises an unbiased likelihood estimator based on Probability Proportional-to-Size (PPS) sampling, allowing information

on the performance of the memory system to be taken into account during the sampling stage. The proposed design on FPGA achieved a speedup of 3.37x over a highly optimised traditional MCMC design on FPGA, and 13.5x over the respective CPU design.

The main limitation of the proposed algorithm is that it still required performing the regular Metropolis step with a high frequency, which needs to access all the data points to evaluate the full likelihood functions. Thus it limits the maximum speedup we can achieve on FPGA even when a larger size of on-chip memories is available. Improvement can be made by proposing more efficient way to compute the approximate likelihood contribution, such as using either a noise-free Gaussian process or a thin plate spline surface as proposed in [69], in order to further improve the performance of the current system.

Previous work related to Chapter 5 is quite limited in literature, although lots of variants of MCMC algorithms use subsampling method. The key problem for MCMC algorithms to apply to big data is the communication time between datapaths and the data memory. It is unavoidable to keep all the data in memories and the question is that how we can reduce the main memory access which has large latency. Advanced data reduction techniques such as random projection can be investigated in order to reduce the memory overhead of MCMC methods brought by big data applications.

Overall, the main contribution and core breakthrough of this research is that we tackle both the computation bound and memory bound problem for Bayesian methods (MCMC and SMC) in the big data regime using FPGAs. This is achieved by taking into account the unique capabilities of FPGAs such as highly-parallel bit-oriented architecture and custom precision support. With the proposed hardware accelerators for MCMC and SMC, these methods can be applied to real data analysis applications and solve the intractable or computationally intensive tasks in Bayesian inference.

6.2 Extensions and Future Work

This work can be extended in several directions including comparison of different types of resampling algorithms, developing custom precision resampling and particle filter architectures and using multiple FPGA devices and/or heterogeneous multi-core computing platforms such as FPGAs and GPUs

to further improve the performance per energy unit in the field of SMC and MCMC.

- There are some other resampling algorithms that are commonly used in particle filters besides the four algorithms presented in this thesis. For instance, the Alias method [73] is a family of efficient algorithms for sampling from a discrete probability distribution, which is actually the optimized rejection sampling algorithms by stacking the weights. The Alias method's complexity doesn't depend on the variance of the weight set which is not the case of rejection sampling. A possible research direction would involve the comparison of the Alias method with the rejection and Metropolis sampling from the software and hardware implementation point of view. The main disadvantage of the Alias methods to be implemented in FPGA is that they need more memory to construct the Alias table and a large number of memory accesses in the step of initialization of the work-lists.

- Another important direction to accelerate particle filters is the conversion of double floating point to any custom precision in hardware implementation. It is beneficial to use small number of bits in floating point or fixed point representation in order to reduce area requirements. A low-complexity residual resampling in fixed-point arithmetic has been explored in [13]. Nevertheless, finite precision analysis and further work should be performed to allow the rapid and automated design space exploration involving optimisation of the precision configurations in particle filter by performance and speed trade-off.

- The resampling architectures proposed in this thesis is straightforward to be used in the distributed particle filtering. Future work can focus on the implementation of different application problems with comparisons of the proposed resampling architectures in different application domains.

- Current GPUs also support custom precision arithmetic, therefore it will be an interesting research direction to implement our proposed MCMC algorithm in Chapter 4 in GPU to compare its performance to that of FPGA in this work. The main emphasis of the design then should be in exploiting the construction of the tight and low bound functions of the likelihood in GPU.

- Another possible and promising research direction of custom precision MCMC design is using

fixed precisions. As the results shown in Chapter 4, even in a much smaller floating point setting, still a small speedup chan be achieved with the proposed design. It would be interesting to see how this will change if we move to the fixed point precision. Obviously the trade-off is that by allowing a far low bound function we can achieve more parallelism.

- Since the main goal of Chapter 5 is to reduce the memory accesses, current method uses application specific architectures. Nevertheless, this methodology can be applied to other problems with potential gains as it couples the "contribution" of the data with its storage location. Automated parameter tuning and mapping, taking into account the application data size and device size, that enables application independent optimizations can be explored to extend the application fields of this method.

- Currently as we have seen the results shown in Figure 5.6 of Chapter 5, there is a limitation of the speedups even increasing the on-chip size. This is due to the required steps to perform the regular Metropolis step in the current communication-aware algorithm. New method can focus on how to propose computationally efficient way to compute the approximate likelihood contribution, i.e., the inclusion probability to avoid the regular step, thus further improve the performance.

- We have provided results on the speed improvement of the communication-aware MCMC algorithm. In addition, it is very interesting to see how this approach may have effect on the power consumption, since the proposed approach improves the energy efficiency related to the data movement. Some power monitor tool can be used to analysis the power consumption of the three algorithms implemented in FPGA in Chapter 5 to have a comparison.

- A more general direction to accelerate the MCMC and SMC methods may be to utilize multiple FPGAs, and/or heterogeneous multi-core computing platforms with accelerators such as FPGAs and GPUs, since current big data problems can be too large to be comfortably processed on a single device. The memory bottlenecks can be eliminated by splitting data across multiple devices. Tools and methods need to be developed for optimisation of workload distribution for heterogeneous multi-core systems, in order to improve the performance per energy unit in the field of SMC and MCMC.

Bibliography

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

[2] C. Andrieu, A. Doucet, and R. Holenstein. Particle markov chain monte carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

[3] E. Angelino, M. J. Johnson, R. P. Adams, et al. Patterns of scalable bayesian inference. *Foundations and Trends® in Machine Learning*, 9(2-3):119–247, 2016.

[4] E. Angelino, E. Kohler, A. Waterland, M. Seltzer, and R. P. Adams. Accelerating MCMC via parallel predictive prefetching. In *Proceedings of the 30th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 22–31, 2014.

[5] 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*, pages 203–211. ACM, 2008.

[6] A. U. Asuncion, P. Smyth, and M. Welling. Asynchronous distributed estimation of topic models for document analysis. *Statistical Methodology*, 8(1):3–17, 2011.

[7] A. Athalye, M. Bolić, S. Hong, and P. M. Djurić. Generic hardware architectures for sampling and resampling in particle filters. *EURASIP Journal on Advances in Signal Processing*, 2005(17):2888–2902, 2005.

[8] R. Bardenet, A. Doucet, and C. Holmes. Towards scaling up Markov chain Monte Carlo: an adaptive subsampling approach. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 405–413, 2014.

[9] R. Bardenet, A. Doucet, and C. Holmes. On markov chain monte carlo methods for tall data. `http://arxiv.org/abs/1505.02827`, 2015.

[10] R. Bardenet and O.-A. Maillard. A note on replacing uniform subsampling by random projections in MCMC for linear regression of tall datasets. `https://hal.archives-ouvertes.fr/hal-01248841`, 2015.

[11] F. Belletti, M. Cotallo, A. Cruz, L. A. Fernandez, A. Gordillo-Guerrero, M. Guidetti, A. Maiorano, F. Mantovani, E. Marinari, V. Martin-Mayor, et al. Janus: An fpga-based system for high-performance scientific computing. *Computing in Science & Engineering*, 11(1):48–58, 2009.

[12] A. Bharath and M. Petrou. *Next generation artificial vision systems: Reverse engineering the human visual system*. Artech House, 2008.

[13] M. Bolić. *Architectures for efficient implementation of particle filters*. PhD thesis, Stony Brook University, 2004.

[14] M. Bolić, P. M. Djurić, and S. Hong. Resampling algorithms for particle filters: A computational complexity perspective. *EURASIP Journal on Advances in Signal Processing*, 2004(15):2267–2277, 2004.

[15] M. Bolić, P. M. Djurić, and S. Hong. Resampling algorithms and architectures for distributed particle filters. *Signal Processing, IEEE Transactions on*, 53(7):2442–2450, 2005.

[16] L. Bottolo, M. Chadeau-Hyam, D. I. Hastie, T. Zeller, B. Liquet, P. Newcombe, L. Yengo, P. S. Wild, A. Schillert, A. Ziegler, et al. Guess-ing polygenic associations with multiple phenotypes using a gpu-based evolutionary stochastic search algorithm. *PLoS Genet*, 9(8):e1003657, 2013.

[17] J. Butler and T. Sasao. Hardware index to permutation converter. In *Proc. IPDPSW*, pages 431–436, May 2012.

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

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

[20] 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*, pages 57–66, 2012.

[21] A. F. da Silva. cudaBayesreg: Bayesian computation in CUDA. *The R Journal*, 2(2):48–55, 2010.

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

[23] R. Douc and O. Cappe. Comparison of resampling schemes for particle filtering. In *Proc. ISPA*, pages 64–69, 2005.

[24] R. Douc, E. Moulines, and D. Stoffer. *Nonlinear time series: theory, methods and applications with R examples*. CRC Press, 2014.

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

[26] A. Doucet, N. De Freitas, and N. Gordon. *Sequential Monte Carlo methods in practice*. Springer Verlag, New York, 2001.

[27] A. Doucet and A. M. Johansen. A tutorial on particle filtering and smoothing: Fifteen years later. *Handbook of nonlinear filtering*, 12(656-704):3, 2009.

[28] P. Fearnhead. Markov chain monte carlo, sufficient statistics, and particle filters. *Journal of Computational and Graphical Statistics*, 11(4):848–862, 2002.

[29] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[30] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian data analysis*, volume 2. Taylor & Francis, 2014.

[31] L. N. Geppert, K. Ickstadt, A. Munteanu, J. Quedenfeld, and C. Sohler. Random projections for Bayesian regression. *Statistics and Computing*, pages 1–23, 2015.

[32] C. Geyer. Introduction to Markov Chain Monte Carlo. *Handbook of Markov Chain Monte Carlo*, pages 3–48, 2011.

[33] A. Gothandaraman, G. D. Peterson, G. L. Warren, R. J. Hinde, and R. J. Harrison. FPGA acceleration of a quantum Monte Carlo application. *Parallel Computing*, 34(4):278–291, 2008.

[34] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

[35] G. Hendeby, R. Karlsson, and F. Gustafsson. Particle filtering: the need for speed. *EURASIP Journal on Advances in Signal Processing*, 2010(22), 2010.

[36] G. E. Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009.

[37] M. Hoffman, F. R. Bach, and D. M. Blei. Online learning for latent dirichlet allocation. In *advances in neural information processing systems*, pages 856–864, 2010.

[38] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley. Stochastic variational inference. *The Journal of Machine Learning Research*, 14(1):1303–1347, 2013.

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

[40] C. Holmes. Markov chain monte carlo and applied bayesian. `http://www.stats.ox.ac.uk/~cholmes/Courses/BDA/bda_mcmc.pdf`, 2008.

[41] S. Hong, J. Jiang, and L. Wang. Improved residual resampling algorithm and hardware implementation for particle filters. In *Proc. WCSP*, pages 1–5, 2012.

[42] S.-H. Hong, Z.-G. Shi, J.-M. Chen, and K.-S. Chen. A low-power memory-efficient resampling architecture for particle filters. *Circuits, Systems and Signal Processing*, 29(1):155–167, 2010.

[43] M. Hoy, C. Weng, J. Yuan, and J. Dauwels. Bayesian tracking of multiple objects with vision and radar. In *Control, Automation, Robotics and Vision (ICARCV), 2016 14th International Conference on*, pages 1–6. IEEE, 2016.

[44] Z. Huang and A. Gelman. Sampling for bayesian computation with large datasets. `http://dx.doi.org/10.2139/ssrn.1010107`, 2005.

[45] K. Hwang and W. Sung. Load balanced resampling for real-time particle filtering on graphics processing units. *Signal Processing, IEEE Transactions on*, 61(2):411–419, 2013.

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

[47] G. Kitagawa. Monte Carlo filter and smoother for non-gaussian nonlinear state space models. *Journal of computational and graphical statistics*, 5(1):1–25, 1996.

[48] A. Korattikara, Y. Chen, and M. Welling. Austerity in MCMC Land: Cutting the Metropolis-Hastings Budget. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 181–189, 2014.

[49] J. H. Kotecha and P. M. Djuric. Gaussian particle filtering. *IEEE Transactions on signal processing*, 51(10):2592–2601, 2003.

[50] C. Kwok, D. Fox, and M. Meila. Real-time particle filters. *Proceedings of the IEEE*, 92(3):469–484, 2004.

[51] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[52] M. D. Linderman, M. Ho, D. L. Dill, T. H. Meng, and G. P. Nolan. Towards program optimization through automated analysis of numerical precision. In *Proc. CGO*, pages 230–237, 2010.

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

[54] S. Liu and C.-S. Bouganis. Communication-Aware MCMC Method for Big Data Applications on FPGAs. In *Proc. FCCM*, May 2017.

[55] S. Liu, G. Mingas, and C.-S. Bouganis. Parallel resampling for particle filters on FPGAs. In *Proc. FPT*, pages 191–198, Dec 2014.

[56] S. Liu, G. Mingas, and C.-S. Bouganis. An exact MCMC accelerator under custom precision regimes. In *Proc. FPT*, pages 120–127, Dec 2015.

[57] S. Liu, G. Mingas, and C.-S. Bouganis. An Unbiased MCMC FPGA-Based Accelerator in the Land of Custom Precision Arithmetic. *IEEE Transactions on Computers*, 66(5):745–758, May 2017.

[58] D. Maclaurin and R. P. Adams. Firefly monte carlo: Exact mcmc with subsets of data. In *Thirtieth Conference on Uncertainty in Artificial Intelligence (UAI)*, 07/2014 2014.

[59] C. Mak. Stochastic potential switching algorithm for monte carlo simulations of complex systems. *The Journal of chemical physics*, 122(21):214110, 2005.

[60] G. Mingas. *Algorithms and architectures for MCMC acceleration in FPGAs*. PhD thesis, Imperial College London, 2015.

[61] 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*, pages 153 –156, 2012.

[62] G. Mingas and C.-S. Bouganis. Population-Based MCMC on Multi-Core CPUs, GPUs and FPGAs. *IEEE Transactions on Computers*, 65(4):1283–1296, April 2016.

[63] G. Mingas, F. Rahman, and C.-S. Bouganis. On Optimizing the Arithmetic Precision of MCMC Algorithms. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 181–188, April 2013.

[64] I. Murray. *Advances in Markov chain Monte Carlo methods*. PhD thesis, University College London, 2007.

[65] L. M. Murray, A. Lee, and P. E. Jacob. Parallel resampling in the particle filter. *Journal of Computational and Graphical Statistics*, 25(3):789–805, 2016.

[66] H.-C. Ng, S. Liu, and W. Luk. Reconfigurable Acceleration of Genetic Sequence Alignment: A Survey of Two Decades of Efforts. In *Proc. FPL*, Sep 2017.

[67] R. D. Payne and B. K. Mallick. Bayesian Big Data Classification: A Review with Complements. *arXiv preprint arXiv:1411.5653*, 2015.

[68] M. K. Pitt and N. Shephard. Filtering via simulation: Auxiliary particle filters. *Journal of the American statistical association*, 94(446):590–599, 1999.

[69] M. Quiroz, M. Villani, and R. Kohn. Speeding up mcmc by efficient data subsampling. *Riksbank Research Paper Series*, (121), 2015.

[70] M. Quiroz, M. Villani, and R. Kohn. Exact Subsampling MCMC. *arXiv preprint arXiv:1603.08232*, 2016.

[71] C. Robert and G. Casella. *Monte Carlo statistical methods*. Springer Science & Business Media, 2013.

[72] R. Salakhutdinov. Learning deep generative models. *Annual Review of Statistics and Its Application*, 2:361–385, 2015.

[73] K. Schwarz. Darts, dice, and coins: Sampling from a discrete distribution. `http://www.keithschwarz.com/darts-dice-coins/`, 2011.

[74] S. L. Scott, A. W. Blocker, F. V. Bonassi, H. Chipman, E. George, and R. McCulloch. Bayes and big data: The consensus monte carlo algorithm. In *Proceedings of the Bayes 250 conference*, volume 16, 2013.

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

[76] B. Sileshi, C. Ferrer, and J. Oliver. Particle filters and resampling techniques: Importance in computational complexity analysis. In *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pages 319–325. IEEE, 2013.

[77] G. K. Smyth. Numerical integration. *Encyclopedia of biostatistics*, 1998.

[78] D. Sorensen and D. Gianola. *Likelihood, Bayesian, and MCMC methods in quantitative genetics*. Springer Science & Business Media, 2002.

[79] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[80] L. D. Stone, R. L. Streit, T. L. Corwin, and K. L. Bell. *Bayesian multiple target tracking*. Artech House, 2013.

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

[82] X. Tian and K. Benkrid. Design and implementation of a high performance financial Monte-Carlo simulation engine on an FPGA supercomputer. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 81–88, Dec 2008.

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

[84] Y. Tillé. *Sampling algorithms*. Springer, 2011.

[85] M. Welling and Y. W. Teh. Bayesian learning via stochastic gradient Langevin dynamics. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 681–688, 2011.

[86] J. Wu, K. Liu, J. Wei, D. Han, and J. Xiang. Particle filter using a new resampling approach applied to leo satellite autonomous orbit determination with a magnetometer. *Acta Astronautica*, 81(2):512–522, 2012.

[87] S. Zierke and J. D. Bakos. FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods. *BMC bioinformatics*, 11(1):184, 2010.