

Imperial College London
Department of Electrical and Electronic Engineering

**Algorithms and architectures for
MCMC acceleration in FPGAs**

Grigorios Mingas

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, February 2016

Abstract

Markov Chain Monte Carlo (MCMC) is a family of stochastic algorithms which are used to draw random samples from arbitrary probability distributions. This task is necessary to solve a variety of problems in Bayesian modelling, e.g. prediction and model comparison, making MCMC a fundamental tool in modern statistics. Nevertheless, due to the increasing complexity of Bayesian models, the explosion in the amount of data they need to handle and the computational intensity of many MCMC algorithms, performing MCMC-based inference is often impractical in real applications. This thesis tackles this computational problem by proposing Field Programmable Gate Array (FPGA) architectures for accelerating MCMC and by designing novel MCMC algorithms and optimization methodologies which are tailored for FPGA implementation. The contributions of this work include: 1) An FPGA architecture for the Population-based MCMC algorithm, along with two modified versions of the algorithm which use custom arithmetic precision in large parts of the implementation without introducing error in the output. Mapping the two modified versions to an FPGA allows for more parallel modules to be instantiated in the same chip area. 2) An FPGA architecture for the Particle MCMC algorithm, along with a novel algorithm which combines Particle MCMC and Population-based MCMC to tackle multi-modal distributions. A proposed FPGA architecture for the new algorithm achieves higher datapath utilization than the Particle MCMC architecture. 3) A generic method to optimize the arithmetic precision of any MCMC algorithm that is implemented on FPGAs. The method selects the minimum precision among a given set of precisions, while guaranteeing a user-defined bound on the output error. By applying the above techniques to large-scale Bayesian problems, it is shown that significant speedups (one or two orders of magnitude) are possible compared to state-of-the-art MCMC algorithms implemented on CPUs and GPUs, opening the way for handling complex statistical analyses in the era of ubiquitous, ever-increasing data.

Acknowledgements

First of all, I must thank my supervisor Christos, without whom this thesis would not be what it is. He gave me absolute freedom in choosing my thesis topic, encouraged me to become independent, taught me to aim for high quality research and was always available to discuss and to provide valuable insights. He also gave me the opportunity (and funds) to travel around the world. All these experiences have not only moulded me as a researcher but also contributed to my evolution as a person.

I am also indebted to Leonardo Bottolo, who showed me how the statistical community sees my research and what I should do to improve it. Also, for providing the initial motivation for one of the chapters of this thesis and for coordinating several exciting collaborations. All the people in CAS, who helped me overcome many obstacles and provided distractions from work. Andrea Suardi, Nikos Kantas, Pierre Jacob, Iain Murray, Ajay Jasra, Anthony Lee, Arnaud Doucet and Petros Dellaportas who provided feedback and advice in various stages, especially in the first months of my PhD where things were still vague.

I also have to thank my various sources of escape from the small world of academia. My parents Vaggelis and Tasoula who put enormous efforts into bringing me up and supported me in every decision I ever made. Their encouragement to learn and to develop my mind and their unconditional love have shaped me. My sister Eleni, who has tolerated me for so long, gave me advice and was there for a chat when I got home late. She has contributed more than anyone to my way of thinking and has helped me realise so many things I would never have thought by myself.

Finally, many dear friends made sure I never lacked good company and gave meaning to my existence. I have to thank Markos, without whom my quality of life in London would have been markedly inferior. Akis, Eralia, Stratos and Chrissa for making my days more interesting and unpredictable and for being true friends. Finally, Katerina for everything. I omit many others, both in the UK and in Greece, but they already know how much I am indebted to them.

Declaration

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

Copyright Declaration

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

Contents

Abstract	i
Acknowledgements	iii
1 Introduction	1
1.1 Motivation	1
1.2 Aims and outline of this study	2
1.3 Statement of Originality	5
1.4 Publications	6
2 Background and related work	8
2.1 Introduction to Bayesian modelling and inference	9
2.2 Background on Markov Chain Monte Carlo	13
2.2.1 MCMC principles	15
2.2.2 Convergence and mixing	16
2.3 MCMC algorithms	17
2.3.1 The Metropolis and Metropolis-Hastings algorithms	18
2.3.2 Multi-modal posteriors and Population-based MCMC	19
2.3.3 State Space Models and Particle MCMC	23

2.3.4	Other methods	32
2.4	The need for faster inference	34
2.4.1	Existing approaches and trends in MCMC methodology	37
2.4.2	The approach of this thesis	38
2.5	Hardware acceleration technologies	38
2.5.1	Central Processing Units	39
2.5.2	Multi-core CPUs	41
2.5.3	Graphics Processing Units	43
2.5.4	Field Programmable Gate Arrays	44
2.5.5	Other platforms	47
2.6	Arithmetic precision	48
2.6.1	Floating point arithmetic	48
2.6.2	Custom precision floating point in CPUs, GPUs and FPGAs	49
2.6.3	Custom precision in MCMC	50
2.7	Related work	54
2.7.1	MCMC parallelization and hardware acceleration	54
2.7.2	Choice of number of particles and use of tempering in Particle MCMC	64
2.7.3	Custom precision in MCMC and in other Monte Carlo algorithms	65
2.7.4	Other related work	67
2.8	Summary	67
3	Algorithms and architectures for Population-based MCMC	69
3.1	Introduction	69

3.2	Parallel Tempering	71
3.3	Accelerating PT	73
3.3.1	Baseline accelerators	74
3.3.2	Custom precision methods for PT	81
3.3.3	Custom precision accelerators	87
3.4	Case study: Bayesian inference on mixture models	91
3.5	Implementation	92
3.5.1	IP implementation and FPGA system integration	92
3.5.2	Platforms and devices	93
3.5.3	Runs in hardware and software	94
3.6	Investigation and results	96
3.6.1	Performance metrics	96
3.6.2	FPGA resource utilization	99
3.6.3	Performance evaluation	99
3.6.4	Precision optimization for FPGAs and kernel optimization for GPUs	109
3.6.5	FPGA memory considerations	113
3.7	Discussion	115
4	Algorithms and architectures for Particle MCMC	118
4.1	Introduction	118
4.2	State-space models and Particle MCMC	120
4.3	ppMCMC: A new population-based pMCMC method	121
4.4	FPGA architectures for pMCMC and ppMCMC	127

4.4.1	Parallelism in the algorithms	127
4.4.2	pMCMC architecture	129
4.4.3	ppMCMC architecture	135
4.4.4	Performance models	139
4.5	Case Study	144
4.6	Implementation	148
4.6.1	IP implementation and FPGA system integration	148
4.6.2	Random number generators	150
4.7	Investigation and results	150
4.7.1	Platforms and devices	151
4.7.2	Resource utilization	151
4.7.3	pMCMC: Hardware comparison (uni-modal posterior)	152
4.7.4	ppMCMC vs. pMCMC: Algorithm comparison and trade-offs (multi-modal posterior)	161
4.7.5	ppMCMC vs. pMCMC: Hardware comparison and trade-offs (multi-modal posterior)	162
4.8	Conclusions	165
5	Arithmetic precision optimization for generic MCMC	169
5.1	Introduction	169
5.2	MCMC estimates and the effect of custom precision	172
5.2.1	MCMC in infinite and double precision	172
5.2.2	Parts of MCMC - precision domains	173
5.2.3	MCMC estimate when using custom precision probability densities	175

5.3	Proposed bias estimator	176
5.3.1	Bias estimator	176
5.3.2	Variance: Proposed bias estimator vs. Straightforward bias estimator vs. Custom precision output estimator	179
5.4	Optimization method	181
5.4.1	User input	182
5.4.2	Steps 1 and 2: Pre-runs (mixed-precision bitstream)	184
5.4.3	Step 3: Selection (software)	186
5.4.4	Step 4: Final runs (optimized-precision bitstream)	186
5.4.5	Assumptions	186
5.5	Case studies: Models and MCMC method	188
5.5.1	Mixture model	188
5.5.2	Neural network model	189
5.5.3	MCMC algorithm	190
5.6	Implementation	190
5.6.1	IP implementation and FPGA system integration	190
5.6.2	Transferring data between the host and the FPGA	191
5.7	Investigation and results	192
5.7.1	Resource utilization	192
5.7.2	Performance evaluation - trade-off between speed and bias	193
5.7.3	Comparison to an unbiased precision optimization approach	199
5.8	Discussion	201

6 Conclusion	203
6.1 Overview of MCMC acceleration research	203
6.2 Summary and discussion of thesis achievements	205
6.3 Future work	212
6.3.1 Targeting other MCMC methods	212
6.3.2 Extension to distributed and heterogeneous platforms	214
6.3.3 Tools to improve accessibility of hardware for MCMC	214
6.3.4 Off-line precision optimization	216
Bibliography	217

List of Tables

2.1	Combinations of precision configurations in the two precision domains of MCMC and the effects on: 1) convergence to the target distribution, 2) area savings/sampling throughput in an FPGA implementation. DP stands for double precision floating point, CP stands for custom precision floating point.	52
3.1	PT algorithm parameters	73
3.2	Baseline architecture memories. P is the number of sub-density pipelines, M , n and s are defined in Table 3.1	76
3.3	Case study parameters	92
3.4	Detailed list of platforms and devices	95
3.5	This table shows the PT parameter combinations for which actual FPGA bitstreams were generated and FPGA runs were performed, separately for each PT algorithm (baseline, WPT, MPPT). Also, it shows the combinations for which software runs were performed instead of FPGA runs. Software runs were implemented in C++ code, using the MPFR library for custom precision calculations.	96
3.6	Resource utilization of the various processing blocks. The numbers in parentheses next to each probability evaluation block pipeline are the numbers of mantissa and exponent bits of the precision configuration respectively. DP stands for double precision (53 mantissa and 11 exponent bits).	100

3.7	Power efficiency of the proposed algorithms and platforms for various (M, n) combinations. The efficiency improvement compared to the E5-2660 v2 CPU is shown in parenthesis (separately for Baseline and MPPT). The best platform per algorithm and per parameter combination is shown in bold.	111
4.1	Constant and variable parameters in the algorithms and architectures of this chapter .	131
4.2	Memories of the pMCMC/ppMCMC architectures	132
4.3	Resource utilization of the pMCMC/ppMCMC IPs and the respective FPGA systems. Architecture parameters were set to $B = 2$, $d_{rng} = 30$, $P_{max} = 8192$, $T_{max} = 8192$ (for both samplers), $M_{max} = 5$ (for ppMCMC). The numbers in the parentheses show what percentage of the available Z-7045 resources is needed for the given block.	152
5.1	Variables and constants in this chapter.	180
5.2	User parameters in the optimization method	184
5.3	Optimization results for various estimates and user parameters from the Mixture Model (MM) and the Neural Network (NN) case studies. In all cases: $Pr_{min} = 0.95$, $Term_{\%} = 0.05$ and $S = \{(24, 11), (20, 11), (16, 11), (14, 11), (12, 11), (10, 11)\}$	196

List of Figures

2.1	Prior, likelihood and normalized posterior for one-dimensional Bayesian model. . . .	11
2.2	MCMC algorithm drawing samples from a bivariate Gaussian with mean (5,5). A few burn-in samples are needed for convergence. Most samples are concentrated around the mean and fewer samples are found in the tails of the distribution.	13
2.3	The successive states (samples) of the Markov chain in MCMC. The transition kernel is also shown.	15
2.4	Example of a multi-modal target distribution. The distribution is a mixture of four bivariate Gaussian components with means (2,5), (4,8) (4.5, 5.5) and (1,7). Basic MCMC samplers tend to get “stuck” in one mode and need a lot of time to jump between modes (because it is rare to propose and accept a sample that leads to another mode when using simple proposals).	20
2.5	Parallel Tempering updates (GU) and exchanges (GE) with four tempered chains. Each chain samples from the density shown on the left side of the figure.	22
2.6	Hidden states, observations and latent parameters of state-space model.	25
2.7	High-level abstraction of a typical CPU. The instruction fetches read instructions from main memory, the instruction decoder sends the necessary control signals to implement the instruction and the ALU executes the instruction using inputs from the registers as operands and writing the results back to the registers or the main memory. .	40

2.8	Comparison of multi-core CPU and GPU architectures. SP stands for Stream Processor, the main processing unit of the GPU (following Nvidia's terminology). It is clear that the GPU devotes a much larger percentage of total chip area on compute units instead of control and cache.	42
2.9	Simplified FPGA architecture.	46
2.10	Shape of $p_c(\theta)$ when changing the precision configuration c . DP stands for double precision. Exponent bits are constant (=11).	53
3.1	Baseline FPGA architecture	75
3.2	Chain streaming through the Sample Proposal, Probability Evaluation and Update pipelines. Occupied stages are grey, unoccupied white. Numbers represent the chains that occupy each stage. There are four pipelines in the Probability Evaluation block ($P = 4$). The data size is $n = 16$. A probability value is generated every four cycles ($\lceil \frac{n}{P} \rceil = 4$). The Sample proposal and Update pipelines are under-utilized.	76
3.3	The GPU update kernel. Here, each block within the kernel handles two chains and four threads run in parallel inside each block, i.e. two threads per chain. The number of data is $n = 8$, which means that four data (i.e. four sub-densities or four tasks) are assigned to each thread. The threads which operate on the same chain pass through a reduction stage in order to give the total density of the chain. The shared memory of each block stores all the data. In a case where the shared memory is not large enough to store all data, the data are moved and processed in chunks.	82
3.4	FPGA architecture for WPT: The Custom precision PT block is the same as the system in Figure 3.1 but uses custom precision for probability evaluation. DP stands for double precision, CP stands for custom precision	88
3.5	FPGA architecture for MPPT: The double precision (DP) Probability Evaluation block has one pipeline and the custom precision (CP) Probability Evaluation block has multiple pipelines.	90

3.6	Scaling of $Speedup_{eff}$ with number of chains M . Baseline and custom precision accelerators are included. $n = 128$ were used. V7 measurements are based on projections and GTX285 measurements are based on GPU-Sim simulations.	103
3.7	Scaling of $Speedup_{eff}$ with number of data n . Baseline and custom precision accelerators are presented. $M = 32$ chains are used. V7 measurements are based on projections and GTX285 measurements are based on GPU-Sim simulations.	105
3.8	Scaling of $Speedup_{eff}$ when changing the size of the multi-core CPU (number of CPU cores). The exact CPU models used are listed in Table 3.4. Speedups are over the reference CPU version with no compiler optimizations. The $Speedup_{eff}$ of the second reference CPU implementation (which uses compiler optimizations) is also shown.	108
3.9	Scaling of $Speedup_{eff}$ when scaling the size of the GPU (number of SPs). All devices are based on the Fermi architecture (see Table 3.4).	109
3.10	Scaling of $Speedup_{eff}$ of the Virtex 7 FPGA series when varying the device size (number of DSP blocks). The points correspond to the devices shown in Table 3.4. .	110
3.11	WPT on FPGA: The effect of precision on the factors of (3.20). $(M, n) = (128, 128)$. The FPGA device used is the LX240T. The optimal number of mantissa bits is 14. All values in the vertical axis are ratios (either ratios of ESS values or ratios of speedups).	112
3.12	MPPT on FPGA: The effect of precision on the factors of (3.20). $(M, n) = (128, 128)$. The FPGA device used is the LX240T. The optimal number of mantissa bits is 14. All values in the vertical axis are ratios (either ratios of ESS values or ratios of speedups)	113
3.13	GPU kernel optimization: Cycles of global update kernel when the number of blocks and the number of tasks (data) per thread change. The parameter combination used is $(M, n) = (8192, 128)$. The GPU device used is the GTX285. Cycles are minimized when using 512 blocks and 32 tasks per thread.	114
4.1	FPGA architecture for pMCMC. Blocks with red dotted lines operate in fixed point arithmetic (see [1]).	130

4.2	FPGA architecture for ppMCMC. Blocks with red dotted lines operate in fixed point arithmetic (see [1]).	136
4.3	Coarse-grain pipelining in ppMCMC architecture for a specific SSM model and specific parameter configuration (P, T, M). The Resampling stage in this example has latency $Lat_{re} = 350$, which is the largest latency among the three PF stages. Thus a new chain can be fed to the PF datapath every $Lat_{re} = 350$ clock cycles. Here, three chains ($j = 3, j = 4, j = 5$) are inside the datapath, all of them at the same time step $t = 2$. Chain $j = 4$ will enter the Resampling stage one cycle after chain $j = 3$ exits the Resampling stage.	138
4.4	FPGA/host system prototype.	149
4.5	Raw speedup of GPU [2] and FPGA vs. multi-core CPU [2] implementation of pMCMC. The number of SSM states/time steps is set to $T = 1000$. Measured runtimes include time to transfer data between the devices and the hosts.	155
4.6	Raw speedup of GPU [2] and FPGA vs. multi-core CPU [2] implementation of pMCMC. The number of SSM states/time steps is set to $T = 16000$. Measured runtimes include time to transfer data between the devices and the hosts.	156
4.7	Total clock cycles consumed by each stage of the pMCMC architecture as the number of particles (P) changes. The remaining parameters are set to $T = 1000, B = 2, N = 10000, T_{max} = 16384, P_{max} = 16384$	157
4.8	Total clock cycles consumed by each stage of the pMCMC architecture as the degree of parallelism (B) changes. The remaining parameters are set to $T = 1000, P = 16384, N = 10000, T_{max} = 16384, P_{max} = 16384$. Five FPGA devices are shown in the top horizontal axis in order to demonstrate what degree of parallelism is possible with each device's available resources (assuming full resource utilization). An actual implementation was done only for Zynq Z-7045. The other four devices were placed in the graph based on projections.	158

4.9	<i>ESS</i> achieved by pMCMC sampler for various P when sampling from the uni-modal SSM posterior. The other parameters are set to $T = 1000$ and $N = 10000$. The fluctuations in the graph are due to the fact that the <i>ESS</i> value is approximated and thus variance is present.	159
4.10	<i>ES/sec</i> achieved by the multi-core CPU, GPU and FPGA pMCMC samplers for various P when sampling from the uni-modal SSM posterior. The other parameters are set to $T = 1000$ and $N = 10000$	160
4.11	<i>ESS</i> and <i>ES/sec</i> of pMCMC and ppMCMC in sequential Matlab implementations when the number of chains (M) and the number of particles (P) change. The multi-modal, multi-tissue SSM of section 4.5 is used as the target distribution. The SSM time steps are fixed to $T = 200$. $N = 10000$ samples are generated for each combination and the <i>ESS</i> and runtime are measured in order to compute <i>ES/sec</i>	163
4.12	Raw speedup of FPGA ppMCMC sampler compared to sequential Matlab implementation of ppMCMC and raw speedup of FPGA pMCMC sampler compared to sequential Matlab implementation of pMCMC. the number of chains (M) and the number of particles (P) change. The multi-modal, multi-tissue SSM of section 4.5 is used as the target distribution. The SSM time steps are fixed to $T = 200$	165
4.13	<i>ES/sec</i> of FPGA ppMCMC, FPGA pMCMC, GPU pMCMC and CPU pMCMC samplers when sampling from the multi-modal, multi-tissue SSM of section 4.5. The number of chains (M) and the number of particles (P) change. The SSM time steps are fixed to $T = 200$ and $N = 10000$	166
5.1	The same Normal target density (Mean = 0, Variance = 1) for different precision configurations.	174
5.2	Double and custom precision domains in an FPGA implementation of MCMC (Combination D/C in Table 2.1). The double precision domain corresponds to the generic MCMC operations. The custom precision domain corresponds to the Probability Evaluation block.	175

- 5.3 Double- and custom-precision ($c = (6, 11)$) sampling for the same output estimate $\tilde{I}$. The error bars are equal to two times the standard deviation of the estimate ($\pm 2 * \sigma_{\tilde{I}}$ for double and $\pm 2 * \sigma_{\tilde{I}_c}$ for custom precision. The true value of the estimated integral is $I = 4$ 177
- 5.4 Histograms of $f(\theta)$ and $f_{b_c}(\theta)$ for the mean estimator under $p_c(\theta)$ with $c = (13, 11)$, where $p_c(\theta)$ is a Gaussian mixture distribution (Section 5.5). The variance of $f(\theta)$ is significantly larger. 181
- 5.5 Ratios $\frac{\sigma_{b_c}^2}{\sigma_{\tilde{I}_c}^2}$ (variance of proposed bias estimator over variance of custom precision output estimator) and $\frac{\sigma_{str}^2}{\sigma_{\tilde{I}_c}^2}$ (variance of “straightforward” bias estimator over variance of custom precision output estimator) for various mantissa bit configurations (exponents bits=11). $p_c(\theta)$ is a Gaussian mixture distribution (Section 5.5) and $f(\theta)$ is the mean estimator. $N = 10^6$ samples are used for all estimators. The proposed estimator’s variance ($\sigma_{b_c}^2$) is orders of magnitude smaller than that of the output estimator ($\sigma_{\tilde{I}_c}^2$) for most precisions. The straightforward estimator’s variance (σ_{str}^2) is always larger than $\sigma_{\tilde{I}_c}^2$ 182
- 5.6 Optimization method flow: Steps 1,2 (mixed-precision FPGA bitstream), 3 (host PC software) and 4 (optimized-precision FPGA bitstream) are shown. During Step 1, the double-precision module is used to draw MCMC samples. During Step 2, it evaluates the probabilities of all generated samples, which are used for weight evaluations. Grey-coloured blocks (Step 1) are unused. The top part of the figure shows the information passed by each step to the next. 183
- 5.7 Resource utilization of a single probability evaluation pipeline (many of which can exist in a probability evaluation block). The pipeline implements the MM posterior using precisions with various mantissa sizes (exponent bits are always 11). The generic MCMC block’s resources (which always operate in double precision and are not part of the precision optimization) are also shown. 192
- 5.8 Step 2: Bias estimates for 3 out of the six 6 candidate precisions. The MM case study is targeted and the output estimate is the 2nd moment of μ_1 . Error bars represent $\pm 2\sigma_{b_c}$. The vertical line shows when the pre-runs need to stop to avoid large overheads to the final run (the line is set based on the output of Step 1). 194

5.9	Step 3: Probability that the bias is within tolerance for five out of the six candidate precisions (Precision $c = (24, 11)$ is omitted). The output estimate is the 2nd moment of μ_1 (MM case study). Parameters are set to $SD_T = 0.02$, $T_{bias} = 0.5$, $Pr_{min} = 0.95$, $Term_{\%} = 0.05$. The optimized precision is $c = (16, 11)$	195
5.10	Trade-off between tolerable bias and speedup (vs. double precision FPGA sampler). Bias tolerance is represented by the two parameters SD_T and T_{bias} . The output estimate is the mean of μ_1 (MM case study). The other user parameters are set to $Pr_{min} = 0.95$, $Term_{\%} = 0.05$ and $S = \{(24, 11), (20, 11), (16, 11), (14, 11), (12, 11), (10, 11)\}$	198
5.11	Speedup of the method proposed in this chapter against ported unbiased method of Chow et al. [3] for various values of T_{bias} . The target standard deviation is $SD_T = 0.007$. The output estimate is the mean of μ_1 (MM case study). The other user parameters are set to $Pr_{min} = 0.95$, $Term_{\%} = 0.05$ and $S = \{(24, 11), (20, 11), (16, 11), (14, 11), (12, 11), (10, 11)\}$. The chosen (optimized) precisions of the method proposed in this chapter are shown above each point in the graph. The low precision of the ported unbiased method is always $c = (16, 11)$ (optimized using the process described in [3]). The runtimes of both methods include the time spent for pre-runs, as well as the time spent for the final runs.	202

Chapter 1

Introduction

1.1 Motivation

Generating random samples according to a given probability distribution (i.e. sampling from a distribution) is a common task in a wide range of scientific fields. Depending on the application, the random samples can be used to estimate integrals, infer model parameters, make predictions based on previous observations, compare competing models or perform stochastic optimization (although this list is far from complete).

There are various techniques to sample from a probability distribution [4]. Many of them are limited to particular classes of distributions and/or cannot address large dimensions. For example, several well-known methods exist to sample from standard distributions such the Normal and Uniform distributions [5]. More general techniques like Importance Sampling and Rejection Sampling can theoretically sample from any given distribution but in practice they can be successfully applied only to non-complex distributions with few dimensions [4, 6].

In contrast, Markov Chain Monte Carlo (MCMC) is a class of stochastic methods which are designed to draw samples from any probability distribution, regardless of dimension or complexity. Complexity here refers to features of the distribution such as multi-modality and correlations [7, 8, 9]. MCMC has become the mainstream tool to sample from posterior distributions in Bayesian modelling during the last 25 years. This is because the structure of Bayesian models and the large amounts of data they handle lead to complex, high-dimensional posterior distributions. MCMC's flexibility and efficiency make it capable of sampling from these posteriors. In fact, MCMC has allowed the application of

Bayesian modelling to many new domains and the analysis of massive data sets [9, 10, 11, 12], practically revolutionizing Bayesian statistics and statistical computing in general. The development of new MCMC methods is still a very active research topic today and MCMC's position as the standard method to perform Bayesian inference is likely to be maintained in the future.

Despite their success, MCMC samplers are often not as fast as modern Bayesian applications require. Runtimes can easily reach weeks or months [9, 13]. The two main reasons for this are: 1) The widespread use of large-scale data sets. For example, models used in genetics require the processing of large sets of genetic predictors [10] or even whole genomes [14]. 2) The use of advanced MCMC methods with significant computational overheads. For example, Population-based MCMC (popMCMC) and Particle MCMC (pMCMC) (which are examined in this thesis) are two algorithms which allow the tackling of extremely complex posterior distributions but their computational intensity limits their applicability in many real situations. Today, these two trends (the ubiquitousness of “big data” in many sectors of industry and research, as well as the increasing adoption of non-standard, computationally intensive MCMC methodology) are fuelling a large increase in the computational workload of Bayesian inference, which is likely to continue in the future. Practically, this means that MCMC practitioners are often forced to accept less precise results, use simpler models or exclude data from the analysis in order to reduce runtimes.

1.2 Aims and outline of this study

In order to allow statisticians to keep drawing meaningful conclusions from the exploding amount of data that is generated today, this thesis claims that it is insufficient to rely solely on better models or smarter, more efficient MCMC methods (the two main research directions pursued by that the statistics community until recently). It is equally important to leverage the power of modern, massively parallel hardware accelerators. What is more, an effort has to be made to combine statistical and engineering expertise when performing hardware-accelerated MCMC inference. There are many potential gains in MCMC performance when: 1) MCMC algorithm design takes the underlying hardware platform into consideration and 2) The mapping of MCMC to hardware is supported by a full understanding of the features of the algorithm and how these can be exploited in the employed hardware platform.

Due to the increasing computational demands on MCMC, various studies on the use multi-core Central Processing Units (CPUs), CPU clusters and Graphics Processing Units (GPUs) for accelerating

MCMC have been published in the last five years, e.g. [15, 16, 17, 18, 19]. However, the largest part of previous literature is limited to blindly mapping the aforementioned methods into the targeted platform, without any concern for adapting the algorithm to make it more suitable for the platform. Moreover, the use of Field-Programmable Gate Arrays (FPGAs), a unique and powerful family of digital circuits, as MCMC accelerators has been largely unexplored. FPGAs are armed with a number of characteristics that match well with the needs of MCMC sampling, e.g. massive parallelism and pipelining capabilities, fully customizable architecture, custom arithmetic precision, fast inter-circuit communication. Although some existing studies have shown the potential of FPGAs to make MCMC sampling feasible in problems where it was previously impractical [19, 18], these works have focused on the exploitation of model-specific parallelism, i.e. taking advantage of the form of the posterior density to improve performance. There is a lack of results on algorithm-specific mappings, i.e. taking advantage of the characteristics of specific MCMC algorithms to create tailored FPGA architectures. Also, no previous work has investigated how certain characteristics of the underlying platform (e.g. custom precision) can be exploited regardless of the form of the posterior density.

This thesis aims to fill the above gaps by proposing ways in which MCMC algorithm design and FPGA architecture design can be done jointly, with the purpose of exploiting the various special features of FPGAs to increase MCMC sampling efficiency in complex problems. The main chapters of the thesis propose MCMC algorithms and FPGA architectures that offer significant performance improvements over existing solutions. Moreover, special attention is given on how to increase performance by reducing arithmetic precision in the FPGA. Reduced precision translates to arithmetic operators that consume less area in the FPGA fabric. This allows more parallel operators to be instantiated in the same device, thus increasing performance. The thesis also investigates the consequences of such a strategy on sampling accuracy. The performance metrics employed for comparison with state-of-the-art samplers are the sampling throughput (how many samples can be produced per second), the scalability of each architecture (how sampling throughput scales with the size of the problem) and the power consumption of each system. The results presented in the thesis indicate that it is possible to achieve speedups of one to two orders of magnitude against state-of-the-art algorithms and hardware accelerators using the proposed algorithmic techniques and FPGA implementations. The contributions contained in this thesis open the way for the handling of extremely complex Bayesian models and for performing previously intractable “big data” analyses in a variety of research fields.

Chapter 2 contains all the background information needed to follow the remaining chapters. Bayesian

inference and the basic principles of MCMC are explained and the most common MCMC algorithms are described, with particular attention on the algorithms that this thesis targets. Moreover, the reasons behind MCMC's computational intensity are clarified and an overview of the most popular computing platforms (which are candidates for acting as MCMC accelerators) is given. A separate section is devoted to arithmetic precision and the ways in which it can be customized in MCMC. Finally, a literature review is provided.

Chapter 3 investigates ways in which popMCMC samplers, a class of MCMC methods designed to sample from multi-modal distributions, can be accelerated using FPGAs. A tailored FPGA architecture is proposed and compared to CPU and GPU accelerators. Most importantly, the chapter explores ways in which custom precision can be used to improve performance in popMCMC. Two modifications to the basic popMCMC algorithm are introduced which allow for precision reduction in the largest part of the FPGA system without any adverse effect on sampling quality. Precision reduction enables more parallelism on the FPGA (since arithmetic operators consume less FPGA area) and thus higher sampling throughput. The two custom precision techniques are also mapped on the CPU and GPU platforms. The proposed implementations are evaluated using a mixture model inference case study.

Chapter 4 focuses on another computationally intensive MCMC variant, pMCMC, whose runtime is dominated by the Particle Filter run which is necessary in each MCMC step. The chapter proposes a custom FPGA architecture which exploits parallelism and pipelining to increase pMCMC performance. Moreover, a novel MCMC algorithm, denoted Population-based Particle MCMC (ppMCMC), is proposed; ppMCMC is designed to address multi-modal posteriors which are a challenge for regular pMCMC samplers. The new algorithm is accompanied by a tailored FPGA architecture which takes advantage of the multiple chains of ppMCMC in order to increase the utilization of the PF datapath. The algorithm and architectures are compared to state-of-the-art solutions on CPUs and GPUs. Evaluation is performed using a large-scale State-Space Model inference problem taken from genetics.

Chapter 5 pursues the goal of accelerating MCMC via a different path. Instead of focusing on a particular MCMC algorithm, it proposes a generic FPGA-based methodology to optimize the precision used in the most computationally intensive part of MCMC, the target density evaluation. The methodology is based on performing short MCMC pre-runs on a set of candidate precisions (using a first FPGA bitstream). These pre-runs are used to estimate the bias that each precision introduces in the output, using an efficient bias estimator proposed in this chapter. A probabilistic criterion is then

used to choose the minimum precision which satisfies a user-defined bias tolerance. A second FPGA bitstream which operates in the chosen (optimized) precision is then loaded on the FPGA and used to get the final estimate. The methodology is compared to a double-precision FPGA sampler, as well as a competing methodology which does not introduce bias. The case studies used are a mixture model and a neural network inference problem.

Finally, Chapter 6 summarizes the work presented in this thesis and gives an overview of the current state of MCMC acceleration research. Potential avenues for future research based upon the findings of this thesis are also proposed.

1.3 Statement of Originality

This thesis contains a number of original contributions to the field. The following list summarizes them (a more detailed discussion of contributions is given in the introductory section of each chapter):

- An FPGA architecture for popMCMC is proposed, which maps the various parts of the algorithm to hardware, taking advantage of its inherent parallelism. This architecture delivers significant speedups over CPUs and GPUs.
- An investigation is performed on how custom precision arithmetic can be reduced in popMCMC without affecting sampling accuracy. Based on this investigation, two custom precision-based modification to the popMCMC algorithms are introduced, which permit the reduction of the employed precision for most popMCMC computations without introducing error in the output result. The first approach corrects sampling errors caused by reduced precision, while the second approach avoids them altogether. Justification of the correctness of the two approaches (i.e. proof that they sample from the distribution without introducing bias) is also included.
- Two novel architectures which map the two custom-precision popMCMC algorithms to an FPGA are proposed. These architectures exploit the area saving that an FPGA implementation enjoys when reduced precisions are used, thus achieving higher computational efficiency than the baseline popMCMC architecture. A precision optimization process which maximizes the gains from using custom precision is also described. A multi-core CPU and a GPU implementation of the two custom precision methods are presented.

- A novel FPGA architecture for accelerating the pMCMC sampler is proposed, which maps the various parts of the algorithm to hardware, parallelizing all Particle Filter operations to increase computational efficiency. The architecture is shown to be faster than CPU and GPU pMCMC samplers.
- The inefficiency of pMCMC when it samples from multi-modal distributions is tackled by introducing a novel MCMC algorithm, denoted ppMCMC, which is a combination of pMCMC and popMCMC. ppMCMC uses multiple MCMC chains (instead of the one chain used in pMCMC) and achieves higher sampling efficiency for multi-modal distributions compared to pMCMC. A design space exploration which chooses the optimal combination of ppMCMC's parameters is also included.
- An FPGA architecture tailored for ppMCMC is presented, which exploits the structure of the algorithm to maximize computational efficiency. The architecture pipelines the multiple chains of ppMCMC in order to increase datapath utilization in the Particle Filter module of the FPGA system. A separate design space exploration is performed for the FPGA-mapped version of ppMCMC.
- A novel precision optimization methodology applicable to any FPGA-mapped MCMC algorithm is devised. This methodology estimates the output error introduced by various candidate precisions and chooses an optimized precision which probabilistically satisfies a user-defined error tolerance. This is the first approach in the literature towards generic precision optimization, i.e. irrespective of the employed MCMC algorithm or the target probability density.

1.4 Publications

The following articles have been published during the course of this thesis:

- G. Mingas and C.-S. Bouganis. "Parallel tempering MCMC acceleration using reconfigurable hardware". *Reconfigurable Computing: Architectures, Tools and Applications (ARC)*, Springer Berlin Heidelberg, 227-238, 2012.
- G. Mingas and C.-S. Bouganis. "A custom precision based architecture for accelerating parallel tempering MCMC on FPGAs without introducing sampling error". *IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012.

- G. Mingas, F. Rahman and C.-S. Bouganis. “On Optimizing the Arithmetic Precision of MCMC Algorithms”. *IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2013.
- G. Mingas and C.-S. Bouganis. “Population-based MCMC on multi-core CPUs, GPUs and FPGAs”. *IEEE Transactions on Computers*, 2015 (to appear - pre-print available online)
- G. Mingas and C.-S. Bouganis. “Accelerating MCMC for large-scale Bayesian inference using FPGAs”. *IEEE/RSJ IROS Workshop on Unconventional computing for Bayesian inference (UCBI)*, 2015

Chapter 2

Background and related work

Since this thesis lies at the boundary between Markov Chain Monte Carlo algorithms and hardware architecture design, this chapter gives the necessary background on each of these topics. It begins with a short introduction to the basic concepts of Bayesian modelling and inference (Section 2.1). Bayesian inference is by far the most important class of MCMC applications and throughout this thesis, examples from this class are used to evaluate the performance of the proposed algorithms and architectures. Section 2.2 provides an introduction to the basic ideas behind MCMC as well as some theoretical background. Section 2.3 contains detailed descriptions of popular MCMC methods, focusing mostly on the ones targeted in this thesis. Section 2.4 highlights the reasons why faster MCMC is a necessity given the complexity of modern applications and lists some of the most recent developments in MCMC methodology which attempt to address the issue of long runtimes. Section 2.5 comprises a comprehensive introduction and comparison of the various computing platforms which can be employed to accelerate MCMC. Moreover, a separate section (Section 2.6) is devoted to custom arithmetic precision and what it means in an MCMC setting, since several of the novelties of this thesis are based on tuning the precision of FPGA implementations.

Following the above background information, Section 2.7 gives an extensive review of the current literature and highlights the issues that remain unresolved until today. It examines previous work on parallelizing and accelerating MCMC using various hardware platforms, methods based on customizing the precision of stochastic algorithms, as well as other techniques which cannot be classified in the above categories. The section also contains comparisons between current literature and the contributions included in this thesis, in order to clarify the gaps that the present work attempts to cover.

2.1 Introduction to Bayesian modelling and inference

Modelling is one of the most fundamental tools used by scientists in order to understand natural phenomena, make predictions, perform classification and test hypotheses. Modelling is also a necessary tool for machines to exhibit intelligence, since intelligence consists in being able to learn from past observations (e.g. through classification) and make predictions for the future [20].

All models attempt to provide a simplified representation of reality, i.e. a representation of observable and non-observable quantities as well as the relationships between them. Bayesian modelling is no different; the core idea behind it is to use the mathematics of probability theory to represent all forms of uncertainty in the model and the relationships between them. In a nutshell, the Bayesian framework treats all variables (known and unknown) as random variables and attempts to infer the probability distributions of the unknown variables given the observed data.

In more detail, let $\mathbf{D} = \{\mathbf{D}_1, \dots, \mathbf{D}_{n_D}\} = \{\{D_{11}, \dots, D_{1n_d}\}, \dots, \{D_{n_D1}, \dots, D_{n_Dn_d}\}\} \in \mathbb{R}^{n_D \times n_d}$ be the set of observed data. Here, for reasons of simplicity, the data are represented as a set of data vectors. Each data vector has dimension n_d and the whole data set consists of n_D data vectors. The data vector index could express time, space, etc but this is not necessarily applicable to all problems. The aim is to explain these data, i.e. infer a model which could have “predicted” them (or “anticipated” them). This model can be then used to make new predictions in the future, etc. Bayesian modelling requires the specification of two probability distributions:

- The probability distribution of the model, whose density is $p(\mathbf{Y} \mid \theta)$, where $\theta \in \mathbb{R}^{n_\theta}$ are the unknown parameters of the model and $\mathbf{Y} \in \mathbb{R}^{n_D \times n_d}$ is some set of observations. Note that although θ can be a vector, the bold font is not used to maintain consistency with the literature. The above density gives information on which sets of observations are likely and which are not, assuming that the model is known (i.e. θ is known). In other words, it is a function that predicts likely observations when the model is known. This density is also called the **likelihood function** (or **likelihood of the parameters**) in Bayesian modelling. It summarizes the information that can be inferred about the model based only on some set of observed data.
- The **prior probability distribution** on the unknown parameters θ , which has density $p(\theta)$. This density represents the information about the model that exists before any data are collected. This information could be provided by an expert or it could come from previous models which have

been calibrated based on older observations. The prior sets a range for the unknown parameters and also gives information which values of the parameters are more likely before any data are observed.

The above two distributions (likelihood and prior) can be combined using the law of Bayes [21] to give the **posterior probability distribution** of the unknown parameters θ , given the data $\mathbf{Y}$:

$$p(\theta | \mathbf{Y}) = \frac{p(\mathbf{Y}|\theta)p(\theta)}{p(\mathbf{Y})} = \frac{p(\mathbf{Y}|\theta)p(\theta)}{\int p(\mathbf{Y}|\theta')p(\theta')d\theta'} \quad (2.1)$$

where the denominator $p(\mathbf{Y}) = \int p(\mathbf{Y} | \theta')p(\theta')d\theta'$ is called the marginal or integrated likelihood in the literature and it is typically hard or impossible to compute. Nevertheless, computing it is not necessary when using MCMC, as will be explained in the following sections. It has to be noted that integral limits (which are $-\infty$ and $+\infty$ when the integral is over probability densities) are omitted in the remaining of this thesis, as is the common practice in MCMC literature.

Figure 2.1 shows a simple one-dimensional example where a Gaussian likelihood and a Gaussian prior are combined to give the posterior of the parameter θ . The prior is wide, representing the initial uncertainty about the value of the unknown θ . The likelihood is more narrow, signifying that the data provide strong evidence that the true value of θ is close to 5. The posterior is affected by both the prior and the likelihood but it is much closer to the likelihood, since the data's influence is stronger. This is a common pattern in Bayesian modelling; large amounts of data and concentration of the data move the posterior closer to the likelihood function and away from the prior (see [21] for more details on this mechanism).

After setting $\mathbf{Y} = \mathbf{D}$ in the above equation (i.e. using the available data), it is possible to use the posterior to perform a series of inference tasks, e.g. infer the unknown model parameters from the observed data, make predictions about future data and compare competing models based on the available data. All of these tasks can be expressed as the evaluation of an integral of the following form:

$$E_{p(\theta|\mathbf{D})}[f(\theta)] = \int f(\theta)p(\theta | \mathbf{D})d\theta \quad (2.2)$$

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 posterior distribution.

For example, in order to infer the mean of θ given the data $\mathbf{D}$, it is enough to set $f(\theta) = \theta$. The integral

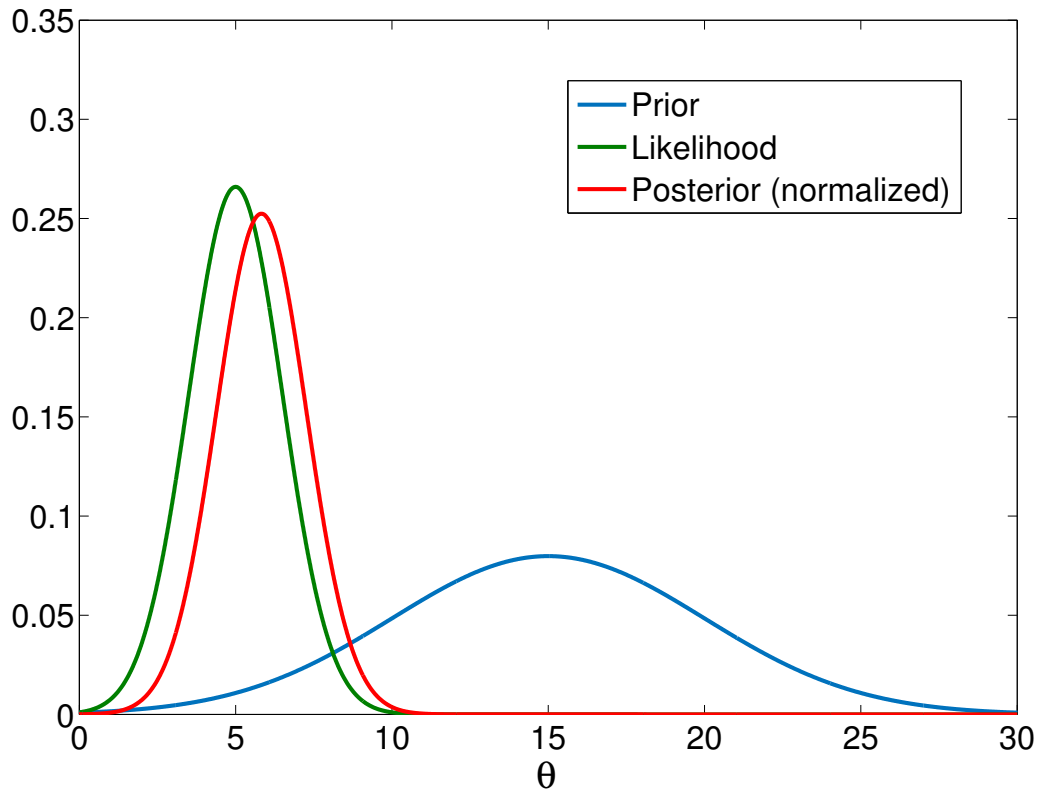


Figure 2.1: Prior, likelihood and normalized posterior for one-dimensional Bayesian model.

(2.2) then takes the form $\int \theta p(\theta | \mathbf{D}) d\theta$, i.e. the expectation of θ under the posterior. In a similar way, it is possible to infer the various moments of θ , its tails, etc. Similarly, by setting $f(\theta) = p(\mathbf{Y} | \theta)$, where again $\mathbf{Y}$ is some set of possible observations, it is possible to predict future observations. In this case the integral (2.2) takes the form $\int p(\mathbf{Y} | \theta) p(\theta | \mathbf{D}) d\theta = p(\mathbf{Y} | \mathbf{D})$, which gives the probability of observing the data $\mathbf{Y}$ in the future, given that $\mathbf{D}$ has already been observed.

This simple (yet powerful) framework has been applied to a vast range of scientific areas: Machine learning [22, 11, 18, 20], statistical physics [19, 6], economics [23], ecology [12], geostatistics [24], medical imaging [9], genetics [13], phylogenetics [25], computational biology [26, 27] and stochastic optimization [15, 10], among others.

An example application: A simple example is described here to clearly illustrate how the Bayesian framework is applied in practice. Let $\mathbf{D} = \{D_1, \dots, D_{1000}\}$ be the measured weights of 1000 children in London right after childbirth. Comparing with the notation presented previously in this section, it is clear that $n_D = 1000$ and $n_d = 1$. Assume that the goal is to infer a model which can predict the likely weight of the next child born in London, taking into account the above data. Assume also that

the weight of a child born in London is distributed according to a Normal distribution with unknown mean μ (measured in kilograms) and known variance $\sigma^2 = 0.3$. In practice, the variance is not known but this is ignored here for simplicity. Also, other models could be used instead of a Normal model, e.g. the Student-t distribution.

In this case, the unknown parameter of the Bayesian model is $\theta = \mu$ (the mean weight of children born in London) and the likelihood of θ given the known data set $\mathbf{D}$ is:

$$p(\mathbf{D} | \theta) = \prod_{i=1}^{1000} N(D_i | \theta, 0.3) \quad (2.3)$$

where $N(D_i | \theta, \sigma^2)$ is the density of the Normal distribution with mean θ and variance $\sigma^2 = 0.3$ evaluated at point $D_i, i \in \{1, \dots, 1000\}$. Moreover, let the prior on the parameter θ be Normal with mean 3.4 and variance 0.3:

$$p(\theta) = N(\theta | 3.4, 0.3) \quad (2.4)$$

The above prior could come from previous UK-wide studies on the weight of children at childbirth. The parameters of the prior are called hyperparameters in Bayesian literature.

The resulting posterior of θ is the following:

$$p(\theta | \mathbf{D}) = \frac{p(\mathbf{D} | \theta)p(\theta)}{p(\mathbf{D})} = \frac{\prod_{i=1}^{1000} N(D_i | \theta, 0.3)N(\theta | 3.4, 0.3)}{\int \prod_{i=1}^{1000} N(D_i | \theta', 0.3)N(\theta' | 3.4, 0.3)d\theta'} \quad (2.5)$$

Using the above posterior, it is possible to predict the weight Y of the next child to be born in London:

$$p(Y | \mathbf{D}) = \int p(Y | \theta)p(\theta | \mathbf{D})d\theta = \int N(Y | \theta, 1)p(\theta | \mathbf{D})d\theta \quad (2.6)$$

Note that prediction in Bayesian inference is not done based on a simple value of the unknown parameter, but rather on the whole posterior probability distribution of the unknown parameter. Apart from prediction, other tasks (e.g. model comparison, finding moments of the posterior) can be performed in a similar way.

The only remaining problem is how to compute the above integrals, which is far from straightforward. In real applications, both the unknown parameters and the data can have much larger dimensions than in this example, making it impossible or impractical to evaluate the integrals analytically or using numerical methods [6]. The following section explains how MCMC tackles this problem.

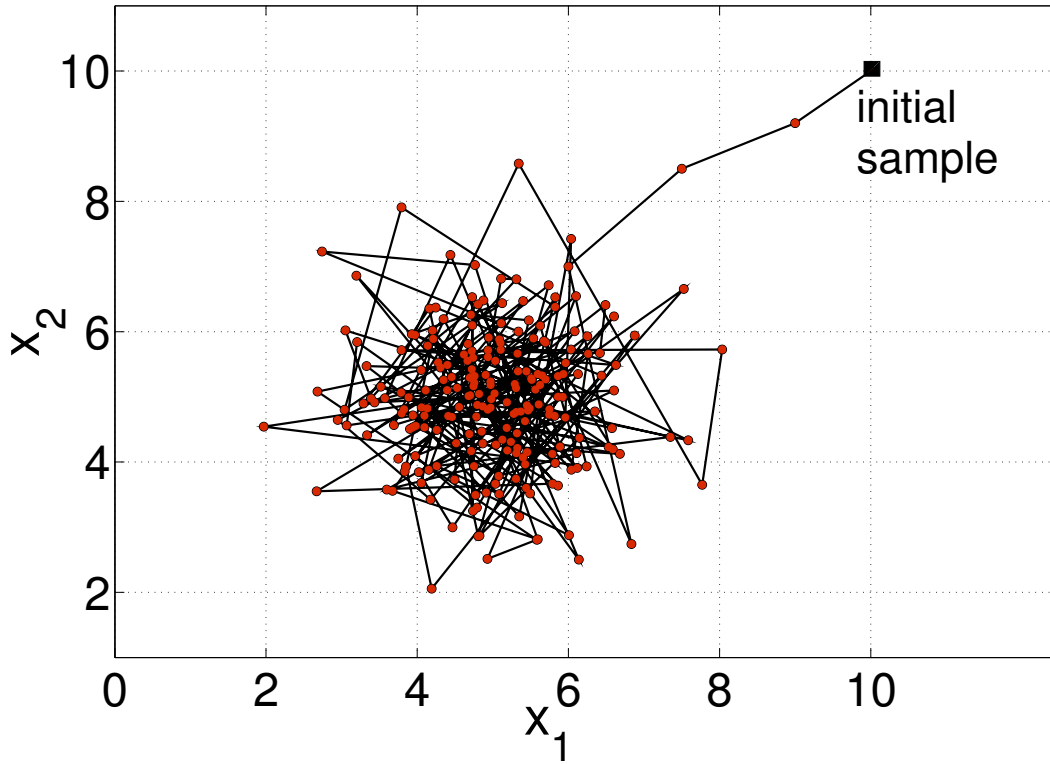


Figure 2.2: MCMC algorithm drawing samples from a bivariate Gaussian with mean (5,5). A few burn-in samples are needed for convergence. Most samples are concentrated around the mean and fewer samples are found in the tails of the distribution.

2.2 Background on Markov Chain Monte Carlo

Integrals that take the form (2.2) can be evaluated analytically only for simple problems. Also, numerical integration works only for small dimensions of θ and $\mathbf{D}$ [6, 20]. For most real problems, which have larger dimensions, these algorithms become intractable because they scale exponentially with the dimension. In these cases, stochastic methods, such as MCMC [4, 9, 6], perfect/exact sampling [28] and particle-based methods (i.e. Sequential Monte Carlo) [29, 6], are typically used. MCMC is by far the most popular among these algorithms and this thesis focuses on it (although one of the MCMC methods that this thesis targets is a combination of MCMC and particle-based methods - see Chapter 4).

MCMC is a method designed to generate random samples from any given probability distribution. Figure 2.2 shows the samples generated by an MCMC algorithm which targets a bivariate Normal distribution. The sampler moves around the support of the distribution spending most of the time in areas of high probability. Most samples are generated in areas of high probability and fewer samples in areas of low probability.

MCMC was originally designed for use in Physics simulations and not for Bayesian inference problems [30]. Nevertheless, almost three decades ago [31] it was rediscovered as a method suitable for estimating the integral (2.2); MCMC can generate samples distributed according to the posterior (2.1) (which is called the target distribution of MCMC in this setting). Given these samples, the following approximation can be evaluated:

$$\tilde{E}_{p(\theta|\mathbf{D})}[f(\theta)] = \frac{1}{N} \sum_{i=1}^N f(\theta^{(i)}) \quad (2.7)$$

where $\theta^{(i)}$, $i \in \{1, \dots, N\}$ are samples taken from $p(\theta | \mathbf{D})$. The variance of this estimator reduces linearly with N and the estimate converges to the true value of the integral (2.2) asymptotically (i.e. for $N \rightarrow \infty$). Therefore, there is a trade-off between runtime (N) and the accuracy of the estimate (which is expressed by its variance). Practitioners are typically interested in reducing variance to a level that is adequate for their purposes.

In the remaining of this thesis, instead of the equations (2.2) and (2.7), the following two equations are used respectively:

$$E_{p(\theta)}[f(\theta)] = \int f(\theta) p(\theta) d\theta \quad (2.8)$$

$$\tilde{E}_{p(\theta)}[f(\theta)] = \frac{1}{N} \sum_{i=1}^N f(\theta^{(i)}) \quad (2.9)$$

where the conditioning on the data $\mathbf{D}$ has been dropped for simplicity and thus the posterior is now denoted $p(\theta)$. Conditioning on the data will be used only in specific cases (Section 2.3.3). Whenever $p(\theta)$ refers to the prior, this will be explicitly mentioned.

Using random samples to estimate integrals in the way described above is known as Monte Carlo integration. MCMC is not the only stochastic sampling method that can be used in this context. If the distribution inside the integral has some simple form (e.g. Normal), there are many other methods to generate samples from it [5]. Also, other Monte Carlo methods such as Quasi Monte Carlo [32] can be used for Monte Carlo integration, instead of MCMC.

MCMC's biggest advantage compared to other Monte Carlo methods is its ability to sample from any target distribution (even if it is complex and multi-dimensional) as long as its density can be evaluated up to a normalizing constant (although some newer methods [33] do not even require this). The way the normalizing constant evaluation is avoided is explained later in this chapter. Apart from the flexibility that MCMC offers to practitioners, it is also a good match for Bayesian inference problems,

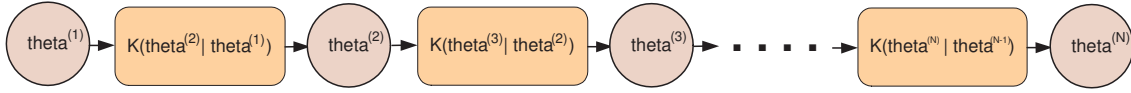


Figure 2.3: The successive states (samples) of the Markov chain in MCMC. The transition kernel is also shown.

since the posterior (2.1) can typically be evaluated only up to a normalizing constant; the marginal likelihood in the denominator of the posterior is intractable in most cases, as mentioned above.

As a result of the above feature, the numerous variants of MCMC have become the mainstream tools to perform Bayesian inference during the last two decades. In fact, MCMC's flexibility and efficiency have enabled the application of the Bayesian framework to many new domains and the analysis of massive data sets, practically revolutionizing Bayesian statistics (reference can be found in the previous section).

2.2.1 MCMC principles

MCMC generates samples from the posterior by constructing a Markov chain [4]. Each state of the chain is equivalent to a random sample ($\theta^{(i)}$ for state i). The basic idea behind all MCMC algorithms is the following: At each step $i \in \{1, \dots, N\}$ of the algorithm, the next state of the chain ($\theta^{(i+1)}$) is generated using a kernel $K(\theta^{(i+1)} | \theta^{(i)})$. The kernel function depends only on the previous state of the chain ($\theta^{(i)}$). All Markov chains have the property that each state depends only on the previous one. The kernel is called the transition kernel of the chain. Figure 2.3 shows how the transition kernel is applied between successive states.

Subject to regulatory conditions [34], the distribution $K(\cdot | \theta^{(1)})$ will gradually *converge* to a stationary distribution $g(\theta)$ which does not depend on i or the initial sample. This means that after an initial burn-in period, during which samples are not distributed according to the stationary distribution, the chain starts generating samples from $g(\theta)$ (as shown in the example of Figure 2.2).

In order for the stationary distribution to be equal to the target distribution ($g(\theta) = p(\theta | \mathbf{D})$), a suitable transition kernel, i.e. a suitable way to draw a new sample given the current one, has to be used. In this context, “suitable” means that the design of these kernels has to adhere to a series of theoretical limitations (see [34]). Transition kernel design is supported by a vast literature on Markov chains and their use for MCMC [4, 34, 31, 6]. Typically, a transition kernel proposes a candidate sample using a simpler distribution and then accepts or rejects the sample based on some criterion (which requires the

evaluation of the target density, i.e. posterior). Nevertheless, this is not the case in all MCMC variants.

After a large enough number of samples has been generated in order to adequately approximate the integral (2.8), the algorithm stops and the estimate (2.9) is evaluated. It is worth noting that, in contrast to other Monte Carlo methods like Quasi Monte Carlo which generate independent samples, MCMC generates dependent (i.e. correlated) samples from the given distribution. This happens due to the Markov property of the chain. Practically, the use of dependent samples means that the estimate (2.9) when using N MCMC samples has larger variance than the same estimate when using N independent samples. The difference in variance can be quantified using the Effective Sample Size (see Section 2.2.2).

2.2.2 Convergence and mixing

As mentioned above, the Markov chain in MCMC converges to the stationary distribution (posterior) gradually. This means that the initial samples are typically not distributed according to the correct posterior (they are “irrelevant”) due to bad initialization (the initial sample is chosen at random or based on some simple distribution, e.g. the prior). The sampler thus needs some time to converge. The number of “erroneous” samples needed before convergence occurs depend on the initial sample, the shape and dimension of the posterior, the MCMC algorithms and how this algorithm has been tuned. The “erroneous” samples, which are called burn-in samples, need to be discarded in order for the final samples to be distributed according to the desired posterior. Nevertheless, there is no rule to determine the length of the burn-in sequence. The decision is often made based on observation of the samples to understand when the sampler has converged. It is also usual practice to perform multiple independent MCMC runs, compare the resulting samples and set the burn-in length so that the samples of all runs “look alike”. Several criteria have been proposed to decide when the independent runs “look alike” and thus the sample has converged, e.g. the Gelman-Rubin criterion [35]. Nevertheless, these criteria do not actually guarantee that convergence has occurred; it is possible that all independent runs look like they have converged for a long time but if the runs continue they eventually move to other areas which were unknown before.

After the sampler has converged, the newly generated samples explore the support of the distribution. It is crucial that this exploration happens fast in order to get a satisfactorily accurate estimate of the integral (i.e. an estimate with satisfactorily small variance) without running the sampler for too long.

Fast exploration means that the sampler moves quickly around the support of the posterior and covers all areas without getting stuck in one particular part of the support (e.g. one mode). The speed of exploration is called *mixing speed* in MCMC literature [9, 34]. Mixing is related to the dependencies (correlations) between MCMC samples. If the MCMC kernel is designed so that subsequent samples are not highly correlated, mixing is fast. In contrast, if subsequent samples are highly correlated, the algorithm mixes slowly. The main aim of all MCMC algorithms is to maximize mixing speed, so as to achieve accurate output estimates with few samples.

The Effective Sample Size (or *ESS* or Effective Sample Size due to autocorrelation) [9] is the most common metric of mixing speed in MCMC literature. *ESS* gives an estimate of how many independent (or “effective”) samples the dependent MCMC samples are equivalent to, i.e. it quantifies the samples’ “exploration value”. Moreover, the ratio of N over *ESS* can be interpreted as the factor by which the variance of the output estimate (2.9) under MCMC is larger than the variance of the same estimate under a theoretical algorithm which generates independent samples. This interpretation will be used in Chapters 3 and 4 of this thesis. *ESS* can be estimated using the MCMC samples’ autocorrelations [36, 9]:

$$ESS = \frac{N}{1 + 2 \sum_{k=1}^N \rho(k)} \quad (2.10)$$

where $\rho(k)$ is the autocorrelation at lag k . The summation in (2.10) is typically truncated when $\rho(k)$ drops below 0.1 [36, 8] to reduce the variance of the *ESS* estimator.

2.3 MCMC algorithms

The first MCMC algorithm is the Metropolis method, which was introduced in the seminal paper of Metropolis et al. in 1953 [30]. Nevertheless, the realization that Markov chains can be used in the context of Bayesian statistics only came with the papers of Geman and Geman in 1984 [31] and especially Gelfand and Smith in 1990 [37].

Since then, and especially during the last 20 years, the increasing use of MCMC in Bayesian statistics has fuelled a vast amount of research on new MCMC algorithms. The design of an MCMC method essentially consists in proposing a transition kernel to generate a state of the chain given the previous state. Different kernels are suitable for different kinds of distributions, e.g. some kernels are designed to address multi-modal distribution, while some others are particularly efficient in tackling strong

correlations between the dimensions of the state space. The ultimate goal of all kernels is to achieve fast convergence and mixing for the kinds of problems they are designed for. For an extensive history of the evolution of MCMC algorithms see [38], [17] and [9]. Sections 2.3.1-2.3.4 present some popular kernels and give particular focus on the kernels that this thesis targets in later chapters.

2.3.1 The Metropolis and Metropolis-Hastings algorithms

The Metropolis algorithm (shown in Code 1) is very simple. Each iteration of the loop in line 2 comprises the following steps:

1. Propose a sample θ^* using a symmetric proposal distribution, which is denoted $q(\cdot | \cdot)$ (line 3). A symmetric distribution is one whose density satisfies the following condition: $q(a | b) = q(b | a)$. The proposal distribution must be easy to sample from. It is usual practice to use a Normal distribution with mean equal to the previous sample $\theta^{(i-1)}$ and some fixed variance, since many algorithms exist to efficiently generate Normal samples [5].
2. Evaluate the probability of the proposed sample according to the target density and then evaluate the acceptance ratio a (line 4). In Bayesian inference, the target density is the posterior $p(\theta) = p(\theta | \mathbf{D})$. Note that the normalizing constant in the posterior (see equation 2.1) is cancelled out when computing the ratio a . Only the likelihood and prior need to be evaluated. This is why Metropolis and all other MCMC methods need the posterior to be known only up to a normalizing constant, as mentioned earlier, a fact that makes them suitable for Bayesian inference.
3. Generate a uniform random number u in the range $[0, 1]$ and compare it with a in order to decide whether the proposed sample will be accepted or rejected (lines 5-9). If the sample is rejected, copy the previous sample $\theta^{(i-1)}$ to the next iteration.

The algorithm needs an initial sample $\theta^{(1)}$ as input. This is usually chosen at random (e.g. from a uniform distribution) or sampled from the prior distribution or found using some initialization process (typically using a simpler model). The output of the algorithm is the sequence of MCMC samples $\theta^{(1:N)}$.

The requirement for the proposal distribution to be symmetric is crucial for the correctness of the Metropolis algorithm. In 1970, Hastings overcame this limitation by proposing the Metropolis-

Algorithm 1 Metropolis MCMC

```

1: procedure METROPOLIS( $N, \theta^{(1)}$ ) - Inputs:  $N$  (number of MCMC samples),  $\theta^{(1)}$  (initial MCMC sample)
2:   for  $i = 2, \dots, N$  do
3:      $\theta^* \sim q(\theta^* | \theta^{(i-1)})$ 
4:     Evaluate acceptance ratio  $a \leftarrow \frac{p(\theta^*)}{p(\theta^{(i-1)})}$ 
5:     Generate uniform random number  $u \sim U[0, 1)$ 
6:     if  $a \geq u$  then
7:       Accept proposed sample:  $\theta^{(i)} \leftarrow \theta^*$ 
8:     else
9:       Reject proposed sample and replicate previous sample:  $\theta^{(i)} \leftarrow \theta^{(i-1)}$ 
10: return  $\theta^{(1:N)}$  ( $N$  MCMC samples)

```

Algorithm 2 Metropolis-Hastings MCMC

```

1: procedure M-H( $N, \theta^{(1)}$ ) - Inputs:  $N$  (number of MCMC samples),  $\theta^{(1)}$  (initial MCMC sample)
2:   for  $i = 2, \dots, N$  do
3:      $\theta^* \sim q(\theta^* | \theta^{(i-1)})$ 
4:     Evaluate acceptance ratio  $a \leftarrow \frac{p(\theta^*) q(\theta^{(i-1)} | \theta^*)}{p(\theta^{(i-1)}) q(\theta^* | \theta^{(i-1)})}$ 
5:     Generate uniform random number  $u \sim U[0, 1)$ 
6:     if  $a \geq u$  then
7:       Accept proposed sample:  $\theta^{(i)} \leftarrow \theta^*$ 
8:     else
9:       Reject proposed sample and replicate previous sample:  $\theta^{(i)} \leftarrow \theta^{(i-1)}$ 
10: return  $\theta^{(1:N)}$  ( $N$  MCMC samples)

```

Hastings (M-H) method [39] (Algorithm 2). In M-H, the proposal can be non-symmetric and the acceptance ratio is changed to accommodate this (line 4).

Metropolis and M-H are the most fundamental algorithms in MCMC literature (along with Gibbs sampling which will be discussed in Section 2.3.4). Most subsequent MCMC variants are improvements on these core algorithms.

2.3.2 Multi-modal posteriors and Population-based MCMC

As mentioned above, the target distribution of MCMC (here, the posterior) can have any form. One particular form of posterior which is difficult to sample from using basic MCMC methods (like M-H) is a multi-modal posterior distribution. These distributions have multiple modes in different parts of their support. An example is shown in Figure 2.4. When targeting these distributions, elementary kernels like M-H tend to get “stuck” in one of the modes [7, 24], resulting in slow convergence and mixing. This problem is particularly intense when the sampled space has a lot of dimensions and the number of modes is large. Multi-modal distributions appear in many Bayesian inference application, e.g.

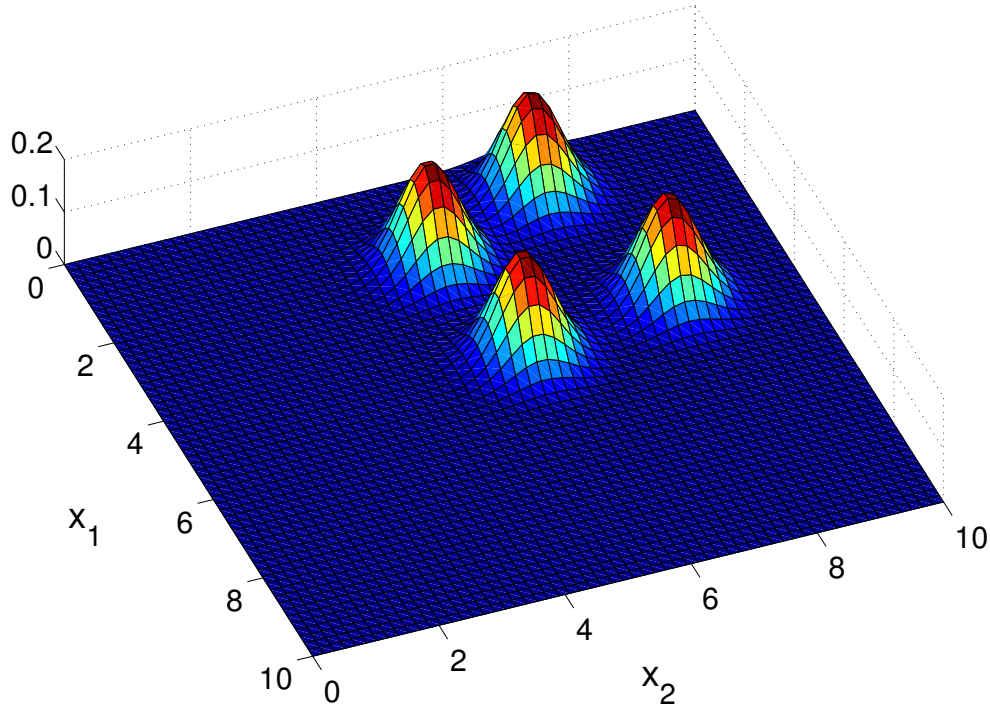


Figure 2.4: Example of a multi-modal target distribution. The distribution is a mixture of four bivariate Gaussian components with means (2,5), (4,8) (4.5, 5.5) and (1,7). Basic MCMC samplers tend to get “stuck” in one mode and need a lot of time to jump between modes (because it is rare to propose and accept a sample that leads to another mode when using simple proposals).

machine learning using Restricted Boltzmann Machines or mixture models [22, 40, 7], computational genetics [13, 25] and biological simulations [27].

Chapter 3 focuses on a family of MCMC methods designed to tackle the issue of multi-modality; population-based MCMC (popMCMC). popMCMC constructs a Markov chain in the joint space of the original sampled variable θ and a series of auxiliary variables, using a joint transition kernel. This can also be expressed as having a population of Markov chains, each one sampling from one variable, instead of the single chain used by M-H. Each chain samples from a slightly different distribution. Certain types of interactions are introduced between the chains, which help to improve the algorithm’s convergence and mixing properties [7]. Here, a description of the most popular Population-based method, called Parallel Tempering (PT) is given. PT was first proposed by Geyer in 1991 [41]. This method will be re-examined in Chapter 3 from a hardware implementation perspective.

Algorithm 3 shows the pseudocode of PT. The chain population consists of M chains and chain j

Algorithm 3 Parallel Tempering

```

1: procedure PT( $N, M, Temp_{1:M}, \theta_{1:M}^{(1)}, \sigma_{1:M}^2$ ) - Inputs:  $N$  (number of MCMC samples),  $M$  (number
   of chains),  $Temp_{1:M}$  (temperatures for all chains),  $\theta_{1:M}^{(1)}$  (initial MCMC samples for all chains),
    $\sigma_{1:M}^2$  (variances of proposal densities for all chains)
2:   for  $i = 2, \dots, N$  do
3:     for  $j = 1, \dots, M$  do // Global update
4:        $\theta^* \sim q(\theta^* | \theta_j^{(i-1)}) = N(\theta^* | \theta_j^{(i-1)}, \sigma_j^2)$ 
5:       Evaluate acceptance ratio  $a \leftarrow \frac{p_j(\theta^*)}{p_j(\theta_j^{(i-1)})}$ 
6:       Generate uniform random number  $u \sim U[0, 1)$ 
7:       if  $a \geq u$  then
8:         Accept proposed sample:  $\theta_j^{(i)} \leftarrow \theta^*$ 
9:       else
10:        Reject proposed sample and replicate previous sample:  $\theta_j^{(i)} \leftarrow \theta_j^{(i-1)}$ 
11:     Choose even chain pairs  $((1, 2), (3, 4), \dots)$  or odd chain pairs  $((2, 3), (4, 5), \dots)$  (in turn)
12:     for all chosen chain pairs  $(q, r)$  do // Global exchange
13:       Evaluate exchange acceptance ratio  $e(\theta_q^{(i)}, \theta_r^{(i)}) \leftarrow \min\left(1, \frac{p_q(\theta_r^{(i)})p_r(\theta_q^{(i)})}{p_q(\theta_q^{(i)})p_r(\theta_r^{(i)})}\right)$ 
14:       Generate uniform random number  $u \sim U[0, 1)$ 
15:       if  $e(\theta_q^{(i)}, \theta_r^{(i)}) \geq u$  then
16:         Exchange the samples  $\theta_q^{(i)}$  and  $\theta_r^{(i)}$ 
17:       else
18:         Do not exchange any samples
19: return  $\theta_1^{(1:N)}$  ( $N$  MCMC samples from the first chain)

```

($j \in \{1, \dots, M\}$) samples from a distribution with density $p_j(\theta)$:

$$p_j(\theta) = p(\theta)^{1/Temp_j}, j \in \{1, \dots, M\} \quad (2.11)$$

where $Temp_j$ (with $1 = Temp_1 < Temp_2 < \dots < Temp_M$) is the temperature of chain j . The density $p_1(\theta)$ is the target density $p(\theta) = p(\theta | \mathbf{D})$ (since $Temp_1 = 1$). The remaining densities are “tempered” versions of $p(\theta)$. Temperatures increase with higher j , which results in gradually smoother densities, i.e. closer to uniform. Practically, this means that “hot” chains move quickly in the state space (they jump more easily between the now smoothed modes), while “cold” chains move slowly but sample from distributions closer to the target $p(\theta)$.

The algorithm performs N iterations (loop in line 2 of Algorithm 3) in order to draw N samples from the target distribution $p_1(\theta) = p(\theta)$. Samples from the tempered (auxiliary) distributions are also generated (N samples for each distribution) but they are discarded in the end. Every iteration comprises a Global update (line 3-10) and a Global exchange (line 11-18).

During the Global update of iteration i , the current samples of all chains ($\theta_1^{(i)}, \dots, \theta_M^{(i)}$) are updated

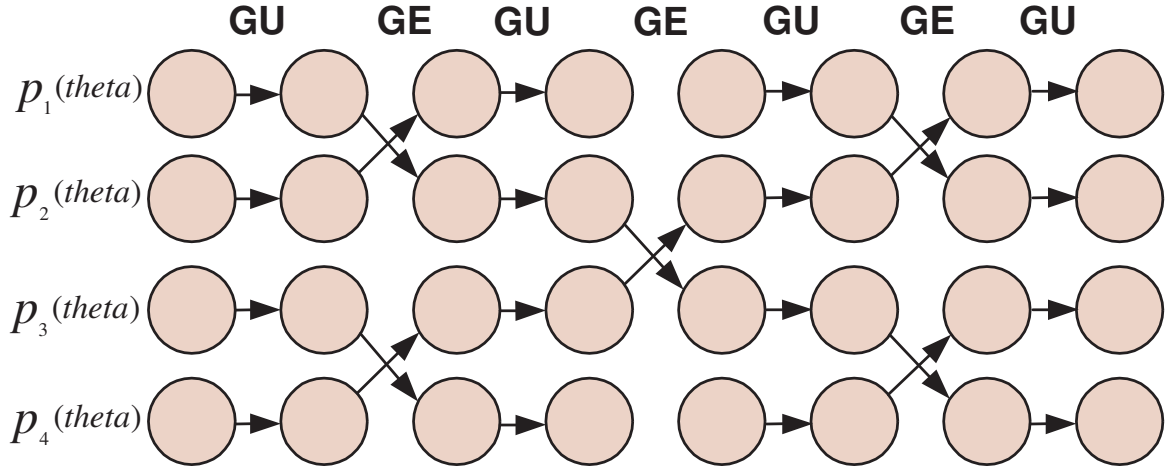


Figure 2.5: Parallel Tempering updates (GU) and exchanges (GE) with four tempered chains. Each chain samples from the density shown on the left side of the figure.

using separate kernels (loop in line 3). Here, and in the remaining of the thesis, it is assumed that Metropolis kernels with Normal proposals are used inside the Global update step of PT, since it is the most common way to implement PT. The variances of the proposals ($\sigma_{1:M}^2$) are input parameters. Each kernel samples from the distribution of the corresponding chain (see (2.11)). The proposal and accept/reject steps of the kernel are shown in lines 4 and 5-10. The latter requires computing the probability of the proposed sample using the target density of the chain ($p_j(\theta^*)$ for chain j).

During Global exchanges, PT attempts interactions between chain pairs; sample exchanges are proposed between all odd pairs of neighboring chains, i.e. $(1,2), \dots, (M-1, M)$ or all even pairs of neighboring chains, i.e. $(2,3), \dots, (M-2, M-1)$ (in a rotating order - line 11). These exchanges push samples from the “hot” chains (large index numbers) to the “colder” ones (small index numbers) and eventually to the first (coldest) chain, helping it escape from isolated modes and thus enhancing mixing.

Figure 2.5 shows a graphical illustration of the Global Updates and Global Exchanges in PT. In real scenarios, tens or hundreds of chains are used. The choice of the number of chains and their temperatures is important for this enhancement to be significant [27, 7] but it is outside the scope of this thesis. Only the samples $\theta_1^{(1:N)}$ from chain 1 are kept and used to get the estimate (2.9) (substituting $\theta^{(i)}$ for $\theta_1^{(i)}$). Some initial samples must be removed as burn-in, as in all MCMC algorithms.

2.3.3 State Space Models and Particle MCMC

Most MCMC methods, including the popMCMC sampler of the previous section, are based on the assumption that the target probability density $p(\theta)$ (i.e. the posterior in Bayesian applications) can be evaluated pointwise up to multiplicative constant. This is necessary to evaluate the acceptance ratio in each MCMC step. Nevertheless, there are several situations where it is not possible to evaluate the probability density. These include inference for State Space Models (SSMs) [33], stochastic kinetic models [42], undirected graphical models [43] and all other Bayesian model classes where the posterior does not admit a closed form expression. These cases are often called analytically intractable in the MCMC literature.

A natural workaround is to devise some estimator of the target probability density and use it inside the MCMC sampler to enable inference. Nevertheless, the effect of this estimator on sampling accuracy is not straightforward to assess. A major breakthrough in the field came from an idea first introduced by Beaumont [44] and later extended and validated by Andrieu and Roberts [45]; it can be shown that if the estimator of the target density is unbiased, the MCMC sampler will converge to the correct target distribution (even though the density cannot be evaluated), i.e. there will be no bias in the estimation of (2.9). Based on this remarkable property, it is possible to construct various unbiased estimators for well-known likelihoods/priors with no closed form expression [46] (like the ones mentioned in the previous paragraph) and use them inside MCMC. MCMC methods based on this technique are collectively called pseudo-marginal MCMC methods.

By far the most important and widely applied pseudo-marginal method is Particle MCMC (pMCMC), proposed Andrieu et al. [33]. It is designed for inference on SSMs with unknown parameters, a task known to lead to analytically intractable likelihoods. pMCMC uses a Particle Filter (PF) to generate an unbiased estimate of the likelihood, which is a natural choice for SSMs. It is thus a combination of the two most popular methods in Bayesian inference (MCMC and PF). pMCMC has created many new application areas for MCMC-based inference on SSMs, including ecology [12], communication networks [47], marine biogeochemistry [48] and economics [23].

This section contains the necessary background information on SSMs, PFs and pMCMC. Chapter 4 will re-examine pMCMC from a hardware implementation point of view.

State-space models

State-space models (also known as Hidden Markov models) are used in situations where there is some unobservable (hidden) sequence of states and this sequence generates observable data (observations) through some function. The hidden state sequence evolves with time or with some other variable (e.g. space) through some other function. The goal is to estimate the state sequence based on the observations and (in some cases) estimate unknown parameters within the functions of the model.

In order to make this more formal, consider $\mathbf{X}_t \in \mathbb{R}^{n_X}$ where $t \in \{1, \dots, T\}$ and $n_X \in \mathbb{N}$ to be the state vector at time t , so that $\mathbf{X}_{1:T}$ is the whole state sequence. Moreover, let $\mathbf{Y}_t \in \mathbb{R}^{n_Y}$ where $t \in \{1, \dots, T\}$ and $n_Y \in \mathbb{N}$ be the observation vector at time t . Notice that the symbol $\mathbf{Y}$ is used to represent the data, instead of the $\mathbf{D}$ that was used in previous sections. This is done to keep notation consistent with most of the literature.

Treating both $\mathbf{X}_{1:T}$ and $\mathbf{Y}_{1:T}$ as random variables, a state-space model is defined by the following three equations:

$$\mathbf{X}_1 \sim p(\mathbf{X}_1) \quad (2.12)$$

$$\mathbf{X}_t \sim p(\mathbf{X}_t | \mathbf{X}_{t-1}, \theta), \quad t > 1 \quad (2.13)$$

$$\mathbf{Y}_t \sim p(\mathbf{Y}_t | \mathbf{X}_t, \theta), \quad t > 0 \quad (2.14)$$

Here, $p(\mathbf{X}_1)$ is the initial probability density of the first state, $p(\mathbf{X}_t | \mathbf{X}_{t-1}, \theta)$ is the probability density of moving to the current state given the previous state (**transition density**) and $p(\mathbf{Y}_t | \mathbf{X}_t, \theta)$ is the probability density of observing the current observation given the current state (**observation density**). Notice that the transition and observation densities can also depend on a set of unknown random parameters, $\theta \in \mathbb{R}^{n_\theta}$ (although this is not necessary). The vector θ can be separated into parameters related to the transition density $\theta_{tr} \in \mathbb{R}^{n_{tr}}$ and parameters related to the observation density $\theta_{obs} \in \mathbb{R}^{n_{obs}}$, so that $\theta = \{\theta_{tr}, \theta_{obs}\}$. Finally, each transition density at time t might depend on a vector of known values $\mathbf{Z}_t \in \mathbb{R}^{n_Z}$. These are ignored here to simplify notation (they are later used inside the proposed FPGA architecture for pMCMC in Chapter 4). Figure 2.6 depicts the structure of the SSM.

The typical assumptions in SSMs (which are easy to see in the above formulation) are that: 1) $\mathbf{X}_t$ is Markovian, i.e. its probability density depends only on $\mathbf{X}_{t-1}$ and not on other previous states. 2) The probability density of $\mathbf{Y}_t$ depends only on $\mathbf{X}_t$ and not on previous states or observations. The SSM framework is widely applicable in many areas of science and engineering such as finance, communi-

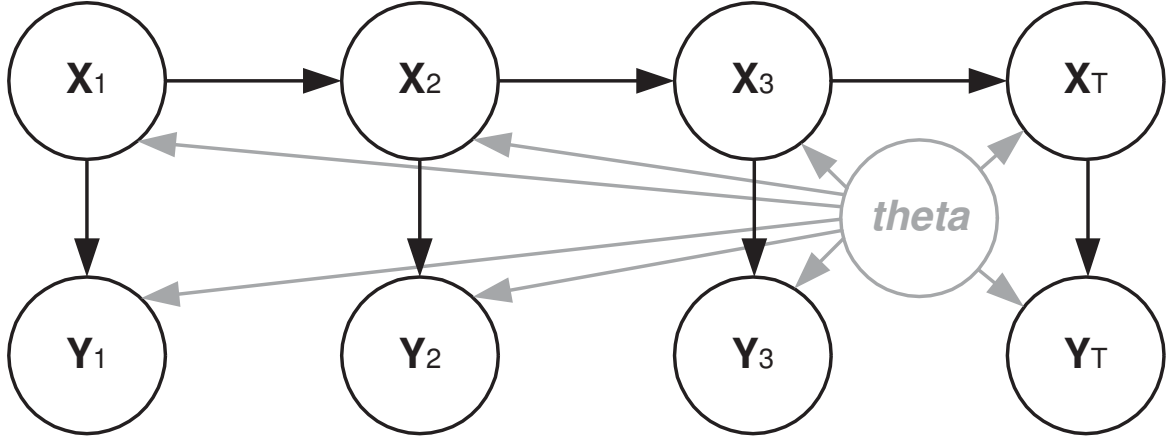


Figure 2.6: Hidden states, observations and latent parameters of state-space model.

cations, genetics and machine learning.

Example of an SSM: A concrete example, which can help understand the framework, is target tracking. e.g. tracking of a robot that moves in a 2-dimensional space using a set of sensors located in known positions. In such a typical tracking scenario, the state $\mathbf{X}_t = \begin{bmatrix} x_t^1 \\ x_t^2 \end{bmatrix}$ represents the horizontal (x_t^1) and vertical (x_t^2) position of the robot at time t . The position changes according to the random walk equation $\mathbf{X}_t = \mathbf{X}_{t-1} + \mathbf{u}_t$, where $\mathbf{u}_t$ is a Normal random vector $\mathbf{u}_t \sim N(\mathbf{u}_t | \mathbf{0}, \theta_{tr})$ and $N(\mathbf{u}_t | \mathbf{0}, \theta_{tr})$ is the bivariate normal density with mean $\mathbf{0}$ and covariance matrix θ_{tr} at point $\mathbf{u}_t$. The density transition density then takes the following form:

$$p(\mathbf{X}_t | \mathbf{X}_{t-1}, \theta_{tr}) = N(\mathbf{X}_t | \mathbf{X}_{t-1}, \theta_{tr}) \quad (2.15)$$

Equation (2.15) means that the robot is assumed to do a random walk. At time $t = 1$ the initial density of the state is a standard normal:

$$p(\mathbf{X}_1) = N(\mathbf{X}_1 | \mathbf{0}, \mathbf{I}) \quad (2.16)$$

In order to track the robot, assume there are measurements available from three sensors. The position of the i -th sensor is $\mathbf{S}_i = \begin{bmatrix} s_i^1 \\ s_i^2 \end{bmatrix}$, where s_i^1 and s_i^2 are the horizontal and vertical positions respectively. The measurements (observations) are the Euclidean distances between the robot and the sensors at time t , plus some measurement noise. For sensor i , the observations are: $\mathbf{Y}_{i,t} = \|\mathbf{X}_t - \mathbf{S}_i\| + \mathbf{u}_t = (x_t^1 - s_i^1)^2 + (x_t^2 - s_i^2)^2 + N(\mathbf{0}, \theta_{obs})$, where θ_{obs} is the covariance matrix of the observation noise. The relationship between the observations captured by sensor i and the position of the robot at time t is

summarized by the observation density:

$$p(\mathbf{Y}_{i,t} | \mathbf{X}_t, \theta_{obs}) = N(\mathbf{Y}_{i,t} | \|\mathbf{X}_t - \mathbf{S}_i\|, \theta_{obs}) \quad (2.17)$$

The full observation density (which includes all sensors) at time t is $p(\mathbf{Y}_t | \mathbf{X}_t, \theta_{obs}) = \prod_{i=1}^3 p(\mathbf{Y}_{i,t} | \mathbf{X}_t, \theta_{obs})$. Notice that in equations (2.15) and (2.17), the unknown parameter θ has been separated into its two components θ_{tr} and θ_{obs} for clarity.

State estimation in SSMs - The Particle Filter

When using an SSM like the one above, the goal is either to estimate the hidden state sequence $\mathbf{X}_{1:T}$ given the observations $\mathbf{Y}_{1:T}$ or to estimate the unknown parameter vector θ given the observations $\mathbf{Y}_{1:T}$ or both. For example, in the robot tracking problem, the main aim is to estimate the sequence of positions of the robot (state $\mathbf{X}_{1:T}$) using the sensor measurements but one might also want to estimate the covariance of the measurement noise and the covariance of the robot's random walk (which together form the unknown parameter θ). The pMCMC algorithm (on which this section focuses) aims at estimating both the hidden states and the unknown parameters jointly. Nevertheless, for reasons of clarity, this part of the section will examine the simpler problem of estimating only the state sequence when the parameter set θ is known. In the next part of the section, this will be used to tackle the joint estimation problem.

The problem will be formulated as a Bayesian inference task. The goal is to sample from the Bayesian posterior distribution of the unknown states, given the data and the fixed parameter θ . The posterior is:

$$p_{\theta}(\mathbf{X}_{1:T} | \mathbf{Y}_{1:T}) \propto p_{\theta}(\mathbf{X}_{1:T}) p_{\theta}(\mathbf{Y}_{1:T} | \mathbf{X}_{1:T}) = p(\mathbf{X}_1) \left(\prod_{t=2}^T p_{\theta}(\mathbf{X}_t | \mathbf{X}_{t-1}) \right) \left(\prod_{t=1}^T p_{\theta}(\mathbf{Y}_t | \mathbf{X}_t) \right) \quad (2.18)$$

where θ has been moved to the subscript of the densities which are affected by its value. The posterior consists of the prior $p_{\theta}(\mathbf{X}_{1:T})$ and the likelihood $p_{\theta}(\mathbf{Y}_{1:T} | \mathbf{X}_{1:T})$. The right-most part of equation (2.18) is easily derived from the densities (2.12)-(2.14).

In order to perform Bayesian inference in this context (using MCMC, IS or some other sampling technique), it is necessary to evaluate the posterior multiple times (e.g. in every MCMC step). Nevertheless, this posterior does not admit a closed form. Fortunately, several alternative methods have been

developed in the Bayesian filtering literature to get the exact posterior or sample from it [29]: 1) When the state $\mathbf{X}_t$ can take only a finite set of values (finite state-space SSMs), the posterior can be evaluated exactly in each point because the integrals involved are converted to finite sums. 2) When the transition and observation densities are linear and Gaussian (linear Gaussian SSMs), the posterior is a Gaussian distribution and its mean and variance can be computed using Kalman filtering techniques [49]. 3) In all the remaining non-linear, non-Gaussian SSM cases, the posterior cannot be evaluated exactly but samples can be drawn from it using Particle Filtering (also known as Sequential Monte Carlo) techniques, instead of MCMC or IS. This section focuses on cases where the posterior cannot be evaluated exactly, since these represent the large majority of SSM applications, including the ones that motivate the work in Chapter 4.

PFs are a group of algorithms which are used for state estimation in SSMs. PFs use a set of samples (particles) to estimate either the latest state at time t ($\mathbf{X}_t$) or the whole state sequence up to time t ($\mathbf{X}_{1:t}$). PF has many variants, which are suitable for different situations. Doucet and Johansen [29], Kantas et al. [50] and Cappe et al. [51] contain good overviews of the various methods and applications. Here, the bootstrap PF [52] will be used to estimate the whole state sequence (since the posterior over the whole sequence is required). The bootstrap PF was chosen because it is the most popular PF variant, it can be applied to most problems and it is easy to implement.

The bootstrap PF algorithm is the following:

In Algorithm 4, the input arguments are the number of particles of the PF (P), the number of states (T), the fixed parameter vector (θ) and the observations ($\mathbf{Y}_{1:T}$). The output is T sets of P particles each ($\mathbf{X}_{1:T}^{1:P} = \{\mathbf{X}_1^{1:P}, \dots, \mathbf{X}_T^{1:P}\}$) and an unbiased estimate (L) of the likelihood of θ given the observations $\mathbf{Y}_{1:T}$, marginalised over $\mathbf{X}_{1:T}$, i.e. $L \simeq p_\theta(\mathbf{Y}_{1:T})$.

The main idea behind Algorithm 4 is that it uses a set of P particles to estimate each state. In every iteration it propagates the particles to the next state, etc. The particle set $\mathbf{X}_t^{1:P}$ is an estimate to the posterior density of state t , given the observations up to time t , i.e. $p(\mathbf{X}_t | \mathbf{Y}_{1:t})$. In other words, the particles are samples distributed according to this posterior. The ensemble of all particle sets $\mathbf{X}_{1:T}^{1:P}$ is an estimate to the posterior density of the whole state sequence, given all observations, i.e. $p(\mathbf{X}_{1:T} | \mathbf{Y}_{1:T})$.

Algorithm 4 first samples P times from the initial density $p(\mathbf{X}_1)$ in order to create an initial set of particles at time step $t = 1$ (lines 3-4). The k -th particle is denoted $\tilde{\mathbf{X}}_1^k$. Then it calculates the weight W_1^k of each particle $\tilde{\mathbf{X}}_1^k$ using the observation function (which works here as a likelihood) at lines 5-

Algorithm 4 Bootstrap Particle Filter

```

1: procedure BOOTSTRAPPF( $P, T, \theta, \mathbf{Y}_{1:T}$ ) - Inputs:  $P$  (number of particles),  $T$  (number of SSM
   states),  $\theta$  (parameter values for transition and observation densities),  $\mathbf{Y}_{1:T}$  (observations)
2: Initial state ( $t = 1$ ):
3:   for  $k = 1, \dots, P$  do
4:     Sample particle from initial density  $\tilde{\mathbf{X}}_1^k \sim p(\mathbf{X}_1)$ 
5:   for  $k = 1, \dots, P$  do
6:     Calculate initial weight  $W_1^k \leftarrow p_\theta(\mathbf{Y}_1 | \tilde{\mathbf{X}}_1^k)$ 
7: Remaining states:
8:   for  $t = 2, \dots, T$  do
9:     for  $k = 1, \dots, P$  do
10:      Sample ancestor index  $a^k$  from  $\{1, \dots, P\}$  with probabilities proportional to
11:       $\{W_{t-1}^1, \dots, W_{t-1}^P\}$  and set resampled particle  $\mathbf{X}_{t-1}^k \leftarrow \tilde{\mathbf{X}}_{t-1}^{a^k}$ 
12:     for  $k = 1, \dots, P$  do
13:       Sample particle from transition density  $\tilde{\mathbf{X}}_t^k \sim p_\theta(\mathbf{X}_t | \mathbf{X}_{t-1}^k)$ 
14:     for  $k = 1, \dots, P$  do
15:       Calculate weight  $W_t^k \leftarrow p_\theta(\mathbf{Y}_t | \tilde{\mathbf{X}}_t^k)$ 
16: Likelihood estimate:
17:    $L \leftarrow \prod_{t=1}^T \left( \frac{1}{P} \sum_{k=1}^P W_t^k \right)$ 
18: return ( $L, \mathbf{X}_{1:T}^{1:P} = \{\mathbf{X}_1^{1:P}, \dots, \mathbf{X}_T^{1:P}\}$ ) (likelihood estimate and  $P$  particles for every SSM state)

```

6. The weight quantifies the quality of the particle as an estimate of $\mathbf{X}_1$ based on information from observation $\mathbf{Y}_t$. After the initial P particles and their weights are known, the algorithm moves to the loop in lines 7-15. For every time step t , three steps are performed: 1) Resampling, 2) Sampling, 3) Weight.

In the resampling step (lines 9-11), for each particle k , an index a^k from 1 to P is sampled with probabilities proportional to $\{W_{t-1}^1, \dots, W_{t-1}^P\}$ (this is equivalent to sampling from a multinomial distribution). The particle with index a^k from the previous time step ($\tilde{\mathbf{X}}_{t-1}^{a^k}$) is then assigned to the variable $\mathbf{X}_{t-1}^k$. When all P iterations finish, the set $\mathbf{X}_{t-1}^{1:P}$ is the set of resampled particles. The resampled particles are the ones which are propagated to the next time step. Resampling practically means that particles with large weights tend to be copied more often and therefore survive into the next generation, while particles with small weights tend to disappear. This process is critical for the stability of the method. Without resampling, the variance of the t -th state estimate (given by the particles at time t) increases exponentially with t ; with resampling, the method can reach much bigger t without introducing large variances (see Doucet and Johansen [29] for more comments on why resampling is important). Notice that the set of particles is distinct from the set of resampled particles, thus the difference in notation ($\tilde{\mathbf{X}}_t^{1:P}$ for the former, $\mathbf{X}_t^{1:P}$ for the latter). It is worth noting that the resampling approach shown here (which is called Multinomial resampling) is only one of the existing resampling

algorithms. Many other algorithms have been proposed, which are more efficient or more suitable for parallel implementation [53, 54].

After the resampling step has finished the algorithm performs the sampling and weight steps in lines 12-13 and 14-15 respectively. The sampling step is similar to the sampling during particles initialisation, only now the transition density is used to propagate the resampled particles of time step $t - 1$ and form a new set of particles at time step t . The weight step is the same as during initialisation.

After all T time steps have finished, the algorithm computes an estimate of the likelihood of the fixed parameters θ , where the states have been marginalised out, i.e. an estimate of $p_\theta(\mathbf{Y}_{1:T})$. This is done by finding the mean of the weights of each step and multiplying the means from all steps (line 17). This estimate is unbiased [29]. The likelihood is a by-product of the whole process, which is not necessary for state estimation but, in combination with the state estimate $\mathbf{X}_{1:T}^{1:P}$, will prove crucial when the PF will be used within an MCMC sampler (see next section).

Finally, the algorithm returns the likelihood estimate and the full history of (resampled) particles; T sets of P particles each. In fact, the full history of non-resampled particles $\tilde{\mathbf{X}}_{1:T}^{1:P}$ coupled with their respective weights $W_{1:T}^{1:P}$ can also be used to estimate the state sequence in exactly the same way as Importance Sampling estimates densities [29]. Nevertheless, the resampled particles do not have to be accompanied by weights (their weights are all equal), therefore they are easier to store and transfer.

Joint estimation of states and parameters - The Particle MCMC sampler

Although the basic bootstrap PF can handle state estimation, it is often the case that the parameter set θ is unknown. For example, the measurement noise in the robot tracking problem is typically unknown, since it depends on various factors, e.g. sensor technology. In this case, the PF presented above cannot be used.

To define the problem in Bayesian terms, the posterior of equation (2.18) has to be supplemented with an extra parameter (θ):

$$\begin{aligned} p(\mathbf{X}_{1:T}, \theta \mid \mathbf{Y}_{1:T}) &\propto p(\theta) p(\mathbf{X}_{1:T} \mid \theta) p(\mathbf{Y}_{1:T} \mid \mathbf{X}_{1:T}, \theta) \\ &= p(\theta) p(\mathbf{X}_1) \left(\prod_{t=2}^T p(\mathbf{X}_t \mid \mathbf{X}_{t-1}, \theta) \right) \left(\prod_{t=1}^T p(\mathbf{Y}_t \mid \mathbf{X}_t, \theta) \right) \end{aligned} \quad (2.19)$$

Here, $p(\theta)$ is the prior density of the unknown parameter set θ . Notice that θ is no longer shown as a

subscript in the related densities (like in equation (2.18)), since it is not fixed. Also, note that in this section $p(\theta)$ does not represent the posterior target distribution, as in previous sections. The posterior of pMCMC is the joint posterior of states and parameters ($p(\mathbf{X}_{1:T}, \theta \mid \mathbf{Y}_{1:T})$). Also, the conditioning on the data ($\mathbf{Y}_{1:T}$) is not omitted.

pMCMC [33] (a combination of MCMC and the PF presented above) is the most common approach to sample from this joint posterior (for other approaches see Murray [2]). As already described earlier, all MCMC algorithms need to evaluate the acceptance ratio of a proposed sample in every iteration, which requires the evaluation of a ratio of posteriors. This evaluation is impossible in most SSM situations (as emphasized previously).

Nevertheless, using a PF as described above, it is possible to attain unbiased samples from $p(\mathbf{X}_{1:T} \mid \mathbf{Y}_{1:T}, \theta)$ and an unbiased estimate of $p(\mathbf{Y}_{1:T} \mid \theta)$ (equivalent to $L \simeq p_\theta(\mathbf{Y}_{1:T})$, a by-product of Algorithm 4). Andrieu and Roberts [45] and Andrieu et al. [33] showed how to use these PF outputs to construct an MCMC algorithm which samples correctly from the posterior (2.19). The two core ideas of this algorithm are the following:

1) Proposal density: It is possible to cancel out the state variable $\mathbf{X}_{1:T}$ in (2.19) from the MCMC acceptance ratio by using a suitable MCMC proposal density. In every MCMC step, this proposal uses a simple Metropolis-Hastings (M-H) kernel to propose a new θ sample, called θ^* . Given θ^* , it then proposes a new $\mathbf{X}_{1:T}$ sample, called $\mathbf{X}_{1:T}^*$ from the distribution $p(\mathbf{X}_{1:T}^* \mid \mathbf{Y}_{1:T}, \theta^*)$ (θ^* is fixed). In other words, the proposal density is:

$$q((\mathbf{X}_{1:T}^*, \theta^*) \mid (\mathbf{X}_{1:T}, \theta)) = q(\theta^* \mid \theta) p(\mathbf{X}_{1:T}^* \mid \mathbf{Y}_{1:T}, \theta^*) \quad (2.20)$$

The purpose of this approach is that the particular proposal leads to the following MCMC acceptance ratio, where the variable $\mathbf{X}_{1:T}^*$ is integrated out:

$$\begin{aligned} a &= \frac{p(\mathbf{X}_{1:T}^*, \theta^* \mid \mathbf{Y}_{1:T}) q((\mathbf{X}_{1:T}, \theta) \mid (\mathbf{X}_{1:T}^*, \theta^*))}{p(\mathbf{X}_{1:T}, \theta \mid \mathbf{Y}_{1:T}) q((\mathbf{X}_{1:T}^*, \theta^*) \mid (\mathbf{X}_{1:T}, \theta))} \propto \frac{p(\theta^* \mid \mathbf{Y}_{1:T}) p(\mathbf{X}_{1:T}^* \mid \mathbf{Y}_{1:T}, \theta^*) q(\theta \mid \theta^*) p(\mathbf{X}_{1:T} \mid \mathbf{Y}_{1:T}, \theta)}{p(\theta \mid \mathbf{Y}_{1:T}) p(\mathbf{X}_{1:T} \mid \mathbf{Y}_{1:T}, \theta) q(\theta^* \mid \theta) p(\mathbf{X}_{1:T}^* \mid \mathbf{Y}_{1:T}, \theta^*)} \\ &= \frac{p(\theta^* \mid \mathbf{Y}_{1:T}) q(\theta \mid \theta^*)}{p(\theta \mid \mathbf{Y}_{1:T}) q(\theta^* \mid \theta)} \propto \frac{p(\theta^*) p(\mathbf{Y}_{1:T} \mid \theta^*) q(\theta \mid \theta^*)}{p(\theta) p(\mathbf{Y}_{1:T} \mid \theta) q(\theta^* \mid \theta)} \end{aligned} \quad (2.21)$$

By integrating out the variable $\mathbf{X}_{1:T}$ in a , there is no longer need to evaluate the term $p(\mathbf{X}_{1:T} \mid \mathbf{Y}_{1:T}, \theta)$. Note that the decomposition of the posterior in the first line of the above equation is different from the decomposition shown in equation (2.19). In order to implement the proposed MCMC proposal, Andrieu and Roberts [45] recommended using a PF. As shown in the previous section, a PF can indeed

generate P samples from $p(\mathbf{X}_{1:T} \mid \mathbf{Y}_{1:T}, \theta)$ (or $p(\mathbf{X}_{1:T}^* \mid \mathbf{Y}_{1:T}, \theta^*)$ in this case), so it is enough to keep one of them as a proposed sample.

2) Use of unbiased likelihood estimate: Although the term $p(\mathbf{X}_{1:T} \mid \mathbf{Y}_{1:T}, \theta)$ is no longer needed to find the acceptance ratio, the exact evaluation of the ratio is still impossible, since $p(\mathbf{Y}_{1:T} \mid \theta)$ is not known in closed form. The critical contribution in [45] (based on prior work done in [44]) was a proof that it is not necessary to evaluate the likelihood $p(\mathbf{Y}_{1:T} \mid \theta)$ exactly to sample from the posterior (2.19). It is, rather, enough to get an unbiased estimate of $p(\mathbf{Y}_{1:T} \mid \theta)$. Regardless of the variance of the estimate, the sampler will converge to the correct posterior (though smaller variances accelerate MCMC mixing). Therefore, the following acceptance ratio can be used:

$$\tilde{a} = \frac{p(\theta^*) \tilde{p}(\mathbf{Y}_{1:T} \mid \theta^*) q(\theta \mid \theta^*)}{p(\theta) \tilde{p}(\mathbf{Y}_{1:T} \mid \theta) q(\theta^* \mid \theta)} \quad (2.22)$$

where $\tilde{p}(\mathbf{Y}_{1:T} \mid \theta)$ is the unbiased estimate of the likelihood produced by the PF (L in Algorithm 4).

The algorithm that uses the above acceptance ratio to sample from SSM posteriors is called pMCMC and was described in [33] (using the term Particle Marginal Metropolis-Hastings sampler). It is shown below:

Algorithm 5 takes the same inputs as Algorithm 4, plus the number of MCMC iterations (N) and an initial θ sample (θ^{init}). The main loop of the algorithm (lines 9-20) is very similar to a simple M-H sampler. For each iteration, it uses the proposal distribution $q(\theta^* \mid \theta)$ to propose a new θ^* based on the previous one (line 10). It then calls Algorithm 4 (line 11) to achieve two things:

1. Get a set of P samples from $p(\mathbf{X}_{1:T}^* \mid \mathbf{Y}_{1:T}, \theta^*)$. One of these samples is selected randomly in line 12 and functions as the proposed state sequence sample $\mathbf{X}_{1:T}^*$.
2. Get an unbiased estimate of the likelihood of θ^* ($\tilde{p}(\mathbf{Y}_{1:T} \mid \theta^*)$).

In lines 13-20, the algorithm computes the acceptance ratio $\tilde{a}$ (equation (2.22)) and uses it to accept or reject the proposed sample ($\theta^*, \mathbf{X}_{1:T}^*$) as in a typical MCMC algorithm. Lines 3-7 serve to initialise the algorithm using θ^{init} and they contain a call to Algorithm 4, exactly like in the main loop. pMCMC returns the full history of samples $Sample[1 : N]$ and the full history of posteriors $Posterior[1 : N]$ (the latter is not necessary). It is usual to discard an initial chunk of the N samples as burn-in, in order to make sure the sampler has converged to the target distribution.

Algorithm 5 Particle MCMC

```

1: procedure pMCMC( $P, T, \mathbf{Y}_{1:T}, N, \theta^{init}$ ) - Inputs:  $P$  (number of particles),  $T$  (number of SSM
   states),  $\mathbf{Y}_{1:T}$  (observations),  $N$  (number of MCMC samples),  $\theta^{init}$  (initial MCMC sample)
2: First iteration ( $i = 1$ ):
3:    $(\tilde{p}(\mathbf{Y}_{1:T} | \theta^{init}), \mathbf{X}_{1:T}^{1:P}) \leftarrow \text{BootstrapPF}(P, T, \theta^{init}, \mathbf{Y}_{1:T})$  // get likelihood and state samples
4:   Randomly select an index  $p$  from  $\{1, \dots, P\}$  and set  $\mathbf{X}_{1:T}^{init} = \mathbf{X}_{1:T}^p$ 
5:    $Sample[1] = (\theta^{init}, \mathbf{X}_{1:T}^{init})$  // save initial sample
6:    $Posterior[1] = p(\theta^{init}) \tilde{p}(\mathbf{Y}_{1:T} | \theta^{init})$  // compute and save posterior
7:    $\theta = \theta^{init}$  // temporary variable
8: Remaining iterations:
9:   for  $i = 2, \dots, N$  do
10:     $\theta^* \sim q(\theta^* | \theta)$  // propose new  $\theta$ 
11:     $(\tilde{p}(\mathbf{Y}_{1:T} | \theta^*), \mathbf{X}_{1:T}^{1:P}) \leftarrow \text{BootstrapPF}(P, T, \theta^*, \mathbf{Y}_{1:T})$  // get likelihood and state samples
12:    Randomly select an index  $p$  from  $\{1, \dots, P\}$  and set  $\mathbf{X}_{1:T}^* = \mathbf{X}_{1:T}^p$ 
13:    Accept proposed sample  $(\theta^*, \mathbf{X}_{1:T}^*)$  with probability  $\min(1, \tilde{a})$ 
14:    if accepted then
15:       $Sample[i] = (\theta^*, \mathbf{X}_{1:T}^*)$  // save proposed sample
16:       $Posterior[i] = p(\theta^*) \tilde{p}(\mathbf{Y}_{1:T} | \theta^*)$  // compute and save posterior
17:       $\theta = \theta^*$  // temporary variable
18:    else
19:       $Sample[i] = Sample[i-1]$  // replicate previous sample
20:       $Posterior[i] = Posterior[i-1]$  // replicate previous posterior
21: return  $(Sample[1:N], Posterior[1:N])$  ( $N$  sets of MCMC samples and posterior values)

```

The main tuning parameter of the pMCMC algorithm is the number of particles of the PF (P). The larger the number of particles, the more time-consuming each pMCMC iteration becomes. At the same time, more particles mean that the likelihood estimate $\tilde{p}(\theta | \mathbf{Y}_{1:T})$ becomes more accurate, i.e. the variance of the estimate is smaller. This, in turn, leads to faster pMCMC mixing, since pMCMC moves closer to an exact MCMC algorithm. Therefore, there is a tradeoff between the runtime of the PF and the mixing of pMCMC when changing the number of particles.

2.3.4 Other methods

A large number of other MCMC algorithms are currently being used by practitioners, depending on the characteristics of the target distributions from which they need to sample. Describing these algorithms in detail is outside the scope of this chapter. The reader is referred to [9, 4, 17, 6] for detailed reviews of older and more recent research on MCMC methodology.

Here, a list of the most influential MCMC algorithms is given, along with brief descriptions:

1. *Gibbs sampling*: The Gibbs sampling algorithm [31] decomposes the n_θ -multidimensional vari-

able θ into its n_θ components and updates them one by one. The proposal distribution of the i -th component $q_i(\cdot | \cdot)$ is the full conditional distribution of the i -th component conditional on all the remaining components [34]. Because this conditional distribution is the “ideal” choice of proposal distribution (i.e. it is the actual target distribution for the respective component), all proposed samples are accepted. Gibbs sampling has enjoyed massive popularity for Bayesian hierarchical models (e.g. Bayesian networks) [21, 34]. The reason is that in these models the posterior is naturally expressed in terms of the full conditional distributions of each component. Also, sampling from the full conditional distributions is easy since they are typically standard distributions like Normal or Gamma.

2. *Slice Sampling*: Slice sampling [55] is based on the idea that sampling from a target $p(\theta)$ is equivalent to sampling uniformly from the area under the graph of $p(\theta)$. To achieve this uniform sampling, slice sampling augments the target distribution by an auxiliary variable u , which, conditional on θ , is uniformly distributed on the interval $[0, p(\theta)]$. Sampling is performed from the joint distribution of θ and u in a Gibbs sampling fashion. Several techniques have been proposed to optimize the efficiency of this procedure [55]. Unfortunately, when $p(\theta)$ is multivariate, sampling uniformly from the region under its graph requires the construction of a n_θ -dimensional hypercube. This is a task that scales exponentially with the dimension of the problem [8], making the use of the method expensive for high dimensions. However, slice sampling is popular because it can improve mixing when there are strong correlations between components of the sampling variable [55, 8].
3. *Multiple proposals*: These methods propose multiple candidate samples in each iteration of MCMC in order to increase the possibility that a “good” proposed sample will be found. They achieve better mixing and increase acceptance rates. Multiple-Try Metropolis [56] is an early example of this family of algorithms. Other ideas based on the same principles have been introduced since its publication [17].
4. *Hamiltonian Monte Carlo*: These algorithms use the gradient of the target density to propose better moves. They were first introduced by Duane et al. [57] and later improved by Girolami and Calderhead [58]. Today they are extremely popular because they can handle strong correlations, as well as high dimensionality in the target distribution.
5. *Reversible-Jump MCMC*: This algorithm [59] is applicable when the dimensionality of the sampling variable is unknown - for example, in model determination [59]. It has enjoyed some

success for inference in mixture models [60] as well as in other domains [9] but it is hard to implement and debug.

6. *Adaptive MCMC*: These methods adapt their proposal distribution during runtime according to the form of the target distribution in order to optimize sampling efficiency (i.e. mixing) [61]. Care needs to be taken when designing an adaptive kernel in order to guarantee that the sampler will converge to the stationary distribution.

2.4 The need for faster inference

The large volume of research on MCMC during the last two decades has produced a variety of methods which can significantly improve convergence and mixing in many real scenarios. popMCMC and pMCMC are good examples of advanced MCMC algorithms which have expanded MCMC’s applicability significantly. Despite the successes of these methods, MCMC sampling is still not as fast as modern large-scale applications require. Numerous examples exist for which running MCMC is impractical or impossible on desktop CPUs [10, 13, 26, 11, 19, 18, 16, 2, 62, 48, 25, 63, 64, 65, 66, 8, 67, 9]. For many of these problems runtimes can reach months, or years even when using advanced MCMC methods like the ones described above. This limits the applicability of MCMC, since a practitioner typically wants results in hours or days (although this depends on the application).

The reasons for this high computational burden can be classified into generic (applying to any MCMC method) and method-specific (related to a particular MCMC method). The generic reasons are the following:

1. The massive size and dimensionality of data and the continuously increasing complexity of Bayesian models in many applications. Both of these factors make the evaluation of the posterior density $p(\theta)$ (necessary in each MCMC step) computationally expensive. Moreover, both factors are related to the increasing need to analyse and draw conclusions from “big data”, a trend which has dominated many fields of Bayesian statistics during the last few years. For example, topic models commonly use large text databases for inference [11], genomic data have dimensions in the order 10^7 [13, 10, 14] and climate models process such large volumes of data that even model emulation (using Gaussian processes) is challenging in many scenarios [68, 69, 70]. Other motivation examples can be found in statistical physics [19] and phylo-

genetics [25]. Since the “big data” trend is bound to continue and spread in more fields in both industry and academia, the handling of the computational burden of evaluating $p(\theta)$ is the primary concern for MCMC acceleration.

2. The problems of slow convergence and slow mixing. Despite the many advances in MCMC methodology, these are still major issues for all MCMC samplers, caused by multi-modality, correlations and multi-dimensionality in the posterior. The main caveat of MCMC is that there is never a guarantee that the sampler has converged (even using convergence metrics), so practitioners tend to perform long MCMC runs (large N) to increase their certainty that no area has remained unexplored. This problem is especially acute in large dimensions of the state space. It also results in large runtimes to get a satisfactory variance in (2.9). The better the MCMC algorithm mixes, the more likely it is that the state space will be fully explored and that the variance will be satisfactory within a practical time frame (e.g. days or weeks).
3. The fact that multiple independent MCMC runs are performed in most real settings to increase the certainty about convergence (e.g. by employing the Gelman-Rubin criterion [35]) and approximate the variance of (2.9). This can increase the total runtime by an order of magnitude. Moreover, MCMC practitioners often want to test many candidate models and also “tune” their MCMC algorithms in order to maximize efficiency (i.e. mixing speed). Even more runs are necessary for these purposes.
4. The real-time constraints in some applications of MCMC. Although most research on MCMC has focused on its use for static inference, real-time inference (where data are received constantly and MCMC needs to re-run every time) has also been examined for some applications, e.g. Simultaneous Localization and Mapping [71]. In these scenarios, real-time constraints in MCMC execution time have to be met, which is a strong motivation to accelerate the algorithm. Moreover, power consumption is a major concern in these applications, since the algorithm typically needs to run on mobile platforms (e.g. mobile robots). In these cases, apart from raw acceleration, the performance/Watt ratio that can be achieved is crucial.

Apart from these generic factors, there are computational overheads related to specific MCMC methods that lead to large runtimes:

1. In the case of popMCMC (and PT), the main overhead is related to the use a population of Markov chains. Typically, dozens or hundreds of chains are run [27, 7, 13], instead of the

single chain used by basic MCMC and only the samples from the first chain are kept. This means that the number of necessary posterior density evaluations per generated MCMC sample is multiplied by a factor equal to the number of chains. Chapter 3 proposes ways to treat this problem using parallel MCMC implementations.

2. In the case of pMCMC, there are two particularities that lead to forbiddingly large runtimes and hinder its adoption for SSM inference:

- The need to run a PF to estimate the likelihood in every pMCMC step. This has complexity $O(T \cdot P)$ (where T is the number of hidden SSM states and P is the number of particles of the PF). When the number of states and the number of observations per state in the SSM become large (e.g. millions, which is the case in many applications), running the PF can become expensive computationally [72, 73, 74]. Moreover, many thousands of MCMC iterations are usually needed, as mentioned above. Therefore, for complex SSMs [48, 2], performing inference with pMCMC can become impractical. The work in Chapter 4 was initially motivated by such a complex problem; SSMs in genetics, where the number of SSM states, which correspond to DNA bases, can reach millions.
- The issue of multi-modality in the target distribution. This has been largely unaddressed in the pMCMC literature. Nevertheless, there are modelling scenarios where multi-modality can appear, e.g. when the transition density of the SSM is a mixture of different densities (see Section 4.5 and [14]). In these cases, pMCMC mixes slowly due to the same reasons described earlier for basic MCMC samplers which are applied to multi-modal distribution. Slow mixing means that a lot of samples need to be generated to achieve a satisfactory variance in (2.9).

Other MCMC variants also have features which increase computational intensity but these are omitted here. Some information about the overheads of other methods and how they have been tackled in previous literature can be found in Section 2.7.

As a result of the above factors which negatively affect runtime, practitioners are forced to: 1) Generate only few MCMC samples (which increases the variance of the output estimate), 2) Use simpler models, 3) Exclude some data from the analysis, 4) Use approximate inference methods instead of MCMC. All the above solutions compromise the accuracy of the analysis.

2.4.1 Existing approaches and trends in MCMC methodology

During the last five to ten years, the issue of making MCMC more efficient by tackling the above limitations has attracted increased attention. Most of the work in MCMC literature has focused on improving methodology (by proposing or modifying algorithms) or on using alternative methods for inference. Recent approaches can be categorized as follows:

1. *Algorithms based on data partitioning:* A number of techniques have been proposed to split large data sets into sub-groups, run a separate MCMC for each sub-group and then combine the resulting estimates in some way. Early methods in this category [64, 75] introduced bias in the combined output estimate. More recent methods [76, 77, 78] have overcome this problem by using kernel density estimators or by evaluating the geometric median of sub-posteriors. Nevertheless, all of these methods are based on the assumption that data are independent and identically distributed (i.i.d.) which is not true in all modelling scenarios.
2. *Algorithms based on data sub-sampling:* This category is also oriented towards big data inference and assumes that the data are i.i.d. The idea behind these methods is to select a random sub-set of the data in every MCMC iterations and compute the posterior density conditioned on the sub-set instead of the full data set [62, 79, 80]. The Firefly Monte Carlo [62] method has attracted the most attention because it guarantees sampling from the correct target distribution, in contrast to other methods which introduce bias.
3. *Approximate algorithms:* There is a large variety of non-MCMC methods which can be used for the same Bayesian inference tasks as MCMC but provide biased results, i.e. they sample from an approximate posterior. These methods are typically faster than MCMC. Among these methods, it is worth mentioning Approximate Bayesian Computation (ABC) [81], which is based on proposing candidate θ values (θ^*), simulating data from the likelihood given the candidate ($\hat{\mathbf{D}} \sim p(\hat{\mathbf{D}} | \theta^*)$, which is not to be confused with evaluating the likelihood) and then comparing $\hat{\mathbf{D}}$ with the real data $\mathbf{D}$ using some summary statistic in order to accept or reject the candidate θ^* . ABC avoids the computational burden of computing the likelihood (and the posterior) but selecting a summary statistic that leads to tolerable acceptance rates is challenging [82]. Other notable approximate methods include Integrated Nested Laplace Approximation (INLA) [83], which is a popular substitute of pMCMC for SSMs and Variational Bayes [84] which is particularly popular in machine learning applications.

2.4.2 The approach of this thesis

In order to handle the increasing computational burden of modern MCMC applications and allow statisticians to keep analysing and drawing conclusions from massive datasets, it is insufficient to rely solely on more efficient MCMC methods or on smarter models; despite the above developments, there are still many problems that are out of reach for MCMC samplers due to forbiddingly long runtimes. This thesis builds upon the following conviction: In combination with methodological improvements, it is equally important to leverage the power of modern hardware accelerators, such as multi-core CPUs, GPUs and FPGAs. All these platforms offer massive amounts of parallel resources, which can be exploited when working with MCMC methods and Bayesian models amenable to parallelization, leading to speedups of orders of magnitude compared to sequential code running on conventional CPUs.

Even more crucially, understanding the properties of the underlying hardware platform and exploiting them is necessary not only during the implementation stage but also during the design stage of an MCMC algorithm. This thesis advocates a holistic design approach, where algorithm design is done jointly with the design of the hardware accelerator. Using this approach, existing MCMC algorithms can be modified or new MCMC algorithms can be proposed based on knowledge about which algorithmic and computational structures map favourably to existing parallel architectures. In addition, hardware design can be tailored to existing algorithms (e.g. PT or pMCMC), while the special features of the targeted hardware (e.g. custom precision) can be exploited in a better way when the characteristics of the algorithm are fully understood.

This thesis is a step towards the adoption of this holistic design philosophy; it focuses on how to design FPGA architectures tailored for MCMC sampling but also addresses MCMC algorithm design with a view on how these algorithms map to hardware. Although this thesis is devoted to FPGAs (with the exception of some parts of Chapter 3), comparisons between FPGA and state-of-the-art implementations on other platforms are also included.

2.5 Hardware acceleration technologies

This section provides an introduction to the characteristics of the main computing platforms available today, along with comments on their strengths when used for acceleration of computationally intensive

tasks such as MCMC. Starting from the most basic platform, the general-purpose microprocessor (i.e. Central Processing Unit - CPU), this section then moves to its parallel extension (multi-core CPUs), the massively parallel alternative of Graphics Processing Units (GPUs) and the very unique case of Field Programmable Gate Arrays (FPGAs).

2.5.1 Central Processing Units

The modern conception of a CPU as a stored-program computer was first described by John Von Neumann [85]. The fundamental ideas about how a CPU operates have remained largely unchanged, although various CPU architectures have been proposed since then.

Figure 2.7 provides a high-level abstraction of the architectural parts of a typical CPU. It consists of:

1. An arithmetic logic unit (ALU), which performs arithmetic and logic operations. These operations are encoded as instructions. Most CPU architectures have fixed instruction sets and all high-level code has to be translated and decomposed into these instructions.
2. A set of registers from which the ALU reads its inputs and writes its outputs.
3. A control unit that fetches instructions from the program memory, decodes them and then feeds them to the ALU for execution. This process involves the activation of various control signals and interactions between the necessary parts of the ALU, registers and other components.

The CPU also communicated with the main (off-chip) memory through a memory interface module.

Modern CPUs work are based on the same principles but they are much more sophisticated in terms of how they process instructions and access memory. They are equipped with complex hardware components to optimize the execution of sequential code, e.g. they use techniques such as Instruction Level Parallelism and Out-of-Order Execution to reduce the execution time of successive sequential instructions and speculative execution and branch prediction to improve concurrency and tackle branching statements more efficiently.

Apart from the above techniques, CPU architectures have made extensive use of multiple layers of cache memories in order to reduce memory latency (which is large when accessing off-chip memory). The idea behind on-chip cache hierarchies is that a fast cache with small size is placed close to the ALU in order to achieve low latency, a slower cache with larger size is placed at a large distance from

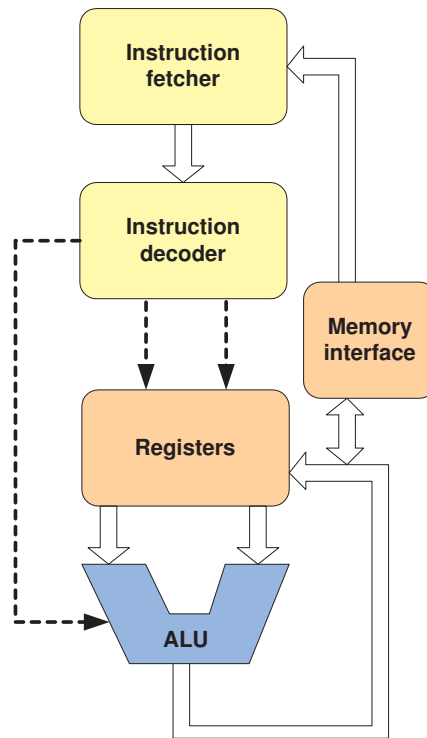


Figure 2.7: High-level abstraction of a typical CPU. The instruction fetches read instructions from main memory, the instruction decoder sends the necessary control signals to implement the instruction and the ALU executes the instruction using inputs from the registers as operands and writing the results back to the registers or the main memory.

the ALU, etc. While CPU registers can typically be accessed in one clock cycle, access times for caches range up to 10 or more cycles and access time for off-chip memory in the range of 100-200 clock cycles. The above cache model is critical for performance in modern CPUs. A large part of the transistors in a CPU is used for implementing large caches, as well as increasing their performance by using complex techniques that exploit the spatial and temporal locality of memory accesses.

The sequential programming model and its decline

Since the first commercially available CPUs were released by Intel in 1970 and especially after 1974 (the year that Intel 8080 was made available), the CPU market has grown at an impressive rate and CPUs can now be found in almost every digital device in the planet. The success of CPUs is due to their flexibility and their constant - until recently - increase in performance. Flexibility refers to their ability to run any sequential software code by decomposing the code into simple generic instructions which are executable by the CPU (using a compiler). This has led to a massive code base which can be easily run in newer CPU generations. The increase of CPU performance is related to the huge innovations in integrated circuit technology over the last decades. This is frequently connected to

Gordon Moore's prediction back in 1965, which has since been known as Moore's law: "The amount of transistors in a given amount of silicon will approximately double every 18 to 24 months" [86].

All these extra transistors have traditionally been used to enhance the various hardware modules and cache memories inside the CPU in order to increase performance, e.g. through improvement in the ALU pipeline, hyper-threading, using speculative execution, increasing the size and capabilities of caches, etc. Moreover, the simultaneous increase in transistors' switching speed has constantly pushed clock frequencies up. These two elements meant that programmers did not have to care about the capabilities of the underlying hardware; increasing frequencies allowed the same sequential program to run faster on newer CPUs. Therefore, developers only needed to wait for one or two extra years for the next CPU model to be released in order to cover the increased computational demands of applications.

This "free lunch" ended in 2004 because integrated circuit technology hit a "power wall" [87, 88]. It is no longer possible to increase clock frequencies while keeping the power envelope of the CPU at safe levels. This is due to limitations in transistor technology, i.e. the fact that power leaks in transistors increase at unsustainable levels as feature size drops. When feature sizes reach 90nm, the leaks become significant and it is no longer straightforward to dissipate the generated heat from the chip. At that point, the industry had to stop relying on increasing frequencies and instead find other ways to improve performance.

2.5.2 Multi-core CPUs

Despite the "power wall", Moore's law is still in effect. This means that the number of transistors per chip still increases but these extra transistors can no longer be used to scale clock frequency. Consequently, the focus has shifted towards parallelism in order to serve the increasing computational needs of programmers. The natural extension of the general, sequential CPUs - multi-core CPUs - has become the mainstream computing platform during the last ten years. Multi-core CPUs integrate two or more CPUs in the same chip, with individual and/or shared caches. Shared caches lead to even more complex and expensive control modules inside the chip. As shown in Figure 2.8 (which is a high-level abstraction), multi-core CPUs consume a lot of area for control hardware and cache memory. This means that relatively fewer resources are devoted to processing units (e.g. ALU). Therefore, the peak performance of a multi-core CPU (e.g. as measured by FLOPS - Floating point

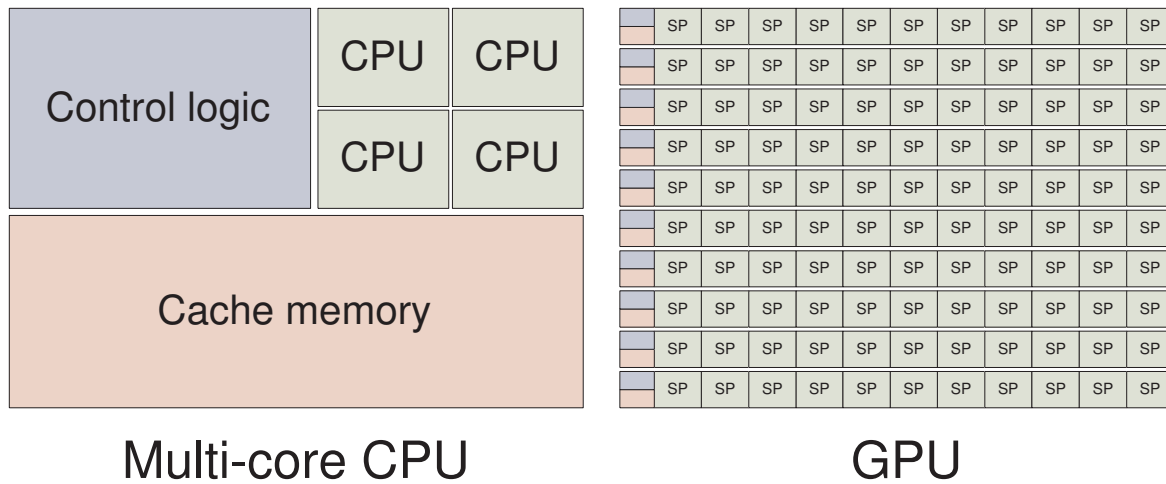


Figure 2.8: Comparison of multi-core CPU and GPU architectures. SP stands for Stream Processor, the main processing unit of the GPU (following Nvidia’s terminology). It is clear that the GPU devotes a much larger percentage of total chip area on compute units instead of control and cache.

operations per second) is typically lower than other platforms where more chip area is devoted to compute units (see next section).

Although multi-core CPUs support legacy software that was written for sequential CPUs, they cannot fully exploit their parallel hardware capabilities without some help from the programmer. This means that programmers need to adopt the new, parallel programming model in order to keep increasing the performance of their applications. Also, for many applications, a multi-core CPU does not provide enough parallelism to satisfy performance requirements as will become clear in the following chapter. The CPU cores inside a multi-core are large, powerful units with a lot of optimization targeted on sequential execution but they are few in number compared to massively parallel devices such as GPUs. Often it is not even possible to achieve acceleration factors equal to the amount of cores, especially if the algorithm requires frequent inter-core communication.

Despite these facts, multi-core are still much easier to use and program than competing platforms. Also, the large code base and the various optimized libraries which are available for CPUs are unparalleled by other platforms. Finally, multi-core CPUs remain competitive when the implemented algorithm demonstrates limited parallelism and/or has a lot of branching statements. All these factors make them extremely popular.

2.5.3 Graphics Processing Units

GPUs are chips whose architecture was originally devised and evolved to address the computational needs of modern video games, i.e. the specific kinds of computations that are necessary in this application field. Nevertheless, their massive computing power (which is a result of the large investment in the field) eventually led GPUs to become serious competitors in the general high performance computing field. Their adoption in various application domains during the last decade has been both wide and successful.

Like multi-core CPUs, GPUs are also based on placing multiple processing cores in the same chip. Nevertheless, the amount and characteristics of these cores are remarkably different compared to multi-core CPUs. GPU cores are less powerful; they do not use all the advanced sequential code optimization techniques found in CPUs and, most importantly, they do not employ complex cache memory structures. This allows them to devote more chip area to increase raw processing power, leading to higher peak performances than CPUs.

The right half of Figure 2.8 shows a typical Nvidia GPU architecture. Following Nvidia's terminology, the processing cores are grouped into Stream Multiprocessors (SMs). Each SM is one row in the figure. Each SM comprises a number of Stream Processors (SPs), shown as green blocks in the figure, which are the main computational units, as well a relatively small amount of cache memory and control logic. It is clear from Figure 2.8 that the SPs occupy most of the chip.

Modern GPUs have hundreds or thousands of SPs. However, this does not translate to massive speedups for any implemented algorithm. A GPU can reach its peak performance only when: 1) The algorithm's code is dominated by data-parallel computations, i.e. the same computation is performed on different sets of data, also known as Single-Instruction-Multiple-Data (SIMD). 2) The algorithm is compute-intensive, i.e. the ratio of computations over memory access operations is high. This ensures that the memory bandwidth is adequate to feed all the GPU SPs and keep them utilized. Moreover, there are other limitations to the types of algorithms that suit the GPU. For example, `if` or `while` statements can result in some of the parallel GPU threads running for longer than others. Due to the limited (compared to CPUs) amount of circuitry to handle these branching operations, this forces all threads to wait for the slowest to finish before continuing execution, causing inefficiency.

A GPU is programmed using languages based on parallel programming principles. Arguably the most popular is Nvidia's CUDA, which is an extension to C and allows the programmer to write functions

(called kernels) which are executed in the GPU. These kernels can be embedded in pre-existing C code and invoke an (ideally) large number of parallel threads, which are independent routines that run in parallel. CUDA threads are grouped into blocks. Threads within a block are assigned to the same SM in the GPU and can efficiently share data through the use of the SM's on-chip shared memory. Each thread also has access to a few local on-chip registers which are extremely fast to read to and write from. Communication between blocks has to happen through the GPU's global off-chip memory and is thus slower. Finally, a fourth type of memory, the constant off-chip memory is read-only during a kernel execution but it is faster than global memory due to caching.

A CUDA implementation needs to be optimized carefully in order to maximize performance. The amount of work per block and per thread and the communication between threads must map well to the GPU architecture. This optimization process is far from trivial for non-experts, making GPU programming more difficult than writing parallel code for multi-core CPUs.

2.5.4 Field Programmable Gate Arrays

FPGAs fundamentally differ from the previous three platforms. While CPUs, multi-cores and GPUs have fixed hardware architectures which are defined before the chip is manufactured, FPGAs consist of a re-programmable “fabric” upon which any custom hardware architecture can be mapped [89]. As Figure 2.9 shows, the FPGA fabric consist of an array of programmable logic blocks and a hierarchy of reconfigurable interconnects which allow the blocks to be wired together. Each block can implement a simple boolean function. By connecting the blocks together any digital circuit can be implemented. Moreover, modern FPGA are equipped with various heterogeneous elements which are hard-wired (they cannot change their functionality). These include multipliers, memories, I/O blocks and CPUs. Also, FPGAs released in 2014 (e.g. Altera's Generation 10 FPGAs [90]) have introduced hard-wired floating point cores to this list.

The re-programming of the fabric can be done as many times as desired and even during runtime and/or partially. This flexibility allows the designer to tailor the hardware to the specific characteristics of the application. The number and granularity of parallel processing elements, the kinds of arithmetic operators which constitute these elements, the size and architecture of cache memories and the arithmetic precision of computations are only some of the properties which can be customized. For example, while a GPU architecture has a pre-defined amount of on-chip memory per SM, an

FPGA can allocate a custom amount of on-chip memory to each “processing element” or even allow all elements to access all the memory, making the exchange of data more efficient (although the latter strategy can incur significant area and performance overheads due to extensive use of FPGA interconnect). Generally, for algorithms that do not adhere to the model described in the GPU section (SIMD, embarrassingly parallel), FPGAs can give better performance by exploiting non-obvious and/or limited parallelism, maximizing pipelining efficiency and adapting the memory architecture to the access pattern of the algorithm. Finally, FPGAs are able to combine the advantages of the CPU and custom hardware worlds. Apart from the hard-wired CPUs embedded in some FPGAs, all FPGAs can also use part of their fabric to implement one or more extra CPUs, which usually handle the sequential and non-critical part of the implemented algorithm, while the custom fabric is responsible for the parallel, computationally intensive part (hardware-software co-design).

Regarding the performance of the memory system, FPGAs enjoy higher on-chip memory bandwidth than GPUs [91] due to the large number of built-in registers and Block RAMs which are accessible in only one clock cycle. In contrast, FPGA typically achieve lower off-chip memory bandwidths than GPUs [91] (although this varies a lot depending on the FPGA board). Therefore, if processed data can be kept inside the FPGA or if a smart caching/reuse scheme can be implemented (see for example Rafique et al. [92]), this benefits the FPGA.

FPGAs can be programmed using Hardware Description Languages (e.g. VHDL, Verilog). Writing code for an FPGA is different and generally more difficult than writing software code. It resembles the description of a function block diagram, where each block computes a simple (e.g. addition) or more complex (e.g. Gaussian distribution) function. The blocks are connected with wires to give the final result. Many identical blocks can be instantiated in order to parallelize computations. A control block is typically included in the design in order to synchronize the operation of the remaining blocks. Also, memory blocks and blocks that take care of the communication with off-chip memory or the memory of the host PC have to be instantiated. All the above parts are completely customizable but pre-existing libraries of such components are available to simplify development. High-level languages (e.g. C, OpenCL) can also be used to program FPGAs and this is currently a very active research area [93]. The main disadvantages of FPGA are the following: 1) They are hard to program, especially when the programmer has no hardware design experience, 2) They are more expensive than GPUs and not widely available in desktop PCs, 3) The compilation times (synthesis, placement and routing) can reach hours for large chips.

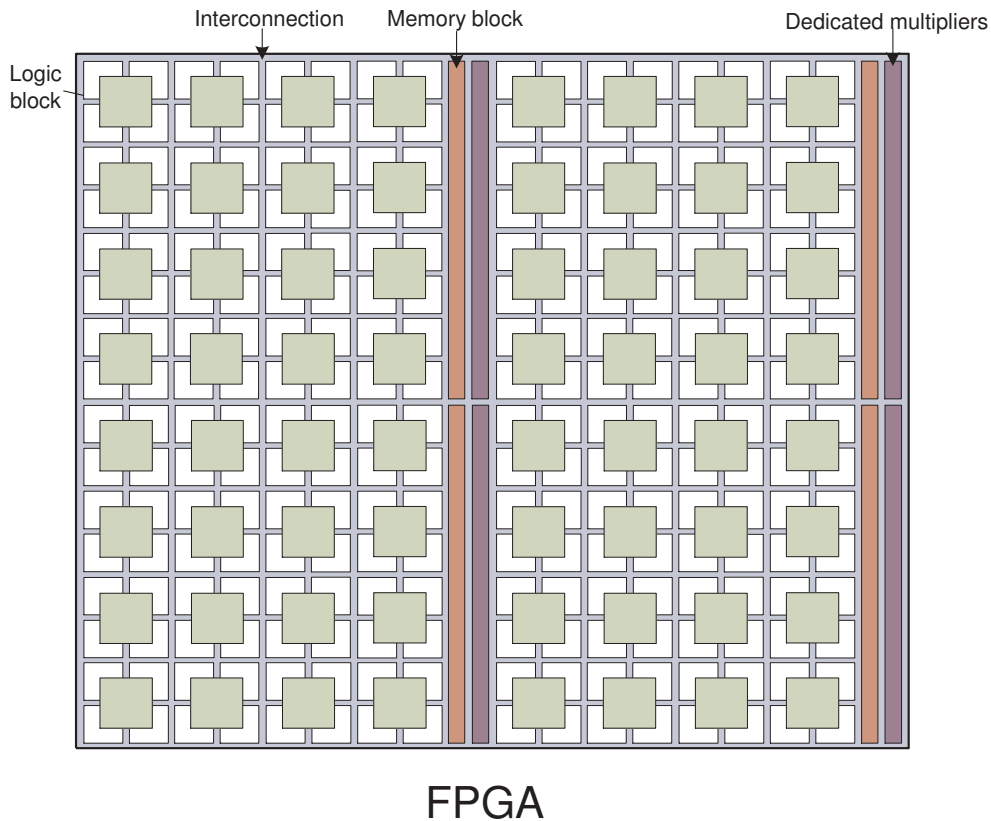


Figure 2.9: Simplified FPGA architecture.

Suitability of FPGAs for MCMC acceleration

This thesis focuses on the use of FPGA as platforms for accelerating MCMC algorithms. The choice to focus on FPGAs, instead of GPUs or CPUs, was based on the fact that the flexibility and unique features of the FPGA offer great promise for performance gains when mapping MCMC.

In more detail, the re-programmability of the chip means that different architectures, optimized for different MCMC algorithms can be designed and then loaded on the chip on demand. The architectures can take advantage of all the special feature of the targeted method in order to improve performance and they do not require a SIMD model of computation as GPUs do. The architectures for popMCMC and pMCMC in Chapters 3 and 4 respectively are good examples of architectures tailored for a specific method. Moreover, with FPGAs, it is possible to build custom processing elements to compute the likelihood and prior of a given model. These elements only have the arithmetic operators necessary to compute the given functions and are thus more efficient than the general-purpose elements of fixed-architecture devices. This is particularly useful considering that the variety of likelihood functions in Bayesian modelling is large, ranging from easily parallelizable Gaussian evaluations to complex

linear algebra operations [10] or specialized algorithms [25]. Also, the fast on-chip memory access of FPGAs is an advantage when frequent communication between parallel processes is required (e.g. exchange step in popMCMC, resampling step in pMCMC).

Another particularly interesting attribute of FPGAs which is relevant to MCMC is their ability to use custom arithmetic precision (instead of the standard double-precision and single-precision floating point arithmetic used in CPUs and GPUs). Lowering precision saves chip area and thus more computations can be done in parallel. However, lower precision also means that more error is introduced in the output of the system. More details about custom precision can be found in Section 2.6.2. The motivation for using custom precision in MCMC comes from the following intuition: The output of stochastic algorithms (such as MCMC) is always given with some variance due to the randomness of these methods (in contrast to deterministic algorithms). This variance can be an advantage when lowering precision, since it can “hide” precision-related errors, if these errors are significantly lower than the variance. This can potentially allow for more savings in FPGA chip area than a deterministic method would allow. Previous work by Chow et al. [3] has already shown that the use of custom arithmetic precision in Monte Carlo algorithms can result in significant speedups without sacrificing accuracy.

Furthermore, FPGAs are extremely efficient at generating uniform or gaussian random numbers. Thomas et al. [5] and Thomas [94] have shown that FPGAs can outperform GPUs when generating random numbers on chip and that certain architectural features of the FPGA are particularly suitable for mapping well-known random number generators. Given the importance of fast random number generation in MCMC, this can prove a significant advantage. Finally, an important property of FPGAs is their low power consumption compared to CPUs and GPUs. Savings of an order of magnitude in energy efficiency are typical [95]. This can be a crucial advantage for High Performance Computing (HPC) or embedded applications of MCMC.

2.5.5 Other platforms

Many other computing platforms, either generic or designed for specific applications, exist today. It is worth mentioning Digital Signal Processors (DSPs), which are CPUs specialized for digital filtering and other signal processing applications. Their architectures and instruction sets are tailored for efficiently performing the kinds of computations necessary in these fields, most notably multiply-

accumulate operations. They are also traditionally built to perform computations in fixed-point arithmetic precision, a feature that could be exploited by the custom precision methods presented in this thesis (although this is not explored here). Another category of non-standard CPUs is the family of Very long Instruction Word (VLIW) processors. These devices allow multiple instructions to be executed simultaneously and this can be determined at compile time. Finally, several FPGA, GPU and CPU vendors have released heterogeneous (mixed-architecture) devices that include CPUs, GPUs and/or FPGAs in the same chip. These include ARM Mali, AMD Fusion, Intel Ivy Bridge and Xilinx Zynq among others. These platforms are suitable for applications which comprise diverse processing tasks, each suitable for mapping on a different architecture, for example a mix of sequential tasks (which run fast on a CPU), matrix operations (which can be implemented efficiently on a GPU) and custom precision computations (which are more suitable for FPGAs). Bayesian inference applications often belong to this category.

2.6 Arithmetic precision

2.6.1 Floating point arithmetic

Arithmetic (or numerical) precision refers to the number of bits used to represent numbers in a computer, as well as the way these bits encode the number. The number and arrangement of bits define which numbers can be represented, making different precision schemes suitable for different applications. In general, the larger the number bits (i.e. the word-length), the more accurate the computations become.

The most common arithmetic format in general-purpose computing is the floating point format. This format will be used in all algorithms and hardware implementations of this thesis. In floating point, numbers are represented using a sign bit, the mantissa a (which consists of m bits that represent a number in the range $[0, 1)$) and the exponent b (which consists of e bits that represent a signed integer):

$$x = \pm 2^{b_e \dots b_2 b_1} \times 0.a_m \dots a_2 a_1 \quad (2.23)$$

The term *precision configuration* will be used in this thesis to describe the number of mantissa and exponent bits of the employed floating point format. This can be represented by $c = (m, e)$.

The use of a separate exponent allows floating point numbers to represent a large range of values (in contrast to the fixed-point format where no exponent is used). This is very useful when there is no information about the dynamic range of variables in a program. Floating point is used as the default format in these cases. Nevertheless, the result of using this format is that representable numbers are not uniformly spaced. The difference between consecutive representable numbers increases when the value of the exponent grows. Practically, this means that floating point is more “accurate” for small numbers and less “accurate” for large numbers.

Of course, as with all arithmetic formats, the accuracy of computations changes with different precision configurations. When the number of mantissa and exponent bits drop, the error in all calculations (e.g. additions, multiplications) increases. The accumulation of these errors as the numbers pass through the datapath can lead to significant bias in the output.

2.6.2 Custom precision floating point in CPUs, GPUs and FPGAs

CPUs, GPUs and most other fixed-architecture platforms support only pre-defined floating point configurations. These are the single-precision configuration, i.e. $c = (24, 8)$ and the double-precision configuration, i.e. $c = (53, 11)$. CPU and GPU architectures have fixed compute units to handle these two formats. Nvidia has announced that its forthcoming Pascal GPU architecture (scheduled for release in 2016) will inherently support a new format, half-precision floating point ($c = (11, 5)$) [96]. The latest version of CUDA already supports half-precision for storing variables and operating on them but the actual computations are still done on single-precision compute units on the GPU device [96]. Half-precision is not employed in the GPU implementations of this thesis. Other floating point formats can only be simulated but with significant cost in terms of performance.

Nevertheless, some applications do not actually require single- or double-precision (or even half-precision) to produce accurate results. Lower precisions are enough in many cases. Moreover, if there is a way to check the dynamic ranges of all variables in an algorithm, it is possible to use even a fixed-point precision format that guarantees coverage of these dynamic ranges.

FPGAs are able to exploit this because, in contrast to fixed architectures, they do not have built-in units that work in a pre-defined precision. In an FPGA system, a designer can instantiate arithmetic operators (e.g. adders, multipliers) which operate in any custom precision configuration. Most importantly, the FPGA area required to implement these operators drops with lower precisions [97, 98] (while for

CPUs and GPUs, computations in low precisions are simulated and thus take more time). This means that more operators can be instantiated in the same FPGA fabric, leading to more parallelism and thus higher performance. FPGAs are thus the ideal platform to experiment with custom precision if there are reasons to believe that the targeted algorithm is robust to precision reduction (i.e. the error in the output of the algorithm is relatively insensitive to precision and thus significant precision reductions can be tolerated). The above approach (i.e. using precision reduction to boost parallelism is employed in Chapter 3 and 5 of this thesis). It has to be noted that reducing precision also results in a reduction of the latency of operators in FPGAs. This effect is not investigated in this thesis, since: 1) Throughput (and not latency) is the critical factor for performance in all the FPGA architectures of this thesis, 2) Latency reduction when lowering precision is limited for most of the floating point operators used in this thesis.

For more information on the trade-offs between precision and FPGA area, see Table 3.6 of Chapter 3 and Figure 5.7 of Chapter 5, as well as Detrey and de Dinechin [98]. For more details on the trade-offs between precision and sampling accuracy in MCMC see the next section (Section 2.6.3), as well as the evaluation results of Chapter 5. It has to be noted that fixed-point precision is not considered in this thesis.

2.6.3 Custom precision in MCMC

As mentioned earlier in this chapter, the typical problem that MCMC aims to solve is the estimation of the integral (2.8). MCMC draws samples $\theta^{(i)}$, $i \in \{1, \dots, N\}$ from $p(\theta)$ (i.e. the target distribution) and uses them to evaluate the output estimate (2.9), which is an unbiased estimator of (2.8).

Equations (2.8) and (2.9) are theoretical; they assume that all computations inside MCMC are performed in infinite precision. This means that MCMC samples are distributed according to the true distribution $p(\theta)$ and the output estimate (2.9) converges to the true value (2.8) for $N \rightarrow \infty$. Nevertheless, in practical MCMC implementations on digital devices, infinite precision is impossible to achieve. As a result, almost all work in MCMC literature uses either double or single precision floating point, since these are the two format that are inherently supported by CPUs. Both formats are considered adequate for most existing problems (although there is no guarantee that they are indeed adequate). In this thesis, double precision is used as equivalent to infinite precision in Chapters 3 and 5. In these chapters, precision optimizations take place and double precision was chosen as the

reference (infinite) precision because it is the highest precision supported by CPU and GPU architectures. On the other hand, Chapter 4 uses single precision for all implementations, since no precision optimization takes place in this case.

All MCMC algorithms consist of two main parts: 1) The evaluation of the probability density of each proposed sample $p(\theta^*)$. This part is specific to the targeted problem (i.e. the likelihood and prior). 2) The generic operations, which mainly include proposing samples and accepting/rejecting them according to their probability density value. This part is generic, i.e. it is the same regardless of the targeted problem, as long as the MCMC method does not change. Examples of what these two parts look like have been given earlier in this chapter when describing various MCMC methods and Bayesian models.

The probability evaluations (first part) take up the bulk of the computation time, especially when complex models and large-scale data are employed. In an FPGA implementation this means that most of the FPGA area will be devoted to probability evaluation modules. The generic MCMC operations are less computationally demanding. Therefore, when interested in achieving high sampling throughput for large-scale problems (which is the goal of this thesis), the crucial task is the minimization of the cost of implementing $p(\theta)$. Part of this thesis (Chapters 3 and 5) focuses on how to achieve this goal by reducing floating point precision in FPGAs. As mentioned above, lower precision leads to area savings in the FPGA fabric, which in turn allows for more parallel modules to be instantiated.

Treating the two parts of MCMC as separate precision domains (all in floating point precision), it is possible to acquire four different precision combinations, shown in Table 2.1. Double precision is used in the place of infinite precision in this section (and in all parts of the thesis that deal with precision optimization). Combination D/D (double precision / double precision) is the mainstream approach followed in MCMC literature; it theoretically guarantees convergence to the “true” probability distribution but it is not efficient computationally, since it requires double precision arithmetic to be used in all parts of the implementation (meaning no area savings in the FPGA).

Combination D/C (double precision / custom precision) performs all generic operations in double precision, thus guaranteeing convergence to *some* probability distribution. Due to the use of custom precision for probability density evaluations, this stationary distribution is not the “true” distribution ($p(\theta)$) but an approximation of it (denoted $p_c(\theta)$, where $c = (\text{mantissa bits}, \text{exponent bits})$ is the precision configuration used). Also, Combination D/C leads to high area savings in the FPGA due

Table 2.1: Combinations of precision configurations in the two precision domains of MCMC and the effects on: 1) convergence to the target distribution, 2) area savings/sampling throughput in an FPGA implementation. DP stands for double precision floating point, CP stands for custom precision floating point.

	Probability density domain in DP	Probability density domain in CP
Generic domain in DP	<i>Combination D/D</i> : “Correct” kernel (all theoretical properties of MCMC maintained) - targets “true” target distribution $p(\theta)$. No area savings, no increase in sampling throughput.	<i>Combination D/C</i> : “Correct” kernel (all theoretical properties of MCMC maintained) - targets approximate but known target distribution $p_c(\theta)$. Large area savings, large increase in sampling throughput.
Generic domain in CP	<i>Combination C/D</i> : “Perturbed” kernel (no guarantee of where/if it converges) applied to the correct target distribution $p(\theta)$. It actually samples from an unknown distribution. Small area savings - small increase in sampling throughput.	<i>Combination C/C</i> : “Perturbed” kernel (no guarantee of where/if it converges) applied to the approximate distribution $p_c(\theta)$. It actually samples from an unknown distribution. Large area savings - large increase in sampling throughput.

to the use of low precision for probability density evaluations (which is the most computationally intensive part and typically takes most of the FPGA area). This approach is thus the most promising of the four approaches in the table.

Figure 2.10 shows the shape of $p_c(\theta)$ (in this case a standard Gaussian density) for various precisions c . Only the number of mantissa bits changes. The effect of reducing the mantissa bits is that the approximation to the “perfect” density becomes coarser. The support of the distribution remains the same, since the number of exponent bits is not reduced.

Combinations C/D (custom precision / double precision) and C/C (custom precision / custom precision) are less interesting. Neither of them guarantee convergence to any target distribution because custom precision is used for generic operations. It is possible that unexpected behaviour will occur when following one of these approaches (i.e. either the sampler does not converge or it converges to an unknown distribution). In addition, Combination C/D does not provide significant area savings (and thus speedup) in the FPGA due to the use of double precision for probability evaluations. Area savings from implementing the generic operations in reduced precision are limited.

Following Combination D/C (which will be employed in Chapter 5 and partly in Chapter 3), the

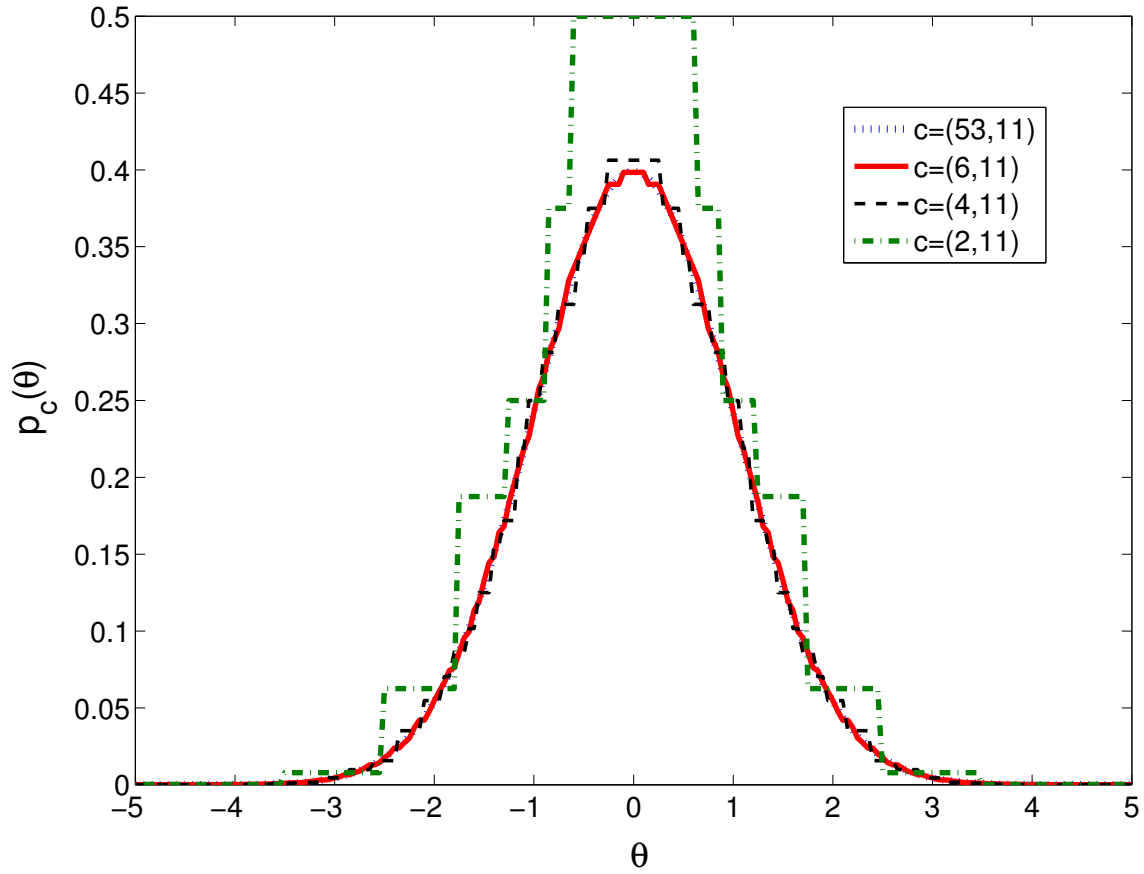


Figure 2.10: Shape of $p_c(\theta)$ when changing the precision configuration c . DP stands for double precision. Exponent bits are constant (=11).

approximated integral is no longer given by Equation (2.8). It is now the following:

$$I_c = E_{p_c}[f(x)] = \int f(\theta) p_c(\theta) d\theta \quad (2.24)$$

The value of the custom precision output estimate is:

$$\tilde{I}_c = \tilde{E}_{p_c}[f(x)] = \frac{1}{N} \sum_{i=1}^N f(\theta^{(i)}) \approx I_c \quad (2.25)$$

where $\theta^{(i)}$, $i \in \{1, \dots, N\}$ are samples drawn from $p_c(\theta)$.

Therefore, combining (2.8) and (2.24), it is clear that the following bias is introduced in the output estimate due to the use of custom precision:

$$b_c = I - I_c = \int f(\theta) p(\theta) d\theta - \int f(\theta) p_c(\theta) d\theta \quad (2.26)$$

In Chapter 3, it will be shown that in the case of popMCMC, it is possible to use custom precision densities (like the ones in Figure 2.10) and at the same time correct or avoid the bias (2.26) in the output estimate. Chapter 5 will propose a way to optimize the precision configuration c given a user-defined threshold for the bias (2.26).

2.7 Related work

This section contains an extensive review of existing literature in fields closely related to the present thesis. These fields are MCMC acceleration using parallel hardware (covering several approaches but focusing particularly on popMCMC, pMCMC and PF implementations), tuning of the number of particles in MCMC, custom precision techniques and optimization methods for Monte Carlo algorithms, as well as other research which cannot be classified in one of the above categories. Each section is supplemented with comments on how the work on this thesis compares to the reviewed literature.

2.7.1 MCMC parallelization and hardware acceleration

Overview of approaches

The main challenge when accelerating MCMC on any parallel platform is its inherently sequential nature; the generation of each sample of the Markov chain requires the previous sample to be available. Therefore, the algorithm cannot be trivially parallelized. Several approaches (comprising algorithmic modifications or hardware implementation techniques or both) have been proposed to bypass this problem:

1. Execution of many independent Markov chains which sample from the same distribution, in order to generate more samples in the same amount of time on a parallel platform. This is a straightforward way to parallelize sampling but it was deemed wasteful even in the early days of MCMC parallelization [99, 100]. An MCMC chain often needs a long time to converge. This means that running many chains with the same convergence time in parallel can be a waste of resources; all of the chains need to converge and therefore burn-in samples need to be thrown away from each chain. No gain in convergence speed is achieved using this strategy and, even worse, it is possible that no chain will converge. Even when the chains have converged, slow

mixing might mean that each of the chains remains stuck in one part of the space for a long time. In other words, this approach does not tackle any of the inherent issues that make MCMC inefficient. Therefore it would be preferable to use the parallel resources available to improve the mixing speed or the sampling throughput of one chain instead of running multiple chains in parallel. Rosenthal [101] and Wilkinson [102] presented results from using the straightforward approach in multi-core CPU systems and demonstrated that the variance of the estimates was reduced compared to a single chain that ran for the same amount of time (for problems where convergence and mixing are rapid).

2. Use of MCMC methods which exhibit natural parallelism. There are several such methods in the MCMC literature: Algorithms that run multiple chains which sample from different distributions in parallel (e.g. population-based MCMC [7]), algorithms that propose multiple samples in parallel (e.g. Multiple-Try Metropolis [56]), algorithms that use a parallelizable particle filter for density estimation (e.g. pMCMC [33]) and algorithms that exhibit some other form of parallelism, e.g. matrix computations [57, 55]). This parallelism can be exploited when implementing these methods on parallel hardware platforms. The speedups that can be achieved depend on the employed MCMC method and the structure of its parallel computations. The subsequent parts of this section will review parallel accelerators for popMCMC and pMCMC in detail. Here, a list of parallel implementations of other methods is given: Gopal and Casella [103] proposed the use of regenerative simulation to introduce parallelism in MCMC, a strategy applicable to any MCMC variant. The reported speedups with an R implementation running on multiple CPU cores were almost proportionate to the number of processors but no details were given on the hardware used. Tibbits et al. [8] mapped multivariate Slice sampling on a GPU and achieved a speedup of 5-6x compared to a CPU. The main focus of this work was the acceleration of the construction of the hypercube (necessary to generate the uniform numbers mentioned in Section 2.3.4), which is the most computationally expensive task in multivariate Slice sampling. Beam et al. [104] used a GPU to accelerate Hamiltonian Monte Carlo; the computations necessary to evaluate the gradient and the posterior density were transformed in order to be expressed in terms of simple matrix computations. These were then sent to the GPU, achieving 100-fold speedups over a CPU implementation for large problems. The decomposition is specific to the model used (multivariate regression) but it can be generalized to other models.

3. Use of speculative execution (i.e. pre-fetching) techniques to perform many MCMC steps in parallel and keep the parallel resources of the device utilized. For each step of the chain, this strategy computes (in parallel) the paths that the chain would follow in case the proposed sample is accepted or rejected. This is done recursively, creating a tree of possible paths. This way, the algorithm does not have to wait until an acceptance step is finished to continue processing. Byrd et al. [105] used such a speculative strategy to accelerate MCMC on a multi-core CPU with reported speedups of up to 2.5x with four cores. Strid [106] and Angelino et al. [107] proposed similar pre-fetching methodologies and demonstrated how they scale with the number of CPU cores. The results indicate that speedups of up to 10x are possible with 32 or 64 cores in CPU cluster environments (containing Intel Xeon processors) and that speedup gains are diminishing as the number of cores grows.

4. Parallelization of the computations inside every step of MCMC (primarily the evaluation of the probability density of each proposed sample). The techniques used in this case depend on the form of the density and how amenable it is to hardware acceleration. A GPU-based example of this approach can be found in Suchard et al. [66], where large-scale mixture model density computations were parallelized, offering speedups of one order of magnitude compared to multi-threaded code run on a desktop CPU. In Asadi et al. [18], a multi-FPGA architecture for MCMC inference in Bayesian networks was described. The achieved acceleration was based on massive parallelization of the intra-chain acceptance step calculations. Part of these calculations were pre-executed and reused repeatedly. Also, a small number of parallel chains were used to enhance mixing but no details were given on how much this strategy helps. Moreover, limited details were given on the software reference implementation and the level of optimization that has been applied to it. The reported speedups (four orders of magnitude) are extremely high and are not justified, indicating that the comparison between software and hardware is not fair. Bayesian network inference using PT was also accelerated in Lebedev et al. [65] as a benchmark for the many-core architectural template proposed in the same paper. Zierke and Bakos [25] and Alachiotis et al. [63] used FPGAs to accelerate phylogenetic inference, taking advantage of the specific form that the likelihood takes in this problem and the specialized algorithms to evaluate it; the achieved performance gains against server-class CPUs were up to 10x. As the above results indicate, speedups in this category of parallel implementations largely depend on the type of target density and how amenable it is to parallelization.

5. Separation of the data set into independent sub-groups and execution of a separate MCMC sampler for each sub-group. This approach has already been mentioned earlier as one of the newest developments in MCMC methodology (Section 2.4.1). Limited related work exists in the parallel hardware-based MCMC literature. In the multi-core CPU field, Whiley and Wilson [64] first proposed strategies to distribute computations for spatial gaussian process inference (although some of them can be applied to other models) and evaluated them on an 8-core Beowulf cluster, achieving a 5x speedup over sequential code. This approach introduces some error in the results because the partitioning of the data does not take into account the dependence between neighbouring partitions. More recent works [76, 77] proposed unbiased data partitioning strategies for MCMC but no runtime results from actual implementations were given. However, it is expected that these techniques can achieve speedups almost proportional to the number of cores, assuming i.i.d. data and small burn-in periods for each sampler. The reason for this is that communication between parallel MCMC samplers is required only after sampling terminates. Some works have also used data partitioning techniques tailored for specific applications: Mansinghka et al. [108] used FPGAs to accelerate inference in Markov Random Fields by splitting the problem into independent sub-groups of data and running a Gibbs sampler for each sub-group in parallel. Belletti et al. [19] used an FPGA-based supercomputer to perform expensive physics simulations (e.g. Ising models, Potts models), again taking advantage of the independence between parts of the model. In the latter work, a lot of application-specific characteristics were exploited in combination with the FPGA implementations (integer computations, table lookups). The approach of data partitioning in combination with parallel hardware is suitable for the era of large-scale data but the most promising techniques [76, 77] are still limited to exploiting i.i.d. data parallelism.

Parts of this thesis belong to the second category described above. In particular, Chapters 3 and 4 propose FPGA architectures for popMCMC and pMCMC respectively. A part of Chapter 3 belongs to the fourth category, since the probability density computations are also parallelized. Chapter 5 cannot be classified in any of the above categories since it applies parallelization only indirectly (through reducing the precision of computations).

While works in categories 2 and 4 have employed CPUs, clusters and GPUs for MCMC acceleration, FPGA use has been limited (in category 4) or non-existent (in category 2). Nevertheless, some of the most popular MCMC methods with inherent parallelism (category 2) are good candidates for

FPGA implementation. In fact, through the use of tailored architectures and custom precision, FPGA accelerators can outperform CPUs and GPUs when doing MCMC sampling, as will be shown in the main chapters of this thesis.

Acceleration of population-based MCMC

A number of works on accelerating popMCMC using parallel hardware have been published in the last five years. In Li et al. [15], a CPU cluster was used to accelerate PT and a method was proposed to limit the communication overhead between processors in the cluster during Global exchange operations (see Section 2.3.2). The runtime of the implementation remained almost constant when increasing the number of parallel processes (each run on a separate processor). This is better than the scaling of centralized PT code, whose runtime increased at a much faster rate in the same scenario. In Earl and Deem [109], a scheme to allocate chains to parallel CPU cores was proposed with the aim of minimizing the CPU idle time. Chapter 3 shows how both these overheads can be avoided in FPGA implementations that use local FPGA BRAMs to store samples.

The first work to tackle PT acceleration on a GPU was published by Lee et al. [16]. GPU implementations of popMCMC were presented and the achieved acceleration ranged from one to two orders of magnitude against sequential (non-optimized) CPU implementations. Lee et al. [16] took advantage of the multiple tempered chains in PT, assigning one chain to each GPU thread. The independence of chains during Global update operations allowed easy parallelization on a GPU, leading to the aforementioned high speedups. These speedups grew with the number of chains, since the GPU gradually exploited a larger percentage of its parallel resources when more chains were used. This work will be used as a reference in Chapter 3 because it provides clear results of how the performance of the implementation scales with the number of the PT chains. Chapter 3 (Section 3.3.1) builds upon this work, proposing an improved GPU accelerator for the same algorithm (which takes advantage of intra-chain parallelism), as well as an optimized multi-core CPU implementation. Moreover, mixed-precision versions of the CPU and GPU accelerators are introduced. These implementations are compared to the respective FPGA accelerators. Moreover, the results of Chapter 3 show how the performance of these accelerators scales with the size of the data, an analysis which is absent from [16]. Gross et al. [26] also addressed PT acceleration on GPUs and applied the algorithm to polymer simulation. The maximum speedup of an Nvidia GTX480 over an Intel Xeon processor was 130x and was achieved with the maximum number of chains (240); this result confirms the findings of Lee et al. [16]. Another

GPU-based accelerator was proposed in Zhu et al. [67], targeting a different popMCMC algorithm (DEMCMC). A speedup of up to 100x compared to software was achieved.

The only previous works that have employed multiple chains in FPGA implementations are Belletti et al. [19] and Asadi et al. [18]; both used small numbers of parallel chains (e.g. four chains) to accelerate mixing in a PT-like fashion but the focus of these works was on exploiting the form of the target density. The mapping of the parallel chain computations was done in a straightforward manner, without the tailored architectures and custom-precision schemes of Chapter 3.

Although it is clear from the above results that GPUs are a good match for popMCMC (due to the embarrassingly parallel nature of the algorithm), Chapter 3 of this thesis shows that FPGAs can outperform GPUs by exploiting custom precision on top of the algorithm's parallelism. Two custom precision PT algorithms are proposed which manage to use low precision in the largest part of the FPGA implementation without affecting sampling accuracy. These algorithms are also applied in a multi-core CPU and GPU setting (where the only choice for the low precision configuration is single precision floating point), where performance gains are smaller than in the FPGA case. This thesis is also the first work that jointly examines how each platform's performance scales with the number of chains and the size of the data. Previous works investigate only one or none of the above. In addition, this thesis presents, for the first time in MCMC literature, a power efficiency assessment of PT in various platforms.

Acceleration of Particle Filters

There is extensive literature on PF acceleration using parallel hardware platforms with some previous work also introducing algorithmic or application-specific modifications to the basic PF algorithm to improve performance. The main focus of most previous work is the resampling step (see Section 2.3.3), since it is the only step of the PF which is not trivially parallelizable (it requires a collective operation involving all particles, e.g. a sum or a cumulative sum). Resampling becomes the dominant computation when the particle population increases [72].

The main interest of this thesis is the acceleration of MCMC algorithms, therefore no direct comparisons with the methods of this section are done in the following chapters. Nevertheless, the use of PFs in pMCMC (which is examined in Chapter 4) makes it necessary to provide an overview of PF acceleration research.

The majority of FPGA accelerators for PFs are used in applications with real-time constraints, e.g. target tracking or computer vision. Moreover, these accelerators are limited in terms of the number of particles they can use (typically, hundreds or a few thousands). The first research efforts to accelerate PFs using FPGAs were published in 2004 and 2005. Bolic [110] and Athalye et al. [111] introduced hardware architectures for PFs, implemented them on FPGAs and applied them to tracking problems. Bolic [110] also developed several techniques to reduce the memory requirements of the PF, while Athalye et al. [111] proposed a new resampling algorithm (Residual Systematic Resampling - RSR), whose FPGA implementation is more efficient (in terms of clock cycles needed to perform resampling) compared to the classic Systematic Resampling (SR) algorithm. A 60% speedup over a DSP processor was reported for a small tracking problem but no other performance comparisons to software were shown. A slightly modified version of RSR, described by Liu et al. [1], is used in Chapter 3 for the resampling step of the FPGA pMCMC accelerator. In 2005, Bolic et al. [112] were the first to propose a distributed PF algorithm, i.e. an algorithm which splits the particles into chunks and assigns the processing of each chunk to a separate sub-PF. Distributed resampling is particularly suitable to many-core computing platforms and clusters but it can negatively affect the accuracy of resampling because sub-PFs do not have access to all the particles. Bolic et al. [112] proposed two methods to avoid or minimize resampling errors in distributed PFs and implemented them on FPGAs.

Some other significant contributions in the field of PF acceleration using FPGAs have focused on proposing automated frameworks for implementing PFs on these devices while maximizing performance and minimizing energy consumption. Saha et al. [113] in 2008 and Happe et al. [114] in 2009 proposed two parameterised frameworks for the implementation of PFs on FPGAs. Saha et al. [113] mapped the whole PF on the FPGA fabric. The results presented showed very small speedups compared to software and did not extend to more than few hundreds of particles. Happe et al. [114] applied a hardware/software co-design approach; a real-time operating system ran on the embedded FPGA CPU and allocated processing tasks to software and hardware threads. An arbitrary number of threads can be chosen by the user separately for each PF stage (sample, weight, resample). A design space exploration was described to optimize performance. Moreover, the authors proposed two run-time reconfiguration schemes to change the hardware/software partitioning at runtime for applications that require it (because of changing real-time constraints). Reported speedups (with resampling done exclusively by software threads) ranged up to 4x against a CPU. A more recent work [115] (in 2014) also proposed a high-level framework for PF implementation on FPGAs, providing a much simpler interface to the user and support for multiple FPGAs. The framework was also equipped with an auto-

mated design space optimization tool. The reported speedups over optimized, parallel CPU and GPU code were up to 10x and up to 4x respectively. Nevertheless, the fact that resampling is performed in software is likely to make the speedup vs the GPU drop significantly for larger particle numbers. While all these FPGA implementations kept the number of particles fixed at runtime, Chau et al. [116] (in 2012) developed an FPGA PF which could adapt the size of the particle population dynamically and targeted at applications with real-time and energy constraints. Particles with small weights were removed in each PF iteration (minimizing runtime), until some predefined lower threshold was reached, at which time resampling occurred and the initial particle number was reset. A method to allocate particles to processing elements in order to minimize energy consumption was also introduced. The processing elements were implemented as soft-core processors. Energy savings of 35%-75% were reported with small particle populations. Finally, several application-specific works on FPGA-based acceleration of PFs have been published [117, 118, 119].

In the GPU field, a significant amount of PF accelerators have been proposed, focusing mostly on how to accelerate the resampling operation. Hendeby et al. [72] and Lee et al. [16] (both in 2010) were among the earliest works on PF acceleration. Hendeby et al. [72] implemented a PF and compared it to a CPU on a tracking problem. They used stratified resampling and employed a parallel prefix sum algorithm to compute the necessary cumulative sum. Nevertheless, the small GPU register size limited resampling performance, while the use of CPU-based random number generation proved a bottleneck; the GPU only slightly outperformed the CPU for large numbers of particles (up to one million) and was slower for small particle populations. Lee et al. [16] presented a straightforward PF implementation using multinomial resampling. Resampling was again the bottleneck computation. Speedups of 8x-37x compared to sequential CPU code were reported for a stochastic volatility model (with up to 128K particles). In 2013, Hwang and Sung [120] proposed a technique to overcome one of the reasons that resampling is inefficient on GPUs, namely the workload imbalance between threads during particle replication. Their method was shown to be 2x-10x faster than a conventional GPU implementation.

Various GPU implementations have resorted to the use of distributed resampling techniques to avoid the need for expensive resampling on the full particle set [121, 122, 123, 124]. In 2010, Chao et al. [121] proposed the use of distributed resampling in combination with the Finite-Redraw-Importance-Maximizing (FRIM) technique in the sampling step, which repeats sampling if the particle's weight is too small. This technique can reduce the number of particles required to achieve a certain level of

estimation accuracy. With the use of a low-end GPU, they reported speedups of up to 5.7x compared to a straightforward, non-distributed, non-FRIM GPU implementation (with few thousands of particles). The distributed implementations of Chitchian et al. [122] in 2013 and Par and Tosun [124] in 2011 achieved speedups up to 100x and 75x respectively compared to sequential software, demonstrating the effectiveness of distributed techniques for massive particle numbers (the former was applied to a robotic control application with 2 million particles, while the latter was applied to a localization and mapping problem with 128K particles). Other modified versions of the basic PF on GPUs can be found in Cabido et al. [125] and Gelencser-Hovath et al. [126], while application-specific PFs can be found in Brown and Capson [73] and Lozano and Otsuka [127].

In 2013, a multi-core CPU implementation of a PF was described in Chitchian et al. [122], where a parallelized, distributed PF was run on a 16-core Intel Xeon CPU and was shown to be up to 6.5x faster than sequential CPU code. A similar setting with a 6-core CPU is found in Par and Tosun [124], with speedups of up to 4.7x. The performance of several distributed resampling algorithms when run on multi-core CPUs was evaluated in Rosen et al. [128] in 2010, where one of the algorithms of Bolic et al. [112] was shown to achieve almost linear speedup with respect to the number of cores due to the limited communication between sub-PFs.

In general, the speedups achieved when accelerating PFs using parallel platforms are smaller than the speedups achieved when accelerating popMCMC methods (typically by one order of magnitude). This is due to the resampling step in PFs, which is not straightforward to parallelize. In contrast, popMCMC chains are easily parallelizable. Distributed resampling can serve to close this gap but the methods to control and guarantee the quality of resampling must be employed.

Acceleration of Particle MCMC

Previous work on accelerating the pMCMC algorithm is limited. The first ever GPU implementation was introduced by Henriksen et al. [129] in 2012. The pMCMC algorithm used was the one presented in Section 2.3.3. The sample and weight steps of the PF were mapped to multiple threads, while a standard systematic resampling algorithm was implemented using the parallel prefix-sum technique of Hendeby et al. [72]. The sampler was applied to two relatively simple SSMs with 5-6 unknown parameters and only 100 states. Also the number of particles in experiments did not exceed 512. The authors demonstrated that the sampler converges to the correct posterior using a total variation

distance metric and found the optimal number of particles that maximized MCMC convergence speed. Nevertheless, there were no details about the GPU device used, the absolute runtimes and the speedups compared to software. Therefore the efficiency of the implementation cannot be evaluated. Moreover, the limited number of states and particles is not representative of real, large-scale problems where hardware acceleration is needed.

A more recent work by Murray [2] (in 2015) proposed a high-level, automated tool for SSM inference on multi-core CPUs and GPUs, denoted Library for Bayesian Inference (LibBi). LibBi provides a domain-specific modelling language, in which the user can describe an SSM with known or unknown parameters. LibBi can be configured to run state estimation on the SSM (using a PF or a Kalman Filter) or perform inference (using pMCMC or SMC², a related method). Many parameters can be selected by the user (e.g. number of particles, frequency of resampling, compiler optimizations). Also, there are five resampling algorithms to choose from when the PF is in use (these algorithms were presented in more detail in [54]). LibBi makes use of a C++ backend and several parallel computing languages and tools like MPI, OpenMP, SSE, CUDA to improve performance. Nevertheless, the language used to define the SSM is not fully flexible in terms of the transition and observation densities that can be defined; some popular densities are not supported and certain types of computations are impossible to define. However, the tool is openly available and can be modified. The evaluation results presented in [2] were limited: Using a Lorenz '96 SSM model with 40 states and 8192 particles, the CPU-only accelerator (which used OpenMP and SSE), achieved a 5x speedup over sequential CPU code. The combined CPU and GPU version (which also used CUDA) provided an extra 4x speedup. An evaluation of the tool for larger SSMs with more particles and a comparison with the pMCMC accelerator proposed here can be found in Chapter 4.

In general, due to the fact that pMCMC has been invented only recently, most of the techniques used to accelerate PFs (described in the previous Section) have not yet been applied to pMCMC (which uses a PF in each iteration). Moreover, the speedups reported by [2] are an order of magnitude smaller than the speedups achieved by parallel implementations of popMCMC.

The work presented in Chapter 3 is the first to use FPGAs to accelerate pMCMC. It is also the first to use a modified version of Residual Systematic Resampling (introduced by Lie et al. [1]) in a PF and the first to scale the number of SSM states and the particle population of pMCMC to many thousands. A comparison of the pMCMC accelerators of Murray [2] with the FPGA-based samplers of Chapter 4 can be found in Section 4.7.

2.7.2 Choice of number of particles and use of tempering in Particle MCMC

As mentioned in Section 2.3.3, the main tuning parameter of the pMCMC algorithm is the number of particles of the PF. The larger the number of particles, the more time-consuming each pMCMC iterations becomes. At the same time, more particles mean that the likelihood estimate $\tilde{p}(\theta \mid \mathbf{Y}_{1:T})$ becomes more accurate, i.e. the variance of the estimate is smaller. This, in turn, leads to faster pMCMC mixing, since pMCMC moves closer to an exact MCMC algorithm. Therefore, there is a tradeoff between the runtime of the PF and the mixing of pMCMC when changing the number of particles. Recent literature [130, 131, 132] has proposed ways to optimize the number of particles based on this tradeoff. They have explored the links between the likelihood estimate variance, the proposal distribution variance, the mixing of pMCMC and the total runtime. Depending on the assumptions made and the mixing metrics used, they recommend that the number of particles and the proposal variance should be chosen so that the variance of the likelihood estimate is somewhere between 1.0 and 3.2, with a corresponding pMCMC acceptance rate of 7%-15%. These numbers lead to maximization of the *mixing per second* metric in pMCMC. The evaluation results in the above papers showed that when using more or fewer particles than the optimal number, mixing per second drops. These recommendations are based on analytical proofs. It has to be noted that the assumptions made in order to construct these proofs do not necessarily hold in all problems.

In spite of the practical value of these results in some scenarios, there are two issues that remain unaddressed: 1) All work mentioned above assumes that the runtime of the PF grows linearly with the number of particles. Nevertheless, this is not true in all cases (i.e. for all numbers of particles - P) when the PF is implemented on parallel platforms. 2) The use of multiple chains to improve pMCMC mixing has not been examined, although pMCMC mixes slowly for multi-modal posteriors, even with massive amounts of particles. Adding chains to the algorithm will add one more parameter to consider during the tuning process; instead of the optimal number of particles, the optimal combination of particles and chains has to be found.

Chapter 4 tackles these two issues by introducing a multi-chain pMCMC sampler based on the principles of popMCMC methods. The new algorithm is denoted ppMCMC and improves efficiency for multi-modal posteriors by using tempered chains (see Section 2.3.2). The algorithm is accompanied by a tailored FPGA architecture which aims at increasing the utilization of the PF datapath by pipelining chain computations. Moreover, a simulation-based design space exploration to jointly optimize

the number of chains and the number of particles is included. Also, this exploration is based on the throughput that a parallelized FPGA PF offers. This is in contrast to previous works [130, 132, 131]), where only P was optimized and the PF's runtime was assumed to grow proportionately to P (which is the case in sequential PF implementations).

It has to be noted that an earlier work [133], has also used a tempering technique to improve the efficiency of a Sequential Monte Carlo algorithm for joint inference on states and parameters in SSMs. The tempering was applied to the successive SSM observation densities, with temperatures increasing with the state index. This served to gradually transform the particle set to an approximation of the desired posterior, through a number of intermediate approximations (an idea very similar to the SMC² algorithm [134]). The tempering strategy used in Chapter 4 is clearly distinct from the work in [133], since it applies tempering to the final PF likelihood estimate and uses a set of PFs within an MCMC algorithm (and more specifically one PF for each of the parallel MCMC chains).

Finally, Chapter 4 contains the first application of pMCMC and FPGAs to a DNA methylation analysis problem [14]. So far, pMCMC was considered intractable for such problems and approximate methods were used [83].

2.7.3 Custom precision in MCMC and in other Monte Carlo algorithms

The only work that has attempted to investigate the effects that custom precision has on the theoretical properties of MCMC (without touching the issue of precision optimization for improving performance) is the work of Breyer et al. [135] in 2001. It was shown that even when using high precision, round-off can cause non-convergence and arbitrarily large perturbations in the stationary distribution of the Markov chain in certain innocuous-looking examples. However, the authors proved that, under certain conditions that usually apply, MCMC still converges to the right distribution when the round-off error is bounded. The findings of this work have limited practical use for deciding whether an employed precision is adequate or not, an issue that is investigated in Chapters 3 and 5 of this thesis. Moreover, results from the Markov chain perturbation literature [136] in 2005 can potentially be related to the issue of customizing precision. These results connect changes to the transition kernel of the chain to alterations in the state probabilities of its stationary distribution (measured using the total variation distance metric). Nevertheless, these results are practically applicable only to discrete state-spaces and the metrics used to measure the “error” in the stationary distribution do not have a

natural or easily interpretable meaning.

Apart from the above works which are based on probability theory and have limited practical applicability, no other study has tackled the issue of arithmetic precision in MCMC. Most importantly, no research exists on how to optimize precision with the aim of improving performance. This thesis is the first to do so, proposing custom precision techniques suitable for a particular MCMC method (see Chapter 3), as well as a generic precision optimization methodology applicable to any MCMC method (see Chapter 5). The former techniques approach the problem from the perspective of limiting precision to parts of the system that do not affect sampling quality, therefore leading to unbiased sampling. The latter approach reduces precision aggressively for all density computations and aims to find the minimum precision that satisfies a user-defined bias tolerance. It therefore leads to biased results.

For Monte Carlo methods other than MCMC, Chow et al. [3] in 2012 and Xiang and Bouganis [137] in 2011 have proposed precision optimization techniques based on performing FPGA runs in different precisions. Chow et al. [3] introduced a precision optimization methodology based on running a low precision and a mixed precision run. The second run was used to estimate and correct the bias of the first run. A workload balancing strategy between the host (which ran the high precision parts) and the FPGA (which ran the custom precision parts) was also described. In contrast, the work presented in Chapter 5 does not remove the bias from the result but guarantees (with some probability) that the bias will be within a user-defined tolerance. This allows for more aggressive accuracy/performance trade-offs. Moreover, Chow et al. [3] focused on simple (non-MCMC) Monte Carlo methods, where the generation of random numbers from $p(\theta)$ is relatively straightforward and the computational burden is due to the evaluation of $f(\theta)$ in (2.8). This means that the evaluation of $f(\theta)$ is the part of the system that is implemented in low precision, leading to bias in the output. On the contrary, Chapter 5 investigates MCMC, where the generation of random numbers from $p(\theta)$ is the bottleneck computation and $f(\theta)$ is typically easy to evaluate. Therefore, in this case the evaluation of $p(\theta)$ is the part of the system that is implemented in reduced precision and thus the source of the bias. This difference is critical, since it leads to a different form of bias estimator. In Xiang and Bouganis [137], an adaptive approach based on the Kolmogorov-Smirnov metric was introduced to optimize precision. The Cumulative Distribution Functions (CDFs) from a high and a low precision run were compared using the metric and the precision of the low precision run was adapted so that a threshold was not violated. Placing such a threshold is empirical and does not constrain the output error, in contrast to the approach presented in Chapter 5 which guarantees an error threshold with some probability.

2.7.4 Other related work

A number of other works are related to the research presented in this thesis. In [138], a modified version of the PT algorithm, similar to the MPPT algorithm of Chapter 3, was proposed; a Gaussian process approximation to the true distribution was used for some of the auxiliary chains of PT without altering the target distribution in the first chain. MPPT also uses approximations for the auxiliary chains but these approximations are reduced precision-based approximations. Moreover, Chapter 3 contains an extensive investigation on how to select the level of crudeness of the approximation (i.e. the precision) in order to maximize performance. Also, results on how performance scales with problem size and a theoretical proof of the correctness (i.e. unbiasedness) of the method are included. None of the above are provided in [138].

The work of Gramacy et al. [139] is related to the second custom precision method of Chapter 3, i.e. WPT. Gramacy et al. [139] showed how samples coming from the auxiliary chains of PT (which are distributed according to tempered versions of the target distribution) can be used to improve the variance of the output estimate. This was done by assigning weights to the auxiliary samples and employing an Importance Sampling approach to contribute to the final estimate. This idea can be useful when one wants to “correct” erroneous samples coming from any altered distribution; the WPT algorithm utilizes the Importance Sampling idea by assigning weights to the samples of the first chain. By doing this, it manages to correct sampling errors caused by using reduced precision in the FPGA implementation.

2.8 Summary

This chapter provided an introduction to the main concepts that are necessary to comprehend the rest of the thesis, highlighted the importance of accelerating MCMC to tackle large-scale problems, gave a brief introduction to current hardware platforms and presented a review of previous related work.

The literature review revealed that, although a significant amount of work on MCMC acceleration has been published during the last five years, this work has been limited in several ways: Research that uses FPGA only tackles the acceleration of the Bayesian likelihood computation (for a specific Bayesian model) and not the acceleration of the MCMC algorithm. Also, the special features of FPGAs (e.g. custom precision) have not been exploited. Even research that employs CPUs and GPUs to accelerate

MCMC is typically limited to direct mappings of algorithms to hardware. The idea of adapting the MCMC algorithm to make it more suitable for the underlying platform is unexplored.

The following chapters of this thesis focus on extending the existing MCMC acceleration research in various ways, so that the above limitations are tackled. In particular, they look at how FPGA technology can be exploited to accelerate classes of MCMC methods for which only CPU and GPU implementations currently exist. Novel algorithms, which are more suitable for FPGA mapping, are also proposed. Finally, the following chapters explore various ways in which custom precision can be used to accelerate MCMC.

Chapter 3

Algorithms and architectures for Population-based MCMC

3.1 Introduction

A common form of complexity in Bayesian posterior distributions is multi-modality, i.e. the existence of two or more separate modes in the probability density. Multi-modal distributions appear in many Bayesian inference application, e.g. machine learning using Restricted Boltzmann Machines or mixture models [22, 40, 7], computational genetics [13, 25] and biological simulations [27]. They cause baseline MCMC samplers (e.g. Metropolis sampler [30]) to get stuck in one of the modes of the distribution for a long time, thus making them inefficient.

Population-based MCMC (popMCMC) [7] is a class of methods specifically designed to address multi-modality in the target distribution. Parallel Tempering (PT) [41] is the most popular of these methods (see Section 2.3.2 and [41]). This chapter proposes ways to tackle the computational challenges of PT, e.g. the processing burden of running multiple MCMC chains instead of the one chain used by basic MCMC methods. The chapter focuses on combining hardware acceleration (using FPGAs but also CPUs and GPUs) with novel algorithmic modifications based on the use of custom arithmetic precision. Both the characteristics of the FPGA architecture and the structure of PT are exploited to accelerate inference. The main questions that this chapter seeks to answer are the following:

- “How can PT be parallelized in multi-core CPU, GPU and FPGA implementations and what are

the gains?”

- “Is there a way to reduce the arithmetic precision in large parts of the algorithm without affecting sampling accuracy?”
- “What extra speedup does such a strategy deliver in each platform?”

The results of this chapter demonstrate that significant speedups are possible when parallelizing PT and that smart modifications to the algorithm permit the reduction of precision in the majority of PT computations without any cost in sampling accuracy. This reduction translates to significant area savings and throughput improvement in FPGA designs.

Chapter outline

Section 3.2 repeats basic background information on the PT algorithm for easier reference (PT has already been presented in Chapter 2). It also describes the available forms of parallelism in the algorithm. The remaining sections contain the main contributions of the chapter, which are the following:

1. An optimized FPGA accelerator for PT, which employs double precision and delivers a speedup of up to 174x over sequential code running on a single-core CPU. Highly optimized implementations of PT on a multi-core CPU and a GPU are also proposed, delivering up to 16.1x and 165x speedup compared to sequential code respectively. Each implementation takes advantage of specific features of the respective hardware platform in order to maximize sampling throughput (Section 3.3.1).
2. Two novel, custom precision methods (i.e. algorithmic modifications) for PT, which allow the use of reduced precision in parts of the algorithm and thus lead to reduced runtimes (Section 3.3.2). Both methods guarantee that the use of reduced precision does not affect sampling quality, i.e. does not introduce error in the Monte Carlo estimate of Equation (2.8). Instead, by reducing precision, the mixing of the PT algorithm is affected, allowing for a trade-off between raw speedup and mixing. The first method uses a weighting scheme to correct errors (Weighted PT - WPT). The second method uses custom precision in parts of the algorithm which do not affect output accuracy (Mixed-Precision PT - MPPT). A theoretical proof that MPPT maintains the detailed balance condition (which is necessary to guarantee convergence to the correct target distribution) is presented. WPT is guaranteed to converge to the correct target distribution

because it is essentially an Importance Sampling (IS) method. All necessary conditions for IS to sample from the correct distribution apply in the WPT case by design.

3. Two tailored architectures which map the two custom precision methods to an FPGA, taking advantage of reduced precision to improve performance (Section 3.3.3). These accelerators offer further speedups of up to 6.5x over the baseline (double precision) FPGA accelerator. The two custom precision methods are also mapped to a CPU and a GPU, delivering speedups of up to 1.4x and 3.2x respectively over the double precision samplers.
4. A precision optimization process for WPT and MPPT on FPGAs, which is able to find the precision configuration which maximizes effective sampling throughput, defined later in the text (Section 3.6.4). The optimization process takes advantage of the trade-off between speedup and mixing to deliver maximum effective performance.

The performance of the various accelerators is evaluated using a case study representative of the types of problems that PT is applied to: Bayesian inference on a mixture model [40, 16]. This case study leads to a multi-modal posterior. An investigation of the way the performance of the accelerators scales with the size of the chain population, the size of the data set and the size of the hardware device is also presented. Finally, results on the power efficiency of each accelerator are included (Section 3.6.3).

3.2 Parallel Tempering

The PT pseudocode, which was initially presented in Chapter 2 (Algorithm 3), is shown in Algorithm 6. A detailed description of the algorithm can be found in Section 2.3.2. In this chapter, the temperature of chain $j \in \{1, \dots, M\}$ (M is the size of the chain population) is set to $Temp_j = (\frac{M}{M+1-j})^2$, following the recommendation of Lee et al. [16]. This setting is suitable for the target distribution which is presented in Section 3.4 and used as a case study in this chapter. For other case studies, a different tempering setting might lead to more efficient sampling (i.e. higher mixing rate) but this investigation is outside the scope of this chapter. The parameters of the PT algorithm, along with the values they take in the evaluation section are listed in Table 3.1. It has to be noted that the symbol $p(\theta)$ is used throughout the chapter to describe the target distribution. This is equivalent to the posterior

Algorithm 6 Parallel Tempering

```

1: procedure PT( $N, M, Temp_{1:M}, \theta_{1:M}^{(1)}, \sigma_{1:M}^2$ ) - Inputs:  $N$  (number of MCMC samples),  $M$  (number
   of chains),  $Temp_{1:M}$  (temperatures for all chains),  $\theta_{1:M}^{(1)}$  (initial MCMC samples for all chains),
    $\sigma_{1:M}^2$  (variances of proposal densities for all chains)
2:   for  $i = 2, \dots, N$  do
3:     for  $j = 1, \dots, M$  do // Global update
4:        $\theta^* \sim q(\theta^* | \theta_j^{(i-1)}) = N(\theta^* | \theta_j^{(i-1)}, \sigma_j^2)$ 
5:       Evaluate acceptance ratio  $a \leftarrow \frac{p_j(\theta^*)}{p_j(\theta_j^{(i-1)})}$ 
6:       Generate uniform random number  $u \sim U[0, 1)$ 
7:       if  $a \geq u$  then
8:         Accept proposed sample:  $\theta_j^{(i)} \leftarrow \theta^*$ 
9:       else
10:        Reject proposed sample and replicate previous sample:  $\theta_j^{(i)} \leftarrow \theta_j^{(i-1)}$ 
11:    Choose even chain pairs  $((1, 2), (3, 4), \dots)$  or odd chain pairs  $((2, 3), (4, 5), \dots)$  (in turn)
12:    for all chosen chain pairs  $(q, r)$  do // Global exchange
13:      Evaluate exchange acceptance ratio  $e(\theta_q^{(i)}, \theta_r^{(i)}) \leftarrow \min\left(1, \frac{p_q(\theta_r^{(i)})p_r(\theta_q^{(i)})}{p_q(\theta_q^{(i)})p_r(\theta_r^{(i)})}\right)$ 
14:      Generate uniform random number  $u \sim U[0, 1)$ 
15:      if  $e(\theta_q^{(i)}, \theta_r^{(i)}) \geq u$  then
16:        Exchange the samples  $\theta_q^{(i)}$  and  $\theta_r^{(i)}$ 
17:      else
18:        Do not exchange any samples
19: return  $\theta_1^{(1:N)}$  ( $N$  MCMC samples from the first chain)

```

$p(\theta | \mathbf{D})$ in Bayesian inference problems, as described in Chapter 2 (also see the posteriors in Section 3.4).

Parallelism in the algorithm

There are two forms of parallelism in PT:

1. Inter-chain parallelism during Global updates (loop in line 3 of Algorithm 6): All chains can be updated in parallel since they do not interact during this stage. More specifically, the computationally intensive evaluations of the probability densities of proposed samples ($p_j(\theta^*)$ in line 5) can be done in parallel for all chains. Global exchanges can also be parallelized since pairs of chains communicate but each exchange is independent. Unfortunately, using more than 100-200 chains does not offer any improvement in mixing [16, 27] and thus using many thousands of chains is not desirable, despite the massive amount of parallelism that this can offer. Therefore it is crucial for a PT accelerator to achieve high performance even for moderate numbers of chains.
2. Intra-chain parallelism during the computation of the probability density of a proposed sample

Table 3.1: PT algorithm parameters

Symbol	Description	Range (in Section 3.6)
M	Number of PT chains	$[8, 32768]$
$Temp_{1:M}$	PT temperature set	$(\frac{M}{M+1-j})^2$ for chain j
N	Number of generated MCMC samples (from each chain)	100000
B	Number of burn-in samples	5000
$\theta_{1:M}^{(1)}$	Initial MCMC samples for each chain	Generated uniformly in the space $[0,10]$
$\sigma_{1:M}^2$	Proposal distribution variances for each chain	
s	Dimension of MCMC samples (i.e. parameter θ)	4

(line 5 in Algorithm 6). This parallelism is density-dependent. Due to the large diversity of models and corresponding densities, it is impossible to investigate the effect of parallelising this part in the general case. Nevertheless, it is common in Bayesian problems to use independent and identically distributed (i.i.d.) data. This leads to a density which is equal to the product of the sub-densities of all the data (or the sum if log-densities are used). This chapter only considers this form of density parallelism (an example of which is presented in Section 3.4), because: 1) It is representative of many applications (e.g. [66, 18, 16]), 2) Several MCMC algorithms are designed to address this specific form of likelihood [62], 3) The kinds of computations involved in most i.i.d. problems (including the case study of Section 3.4), such as reductions, are very common even in non-i.i.d. problems. The above three points demonstrate that the results presented in this chapter on performance and performance scaling can be generalized to many other models and algorithms.

3.3 Accelerating PT

This section contains the main contributions of the chapter: Ways in which PT can be accelerated using 1) parallel hardware and 2) custom precision methods (i.e. algorithmic modifications). The PT code (Algorithm 6) was first implemented in C++ using double precision arithmetic, without exploiting parallelism and without applying any compiler optimizations. This sequential implementation is used as reference. An identical sequential implementation but with all compiler optimization activated is also presented in the results section for reasons of completeness. Section 3.3.1 describes in detail the baseline FPGA accelerator (which uses double precision), as well as two highly optimized implementations of PT on a multi-core CPU and a GPU. Particular focus is given on the FPGA architecture.

Section 3.3.2 proposes two custom precision methods to improve PT's efficiency. Section 3.3.3 maps these methods first to the FPGA (which is the most suitable device due to its fully customizable precision) and then to the CPU and GPU (where the only choice of reduced precision is single floating point precision). In each platform, both forms of parallelism mentioned in Section 3.2 are exploited.

3.3.1 Baseline accelerators

FPGA

Here, a baseline hardware architecture for PT in double floating point precision is proposed. Figure 3.1 shows the block diagram of the architecture. There are four computational blocks (Sample Proposal, Probability Evaluation, Accept/Reject and Exchange) and three memories (Sample, Probability and Data). There is also one Gaussian and two uniform random number generators and four FIFO registers. Most control signals are omitted for clarity.

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

One MCMC iteration (outer loop in line 2 of Algorithm 6) includes the following (for iteration number i): The current samples of all parallel chains ($\theta_j^{(i-1)}$ for all $j \in \{1, \dots, M\}$) are read from Sample memory and forwarded to the Sample Proposal block. The Sample Proposal block proposes candidate samples θ^* by adding Gaussian random numbers to the current samples. If the sample's dimension is larger than one, the block proposes values for all dimensions in parallel (aided by parallel Gaussian RNGs). The candidates are then moved to the Probability Evaluation block (shown in Figure 3.2), which computes the candidate probabilities $p_j(\theta^*)$. This is done by calculating n probability sub-densities (n is the number of data), finding their logarithms and summing them to get the total log-density. For big n , the Probability Evaluation block becomes the bottleneck of the system. To reduce this bottleneck, multiple parallel pipelines can be instantiated inside this block. The number of pipelines is limited only by the FPGA chip size. Moreover, since each sub-density requires that a datum be read from the Data memory, the Data memory is designed to provide data to all pipelines in parallel at the same cycle (each memory address stores multiple data). The sizes of all system memories are shown in Table 3.2. An adder tree performs the reduction. The Accept/Reject block receives

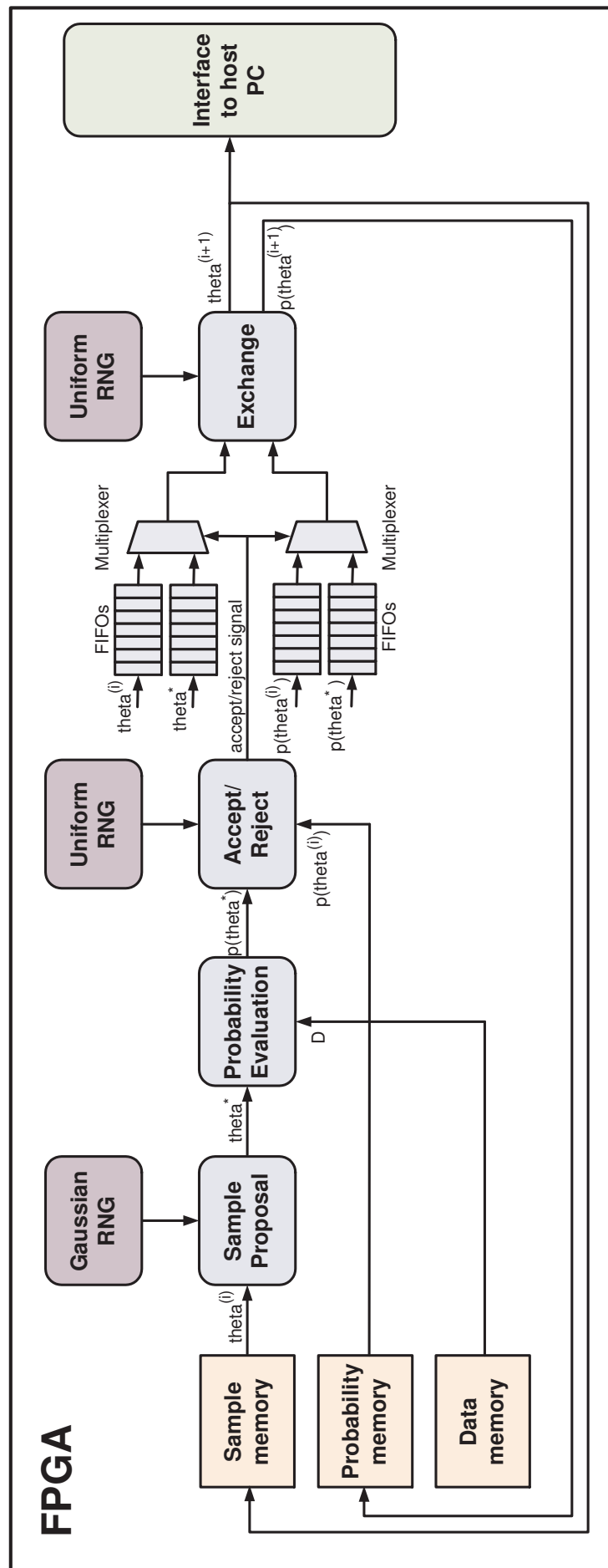


Figure 3.1: Baseline FPGA architecture

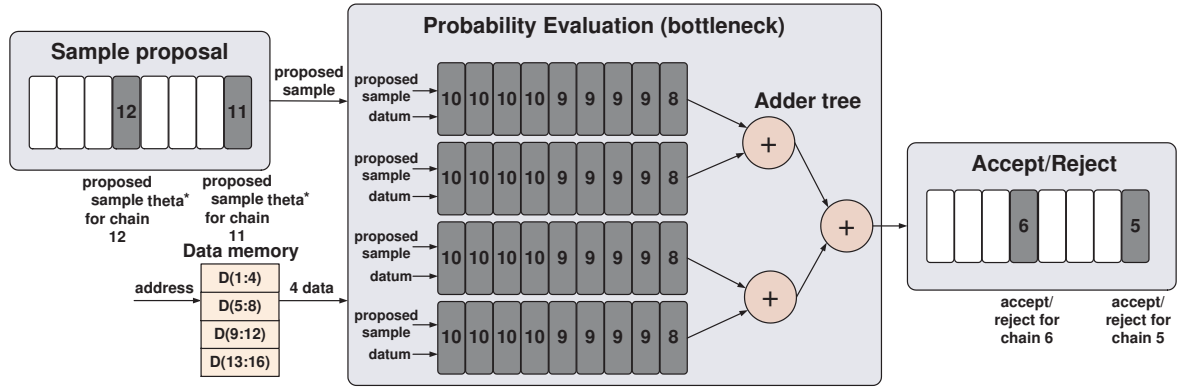


Figure 3.2: Chain streaming through the Sample Proposal, Probability Evaluation and Update pipelines. Occupied stages are grey, unoccupied white. Numbers represent the chains that occupy each stage. There are four pipelines in the Probability Evaluation block ($P = 4$). The data size is $n = 16$. A probability value is generated every four cycles ($\lceil \frac{n}{P} \rceil = 4$). The Sample proposal and Update pipelines are under-utilized.

Table 3.2: Baseline architecture memories. P is the number of sub-density pipelines, M , n and s are defined in Table 3.1

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

the candidate probabilities $p_j(\theta^*)$ and reads the previous probabilities of each chain ($p_j(\theta_j^{(i-1)})$) from Probability memory. It also computes the temperature $Temp_j = (\frac{M}{M+1-j})^2$. All these values (along with a uniform random number) are used to find the Metropolis ratio and accept or reject each candidate sample.

The above steps comprise the update operation. The updated samples also pass through the Exchange block before they are written back to Sample memory. Unlike CPU and GPU implementations (which are presented in the following sections), the Global update (update of all chains) does not need to finish before starting the Global exchange. As soon as a chain is updated, it is forwarded to the Exchange block while the next chains are processed by the Update block.

Each exchange is performed between a pair of chains. Therefore, the block has to wait for two chains to be updated and then attempt the exchange. Because exchanges are performed between neighbouring chains only (see lines 11-12 in Algorithm 6), pairs of potentially exchanged samples conveniently reach the block successively (since chains are updated in order). The temperatures and the updated sample probabilities are used to accept or reject the exchange. FIFOs are used to store these values

when they first become available (earlier in the pipeline), removing the need to write them to the memories after updates and read them back for exchanges (which is necessary in CPU and GPU implementations [15, 16]).

Finally, the new samples and probabilities are written back to memories. When all chains have traversed the pipeline, the Global update and exchange are complete and the next MCMC iteration starts. At every iteration, the current samples and probabilities of the first chain are read from the Sample memory and sent to a BRAM-based buffer, which is able to transmit the data to the host PC in real time (using double buffering). Details on the implementation of the buffer are given in Section 3.5.

Performance model

This section gives exact formulas for the latency and throughput of the various blocks in the PT architecture. The latency of the Sample Proposal block (for generating a candidate sample for one chain) is:

$$Lat_{sp} = Lat_{add} \quad (3.1)$$

where Lat_{add} is the latency of a double floating point adder (needed to add a Gaussian random number to the previous sample of the chain). All dimensions of the proposed sample are generated in parallel.

The latency of the Probability Evaluation block (to compute the probability density of one chain) is:

$$Lat_{pe} = C_{subdensity} + \left\lceil \frac{n}{P} \right\rceil \quad (3.2)$$

where $C_{subdensity}$ is the latency of a single sub-density evaluation pipeline (depends on the target distribution) and the term $\left\lceil \frac{n}{P} \right\rceil$ is the latency for passing all the n data through the P parallel pipelines. Each pipeline can receive one data input per cycle (has a throughput of 1 data per cycle). It is assumed that $n > P$, which is the case for all non-trivial data sets.

The latency of the Accept/Reject block (for accepting or rejecting a candidate for one chain) is given by the following equations:

$$Lat_{ar} = Lat_{mult} + Lat_{sub} + Lat_{comp} \quad (3.3)$$

where Lat_{mult} is the latency of a double floating point multiplier needed to multiply the proposed and previous log-densities with the temperature of the chain. Two multipliers are needed and they work in parallel. The results are the numerator and denominator of the acceptance ratio in line 6 of Algorithm

6 (using log-values). Lat_{sub} is the latency of a double floating point subtracter (needed to subtract the numerator and denominator in order to compute the acceptance ratio) and Lat_{comp} is the latency of a double floating point comparator (needed to compare the ratio with the logarithm of a uniform random number).

The latency of the Exchange block (for accepting or rejecting a candidate for one chain) is given by the following equations:

$$Lat_{ex} = Lat_{mult} + Lat_{add} + Lat_{sub} + Lat_{comp} \quad (3.4)$$

Four parallel multipliers are needed to find the four terms in the exchange ratio of line 13 in Algorithm 6 (using log-values). Two parallel adders compute the numerator and denominator of the exchange ratio and a subtracter is needed to compute the value of the ratio. The comparator compares the value with the logarithm of a uniform random number.

The above four blocks constitute the system pipeline. By making use of chain pipelining, as described in the previous paragraphs, it is possible to feed one new chain to the pipeline every $\lceil \frac{n}{P} \rceil$ clock cycles (since this is the number of cycles for which the Probability Evaluation block is busy processing each chain). The total latency of the baseline architecture's pipeline (the number of cycles necessary to process one Global Update and one Global exchange, i.e. one PT iteration in the loop of line 2 in Algorithm 6) is the following:

$$Lat_{iter-pt} = Lat_{sp} + C_{subdensity} + M \cdot \lceil \frac{n}{P} \rceil + Lat_{up} + Lat_{ex} \quad (3.5)$$

where the term $M \cdot \lceil \frac{n}{P} \rceil$ is the number of cycles needed to pass all chains through the system pipeline. The latencies of the update and exchange blocks are counted only once because they overlap with the Probability Evaluations (which take significantly more cycles for realistic scenarios where n is large). Figure 3.2 demonstrates this point more clearly; it shows the utilization of the Sample Proposal, Accept/Reject and Probability Evaluation pipelines by PT chains when $n = 16$, $P = 4$ and $\lceil \frac{n}{P} \rceil = 4$. A sample reaches the block every four cycles. At the same time, $n = 16$ data are sent to the block (one quadruple per cycle). The Data memory is designed to “match” the consumption rate of the block, as discussed previously.

The latency of the whole PT algorithm is the following:

$$Lat_{pt} = N \cdot Lat_{iter-pt} \quad (3.6)$$

and the total time of the system is:

$$Time_{total-pt} = \frac{Lat_{pt}}{freq} + Time_{input-pt} \quad (3.7)$$

where $freq$ is the clock frequency of the PT IP in Hz and $Time_{input-pt}$ is the time needed to send input arguments from the host PC to the FPGA. No time is spent for outputting the MCMC samples, since this happens simultaneously with processing (using a double buffering memory architecture).

The throughput of the baseline architecture (MCMC iterations it processes per second, where an MCMC iteration comprises a Global update and a Global exchange), excluding $Time_{input-pt}$, is:

$$TP_{pt} = \frac{freq}{Lat_{iter-pt}} \quad (MCMC \text{ iterations} / sec) \quad (3.8)$$

This throughput is equal to the throughput of the Probability Evaluation block. It is clear that the critical factor for the performance of the system is $\lceil \frac{n}{P} \rceil$, the cycles the block needs in order to process each chain. By fitting more parallel sub-density pipelines in the FPGA fabric, P can be increased, resulting in higher throughput.

Multi-core CPU

An optimized implementation of PT was implemented on a multi-core CPU in order to achieve a fair performance comparison. In order to exploit PT's parallelism, pragmas and Intel Cilk keywords [140] were embedded in the sequential C++ code. Also, Intel Compiler optimizations [141] (including the -O3 flag and the optimizations related to the CPU architecture) were applied. More specifically: 1) The Global update loop was transformed into a **cilk_for** loop and the granularity of the parallelization was optimized by ordering the compiler to group loop iterations into groups of a certain number of iterations each (using the **granularity** pragma). Depending on the number and type of CPU cores and the amount of work per iteration, a specific granularity maximizes performance. 2) The reduction operation (necessary to sum the sub-densities and evaluate the total probability density) were parallelized using the **simd reduction** pragma. A parameter is also used here to specify the granularity of paral-

lization [140]. 3) Sub-density evaluations were vectorized by converting the respective functions to Cilk elemental (vectorized) functions, using the `__attribute__((vector))` keyword.

GPU

An optimized GPU implementation was also created based on the state-of-the-art CUDA code of Lee et al. [16]. In Lee et al. [16] the main computational work of the implementation is split into two kernels, the global update and the global exchange kernels. There are also kernels for random number generation and initialization. All of the remaining work is done on the CPU. The global update kernel updates all chains once. It exploits chain parallelism, assigning the work of every PT chain to a separate thread. This results in an implementation which does not exploit all available parallelism; it ignores intra-chain parallelism. The exchange kernel performs exchanges between neighboring chains and these are also parallelized.

Here, a PT implementation which uses an enhanced global update kernel is presented. All the remaining components of the implementation of Lee et al. [16] remain the same (for more details on these components see [16]). The changes to the global update kernel aim at increasing thread utilization and maximizing performance. They are listed below:

- 1) Intra-chain computations are parallelized by assigning the calculation of the sub-densities (or groups of them) of each chain to separate threads, in contrast to Lee et al. [16] where all sub-densities of a chain were assigned to the same thread. This makes the comparison to other platforms fair.
- 2) The global update kernel processes M chains, each of which contains n sub-density evaluations. These Mn tasks can be allocated to CUDA blocks and threads in many combinations. The number of blocks ranges from 1 to M ; within each block, 1 to n tasks are allocated to each thread. Combinations which allocate the work of a chain into separate blocks are not examined because this would require communication between different blocks during chain updates (which is expensive since the Global GPU memory needs to be used instead of the Shared memory). For each (M, n) setting, the combination of blocks and tasks per thread which maximizes the kernel's throughput is chosen. For example, when few PT chains are used, there is not enough parallelism in the inter-chain level. It is then beneficial to assign each sub-density (task) of each chain to a separate thread to introduce as much intra-chain parallelism as possible. On the other hand, when the number of chains reaches a few thousands, it is preferable to assign more one sub-density task to each thread, because 1) there is

now enough inter-chain parallelism to saturate the device, 2) inter-chain parallelism does not require a reduction, unlike intra-chain parallelism. The above optimization is described in detail in Section 3.6.4 and leads to increased GPU utilization compared to Lee et al. [16] (also considering that much larger data sets are used compared to [16]).

3) Reduction operations inside the density computation of each chain are unrolled using the technique proposed in [142]. After the independent sub-densities are computed by the threads, they need to be summed to get the likelihood. This requires communication between threads through the shared memory. Although this reduction can be easily done using a reduction tree, this forces half of the threads to be inactive in the first tree stage, 75% of the threads to be inactive in the second stage, etc. In the proposed implementation, the technique proposed in [142], which completely unrolls the reduction calculations and minimizes thread imbalance is applied.

4) The implementation of Lee et al. [16] stores all the data in the GPU's constant memory (typically limited to a few dozens of KBs). This is possible because the data sizes used are small (100 data point, 4 bytes each). Here, data are stored in global GPU memory, which is a realistic strategy given the data sizes in real applications. During execution, data are moved, in chunks, to the shared memory of all blocks. All the chains of a block can use the data to compute part of the log-density before the next chunk is read, increasing the compute-to-memory ratio of the kernel by a factor equal to the number of chains per block (ranging from 2 to 32).

3.3.2 Custom precision methods for PT

Custom arithmetic precision in MCMC

The baseline accelerators of the previous section use double precision throughout the system. Double and (to a lesser extent) single precision are the configurations used in the overwhelming majority of MCMC implementations in the literature, since these precisions are considered enough for real problems. Double precision was chosen as the baseline precision in this chapter because it is the highest (closest to infinite) precision that is supported by CPU and GPU architectures.

In this section, reduced precision is used in certain parts of PT to lower the area utilization of arithmetic operators in FPGAs (and thus implement more parallel operators or modules). The same technique can be used in CPUs and GPUs to achieve lower runtimes but the lowest floating point precision that

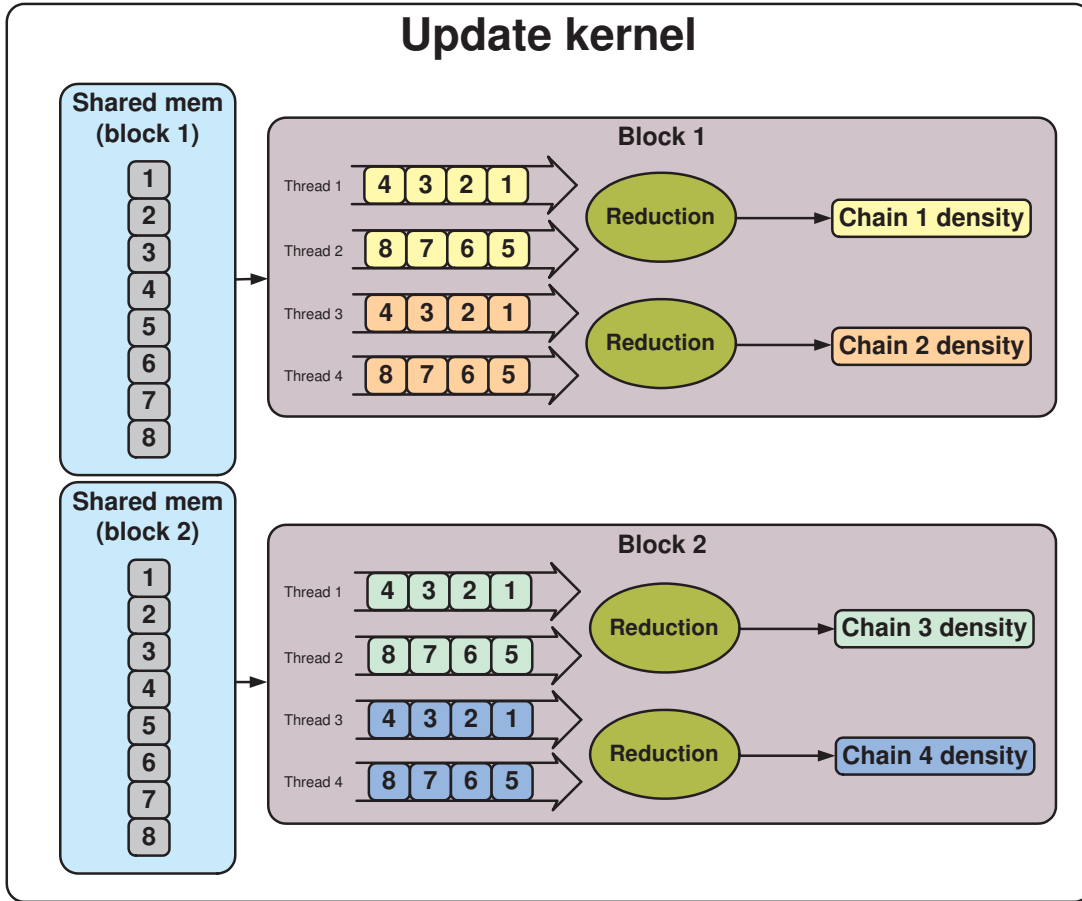


Figure 3.3: The GPU update kernel. Here, each block within the kernel handles two chains and four threads run in parallel inside each block, i.e. two threads per chain. The number of data is $n = 8$, which means that four data (i.e. four sub-densities or four tasks) are assigned to each thread. The threads which operate on the same chain pass through a reduction stage in order to give the total density of the chain. The shared memory of each block stores all the data. In a case where the shared memory is not large enough to store all data, the data are moved and processed in chunks.

these platforms support inherently is single precision, which means the expected runtime gains are smaller. Other precisions in CPUs and GPUs can only be simulated which is much slower than using one of the supported configurations. It is worth noting that this is about to change, since future Nvidia GPUs will inherently support half precision (16-bit) floating point [96]. Although reduced precision results in larger arithmetic errors in calculations, it is shown here that these errors can be avoided or corrected in a PT setting. Precision is customized by changing the number of mantissa bits only. Precision configurations are described by the pair (*mantissa bits*, *exponent bits*).

In both custom-precision methods presented here, reduced precision is used only when evaluating the probability density. All other parts of PT work in double precision. This is equivalent to Combination D/C in Table 2.1 of Chapter 2. The reason for this is twofold: 1) Altering the precision of all MCMC steps can result in unexpected behaviour (non-convergence or convergence to an unknown distribution)

[135]. In contrast, by using custom precision only when computing $p(\theta)$, the properties of MCMC are preserved, i.e. MCMC converges to an altered but known distribution (a custom-precision approximation of $p(\theta)$). Here, it is shown that even this approximation can be avoided using the proposed custom precision schemes. More comments on the use of custom precision in MCMC can be found in Chapter 2. 2) The second reason for using custom precision only for probability evaluations is that this computation takes the large majority of runtime (or hardware resources) for all accelerators. The latter point is confirmed by the resource utilization results in Section 3.6.2.

Weighted PT method

When the probability evaluation is implemented in custom precision, PT samples from an approximation of $p(\theta)$. The approximate density is denoted $\tilde{p}(\theta)$. Examples of such approximate densities have already been given in Figure 2.10 of Chapter 2 (custom precision densities in that figure were denoted $p_c(\theta)$). When samples from $\tilde{p}(\theta)$ are used to estimate (2.8), the estimate will be biased, as was explained in Chapter 2.

To avoid this bias, a novel method, called Weighted PT (WPT), is proposed. The main idea of WPT is to use custom precision in the density evaluation of all chains and assign importance weights to the samples generated by the first chain to correct the first chain's bias (or rather the bias in the output estimate (2.9)). This transforms the algorithm into an Importance Sampling (IS) method [6].

More specifically, the density of chain j in baseline PT is $p_j(\theta) = p(\theta)^{1/Temp_j}$. In contrast, the density of chain j in WPT is:

$$\tilde{p}_j(\theta) = \tilde{p}(\theta)^{1/Temp_j} \quad (3.9)$$

For each chain, WPT computes $p(\theta)$ in custom precision (i.e. it computes $\tilde{p}(\theta)$) and then applies the temperature $Temp_j$ in double precision. All other PT operations are also performed in double precision. Weights are assigned to the samples of the first chain based on the fact that the desired target density in this chain is $p_1(\theta)$ ($= p(\theta)$), while samples are actually taken from $\tilde{p}_1(\theta)$ ($= \tilde{p}(\theta)$), the custom-precision version of $p_1(\theta)$ (no tempering is applied to the first chain since $T_1 = 1$). Thus $\tilde{p}_1(\theta)$ functions as the importance sampling distribution in IS; each sample $\mathbf{x}_1^{(i)}$ from the first chain at time i , where $i \in \{1, \dots, N\}$, is assigned the weight:

$$w_i = \frac{p_1(\theta_1^{(i)})}{\tilde{p}_1(\theta_1^{(i)})} \quad (3.10)$$

The integral (2.8) is then estimated by the IS sum:

$$\tilde{E}_{p-IS}[f(\theta)] = \frac{1}{N} \sum_{i=1}^N w_i f(\theta_1^{(i)}) \quad (3.11)$$

instead of the sum in equation (2.9). The PT steps remain the same as in Algorithm 6, with the only differences being the use of $\tilde{p}(\theta)$ instead of $p(\theta)$ everywhere and the computation of the weights.

Generally, the density $\tilde{p}_1(\theta)$ is a close approximation to $p_1(\theta)$ and therefore it can be considered a good (efficient) importance sampling distribution according to the empirical rules of IS [6]. This efficiency drops only for very low precisions (see Section 3.6). IS also requires that the importance sampling distribution has a wider support than the target distribution [6]. To guarantee this, the computation of $\tilde{p}_1(\theta)$ is saturated, i.e. zero values are transformed to the minimum value representable by the employed precision configuration. This guarantees that the support of $\tilde{p}_1(\theta)$ is wider than that of $p_1(\theta)$. Once meeting this requirement, the samples and weights generated by WPT can be used to estimate (2.8) with zero bias compared to the double precision (baseline) sampler.

The use of weights in population-based MCMC has been previously proposed by Gramacy et al. [139] as a way to use the auxiliary chains' samples to improve the variance of (2.8). Here, the weights are used for a different purpose (to correct precision-related bias) and only for the first chain.

Mixed-precision PT method

PT has the distinctive property of running many Markov chains but keeping samples from the first chain only. Therefore, only the first chain's target distribution affects the output estimate of Equation (2.9). Auxiliary chains only contribute to the acceleration of the first chain's mixing and not to the output estimate. This means that they can sample from any distribution with the only possible effect being a change in the mixing speed.

The second custom precision method proposed in this chapter is called Mixed-Precision PT (MPPT) and it exploits the above fact to increase PT's performance. It uses double precision only when updating the first PT chain, which means that the first chain samples from the "correct" distribution $p_1(\theta) = p(\theta)$. The auxiliary chains (indexes $j \in \{2, \dots, m\}$) sample from tempered versions of the

custom precision approximation $\tilde{p}(\theta)$:

$$\tilde{p}_j(\theta) = \tilde{p}(\theta)^{1/Temp_j} \quad (3.12)$$

These densities are the same as the ones used for WPT (equation (3.9)). In order to guarantee the correctness of the first chain's target distribution, the way the algorithm exchanges samples is also modified. Exchanges between chains $q > 1$ and $r > 1$ at MCMC iteration i are accepted with probability:

$$e(\theta_q^{(i)}, \theta_r^{(i)}) = \min\left(1, \frac{\tilde{p}_q(\theta_r^{(i)})\tilde{p}_r(\theta_q^{(i)})}{\tilde{p}_q(\theta_q^{(i)})\tilde{p}_r(\theta_r^{(i)})}\right) \quad (3.13)$$

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

$$e(\theta_1^{(i)}, \theta_r^{(i)}) = \min\left(1, \frac{p_1(\theta_r^{(i)})\tilde{p}_r(\theta_1^{(i)})}{p_1(\theta_1^{(i)})\tilde{p}_r(\theta_r^{(i)})}\right) \quad (3.14)$$

Notice the use of both double and custom precision densities in Equation (3.14). The above equations replace the one in line 13 of Algorithm 6. By altering the exchange operations which include the first chain, the detailed balance condition [6] is guaranteed to hold for all exchange moves. This is necessary to guarantee that the distribution of the first chain remains the same as in double precision samplers. Thus MPPT induces no loss in sampling accuracy. A proof that detailed balance holds in MPPT is presented in the following section.

In Fielding et al. [138], Gaussian process approximations of the distributions of some PT chains were used, while retaining the target distribution of the first chain. Similarly, MPPT uses an approximation of the distributions of auxiliary chains but the approximation is a custom precision-based approximation rather than a probabilistic approximation. Moreover, in contrast to the present chapter, Fielding et al. [138] do not investigate the effect of the approximation on mixing.

The price for not using double precision for auxiliary chains is that mixing can be negatively affected. This is intuitively expected, since sample exchanges between the first and second chains are sometimes less likely to succeed (this negative effect is demonstrated in the trade-off between precision and mixing in Section 3.6.4). Nevertheless, there is no reason why custom precision approximations for auxiliary chains could not prove more efficient than their double precision counterparts in some scenarios.

Detailed balance proof for MPPT

In this section, a proof that the detailed balance condition is maintained by the MPPT algorithm is presented. For detailed balance to hold, the following must hold for all update and exchange moves:

$$\begin{aligned} p_{full}(\theta_{full}) \times K(\theta'_{full} \leftarrow \theta_{full}) &= \\ p_{full}(\theta'_{full}) \times K(\theta_{full} \leftarrow \theta'_{full}) \end{aligned} \quad (3.15)$$

where $p_{full} = p_1 \prod_{j=2}^M \tilde{p}_j$ is the target distribution of the whole chain population, $\theta_{full} = \theta_{1:M}$ is the state of the whole chain population, θ'_{full} is a different state and $K(\cdot \leftarrow \cdot)$ is the transition kernel of PT, which gives the probability to move from any state to any other state during an update or exchange. Note that working with the joint states and target distributions of PT is equivalent to working with each state and target distribution separately (which was done in the previous sections and in Chapter 2).

Updates: When updating the chains, a standard Metropolis move is used for each chain. Because they are Metropolis moves, they maintain detailed balance for each chain and therefore for the ensemble p_{full} . This is true both for double and for custom precision.

Exchanges: When exchanging samples between any pair of chains in the range $j = 2$ to $j = M$ (auxiliary chains), the exchange move of line 13 in Algorithm 6 is used. This is the standard exchange move of PT but using the custom precision densities $\tilde{p}_j$ instead of the double precision densities p_j . This move maintains detailed balance for p_{full} (proved in the same way as in double precision PT, replacing p_j with $\tilde{p}_j$).

The challenge in proving that detailed balance holds for MPPT lies in showing that it holds for exchanges between the first and the second chains because both the double and custom precision densities are used in the exchange ratio. In more detail, Equation (3.14) is used. In this case, the transition kernel K is equivalent to the probability of proposing the exchange (given the present state), i.e. exchanging samples θ_1 and θ_2 , multiplied by the exchange acceptance probability of Equation (3.14). The symbol $Q(\cdot \leftarrow \cdot)$ is used to refer to the proposal probability. For clarity, the full form of the acceptance ratio is used (which includes the densities of all chains and Q), in contrast to the simplified form of Equation (3.14). Starting from the left-hand side of (3.15), the proof that the exchange move between chains 1 and 2 maintains detailed balance is the following:

$$\begin{aligned}
& p_{full}(\theta_{full}) \times K(\theta'_{full} \leftarrow \theta_{full}) \\
&= p_1(\theta_1) \tilde{p}_2(\theta_2) \prod_{j=3}^M \tilde{p}_j(\theta_j) \times K([\theta_2, \theta_1, \theta_{3:M}] \leftarrow [\theta_1, \theta_2, \theta_{3:M}]) \\
&= p_1(\theta_1) \tilde{p}_2(\theta_2) \prod_{j=3}^M \tilde{p}_j(\theta_j) \times Q([\theta_2, \theta_1, \theta_{3:M}] \leftarrow [\theta_1, \theta_2, \theta_{3:M}]) \\
&\times \min \left(1, \frac{p_1(\theta_2) \tilde{p}_2(\theta_1) \prod_{j=3}^M \tilde{p}_j(\theta_j) \times Q([\theta_1, \theta_2, \theta_{3:M}] \leftarrow [\theta_2, \theta_1, \theta_{3:M}])}{p_1(\theta_1) \tilde{p}_2(\theta_2) \prod_{j=3}^M \tilde{p}_j(\theta_j) \times Q([\theta_2, \theta_1, \theta_{3:M}] \leftarrow [\theta_1, \theta_2, \theta_{3:M}])} \right) \\
&= \min \left(p_1(\theta_1) \tilde{p}_2(\theta_2) \prod_{j=3}^M \tilde{p}_j(\theta_j) \times Q([\theta_2, \theta_1, \theta_{3:M}] \leftarrow [\theta_1, \theta_2, \theta_{3:M}]), \right. \\
&\quad \left. p_1(\theta_2) \tilde{p}_2(\theta_1) \prod_{j=3}^M \tilde{p}_j(\theta_j) \times Q([\theta_1, \theta_2, \theta_{3:M}] \leftarrow [\theta_2, \theta_1, \theta_{3:M}]) \right) \\
&= p_1(\theta_2) \tilde{p}_2(\theta_1) \prod_{j=3}^M \tilde{p}_j(\theta_j) \times Q([\theta_1, \theta_2, \theta_{3:M}] \leftarrow [\theta_2, \theta_1, \theta_{3:M}]) \\
&\times \min \left(\frac{p_1(\theta_1) \tilde{p}_2(\theta_2) \prod_{j=3}^M \tilde{p}_j(\theta_j) \times Q([\theta_2, \theta_1, \theta_{3:M}] \leftarrow [\theta_1, \theta_2, \theta_{3:M}])}{p_1(\theta_2) \tilde{p}_2(\theta_1) \prod_{j=3}^M \tilde{p}_j(\theta_j) \times Q([\theta_1, \theta_2, \theta_{3:M}] \leftarrow [\theta_2, \theta_1, \theta_{3:M}])}, 1 \right) \\
&= p_1(\theta_2) \tilde{p}_2(\theta_1) \prod_{j=3}^M \tilde{p}_j(\theta_j) \times K([\theta_1, \theta_2, \theta_{3:M}] \leftarrow [\theta_2, \theta_1, \theta_{3:M}]) \\
&= p_{full}(\theta'_{full}) \times K(\theta_{full} \leftarrow \theta'_{full})
\end{aligned}$$

Therefore, the detailed balance condition is maintained for exchange moves between chains 1 and 2, which means that it is maintained for MPPT.

3.3.3 Custom precision accelerators

FPGA

WPT architecture: The FPGA architecture for WPT is shown in Figure 3.4, comprising:

1. A custom precision PT system (which is the system of Figure 3.1 with the Probability Evaluation block operating in custom precision).
2. A block which computes the probabilities of all samples of the first chain in double precision ($p_1(\mathbf{x}_1^{(i)})$ for all i).
3. A Weight Evaluation block, which uses the probabilities in custom and double precision to find the weights of Equation (3.10).

The double precision Probability Evaluation block processes samples from the first chain only. The custom precision Probability Evaluation block (inside Custom precision PT) processes samples from

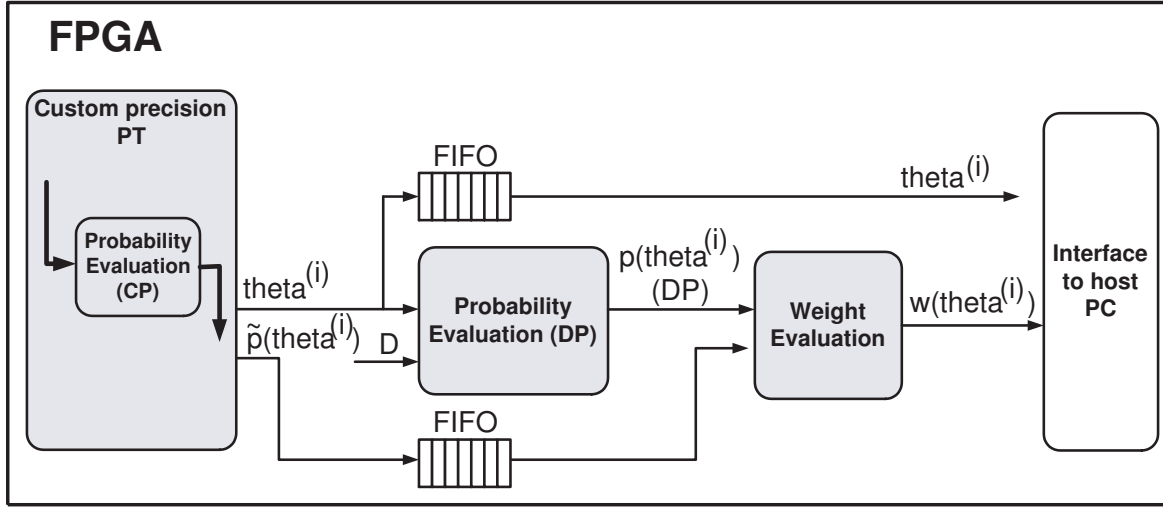


Figure 3.4: FPGA architecture for WPT: The Custom precision PT block is the same as the system in Figure 3.1 but uses custom precision for probability evaluation. DP stands for double precision, CP stands for custom precision

all M chains. This means that the two blocks need to read data from the Data memory at different rates. A dual port Data memory with one port assigned to each block is used to address this issue.

The custom precision PT block works in parallel to the other blocks in Figure 3.4. When a sample is generated by the PT block, it is passed to the other modules while the PT block works on the next sample. The weights are written to the same buffers as the MCMC samples. The latencies of the various blocks inside the Custom precision PT block are given by the same equations that were presented for the baseline architecture in Section 3.3.1. The weight evaluation modules are not in the critical path of the system (they work in parallel, acting on the output of PT). Therefore, the total time of the WPT architecture is given by Equation (3.7) (the latency of the sub-density evaluator $C_{subdensity}$ changes due to the use of custom precision operators but does not affect performance significantly). The accelerator's performance depends on the number of custom precision pipelines in the Probability Evaluation block (inside Custom precision PT), which is again denoted P . Compared to the baseline architecture, the use of reduced precision leads to a net gain in P (and subsequently in throughput), despite WPT's resource overhead due to the extra blocks of Figure 3.4 (the overhead is quantified in Section 3.6.2). Of course, the use of IS to get the final estimate affects the mixing of WPT negatively compared to the double precision sampler. The total combined effect of P and mixing in performance is explored in Section 3.6.4.

MPPT architecture: Figure 3.5 illustrates the FPGA architecture for MPPT. There are two Probability Evaluation blocks. The double precision block is responsible for:

1. Computing $p_1(\theta_1^{(i)})$ for all first chain updates.
2. Computing $p_1(\theta_r^{(i)})$ (where $r = 2$) in (3.14) for all exchanges between chains 1 and 2.

The custom precision block is responsible for:

1. Computing $p_j(\theta_j^{(i)})$ for $j \in \{2, \dots, M\}$ (auxiliary chains).
2. Computing $\tilde{p}_r(\theta_1)$ (where $r = 2$) in (3.14) for exchanges between chains 1 and 2.

Figure 3.5 provides a simplified view of the pipelines inside the two blocks. An extra Sample Proposal block is needed to feed the double precision block (the candidate samples for this block are generated earlier for synchronization). A dual port Data memory is used as in WPT. As in WPT, latency and runtime are not affected by the presence of the double precision block and they are given by the same equations. Performance increases with smaller precisions because a larger P can be used (as in WPT). Although mixing tends to decrease with lower precisions (at least for the case studies of this chapter), the MPPT accelerator achieves a net gain in performance compared to the baseline accelerator (more details in Section 3.6).

Multi-core CPU

In addition to the above FPGA implementations of WPT and MPPT, the two custom precision methods of Section 3.3.2 were also mapped to a multi-core CPU. CPUs have inherent support only for single and double precision floating point arithmetic, so the only choice for the precision of the weight computation in WPT and the auxiliary chains in MPPT is single precision. This precision can reduce runtime as will be shown in Section 3.6.

GPU

The custom precision methods can also be used in a GPU setting. This requires the use of extra kernels compared to the baseline implementation of Section 3.3.1. Only single precision can be used as reduced precision, as in the CPU case.

For WPT, the global update kernel runs exclusively in single precision and, after it terminates, a second kernel is invoked to evaluate the density of the first chain in double precision. The weight is computed

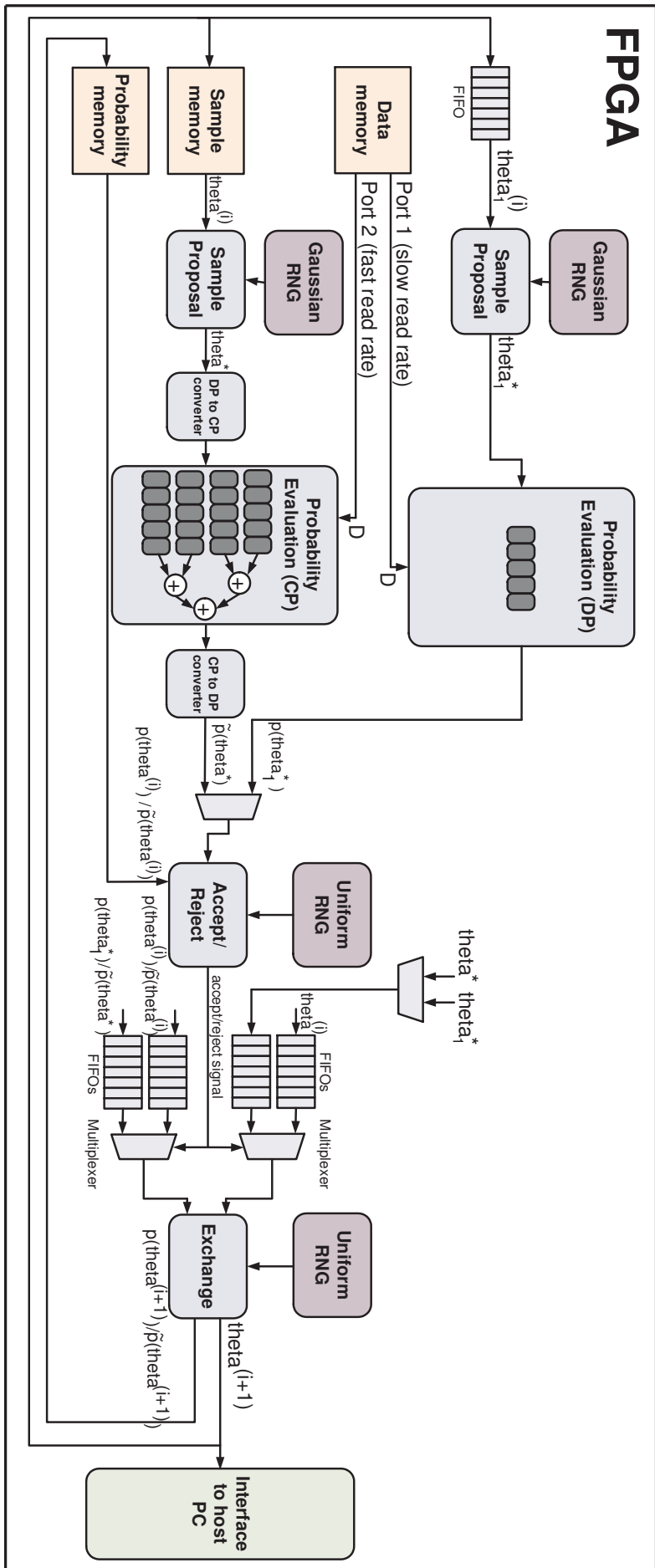


Figure 3.5: FPGA architecture for MPPT: The double precision (DP) Probability Evaluation block has one pipeline and the custom precision (CP) Probability Evaluation block has multiple pipelines.

in software. For MPPT, the global update comprises two kernel invocations, which correspond to the custom and double precision pipelines of the FPGA architecture. The first kernel (single precision) computes:

1. $p_j(\theta_j^{(i)})$ for all auxiliary chain updates ($j \in \{2, \dots, M\}$).
2. $\tilde{p}_r(\theta_1)$ (where $r = 2$) in (3.14) for the exchange between chains 1 and 2.

The second kernel (double precision) computes:

1. $p_1(\theta_1^{(i)})$ for the first chain update.
2. $p_1(\theta_r^{(i)})$ (where $r = 2$) in (3.14) for all the exchange between chains 1 and 2.

The second kernel only processes two density evaluations and therefore underutilizes the GPU. This leads to inefficiency for small numbers of PT chains (see Section 3.6.3). In both methods, the global exchange kernel runs in double precision.

3.4 Case study: Bayesian inference on mixture models

Mixture models are a powerful family of probabilistic models used in numerous fields [40]. Multimodal target distributions often appear when performing inference on these models. Hence, they are representative of the problems on which PT is applied. A Gaussian mixture model taken from [16] is used here. This model obeys the i.i.d. principle described in the previous section and thus leads to a density which is a product (or sum) of sub-densities. By increasing the number of i.i.d. data (and thus the number of sub-densities), the data-related computational load (i.e. the available parallelism in intra-chain computations) can be scaled (see Section 3.6.3).

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

$$p(D_l | \mu_{1:k}, \sigma_{1:k}, a_{1:k-1}) = \sum_{i=1}^k a_i N_{PDF}(D_l | \mu_i, \sigma_i) \quad (3.16)$$

Here, N_{PDF} denotes the density of a univariate Gaussian distribution, k is the number of mixture components and $\mu_{1:k}$, $\sigma_{1:k}$ and $a_{1:k-1}$ are the parameters of the model (means, variances and weights

Table 3.3: Case study parameters

Symbol	Description	Range (in Section 3.6)
k	Number of mixture model components	4
n	Number of data	[128, 32768]
d	Dimension of data	1

of components respectively). The parameters are fixed to $k = 4$, $\sigma_i = \sigma = 0.55$ and $a_i = a = 1/k$ for $i \in \{1, \dots, k\}$. The parameter $\mu_{1:4}$ is the unknown parameter which is sampled by MCMC, i.e. $\theta = \mu_{1:4}$ (same as in [16]). The prior distribution on $\mu_{1:4}$ is a four-dimensional uniform. In order to perform inference, a data set $D_{1:n}$ is simulated using $\mu_{1:4} = (-3, 0, 3, 6)$. The goal is then to infer $\mu_{1:4}$ using PT.

Due to the i.i.d. assumption, the likelihood is a product of sub-densities:

$$p(D_{1:n}|\mu_{1:4}) = \prod_{j=1}^n p(D_j|\mu_{1:4}, \sigma_{1:4}, a_{1:3}) \quad (3.17)$$

If $p(\mu_{1:4})$ is a uniform prior, the posterior density of $\mu_{1:4}$ (which is the target density of PT and admits $4! = 24$ modes [7, 16]) is given by:

$$p(\mu_{1:4}|D_{1:n}) \propto p(D_{1:n}|\mu_{1:4})p(\mu_{1:4}) \quad (3.18)$$

The normalizing constant of the posterior is not needed in MCMC, as already explained in Chapter 2. The main parameters of PT and the parameters of the mixture model target distribution are shown in Table 3.3.

3.5 Implementation

3.5.1 IP implementation and FPGA system integration

All FPGA samplers were implemented in VHDL. The RIFFA framework (version 1.0) [143] was used for prototyping. RIFFA wraps the PT IP and uses a PCI-express connection to transfer data between the FPGA and the host PC. All the I/O modules on the hardware side and the software drivers on the host side are handled by the framework. A small piece of C code was written for the host side in order to initialize the FPGA, start the run and receive the outputs. Moreover, a double buffering architecture was designed on top of RIFFA in order to be able to send output data (MCMC samples and weights)

to the host PC at the same time they are generated by the IP. The measured FPGA-to-host throughput of the double buffering PCI-express memory architecture is 120 MB/sec. This throughput is enough for all experiments presented in this chapter (i.e. it is enough for I/O not to be the bottleneck of the system). This number is close to the reported RIFFA throughput in [143]. It can be significantly improved by using more PCI-express lanes. All FPGA samplers were synthesized, placed and routed using Xilinx XPS 13.1. The clock frequency was set to $freq = 210$ MHz for all designs.

The sequence of operations for a complete run of the PT sampler are the following: The C application in the host allows the user to select the parameters of the PT sampler (e.g. the constants $N, B, Temp_{1:M}, \theta_{1:M}^{(1)}$ and $\sigma_{1:M}^2$). The number of chains (M) must be fixed before synthesis. RIFFA driver functions are used to send the initialization data directly to the FPGA IP and to order the IP to start (no CPU operates on the FPGA). The IP performs the PT run and writes the output MCMC samples (and the weights for WPT) to the double buffer. The double buffers work simultaneously with the IP, sending data back to the host PC, where the RIFFA driver receives them and stores them in a text file.

3.5.2 Platforms and devices

The performance of the proposed PT samplers is evaluated using a number of devices from each platform (Table 3.4 contains details). The devices represent recent and older generation of each platform. The two GPU generations roughly correspond to the two FPGA generations in terms of release dates.

For the multi-core CPU, Intel Xeon devices were used with numbers of cores ranging from 4 to 20. Some of the devices consist of a pair of chips placed on separate sockets. All processors were installed on Imperial College's High Performance Computing cluster. All runs were performed using 16 GBs of RAM and the code was compiled using Intel's C++ compiler (ICC version 2015.1) and applying the -O3 optimization flag and flags designed to optimize for the targeted CPU architecture. Every effort was made to select the combinations of optimizations (including Cilk optimizations) that maximize performance in each scenario.

For the GPU platform, measurements are presented from one device of the Nvidia GeForce 200 series (Tesla architecture) and five devices of the Nvidia GeForce 400 series (Fermi architecture). Actual runs were performed only for the C2050 model (hosted by an Intel Core 2 Q9550 CPU with 8 GBs of RAM, running Linux). The remaining measurements came from the GPGPU-Sim simulator [144]

(version 3.2.2). This simulator can construct a model of any GPU device by configuring various parameters in a text file. It then predicts the time required to run a CUDA kernel on the device accurately. 97-98% accuracy is reported for Tesla and Fermi architectures, which is enough for the purposes of this chapter. CUDA version 1.3 (for Tesla) and version 2.0 (for Fermi) were used. The Nvidia compiler (NVCC) [145] was used for compiling the GPU kernels and the Intel C++ compiler (ICC version 2015.1) was used for compiling the part of the code that runs on the CPU (the same optimization flags mentioned for the CPU sampler were applied here).

For the FPGA platform, results are presented for one device of the Xilinx Virtex 6 series and six devices of the Xilinx Virtex 7 series. Actual runs were performed only for the Virtex 6 LX240T model (placed on an ML605 board and hosted by an Intel Core i7-2600 CPU with 4 GBs of RAM, running Linux). Performance estimates for the other devices come from combining post-place and route resource utilization, device resources and equation (3.7) (either for baseline, WPT or MPPT).

Two sequential reference implementations were used. The first was a sequential implementation in C++, which ran on an Intel Core i7-2600 device with one core activated and with all compiler and Cilk optimizations deactivated. This is an attempt to capture the approach of MCMC practitioners who are not familiar with any form of code optimization or parallelization. The second reference implementation was an identical sequential implementation in C++, which also ran on an Intel Core i7-2600 with one core activated but with all Intel compiler optimizations activated (Cilk optimization were deactivated).

In order to produce power and energy consumption results, the Xilinx Power Estimator [146] was used for the FPGA samplers (assuming full device utilization) and the nominal thermal design power of the CPU and GPU devices was used.

3.5.3 Runs in hardware and software

As mentioned above the three FPGA samplers (baseline, WPT and MPPT) were compiled and run only on the LX240T device. Nevertheless, bitstreams were not generated for all parameter and precision combinations (i.e. number of chains M , number of mantissa bits)¹. The parameter and precision combinations for which a bitstream was generated are shown in Table 3.5, separately for each algorithm. The baseline sampler was compiled for $M = 8$ and $M = 32$. The WPT sampler was compiled for

¹recall that M and precision are set at compile time

Table 3.4: Detailed list of platforms and devices

Platform	Family/Device	Release date	Fabrication process
Multi-core CPU	Intel Xeon E5-2620 (4 cores), 2 x X5650 (2 x 6 cores), 2 x E5-2660 v2 (2 x 10 cores)	2010-13	32nm
	GeForce 200 GTX285	2009	55nm
GPU	GeForce 400 GT420, GT440, GTS450, GTX460SE, GTX465, C2050	2010-11	40nm
	Virtex 6 LX240T	2009	40nm
FPGA	Virtex 7 VX330T, VX415T, VX485T, VX550T, VX690T, VX1140T	2011	28nm

$M = 8$ combined with all custom precision configurations ((4, 11), (6, 11), (8, 11), (10, 11), (14, 11), (20, 11), (24, 11), (40, 11) and (53, 11)). The MPPT sampler was compiled for $M = 8$ combined with custom precision configuration (24, 11) only. Thus the runtimes and mixing results for the above combinations come from real runs on the LX240T FPGA, while the resource utilization results for these combinations are post place and route results. Runs for the remaining combinations of M and precision were performed in software using a C++ implementation (no bitstream was compiled). In order to “emulate” the custom precision calculations, the MPFR library [147] was used. This library allows all arithmetic operators to be performed in any custom precision inside C++ code. The resource utilization of the above combinations (which ran in software) was calculated using the post place and route results of the compiled designs in combination with post place and route resource utilization of the Probability Evaluation blocks in different precisions, i.e. Probability Evaluation blocks were compiled in all precisions but did not run in hardware for all precisions. FPGA runtimes were estimated based on the latency and runtime equations of Section 3.3.1 (where P was defined based on the resource utilization of the generic parts and the custom precision Probability Evaluation blocks).

Table 3.5: This table shows the PT parameter combinations for which actual FPGA bitstreams were generated and FPGA runs were performed, separately for each PT algorithm (baseline, WPT, MPPT). Also, it shows the combinations for which software runs were performed instead of FPGA runs. Software runs were implemented in C++ code, using the MPFR library for custom precision calculations.

Algorithm	Implementation	Parameter combinations
Baseline	On FPGA	$(M = 8)$, $(M = 32)$
Baseline	In software (+MPFR)	$(M = 128)$, $(M = 512)$, $(M = 2048)$, $(M = 8192)$, $(M = 32768)$
WPT	On FPGA	$(M = 8, m = 4, e = 11)$, $(M = 8, m = 6, e = 11)$, $(M = 8, m = 8, e = 11)$, $(M = 8, m = 10, e = 11)$, $(M = 8, m = 14, e = 11)$, $(M = 8, m = 20, e = 11)$, $(M = 8, m = 24, e = 11)$, $(M = 8, m = 40, e = 11)$, $(M = 8, m = 53, e = 11)$
WPT	In software (+MPFR)	All other combinations
MPPT	On FPGA	$(M = 8, m = 24, e = 11)$
MPPT	In software (+MPFR)	All other combinations

3.6 Investigation and results

3.6.1 Performance metrics

The performance criterion used to compare PT accelerators is the mixing (exploration) per second of runtime that the accelerator can achieve, given a fixed selection of PT tuning parameters (number of chains M , temperatures $Temp_{1:M}$) and a particular target distribution (implying a fixed data set size n). Comparisons are made only between identically tuned implementations in different platforms. Of course, various numbers of chains are used to examine performance scaling but each comparison is done after fixing the number of chains (e.g. a GPU with 128 chains is only compared with identically tuned CPU and FPGA implementations with 128 chains, not with samplers that use 32 chains). The issue of selecting the optimal number of chains, tempering scheme, etc in PT has been addressed extensively in previous literature [148, 61, 27, 16].

The mixing per second metric is affected by two factors:

1. How many MCMC samples can be generated per second, i.e. the raw throughput of the sampler (given by Equation (3.8), where each sample is equivalent to an MCMC iteration).
2. How fast the MCMC samples explore the distribution, i.e. how much “exploration” is achieved

with a given amount of samples. This factor is related to the way the algorithm proposes and accepts/rejects new samples.

Here, since the PT algorithm and all its tuning parameters are fixed when comparisons are performed, the second factor depends only on precision (precision affects exploration as explained in Section 3.3.2).

For double precision (baseline) accelerators, only the first factor (how many samples can be generated per second) affects performance. It is enough to find the Raw Speedup ($Speedup_{raw}$) of each accelerator against the first reference sequential CPU sampler (without Intel compiler optimizations) to make comparisons (speedup refers to samples/sec). The second reference sequential CPU sampler (with Intel compiler optimizations) will also be shown in the performance evaluation figures for completeness.

Nevertheless, for custom precision accelerators, the second factor has to be taken into account too. Specifically, the performance of WPT is affected by:

1. Different first chain mixing speed compared to the baseline sampler (due to custom precision), i.e. the difference in mixing when sampling from $\tilde{p}_1$ instead of p_1 .
2. The importance sampling mechanism; the more the importance distribution $\tilde{p}_1$ deviates from the IS target distribution p_1 , the more the importance weights take extreme values. This makes the IS estimator in (3.11) less efficient [149, 6]. A few large weights might end up containing most of the useful “information” and more samples need to be drawn to achieve the same variance as when sampling from the IS target directly.

The sampling efficiency of MPPT is affected by the use of custom precision for the auxiliary chains ($\tilde{p}_{2:M}$), which changes the mixing speed of the first chain (compared to the baseline sampler). Due to these reasons, one MCMC sample from WPT or MPPT has a different “exploration value” compared to one sample from baseline samplers.

To capture the second factor (exploration speed) in comparisons, two metrics are used:

1. The Effective Sample Size due to MCMC autocorrelation (ESS_{mcmc}) [9], a typical metric of mixing speed in MCMC. This metric was described in Chapter 2 and was denoted ESS but ESS_{mcmc}

is used here to distinguish it from ESS_{is} (which is described below). As already mentioned, ESS_{mcmc} gives an estimate of how many independent (or “effective”) samples the dependent MCMC samples are equivalent to. It is computed based on equation (2.10). In both WPT and MPPT, ESS_{mcmc} quantifies the effect that custom precision (in chains 1 to M for WPT and 2 to M for MPPT) has on the mixing of the first chain.

2. The Effective Sample Size due to importance sampling (ESS_{is}) [6], which gives an estimate of how many independent samples from the IS target distribution the samples from the IS importance distribution are equivalent to (when used to find (3.11)). This is a standard metric in IS literature and it can be estimated using the weights of the samples as follows [149]:

$$ESS_{is} = \frac{(\sum_{i=1}^N w_i)^2}{\sum_{i=1}^N w_i^2} \quad (3.19)$$

More information on ESS_{is} can be found in [149] and [6]. In WPT, ESS_{is} quantifies the loss in efficiency due to the use of IS weights.

Combining these metrics with Raw speedup, a performance criterion suitable for all compared samplers, the Effective Speedup, is proposed:

$$Speedup_{eff} = Speedup_{raw} \times \frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}} \times \frac{ESS_{is}^{(CP)}}{ESS_{is}^{(DP)}} \quad (3.20)$$

The ratio $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}}$ represents the loss/gain in the mixing efficiency of the chain 1 when going from double (DP) to custom (CP) precision, i.e. the relative ESS_{mcmc} . The ratio $\frac{ESS_{is}^{(CP)}}{ESS_{is}^{(DP)}}$ represents the loss/gain in efficiency due to IS when going from DP to CP, i.e. the relative ESS_{is} . For the baseline samplers $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}} = 1$ and $\frac{ESS_{is}^{(CP)}}{ESS_{is}^{(DP)}} = 1$ (since no custom precision is used). Therefore, $Speedup_{eff} = Speedup_{raw}$. For WPT, $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}} \neq 1$ and $\frac{ESS_{is}^{(CP)}}{ESS_{is}^{(DP)}} \neq 1$. In this case the (total) effective sample size is the product of the effective sample sizes due to MCMC and due to IS. This has been suggested by Gramacy et al. [139] as a suitable approximation to total effective sample size when IS is used on top of MCMC. For MPPT, $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}} \neq 1$ and $\frac{ESS_{is}^{(CP)}}{ESS_{is}^{(DP)}} = 1$.

$Speedup_{eff}$ measures the performance gain (in effective samples/sec) of an implementation compared to the reference double precision implementation (for identical tuning parameters and target distribution). For WPT and MPPT, $Speedup_{eff}$ changes with precision. In the FPGA case, precision can be optimized to maximize $Speedup_{eff}$ (see Section 3.6.4). In the CPU and GPU case only the single

precision configuration can be used so the optimization choice is between single and double precision ($Speedup_{eff}$ is different for these two choices).

3.6.2 FPGA resource utilization

For FPGA architectures, $Speedup_{raw}$ (and $Speedup_{eff}$) depends on the number of pipelines in the Probability Evaluation block (P), since larger P leads to higher throughput in (3.8). All FPGA implementations presented here use the maximum number of pipelines that can fit in the targeted device (given the device's Look-up Tables (LUTs), Flip Flop registers (FFs) and Digital Signal Processing (DSPs) blocks). In all cases, DSP blocks are the limiting resource, i.e. the one that limits the number of pipelines.

Table 3.6 shows the post place and route resource utilization of the various blocks of the FPGA architectures when mapped to a Virtex-6 LX240T device. The total resources of LX240T are also shown. The overheads of MPPT and WPT comprise one double precision probability evaluation pipeline and the extra modules described in Section 3.3.3. MPPT needs slightly more resources than WPT due to the extra Sample Proposal block. The table also shows the resources needed for a single pipeline in different precisions. Fewer resources allow a larger P to be used. The optimization in Section 3.6.4 (Figures 3.11 and 3.12) shows how the area saving due to reduced precision translate into higher $Speedup_{raw}$.

3.6.3 Performance evaluation

In this section, the number of chains and the size of the data are set to various different values and $Speedup_{eff}$ is measured to compare accelerators (CPU, GPU and FPGA). The distribution of Section 3.4 is targeted. All runs are performed with $N = 10^5$, i.e. until 10^5 MCMC samples are generated. A comparison is also made with the sampler of Lee et al. [16], where each thread is assigned one PT chain without exploiting intra-chain parallelism. The way $Speedup_{eff}$ scales with the number of chains, the amount of data and the size of the device is also investigated. Although the investigation of performance scaling with data size is specific to the i.i.d. data assumption, this does not restrict the applicability of the proposed inter-chain parallelization techniques and custom precision methods.

The precision optimization described in Section 3.6.4 has been applied to all custom precision FPGA implementations of this section. Also, each GPU sampler uses kernels with the optimal combination of

Table 3.6: Resource utilization of the various processing blocks. The numbers in parentheses next to each probability evaluation block pipeline are the numbers of mantissa and exponent bits of the precision configuration respectively. DP stands for double precision (53 mantissa and 11 exponent bits).

Block name	LUTs (% of LX240T)	FFs (% of LX240T)	DSPs (% of LX240T)
Sample proposal	6688 (4.4%)	5286 (1.7%)	48 (6.2%)
Accept/reject	9123 (6.0%)	7422 (2.4%)	43 (5.5%)
Exchange	6277 (4.1%)	5123 (1.6%)	62 (8.0%)
Other functions, control and I/O (including RIFFA)	16965 (11.2%)	12592 (4.1%)	0 (0.0%)
WPT overhead	26329 (17.4%)	29012 (9.6%)	221 (28.7%)
MPPT overhead	29549 (19.6%)	33087 (10.9%)	230 (29.9%)
Probability evaluation - 1 pipeline (DP)	25581 (16.9%)	28408 (9.4%)	218 (28.3%)
Probability evaluation - 1 pipeline (24,11)	8267 (5.4%)	8146 (2.7%)	58 (7.5%)
Probability evaluation - 1 pipeline (20,11)	7598 (5.0%)	7697 (2.5%)	43 (5.5%)
Probability evaluation - 1 pipeline (16,11)	7098 (4.7%)	7355 (2.4%)	38 (4.9%)
Probability evaluation - 1 pipeline (12,11)	5748 (3.8%)	6051 (2.0%)	37 (4.8%)
Probability evaluation - 1 pipeline (8,11)	4732 (3.1%)	5219 (1.7%)	37 (4.8%)
Virtex-6 LX240T total resources	150720 (100.0%)	301440 (100.0%)	768 (100.0%)

CUDA blocks and data per thread (given M and n), as will be demonstrated in Section 3.6.4. Finally, CPU implementations use the optimal granularities for chain and reduction parallelization. These optimizations are all hardware-related (i.e. changes in the architecture/implementation). Algorithm-related parameters (i.e. M and $Temp_{1:M}$) are not optimized as mentioned earlier.

Scaling the number of chains: In Figure 3.6, the amount of data is set to $n = 128$ and the number of chains (M) varies. The actual runtimes of the sequential reference implementation in C++ range from 6 seconds for $M = 8$ to 6.7 hours for $M = 32768$. These runtimes are typically multiplied by the number of independent runs that a practitioner performs, which can range from 10 to many hundreds. Also, the number of samples can be increased depending on the required variance in the output estimate (here it is set to $N = 10^5$). These factors can lead to long runtimes.

The $Speedup_{eff}$ achieved by each accelerator compared to the sequential reference implementation (without compiler optimizations) is shown in Figure 3.6. Moreover, the $Speedup_{eff}$ of the second reference CPU implementation, i.e. a sequential CPU implementation with Intel compiler optimizations, is shown in order to make comparisons with this reference implementation easy. For the CPU, only the measurements for the 20-core accelerator (on the E5-2660 v2 chip) are shown. For the GPU, only the measurements for GTX285 and C2050 (which is the largest GPU device used here) are shown. For the FPGA, measurements for LX240T and VX1140T (which is the largest FPGA used here) are

shown.

The baseline PT on a multi-core CPU achieves a peak speedup of 13.8x. This is not reached with fewer than 2048 chains. The WPT and MPPT CPU samplers reach a speedup of 18.4x. This improvement is due to the use of single precision for the majority of density evaluations. This shows that it is possible to significantly improve the performance of sequential code by using Intel Compiler optimizations and making changes to the code to guide the compiler on how to parallelize computations.

In order to clarify what part of the presented CPU speedups is due to Intel compiler optimizations and what part is due to the use of the Cilk library, the above $Speedup_{eff}$ numbers can be divided by the $Speedup_{eff}$ of the second reference implementation (with compiler optimizations), i.e. the purple starred line in the figure. From observing the figure, it is clear that the compiler optimizations offer an extra 1.38x speedup (with some small variation) over the first reference implementation (which uses no compiler optimizations) for all the range of chain numbers. The constant speedup is expected, since there is no exploitation of parallelism in either of the reference implementations. As a result of the above number, it can be concluded that most of the speedup achieved by the parallel CPU samplers is due to the Cilk pragmas: The baseline PT is up to 10.0x faster than the compiler-optimized reference implementation and the WPT and MPPT are up to 13.3x faster than the compiler-optimized reference implementation.

The baseline PT on the two GPUs achieves significantly higher peak $Speedup_{eff}$. To reach peak performance (165x for C2050, 78x for GTX285), $M = 32768$ chains are used. For M close to a few hundreds, speedups are in the range 15x-50x. This slow scaling is due to lack of enough parallelism to fully utilize the GPU until a large amount of chains are used. These speedups are up to 4.4x higher than those of the state-of-the-art implementation of Lee et al. [16]. Lee et al. [16] reach the peak performance of the implementation of this chapter only for $M = 32768$. This is expected, since Lee et al. [16] only exploit inter-chain parallelism and thus it requires a massive number of chains to fully exploit the processing power of the GPU.

WPT and MPPT samplers achieve similar $Speedup_{eff}$ in the GPU. They outperform baseline implementations by up to 3.2x (GTX285) and 2.4x (C2050). For fewer than 512 chains, baseline samplers are faster than WPT and MPPT by up to 1.3x. This is explained as follows: Two update kernels (in double and single precision) are called in WPT and MPPT (Section 3.3.3) to complete each Global update (one double precision kernel which processes one chain/weights and one single precision kernel

which processes everything else). Because of GPU under-utilization for small problems, the kernels' runtime is stable or increases slightly when the processed chains range between 1 and 200 (i.e. the GPU has free resources to spare). As a result of the almost constant runtime, when e.g. MPPT calls the double precision kernel to process the first chain, the runtime of the kernel is almost the same as the runtime that would be needed by the baseline sampler (which works in double precision) to process 32 or even 128 chains. On top of that, MPPT involves the cost of calling a second (single precision) kernel for the auxiliary chains. Therefore, using MPPT in the GPU results in lower performance than the baseline GPU sampler until a large enough number of chains is used at the second (single precision) kernel; at that point the benefits from processing the auxiliary chains in single precision outweigh the cost of running two kernels (i.e. the single precision kernel dominates the runtime and therefore MPPT becomes faster than the baseline sampler). The same behaviour is observed in WPT, where instead of the first chain, the double precision kernel processes the weight evaluations.

All the above speedups should be divided by the respective speedups of the second reference CPU implementation (approximately 1.38 for all points in the graph) in order to calculate the speedups of the GPU samplers against the second reference CPU implementation (which makes use of compiler optimizations).

On the FPGA platform, the baseline architecture on the LX240T is 26x faster than the reference (for all M) and the VX1140T is 44x-165x faster than the reference (depending on M). WPT increases these speedups to 65x-128x and 66x-854x respectively and MPPT to 98x-138x and 76x-997x respectively. The boost in performance when using WPT/MPPT is due to increased P (the number of parallel sub-density blocks), which in turn is a results of using reduced precision. Peak performance is reached with only 8-32 chains for the baseline and 32-128 chains for the custom precision architectures. The speedups of all custom precision samplers in all platforms come without no compromise in sampling quality. All the above speedups should be divided by (approximately) 1.38 in order to calculate the speedups of the FPGA samplers against the second reference CPU implementation (which makes use of compiler optimizations).

Comparing across platforms, it is clear that the big Virtex 7 FPGA with the baseline sampler is 24x-36x and 0.9x-28x faster than the big baseline CPU (20 cores) and big baseline GPU (C2050) implementations respectively. These speedups increase to 57x-92x and 2.2x-76x when WPT or MPPT are used for all platforms. The increased speedups of the FPGA for WPT/MPPT show that the two custom precision methods are more suitable for mapping on the FPGA because they can exploit the fully custom

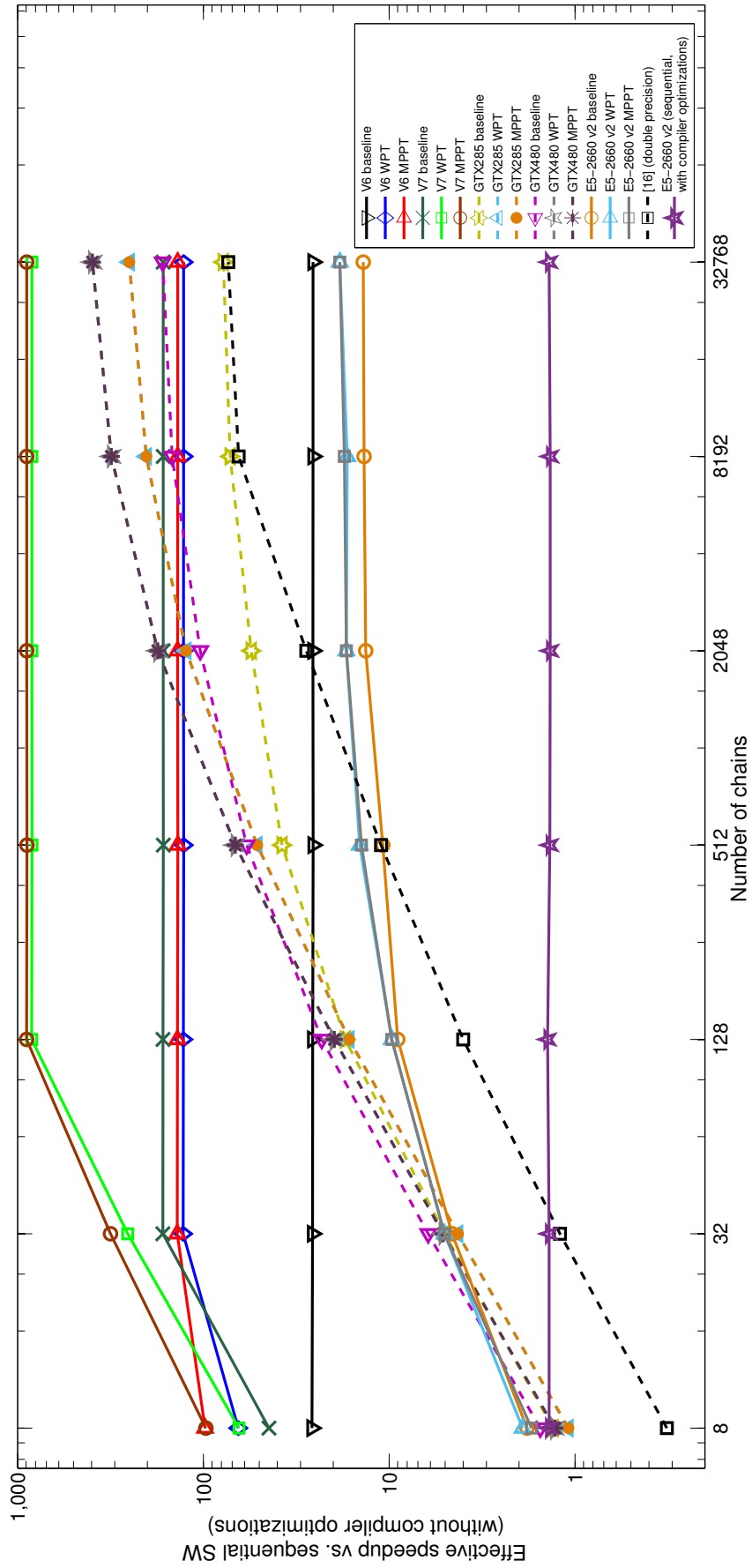


Figure 3.6: Scaling of $Speedup_{eff}$ with number of chains M . Baseline and custom precision accelerators are included. $n = 128$ were used. V7 measurements are based on projections and GTX285 measurements are based on GPU-Sim simulations.

precision of the platform, while CPUs and GPUs can only use single or double precision. Similarly, the small Virtex 6 FPGA is slower than the small GTX285 GPU for $M > 128$ when the baseline sampler is used but only for $M > 2048$ when the WPT/MPPT methods are used. Nevertheless, GTX285's peak performance is higher than LX240T's by 1.8x-3.1x (depending on the method).

All FPGA architectures reach their peak performance much earlier (for smaller M) than GPUs, since their flexibility allows them to utilize resources efficiently and exploit modest amounts of parallelism. For $M < 512$, the Virtex 6 FPGA is faster than both GPUs when using MPPT/WPT. For $M < 128$, even the baseline Virtex 6 is faster than all GPU samplers. In PT literature, the values of M typically range from less than 10 to a few dozens (and rarely 100-200). Thousands of chains do not improve mixing (e.g. see [16] and [27]). This makes the advantage of FPGAs at small M significant. Note also that for $M < 32$ CPU samplers are up to 1.8x faster than GPU samplers due to limited amounts of parallelism. Finally, MPPT outperforms WPT by up to 10% in FPGAs. This is due to the fact that WPT's efficiency is affected by both ESS_{mcmc} and ESS_{is} in (3.20), the latter contributing to performance deterioration. MPPT is only affected by ESS_{mcmc} .

Scaling the number of data: Figure 3.7 shows $Speedup_{eff}$ when the number of chains is set to a realistic $M = 32$ and the number of data (n) varies. The actual runtimes of the sequential reference implementation in C++ range from 24 seconds for $n = 128$ to 1.6 hours for $n = 32768$. Again these runtimes can increase considerably in real analyses due to multiple independent runs and larger N . The second reference implementation (sequential CPU with Intel Compiler optimizations) is also shown, as in the previous figure.

Next to each point corresponding to WPT and MPPT on FPGAs, the selected custom precision is shown (found based on the precision optimization of Section 3.6.4). The choice of precision only affects $Speedup_{eff}$ and not sampling quality.

The peak $Speedup_{eff}$ of the baseline CPU is 16.1x, which is slightly higher than in Figure 3.6. The speedups of the WPT and MPPT methods on the CPU are only marginally higher than the baseline sampler. The performance of the compiler-optimized reference CPU implementation (which is sequential) is again 1.38x higher than the performance of the first reference CPU implementation (this difference can be entirely attributed to the compiler).

$Speedup_{eff}$ for the baseline GTX285 and C2050 ranges from 5x to 99x. These are lower than Figure 3.6 for large M . The reason is that the GPU is able to exploit parallelism in the inter-chain level

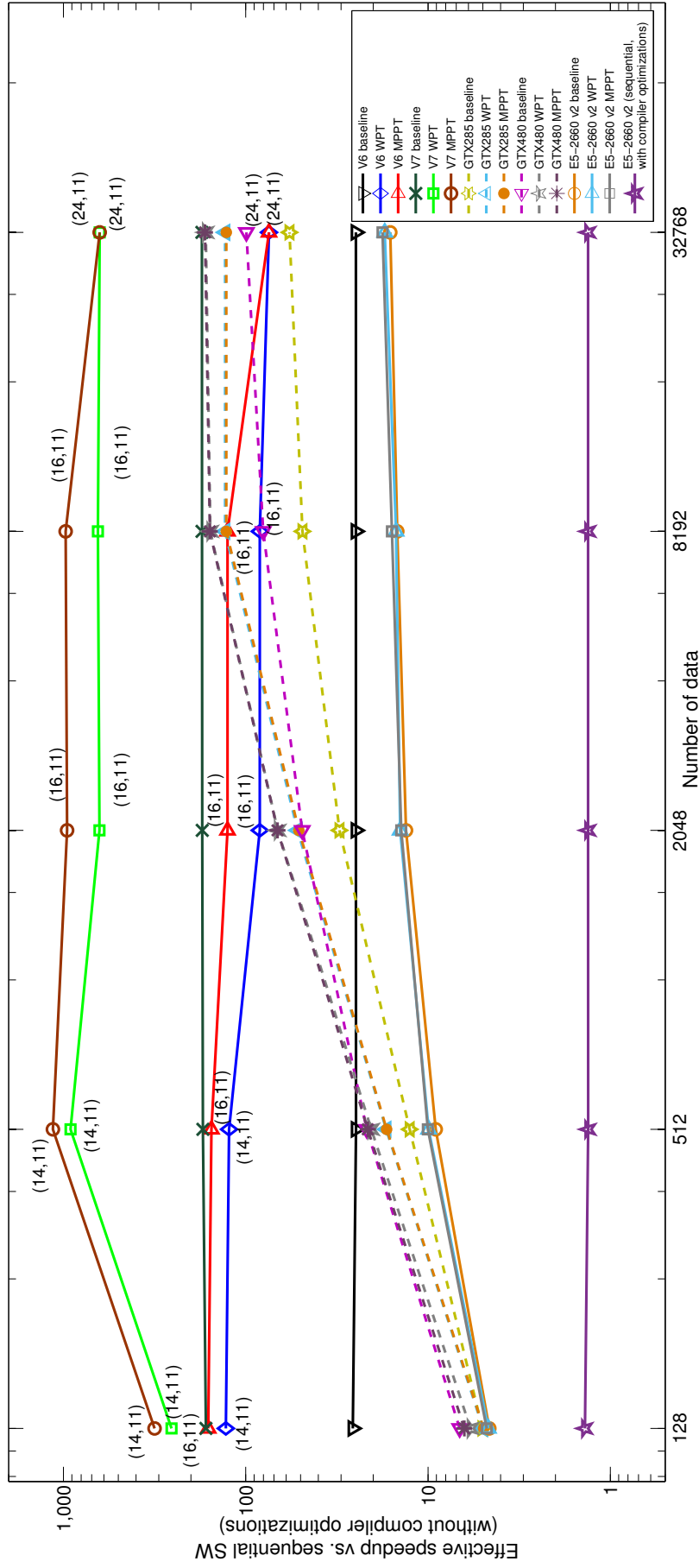


Figure 3.7: Scaling of $Speedup_{eff}$ with number of data n . Baseline and custom precision accelerators are presented. $M = 32$ chains are used. V7 measurements are based on projections and GTX285 measurements are based on GPU-Sim simulations.

more efficiently than parallelism in the intra-chain level. The culprit is the reduction operation in the likelihood computation. While the parallel chains are independent and can simply be assigned to separate threads which do not need to interact during Global updates, the sub-densities within each chain need to be summed after they are calculated, compelling threads to communicate. Even when using unrolled reduction, the GPU is under-utilized during this operation. The WPT and MPPT methods on the GPUs offer an extra speedup of up to 2.2x over the baseline sampler but they are slightly slower for $n = 128$ (in the C2050 case). FPGA samplers reach peak performances similar to Figure 3.6 (174x-1143x for VX1140T and 25x-160x for LX240T), showing that the FPGA exploits intra- and inter-chain parallelism equally well. The main reason is the efficient implementation of reductions; the adder tree which receives the values coming out of the sub-density pipelines is designed to match the number of the pipelines and does not waste resources. This is an instance of the advantages of custom architectures over fixed architectures. All the $Speedup_{eff}$ values mentioned above should be divided by (approximately) 1.38 to find the speedups against the second reference CPU implementation (which uses compiler optimizations).

The figure also reveals the scaling disadvantage of the two custom precision methods on FPGAs. When n increases, more sub-densities are summed during density evaluation, leading to more arithmetic error and rougher density approximations. This negatively affects the mixing/efficiency of WPT/MPPT. In other words, the point at which the mixing/efficiency breaks in Figures 3.11 and 3.12 of Section 3.6.4 is shifted to the right, i.e. to higher precisions (more details on the respective section). Therefore, higher precisions need to be used in the custom precision part to avoid a large penalty in mixing/efficiency. This reduces $Speedup_{eff}$, since pipelines cost more FPGA resources when using larger precisions and thus fewer pipelines are instantiated. For example, MPPT on VX1140T for $n = 32768$ needs 24 mantissa bits to achieve optimal $Speedup_{eff} = 633x$. This is smaller compared to the $Speedup_{eff} = 1143x$ achieved for $n = 512$ (requiring 14 mantissa bits). This problem does not appear in CPUs and GPUs, since only single precision is used. Due to the above issue, WPT/MPPT on GTX280 outperforms WPT/MPPT on LX240T for $n \geq 8192$. WPT/MPPT on VX1140T is still faster than WPT/MPPT on GTX480 by 3.7x-54x.

Scaling the size of the device: Figures 3.8, 3.9 and 3.10 show how $Speedup_{eff}$ scales when the size of the device (CPU, GPU and FPGA respectively) increases. This is where the full list of devices contained in Table 3.4 is employed. The parameters M and n are fixed to $M = 32$ and $n = 32768$. For multi-core CPUs, device size refers to the number of CPU cores, for GPUs it refers to the number

of Streaming Processors and for FPGAs it refers to the number of DSP blocks, which is the limiting resource as mentioned above. The runtimes for the different devices were measured as described in Section 3.5.2: FPGA runtimes were either taken from real runs (for LX240T) or from estimates based on the resources of each device, the resource requirements of the samplers and the throughput equations of each sampler (all other FPGA devices). GPU runtimes were either taken from real runs (C2050) or using the GPGPU-Sim simulator [144] (all other devices). CPU runtimes were taken from real runs (all devices).

In Figure 3.8, the performance of the baseline sampler (implemented on various Xeon processors) increases by almost 3.5x when going from 1 core to 4 cores, by 7.4x when going to 12 cores and by 13x when going to 20 cores. The WPT and MPPT samplers on 4, 12 and 20 cores were measured 3.6x, 4.8x and 8.7x faster respectively over the single core version. It is worth noting that the largest part of the total runtime on a single core is taken by the Global update due to the large data size ($n = 32768$) in this experiment. Therefore, according to Amdahl's law, the theoretical maximum speedups over the single core sampler are close to the number of cores of the CPU. The fact that the actual speedups are lower than the number of cores, shows that the Intel compiler parallelization does not fully exploit all the parallel CPU resources.

Moreover, Figure 3.8 shows the $Speedup_{eff}$ of the second reference CPU implementation (i.e. a sequential C++ implementation with all compiler optimizations activated but without any Cilk pragmas in the code). As in previous comparisons, the speedup that the compiler optimizations offer is approximately 1.38x over the first reference CPU implementation (which does not use compiler optimizations), regardless of the number of cores in the CPU. This is expected, since the implementation is fully sequential. As previously, the speedups of the various parallel CPU samplers over the second reference implementation can be found by dividing with the respective values of the green line in the figure (which are approximately equal to 1.38 for all points in the figure).

In Figure 3.9, the performance of the baseline GPU sampler on various devices from the GT400 series increases almost at the same rate as the number of SPs (and even at a higher rate in some cases due to the different SP and memory clock frequencies of the devices, as well as other architectural differences). Finally, Figure 3.10 shows that the performance of the FPGA also increases at almost the same rate as the number of DSP blocks when targeting various devices from the Virtex 7 series.

As a conclusion, it is clear that the performance of all samplers increases approximately linearly with

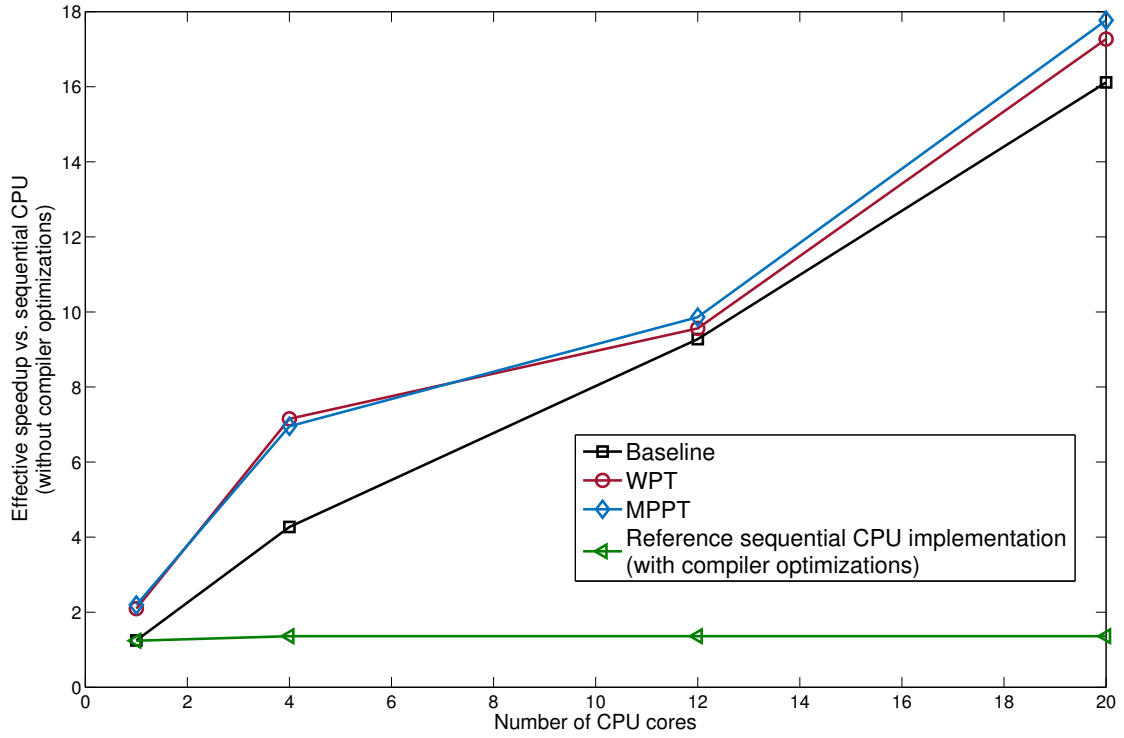


Figure 3.8: Scaling of $Speedup_{eff}$ when changing the size of the multi-core CPU (number of CPU cores). The exact CPU models used are listed in Table 3.4. Speedups are over the reference CPU version with no compiler optimizations. The $Speedup_{eff}$ of the second reference CPU implementation (which uses compiler optimizations) is also shown.

the size of the device for all platforms, which is expected since the workload is compute-bound and not memory-bound.

Power efficiency: Apart from $Speedup_{eff}$, which refers only to sampling speed, power efficiency is also important for many applications, especially when MCMC is used in High Performance Computing platforms or embedded applications (where power consumption is key). The nominal thermal design power of each device was used. For the CPUs and GPUs, this was taken from each device’s specifications. For the FPGAs, the Xilinx Power Estimator [146] was used to generate an estimate of the nominal power consumption. It was assumed that all FPGA resources are utilized. Although this is not the case in the proposed designs, this approach was adopted to make the comparison to the other platforms fair.

Table 3.7 compares accelerators in two (M, n) settings, using two metrics: Effective samples per Joule (i.e. ES/J or Performance per Watt) and Effective Samples per Joule-sec (ES/(J-sec)). FPGA accelerators can generate up to 106x and 200x more Effective samples per Joule than CPUs and GPUs respectively for small M and n (in these cases CPU and GPU resources remain largely under-

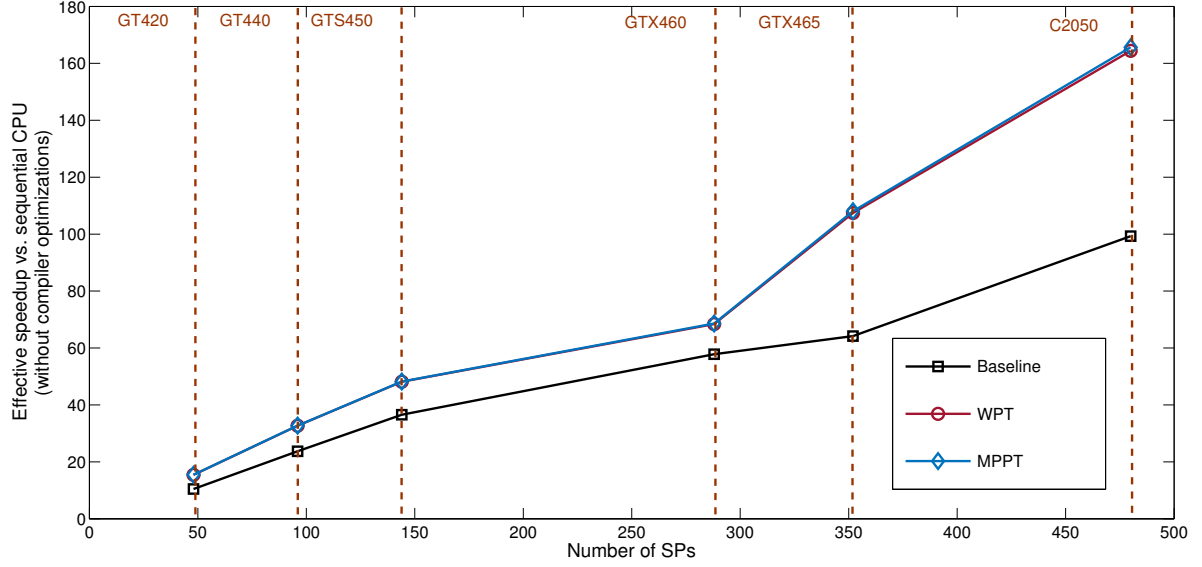


Figure 3.9: Scaling of $Speedup_{eff}$ when scaling the size of the GPU (number of SPs). All devices are based on the Fermi architecture (see Table 3.4).

utilized). The GPU improves its ES/J metric for larger M and n but is still 2.5x-10.4x less efficient than the FPGA. The FPGA's ES/(J-sec) is up to 5629x and 15636x higher than the CPU's and GPU's respectively for small M and n and up to 5159x and 38x for large M and n . The above numbers reveal that the FPGA is able to extract significantly more performance per unit of energy.

3.6.4 Precision optimization for FPGAs and kernel optimization for GPUs

Precision optimization for FPGAs: The $Speedup_{eff}$ of custom precision FPGA samplers can be maximized by optimizing their precision configuration, i.e. the number of mantissa bits. This optimization has to be done for each combination of (M, n) and FPGA device. It comprises pre-runs in all candidate precisions. From these pre-runs, samples and/or weights are collected and used to evaluate the ESS ratios in 3.20 and then get $Speedup_{eff}$. Here, the optimization process is demonstrated for one parameter combination $(M, n) = (128, 128)$ when targeting the LX240T FPGA.

For WPT, Figure 3.11 shows how the three terms in the right-hand side of (3.20) ($Speedup_{raw}$, $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}}$ and $\frac{ESS_{is}^{(CP)}}{ESS_{is}^{(DP)}}$) change with precision. Results come from 30 independent runs of 10^5 samples each. The mixing-related $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}}$ remains close to one as the number of mantissa bits decreases because $p_1(\theta)$ and $\tilde{p}_1(\theta)$ are very similar. Fluctuation in the graph is due to the variance of the ESS_{mcmc} estimator. For 6 mantissa bits the ratio starts growing, which means that WPT mixes faster than the baseline

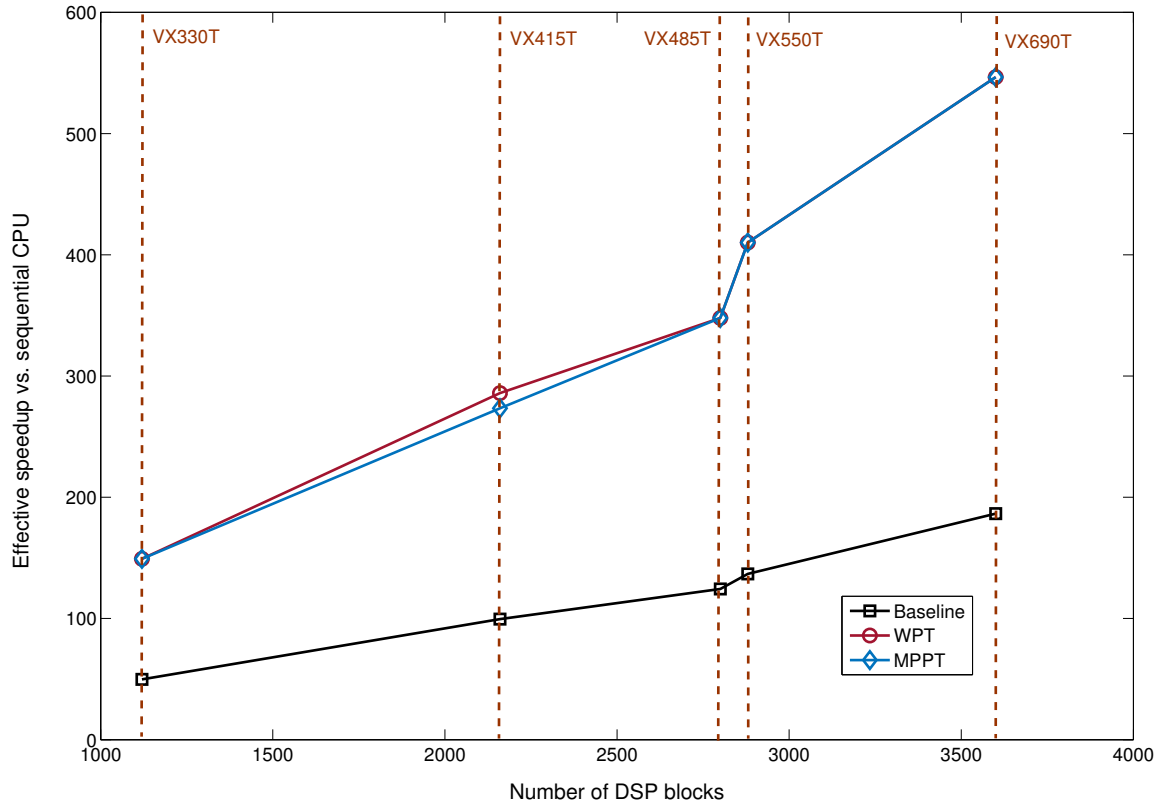


Figure 3.10: Scaling of $Speedup_{eff}$ of the Virtex 7 FPGA series when varying the device size (number of DSP blocks). The points correspond to the devices shown in Table 3.4.

sampler. This is because $\tilde{p}_1(\theta)$ becomes a coarsely quantized or even flattened version of $p_1(\theta)$, since the precision is not high enough to distinguish between different probability density values. This leads to an easy-to-sample target and faster mixing.

The IS-related $\frac{ESS_{is}^{(CP)}}{ESS_{is}^{(DP)}}$ is also very close to one for most precisions, signifying that no loss of efficiency is caused by the IS scheme. For 10 or fewer mantissa bits, the ratio decreases. This is because parts of the support of $p_1(\theta)$ take much lower probability density values under $\tilde{p}_1(\theta)$ or vice versa, especially in the distribution's tails. This results in large weights being generated, making IS inefficient [6, 149]. The increase of the IS-related ratio from 6 to 4 mantissa bits is due to the variance of the estimator.

$Speedup_{raw}$ increases by up to 5.3x (compared to double precision - right-most point in Figure 3.11) when precision drops. This is due to the reduced FPGA area cost per pipeline, which means that a larger P can be used. A larger P translates to increased throughput in (3.8).

Multiplying the three above terms gives $Speedup_{eff}$, also shown in Figure 3.11. The maximum $Speedup_{eff}=128x$ is achieved for configuration (14,11), which is the optimal precision. It is clear that the net effect on $Speedup_{eff}$ by customizing precision is positive for most precision configura-

Table 3.7: Power efficiency of the proposed algorithms and platforms for various (M, n) combinations. The efficiency improvement compared to the E5-2660 v2 CPU is shown in parenthesis (separately for Baseline and MPPT). The best platform per algorithm and per parameter combination is shown in bold.

	Accelerator (device)	ES/J	$ES/(J \cdot sec)$
$(M, n)=(8, 128)$	Baseline (E5-2660 v2)	1.5 (1x)	$4.5 \cdot 10^{-1}$ (1x)
	Baseline (C2050)	$9.7 \cdot 10^{-1}$ (0.64x)	$2.4 \cdot 10^{-1}$ (0.53)
	Baseline (VX1140T)	$7.1 \cdot 10^1$ (47x)	$4.9 \cdot 10^2$ (1121x)
	MPPT (E5-2660 v2)	1.5 (1x)	$4.6 \cdot 10^{-1}$ (1x)
	MPPT (C2050)	$8.0 \cdot 10^{-1}$ (0.53x)	$1.7 \cdot 10^{-1}$ (0.36x)
	MPPT (VX1140T)	$1.5 \cdot 10^2$ (106x)	$2.5 \cdot 10^3$ (5629x)
$(M, n)=(32768, 128)$	Baseline (E5-2660 v2)	$2.8 \cdot 10^{-3}$ (1x)	$1.6 \cdot 10^{-6}$ (1x)
	Baseline (C2050)	$2.5 \cdot 10^{-2}$ (8.9x)	$1.7 \cdot 10^{-4}$ (106x)
	Baseline (VX1140T)	$6.7 \cdot 10^{-2}$ (23x)	$4.5 \cdot 10^{-4}$ (281x)
	MPPT (E5-2660 v2)	$3.7 \cdot 10^{-3}$ (1x)	$2.7 \cdot 10^{-6}$ (1x)
	MPPT (C2050)	$6.1 \cdot 10^{-2}$ (16x)	$9.9 \cdot 10^{-4}$ (386x)
	MPPT (VX1140T)	$3.6 \cdot 10^{-1}$ (99x)	$1.3 \cdot 10^{-2}$ (5159x)
$(M, n)=(32, 32768)$	Baseline (E5-2660 v2)	$1.3 \cdot 10^{-2}$ (1x)	$3.7 \cdot 10^{-5}$ (1x)
	Baseline (C2050)	$6.3 \cdot 10^{-2}$ (4.8x)	$1.1 \cdot 10^{-3}$ (29x)
	Baseline (VX1140T)	$2.8 \cdot 10^{-1}$ (22x)	$8.2 \cdot 10^{-3}$ (222x)
	MPPT (E5-2660 v2)	$1.4 \cdot 10^{-2}$ (1x)	$4.2 \cdot 10^{-5}$ (1x)
	MPPT (C2050)	$1.0 \cdot 10^{-1}$ (7.1x)	$2.9 \cdot 10^{-3}$ (71x)
	MPPT (VX1140T)	$10.4 \cdot 10^{-1}$ (74x)	$1.1 \cdot 10^{-1}$ (2701x)

tions.

Precision optimization for MPPT is similar but without the IS-related effect. The auxiliary chains' precision affects the mixing of the first chain, i.e. the term $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}}$ in (3.20). Figure 3.12 shows that $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}}$ (its mean value from 30 independent runs) varies but remains close to one as precision drops until 8 mantissa bits are reached. This stability is due to the fact that exchange moves between chains $1 \leftrightarrow 2$ (which help chain 1 escape from local modes) can succeed even if the density of chain 2 is calculated in low precision. Although samples of the second (and all auxiliary) chains are not “correctly” distributed around the mode centres, all modes still exist and chains traverse from mode to mode due to tempering. This suffices to occasionally supply chain 1 with samples that help it escape from a mode. This behaviour is confirmed by the percentage of successful exchange moves, which remains constant as precision drops. When very low precisions are used, shifts in the modes' positions are observed, leading $\frac{ESS_{mcmc}^{(CP)}}{ESS_{mcmc}^{(DP)}}$ to collapse. Figure 3.12 also shows $Speedup_{raw}$ and $Speedup_{eff}$. The maximum $Speedup_{eff}=138x$ is achieved by configuration (14,11).

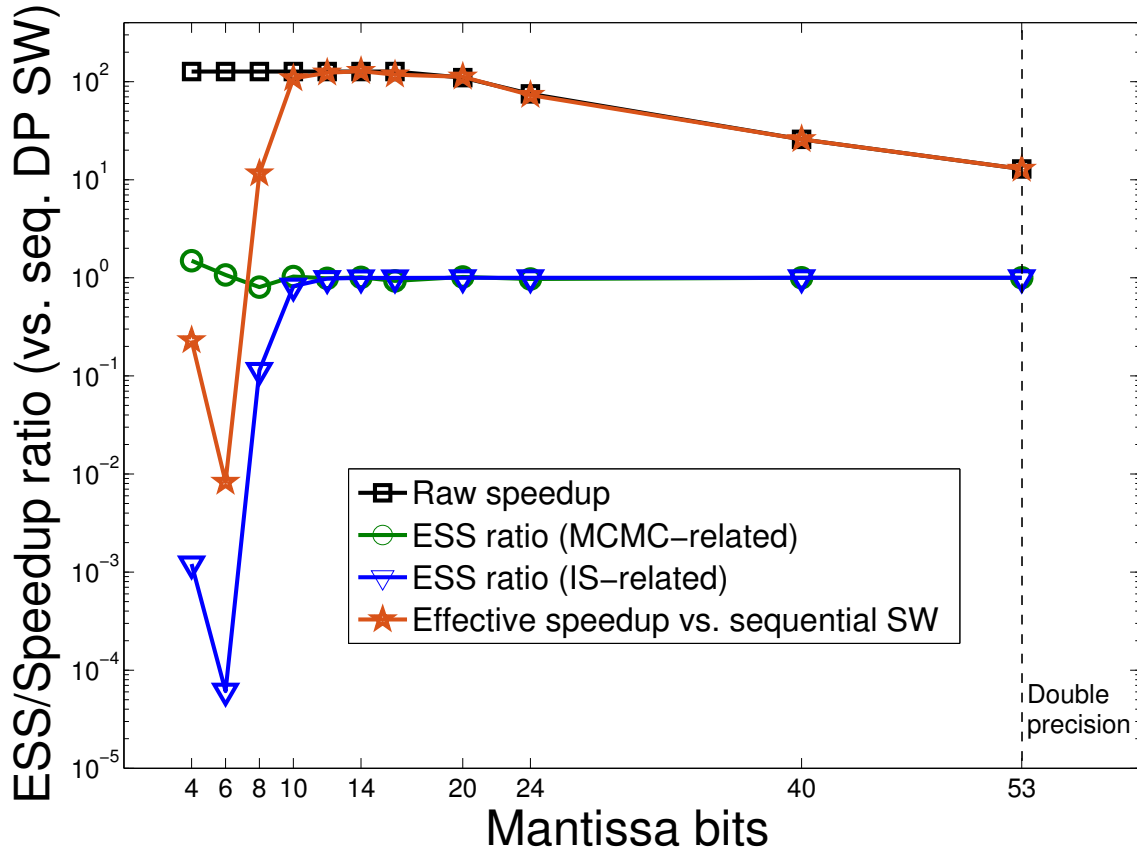


Figure 3.11: WPT on FPGA: The effect of precision on the factors of (3.20). $(M, n) = (128, 128)$. The FPGA device used is the LX240T. The optimal number of mantissa bits is 14. All values in the vertical axis are ratios (either ratios of ESS values or ratios of speedups).

The above optimization shows that both architectures benefit from using custom precision. In the examined case, the optimized $Speedup_{eff}$ of both WPT and MPPT is 4.5x larger than the $Speedup_{eff}$ of the baseline FPGA sampler with no cost in sampling quality. These gains decline when using larger data sets (larger n) as already mentioned in Section 3.6.3, because the drop in ESS ratios happens at larger precisions than the ones shown in the two figures above (where $n = 128$). This forces the optimization to select a larger custom precision, limiting $Speedup_{raw}$ and thus $Speedup_{eff}$. Moreover, a drawback of the optimization process described here is that the optimization has to be done separately for every combination of M , n and FPGA device.

For CPUs and GPUs, WPT and MPPT do not require precision optimization because the only available reduced precision is single precision. In fact, the only precision-related choice is between single and double precision. Of course, using single precision has some effect on the ESS ratios of (3.20). Nevertheless, $Speedup_{eff}$ under single precision on CPUs/GPUs was higher for all parameter combinations and all devices presented above (for both WPT and MPPT). The effect of single precision on

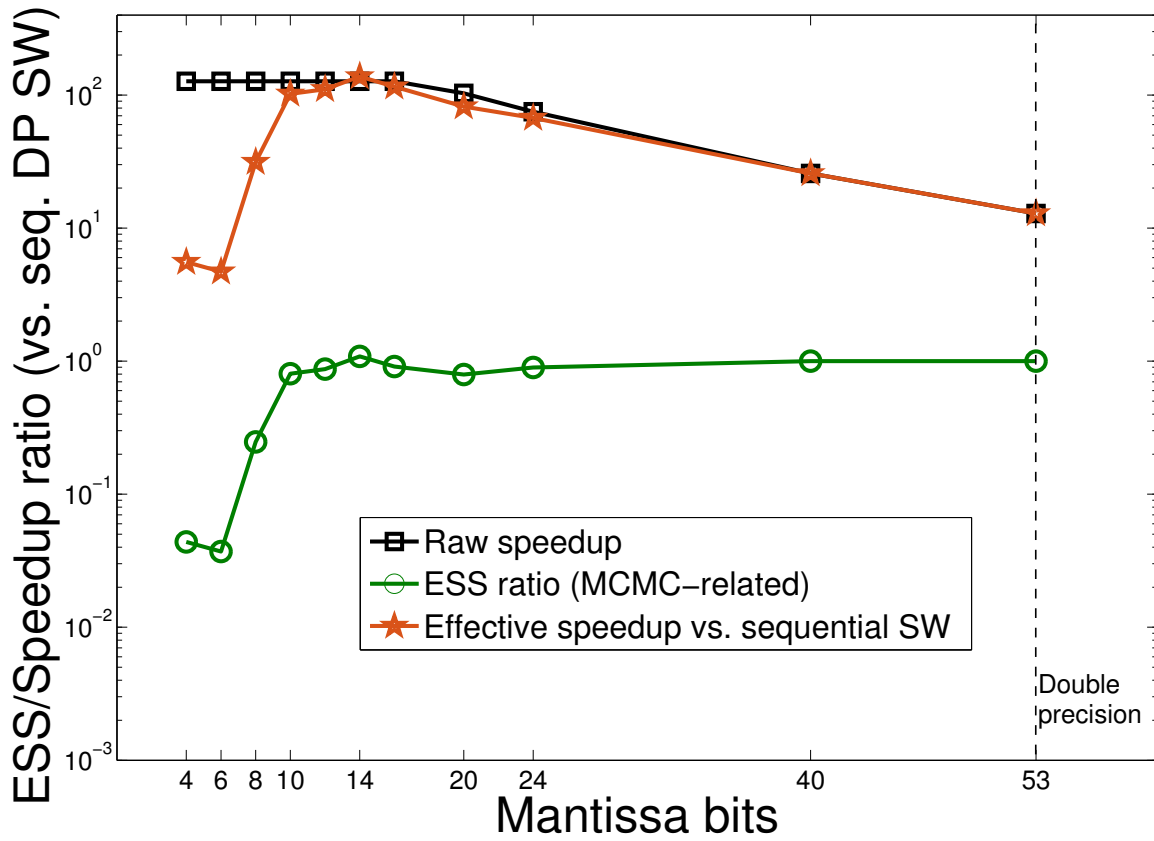


Figure 3.12: MPPT on FPGA: The effect of precision on the factors of (3.20). $(M, n) = (128, 128)$. The FPGA device used is the LX240T. The optimal number of mantissa bits is 14. All values in the vertical axis are ratios (either ratios of ESS values or ratios of speedups)

ESS ratios has been incorporated into all Section 3.6 results.

Kernel optimization for GPUs: As mentioned in Section 3.3.1, the combination of blocks and tasks (sub-densities) per thread in the CUDA kernels of GPU samplers can be optimized to minimize run-time. This optimization is shown in Figure 3.13 for $(M, n) = (8192, 128)$ on the GTX285. Values come from pre-runs. Some combinations are impossible because they lead to more threads per block than the employed CUDA version allows. It is clear from the graph that assigning sufficient tasks per block is necessary to achieve good performance and that processing several tasks per thread is beneficial. This optimization has been applied to all the GPU samplers of the previous section.

3.6.5 FPGA memory considerations

All FPGA implementations have Sample, Probability and Data memories (their sizes given in Table 3.2), which use on-chip memory resources. The on-chip memory of an FPGA is limited. The largest

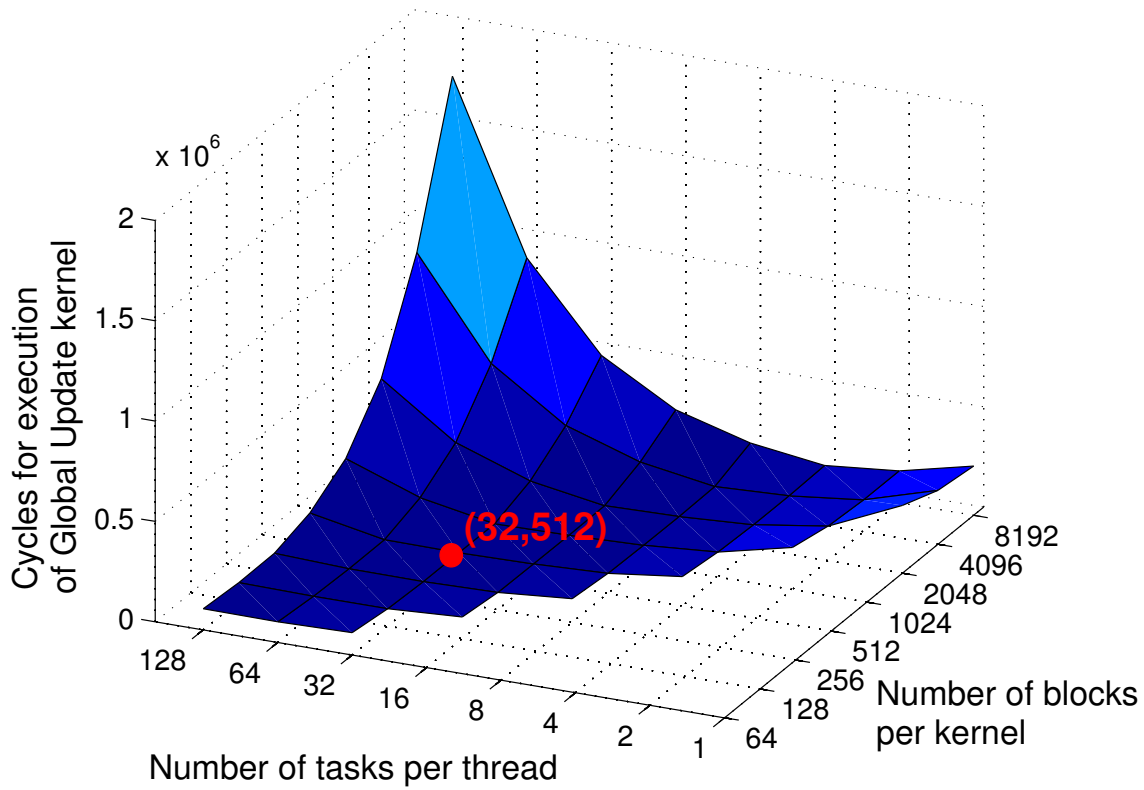


Figure 3.13: GPU kernel optimization: Cycles of global update kernel when the number of blocks and the number of tasks (data) per thread change. The parameter combination used is $(M, n) = (8192, 128)$. The GPU device used is the GTX285. Cycles are minimized when using 512 blocks and 32 tasks per thread.

FPGA used here (VX1140T) has 8.2 MB of memory on chip. When using many chains and large data sets, some or all of these memories have to be moved off chip. Performance can degrade if the off-chip memory throughput is not enough to feed the processing modules. This mainly depends on the ratio of computations over memory accesses in the probability density evaluation. Also, performance can degrade if the FPGA-to-host memory throughput is lower than the rate at which the architecture generates output data (MCMC samples).

For the mixture model examined here this is not an issue. Indicatively, if MPPT is implemented on the LX240T FPGA with $M = 32768$ and $n = 32768$ (a combination not tested in this chapter), the system is able to process 1.3 Global operations per second, based on (3.8) and the number of pipelines that fit in the FPGA. Since not all memories fit in the FPGA, the two largest ones (Samples, Probabilities) have to be implemented as off-chip memory and only the Data memory is kept on chip. The system then needs to read 1.3×32768 samples and 1.3×32768 probabilities per second from off-chip memory (since the Global operation operates on the samples and probabilities of all chains) and

write 1.3×32768 samples and 1.3×32768 probabilities per second back to off-chip memory (since the off-chip memory has to be up to data at all times). This translates to an input memory throughput requirement of 1.62 MB/sec and an output memory throughput requirement of 1.62 MB/sec, which are within the capabilities off-chip memory in FPGA boards. For example, the Xilinx ML605 board with the LX240T device can reach a total I/O throughput of 5.9 GB/sec [91]. Also, other implementations report much higher peak memory throughputs [150]. In contrast, if the ratio of computations over memory accesses is very low (due to a different form of probability density) or if the Data memory is moved off-chip, it is possible that the off-chip memory throughput will not be enough to feed the system, becoming the bottleneck.

Moreover, the PCI-express connection to the host PC needs to be able to transfer the MCMC samples and weights of the first chain as fast as they are generated by the architecture. The worst-case scenario (in contrast to the previous paragraph) is when using a very small problem size ($M = 8$ and $n = 128$). On the LX240T FPGA, the architecture is able to generate first-chain samples very fast in this scenario, i.e. 2.27×10^6 first-chain samples/sec. Each sample comprises four 64 bit numbers, which means that the required PCI-express throughput is close to 70 MB/sec. The RIFFA-based PCI-express connection in the proposed system achieves a throughput of 120 MB/sec, which is sufficient. If the largest FPGA in this chapter is used (VX1140T), the throughput requirement increases to the 382 MB/sec, which cannot be covered by the PCI-express implementation employed here. Nevertheless, this throughput is comfortably within the capabilities of faster PCI-express implementation which use more lanes and/or a more recent generation of the PCI-express protocol.

3.7 Discussion

This chapter focuses on proposing novel FPGA architectures for popMCMC algorithms (more specifically, the popular and representative PT algorithm), introduces methods to exploit custom precision to reduce runtime (without introducing sampling error) and compares the performance of FPGAs to that of other accelerator platforms.

Results showed that the baseline FPGA architecture is faster and more energy efficient than multi-core CPU and GPU accelerators, especially when tackling small and medium-size problems. Moreover, it is clear from the results that custom precision can improve the efficiency of PT without compromising sampling quality, provided that it is used in accordance with the algorithm's properties. Both custom

precision methods proposed in this chapter do this, managing to use reduced precision for the majority of probability density evaluations and limiting double precision computations significantly. The posterior samples of all runs were examined to confirm that the custom precision methods sample from the “correct” posterior. This was the case, as theoretically expected.

Both methods are particularly suitable for mapping on FPGAs, where they deliver up to a 6.5x speedup over the double precision sampler. The gains on CPUs and GPUs are smaller. The use of custom precision in PT in combination with its FPGA implementation is an example of how to combine knowledge about the MCMC method (in this case PT’s tolerance to using inexact computations in parts of the implementation) with understanding of the underlying hardware to maximize efficiency.

The main disadvantage of WPT/MPPT is the way their performance scales with data size. A drop in the effective speedup is observed after a certain point due to error accumulation in the likelihood calculation (although the effective speedup is still higher than that of the GPUs). WPT and MPPT thus lose part (but not all) of their performance advantage over GPUs when handling massive data sets. The strengths and weaknesses of the two methods when mapped on FPGAs are:

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

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

WPT and MPPT can be applied to other sampling scenarios. WPT can be used in any MCMC method in combination with thinning [9], a technique which retains only some of the samples (equivalent to subsampling the MCMC output) to decrease correlation. In such a scenario, a custom precision block could process all samples, while a double precision block could compute weights only for thinned samples (which constitutes a much smaller workload). Moreover, both WPT and MPPT can be applied to other population-based MCMC methods [7, 67], since all of them are based on the idea of utilizing auxiliary chains to improve the mixing of the first chain.

The focus of this chapter is on optimally mapping PT to hardware and exploiting precision. The way performance changes when using different likelihoods is outside the scope of this work. Having said

that, the speedups demonstrated for the mixture model are expected to hold for other models with i.i.d. data, since the same form of parallelism can be exploited. Performance scaling is also expected to be similar to Figures 3.6 and 3.7 when using i.i.d. likelihoods (which is the case in a wide variety of real problems). For non-i.i.d. likelihoods, probability evaluation might require computations which are not SIMD, e.g. [10, 25], so speedups (and the way they scale) depend largely on the problem. The flexibility of FPGA, which can tailor their architecture to a specific type of computation, could prove beneficent in this scenario.

Overall, popMCMC algorithms are particularly suitable for parallel platforms due to their inherently parallel structure. This is the reason why significant speedups over sequential software are possible when implementing them on multi-core CPUs, GPUs and FPGAs. The following chapter will investigate the potential of using FPGA to accelerate a different class of MCMC algorithms (Particle MCMC), which are not as straightforward to parallelize. Moreover, it will extend the approach of co-designing the algorithm and the underlying hardware to this new class. Unlike the present chapter, where the co-design was based on exploiting custom precision, the co-design approach of the next chapter aims to maximize datapath utilization by modifying the algorithm.

Chapter 4

Algorithms and architectures for Particle MCMC

4.1 Introduction

Most MCMC methods, including the Population-based MCMC sampler of Chapter 3, are based on the assumption that the target probability density (i.e. the posterior in Bayesian applications) can be evaluated pointwise up to a multiplicative constant. Nevertheless, this is not possible in several scenarios, notably when performing inference on the joint space of unknown states and unknown parameters in SSMs [33]. The pMCMC method (Algorithm 5 in Chapter 2) is designed to address this challenge. It uses a Particle Filter (PF) inside an MCMC algorithm, in order to generate unbiased estimates of the probability density. Andrieu and Roberts [45] showed that the use of these estimates leads to sampling from the correct SSM posterior.

Despite pMCMC's success since its invention in 2010 [33], its applicability suffers from two main issues: 1) The computational cost of estimating the probability density using a PF is $O(T \cdot P)$ (where T is the number of hidden SSM states and P is the number of particles of the PF) [72]. This becomes prohibitive in modern, big-data applications. Moreover, each pMCMC iteration requires a separate PF run and typically thousands of iterations are needed. Therefore, performing inference for large SSMs [48, 2] is currently impractical. The work in this chapter was initially motivated by such a complex problem; SSMs in genetics, where the number of SSM states, which correspond to DNA bases, can reach millions [14]. 2) The issue of multi-modality in the posterior, which can appear in

certain modelling scenarios but has remained unaddressed in pMCMC literature (for more details on these two issues, see Chapter 2).

In this chapter, the above challenges are tackled using FPGA acceleration and novel algorithmic modifications. The questions that this chapter considers are:

- “How can the inherent parallelism of pMCMC be exploited to achieve high sampling throughput in an FPGA implementation?”
- “Can the pMCMC algorithm be enhanced to increase its efficiency for multi-modal posteriors?”
- “How can these enhancements be combined with extra hardware optimizations in order to further boost performance?”

The results presented in this chapter make it clear that FPGAs can outperform other hardware platforms when running pMCMC and that a new algorithm (which is proposed here) increases sampling efficiency for multi-modal targets, while possessing favourable properties for mapping on FPGAs.

Chapter outline

Section 4.2 recalls the main equations and algorithms related to SSMs, PFs and pMCMC, which were first presented in Chapter 2. The remaining sections contain the contributions of the chapter:

1. A new algorithm which combines existing MCMC methods (popMCMC and pMCMC) to address target densities which are both analytically intractable and multi-modal. The algorithm (named Population-based Particle MCMC - ppMCMC) extends pMCMC by using a population of pMCMC chains instead of a single chain (in the same way that popMCMC uses multiple chains) in order to improve mixing (section 4.3). The section also justifies the correctness of the ppMCMC algorithm, i.e. it shows why it converges to the desired posterior distribution.
2. A novel FPGA architecture for pMCMC, which exploits the parallelism in the three steps of the PF to achieve high sampling throughput. (section 4.4).
3. An FPGA architecture for ppMCMC which is based on the baseline pMCMC architecture but provides support for multiple chains. The existence of multiple chains is exploited to increase the utilization of the PF data path through coarse-grain pipelining. An analytical performance model is introduced in order to quantify the speedup gains from such a strategy (section 4.4).

The new algorithm and the two FPGA accelerators are applied to large-scale SSM inference in genetics. More specifically, an SSM model of DNA methylation with unknown parameters is targeted. This model can lead to uni-modal or multi-modal posteriors. The sampling throughput of the FPGA is compared with the sampling throughput of state-of-the-art, optimized CPU and GPU pMCMC accelerators [2] which are applied to the same problem. Moreover, for the multi-modal posterior case, the trade-off between the number of MCMC chains and the number of PF particles in ppMCMC is explored in order to maximize sampling throughput (Section 4.7).

4.2 State-space models and Particle MCMC

All the necessary background on SSMs and pMCMC can be found in Section 2.3.3 of Chapter 2. Here, the main equations and algorithms are repeated for easier reference. The three SSM equations are the following:

$$\mathbf{X}_1 \sim p(\mathbf{X}_1) \quad (4.1)$$

$$\mathbf{X}_t \sim p(\mathbf{X}_t | \mathbf{X}_{t-1}, \theta), \quad t > 1 \quad (4.2)$$

$$\mathbf{Y}_t \sim p(\mathbf{Y}_t | \mathbf{X}_t, \theta), \quad t > 0 \quad (4.3)$$

The posterior when performing state estimation (with known parameters θ) in SSMs is the following:

$$p_\theta(\mathbf{X}_{1:T} | \mathbf{Y}_{1:T}) \propto p_\theta(\mathbf{X}_{1:T}) p_\theta(\mathbf{Y}_{1:T} | \mathbf{X}_{1:T}) = p(\mathbf{X}_1) \left(\prod_{t=2}^T p_\theta(\mathbf{X}_t | \mathbf{X}_{t-1}) \right) \left(\prod_{t=1}^T p_\theta(\mathbf{Y}_t | \mathbf{X}_t) \right) \quad (4.4)$$

The Bayesian posterior when performing joint state and parameter estimation is the following:

$$\begin{aligned} p(\mathbf{X}_{1:T}, \theta | \mathbf{Y}_{1:T}) &\propto p(\theta) p(\mathbf{X}_{1:T} | \theta) p(\mathbf{Y}_{1:T} | \mathbf{X}_{1:T}, \theta) \\ &= p(\theta) p(\mathbf{X}_1) \left(\prod_{t=2}^T p(\mathbf{X}_t | \mathbf{X}_{t-1}, \theta) \right) \left(\prod_{t=1}^T p(\mathbf{Y}_t | \mathbf{X}_t, \theta) \right) \end{aligned} \quad (4.5)$$

The proposal density of pMCMC is the following:

$$q((\mathbf{X}_{1:T}^*, \theta^*) | (\mathbf{X}_{1:T}, \theta)) = q(\theta^* | \theta) p(\mathbf{X}_{1:T}^* | \mathbf{Y}_{1:T}, \theta^*) \quad (4.6)$$

The acceptance ratio of pMCMC is the following:

$$\tilde{a} = \frac{p(\theta^*) \tilde{p}(\mathbf{Y}_{1:T}|\theta^*) q(\theta|\theta^*)}{p(\theta) \tilde{p}(\mathbf{Y}_{1:T}|\theta) q(\theta^*|\theta)} \quad (4.7)$$

The bootstrap PF and pMCMC algorithms are shown in Algorithms 7 and 8 respectively:

Algorithm 7 Bootstrap Particle Filter

```

1: procedure BOOTSTRAPPF( $P, T, \theta, \mathbf{Y}_{1:T}$ ) - Inputs:  $P$  (number of particles),  $T$  (number of SSM
   states),  $\theta$  (parameter values for transition and observation densities),  $\mathbf{Y}_{1:T}$  (observations)
2: Initial state ( $t = 1$ ):
3:   for  $k = 1, \dots, P$  do
4:     Sample particle from initial density  $\tilde{\mathbf{X}}_1^k \sim p(\mathbf{X}_1)$  // sampling step
5:   for  $k = 1, \dots, P$  do
6:     Calculate initial weight  $W_1^k \leftarrow p_\theta(\mathbf{Y}_1 | \tilde{\mathbf{X}}_1^k)$  // weight step
7: Remaining states:
8:   for  $t = 2, \dots, T$  do
9:     for  $k = 1, \dots, P$  do
10:      Sample ancestor index  $a^k$  from  $\{1, \dots, P\}$  with probabilities proportional to
11:       $\{W_{t-1}^1, \dots, W_{t-1}^P\}$  and set resampled particle  $\mathbf{X}_{t-1}^k \leftarrow \tilde{\mathbf{X}}_{t-1}^{a^k}$  // resampling step
12:     for  $k = 1, \dots, P$  do
13:      Sample particle from transition density  $\tilde{\mathbf{X}}_t^k \sim p_\theta(\mathbf{X}_t | \mathbf{X}_{t-1}^k)$  // sampling step
14:     for  $k = 1, \dots, P$  do
15:      Calculate weight  $W_t^k \leftarrow p_\theta(\mathbf{Y}_t | \tilde{\mathbf{X}}_t^k)$  // weight step
16: Likelihood estimate:
17:    $L \leftarrow \prod_{t=1}^T \left( \frac{1}{P} \sum_{k=1}^P W_t^k \right)$ 
18: return ( $L, \mathbf{X}_{1:T}^{1:P} = \{\mathbf{X}_1^{1:P}, \dots, \mathbf{X}_T^{1:P}\}$ ) (likelihood estimate and  $P$  particles for every SSM state)

```

4.3 ppMCMC: A new population-based pMCMC method

This section presents a novel MCMC algorithm, which is a combination of two existing MCMC methods (popMCMC and pMCMC). The new algorithm is called Population-based Particle MCMC (ppMCMC) and its purpose is to address problems where the SSM posterior presented in Section 2.3.3, i.e. equation (4.5) of this chapter, is multi-modal. Multi-modality here is tackled only for the marginal posterior of θ (i.e. the posterior marginalised over the states). In these cases, the standard pMCMC algorithm faces the well-known issue of slow mixing, which is common to all single-chain MCMC methods. The sampler tends to “get stuck” in one of the modes of the posterior and only rarely manages to move to a different mode. Some of the modes might never be visited unless the sampler runs for a very long time, i.e. the runtime becomes impractical for real-world use by practitioners. Thus the user cannot be certain that the posterior has been explored completely. Of course, full exploration

Algorithm 8 Particle MCMC

```

1: procedure pMCMC( $P, T, \mathbf{Y}_{1:T}, N, \theta^{init}$ ) - Inputs:  $P$  (number of particles),  $T$  (number of SSM
   states),  $\mathbf{Y}_{1:T}$  (observations),  $N$  (number of MCMC samples),  $\theta^{init}$  (initial MCMC sample)
2: First iteration ( $i = 1$ ):
3:    $(\tilde{p}(\mathbf{Y}_{1:T} | \theta^{init}), \mathbf{X}_{1:T}^{1:P}) \leftarrow \text{BootstrapPF}(P, T, \theta^{init}, \mathbf{Y}_{1:T})$  // get likelihood and state samples
4:   Randomly select an index  $p$  from  $\{1, \dots, P\}$  and set  $\mathbf{X}_{1:T}^{init} = \mathbf{X}_{1:T}^p$ 
5:    $\text{Sample}[1] = (\theta^{init}, \mathbf{X}_{1:T}^{init})$  // save initial sample
6:    $\text{Posterior}[1] = p(\theta^{init}) \tilde{p}(\mathbf{Y}_{1:T} | \theta^{init})$  // compute and save posterior
7:    $\theta = \theta^{init}$  // temporary variable
8: Remaining iterations:
9:   for  $i = 2, \dots, N$  do
10:     $\theta^* \sim q(\theta^* | \theta)$  // propose new  $\theta$ 
11:     $(\tilde{p}(\mathbf{Y}_{1:T} | \theta^*), \mathbf{X}_{1:T}^{1:P}) \leftarrow \text{BootstrapPF}(P, T, \theta^*, \mathbf{Y}_{1:T})$  // get likelihood and state samples
12:    Randomly select an index  $p$  from  $\{1, \dots, P\}$  and set  $\mathbf{X}_{1:T}^* = \mathbf{X}_{1:T}^p$ 
13:    Accept proposed sample  $(\theta^*, \mathbf{X}_{1:T}^*)$  with probability  $\min(1, \tilde{a})$ 
14:    if accepted then
15:       $\text{Sample}[i] = (\theta^*, \mathbf{X}_{1:T}^*)$  // save proposed sample
16:       $\text{Posterior}[i] = p(\theta^*) \tilde{p}(\mathbf{Y}_{1:T} | \theta^*)$  // compute and save posterior
17:       $\theta = \theta^*$  // temporary variable
18:    else
19:       $\text{Sample}[i] = \text{Sample}[i-1]$  // replicate previous sample
20:       $\text{Posterior}[i] = \text{Posterior}[i-1]$  // replicate previous posterior
21: return  $(\text{Sample}[1:N], \text{Posterior}[1:N])$  ( $N$  sets of MCMC samples and posterior values)

```

is never certain, even when using an algorithm with superior mixing. Nevertheless, improving mixing makes it more likely that the modes will be found in a reasonable time frame (i.e. days or weeks).

Although most applications of pMCMC lead to uni-modal marginal posteriors for θ , there are modelling scenarios, like the one described in section 4.5, where the posterior admits multiple modes. For example, the methylation profile of whole blood consists of multiple methylation profiles originating in the different cell types that exist in the blood [14]. In order to model this using an SSM, the transition density needs to be a mixture of densities with multiple unknown parameters. Inference on mixtures leads to multi-modal posteriors.

The ppMCMC algorithm is a typical population-based MCMC method: It employs a population of M MCMC chains, where each chain samples from a different target distribution, i.e. a modified version of the SSM posterior in equation (4.5). The differences between the chains' target distributions are due to the use of tempering (in a PT-like fashion, see Chapter 3); each chain uses a separate PF to approximate its likelihood (like in the pMCMC algorithm) but the likelihoods are then tempered. Also, ppMCMC uses exchange moves between the chains in a predefined order (again in the same way as PT). The combination of tempering and exchanges results in better mixing for multi-modal

distributions but also introduces complications (which will be explained shortly). The two actions that take place during each ppMCMC iteration are the update moves and the exchange moves:

Update Moves

During the update moves stage, each chain proposes candidate MCMC samples $(\mathbf{X}_{1:T}^*, \theta^*)$ (for its own unique, modified SSM posterior) using the proposal:

$$q_j((\mathbf{X}_{1:T}^*, \theta^*) | (\mathbf{X}_{1:T}, \theta)) = q_j(\theta^* | \theta) p(\mathbf{X}_{1:T}^* | \mathbf{Y}_{1:T}, \theta^*), \quad j \in \{1, \dots, M\} \quad (4.8)$$

where j is the chain index, $(\mathbf{X}_{1:T}, \theta)$ is the previous sample of the chain, $q_j((\mathbf{X}_{1:T}^*, \theta^*) | (\mathbf{X}_{1:T}, \theta))$ is the full MCMC proposal density (for states and θ) for chain j and $q_j(\theta^* | \theta)$ is the component of the proposal density which is related to θ for chain j . This component can be different between chains, in order to account for the different target densities of each chain, i.e. chains with more diffuse target densities mix faster when their proposal has larger variance. The component of the proposal density which is related to states ($p(\mathbf{X}_{1:T}^* | \mathbf{Y}_{1:T}, \theta^*)$) is the same for all chains. The proposed states are generated by a PF exactly as in pMCMC. The only difference between the proposal schemes of pMCMC and ppMCMC is the use of different variances in the θ component of the proposal. Note that here, and in the remaining of the section (except the pseudo-code), the subscript j is dropped from the candidate and previous samples of the chains to simplify notation.

After the candidate samples have been proposed, they are accepted or rejected based on the following M-H acceptance ratio:

$$\tilde{a}_j = \frac{p(\theta^*) \tilde{p}(\mathbf{Y}_{1:T} | \theta^*)^{\frac{1}{Temp_j}} p(\mathbf{X}_{1:T}^* | \mathbf{Y}_{1:T}, \theta^*) q_j(\theta | \theta^*) p(\mathbf{X}_{1:T} | \mathbf{Y}_{1:T}, \theta)}{p(\theta) \tilde{p}(\mathbf{Y}_{1:T} | \theta)^{\frac{1}{Temp_j}} p(\mathbf{X}_{1:T} | \mathbf{Y}_{1:T}, \theta) q_j(\theta^* | \theta) p(\mathbf{X}_{1:T}^* | \mathbf{Y}_{1:T}, \theta^*)} = \frac{p(\theta^*) \tilde{p}(\mathbf{Y}_{1:T} | \theta^*)^{\frac{1}{Temp_j}} q_j(\theta | \theta^*)}{p(\theta) \tilde{p}(\mathbf{Y}_{1:T} | \theta)^{\frac{1}{Temp_j}} q_j(\theta^* | \theta)}, \quad j \in \{1, \dots, M\} \quad (4.9)$$

where notation is the same as in the standard pMCMC method and $Temp_j$ is the temperature of chain j (with $1 = Temp_1 < Temp_2 < \dots < Temp_M < \infty$). The above ratio is different from the standard pMCMC ratio in equation (4.7), since the estimated likelihood $\tilde{p}(\mathbf{Y}_{1:T} | \theta)$ is tempered. This is done in order to achieve the posterior smoothing effect of all tempering schemes: The first chain of the population has $Temp_1 = 1$, which means that it is the only chain that is not tempered. The chain samples from the exact same SSM posterior as a typical pMCMC algorithm (the “correct” posterior). The auxiliary chains of the population ($j \in \{2, \dots, M\}$) are tempered, which means that they sample from some smoothed (closer to uniform) version of the “correct” SSM posterior. As can be seen in

the acceptance equation, the temperatures are applied only to the likelihood $\tilde{p}(\mathbf{Y}_{1:T} \mid \boldsymbol{\theta})$ (and not the other terms in the left-most part of the equation). This is a typical approach in tempering MCMC algorithms, since the likelihood is the component that usually contributes the most in the posterior shape. Moreover, it is crucial to apply the temperature only to this component in order to simplify the implementation of exchange steps (as will be shown below).

From examining equation (4.9), it is clear that no changes to the Bootstrap PF (compared to the pMCMC case) are needed in order to implement the ratio and temper the likelihoods; the PF of each chain generates a sample $\mathbf{X}_{1:T}^*$ from $p(\mathbf{X}_{1:T}^* \mid \mathbf{Y}_{1:T}, \boldsymbol{\theta}^*)$ which is used in the proposal. It also generates an unbiased estimate $\tilde{p}(\mathbf{Y}_{1:T} \mid \boldsymbol{\theta}^*)$ of the “correct” (non-tempered) likelihood. The temperature is applied after the termination of the PF. The terms $p(\mathbf{X}_{1:T}^* \mid \mathbf{Y}_{1:T}, \boldsymbol{\theta}^*)$ and $p(\mathbf{X}_{1:T} \mid \mathbf{Y}_{1:T}, \boldsymbol{\theta})$ are cancelled out in the acceptance ratio (as in basic pMCMC).

What are the target distributions?

Although applying a temperature to the likelihood is a well-known technique in population-based methods, in the case of ppMCMC it is not clear what the target distribution of each tempered chain is. The term $p(\boldsymbol{\theta})\tilde{p}(\mathbf{Y}_{1:T} \mid \boldsymbol{\theta})^{\frac{1}{\text{Temp}_j}} p(\mathbf{X}_{1:T} \mid \mathbf{Y}_{1:T}, \boldsymbol{\theta})$ (which contains the likelihood estimate) is used in the acceptance ratio of chain j but this does not lead the chain to converge to the posterior $p(\boldsymbol{\theta})p(\mathbf{Y}_{1:T} \mid \boldsymbol{\theta})^{\frac{1}{\text{Temp}_j}} p(\mathbf{X}_{1:T} \mid \mathbf{Y}_{1:T}, \boldsymbol{\theta})$, as one would intuitively expect after comparing to the pMCMC case.

According to the theory presented in [45], a pMCMC chain converges to a target distribution provided that unbiased estimates of the distribution’s density are used in the numerator and denominator of the acceptance ratio. Nevertheless, the term $p(\boldsymbol{\theta})\tilde{p}(\mathbf{Y}_{1:T} \mid \boldsymbol{\theta})^{\frac{1}{\text{Temp}_j}} p(\mathbf{X}_{1:T} \mid \mathbf{Y}_{1:T}, \boldsymbol{\theta})$ is not an unbiased estimator of $p(\boldsymbol{\theta})p(\mathbf{Y}_{1:T} \mid \boldsymbol{\theta})^{\frac{1}{\text{Temp}_j}} p(\mathbf{X}_{1:T} \mid \mathbf{Y}_{1:T}, \boldsymbol{\theta})$: Running a PF on the given SSM produces an unbiased estimate $\tilde{p}(\mathbf{Y}_{1:T} \mid \boldsymbol{\theta})$ of the likelihood (i.e. $\mathbb{E}[\tilde{p}(\mathbf{Y}_{1:T} \mid \boldsymbol{\theta})] = p(\mathbf{Y}_{1:T} \mid \boldsymbol{\theta})$). However, applying the temperature after the likelihood estimate is generated (i.e. finding $\tilde{p}(\mathbf{Y}_{1:T} \mid \boldsymbol{\theta})^{\frac{1}{\text{Temp}_j}}$) does not maintain unbiasedness with respect to the “correct” tempered likelihood (i.e. $p(\mathbf{Y}_{1:T} \mid \boldsymbol{\theta})^{\frac{1}{\text{Temp}_j}}$).

In more detail, because the function $x \mapsto x^{\frac{1}{\text{Temp}_j}}$ is concave for $\text{Temp}_j \geq 1$, applying Jensen’s inequality [151] leads to the following:

$$\mathbb{E}[\tilde{p}(\mathbf{Y}_{1:T} \mid \boldsymbol{\theta})^{\frac{1}{\text{Temp}_j}}] \leq \mathbb{E}[\tilde{p}(\mathbf{Y}_{1:T} \mid \boldsymbol{\theta})]^{\frac{1}{\text{Temp}_j}} = p(\mathbf{Y}_{1:T} \mid \boldsymbol{\theta})^{\frac{1}{\text{Temp}_j}} \quad (4.10)$$

This is true for all ppMCMC chains since all temperatures are greater or equal to one (for $\text{Temp}_j \leq 1$,

the function is convex and the inequality is reversed). Equality holds only for $Temp_j = 1$. Therefore, unbiased estimates of the “correct” tempered likelihood densities $p(\mathbf{Y}_{1:T} | \boldsymbol{\theta})^{\frac{1}{Temp_j}}$ (and thus the respective posterior densities) can be acquired only when $Temp_j = 1$ (i.e. only in the case of the first chain). Nevertheless, this does not mean that the tempered chains do not converge to *any* target distribution. In fact, chain j converges to the distribution whose density is unbiasedly estimated by $p(\boldsymbol{\theta})\tilde{p}(\mathbf{Y}_{1:T} | \boldsymbol{\theta})^{\frac{1}{Temp_j}} p(\mathbf{X}_{1:T} | \mathbf{Y}_{1:T}, \boldsymbol{\theta})$. The densities of these distribution can be written as:

$$p_j(\mathbf{X}_{1:T}, \boldsymbol{\theta} | \mathbf{Y}_{1:T}) = p(\boldsymbol{\theta}) \mathbb{E}[\tilde{p}(\mathbf{Y}_{1:T} | \boldsymbol{\theta})^{\frac{1}{Temp_j}}] p(\mathbf{X}_{1:T} | \mathbf{Y}_{1:T}, \boldsymbol{\theta}), \quad j \in \{1, \dots, M\} \quad (4.11)$$

These are the actual target densities of the M chains of the ppMCMC algorithm. Only for the first chain the density is equal to the “correct” tempered posterior (with $Temp_1 = 1$) and also to the “correct” SSM posterior, i.e. $p_j(\mathbf{X}_{1:T}, \boldsymbol{\theta} | \mathbf{Y}_{1:T}) = p(\mathbf{X}_{1:T}, \boldsymbol{\theta} | \mathbf{Y}_{1:T})$.

The key point here is that it is not necessary for the auxiliary chains to sample from the set of “correct” tempered posteriors, since their samples are not kept. Only the samples of the first chain are kept because they are the ones distributed according to the desired, “correct” SSM posterior. The auxiliary chains are only employed to help the first chain mix faster; they need to explore the distribution space quickly (and therefore their target distributions need to be closer to uniform) and occasionally feed the first chain with samples through exchange moves. These samples help the first chain escape from local modes. It is therefore enough for the auxiliary chains to sample from *some* set of tempered versions of the SSM posterior (and not necessarily from the “correct” set of tempered posteriors). The densities in equation (4.11) provide this tempering effect and therefore fulfil their purpose, i.e. they move fast in the distribution space and help the first chain mix faster through exchange moves. In fact, the term “correct” is only used here for reasons of clarity; there is no reason to believe that the “correct” densities $p(\mathbf{Y}_{1:T} | \boldsymbol{\theta})^{\frac{1}{Temp_j}}$ are the best candidates for use in auxiliary chains (with respect to the mixing gains they offer). On the other hand, this does not mean that *any* density would serve as a good auxiliary density, e.g. uniform auxiliary densities would not help the mixing of the first chain because they are not concentrated around the true modes. In other words, *some* (not complete) smoothing must be applied to the true densities but the exact form of the *optimal* auxiliary densities is not known. In practical situations, the temperature set is tuned (using pre-runs) to improve mixing as much as possible.

The above approach is similar to the MPPT custom precision technique presented in Chapter 3 for the PT algorithm; in that case, custom precision approximations of the auxiliary chains’ target densities

were used instead of the “correct” tempered densities. This did not affect the target distribution of the first chain and also proved effective for improving mixing (for most precisions).

Exchange Moves

In every ppMCMC iteration, after all chains have finished the update moves, the exchange step is performed. Exchange moves are attempted between chain pairs $(1,2), (3,4), \dots$ or chain pairs $(2,3), (4,5), \dots$ (neighbouring chains) in a rotating manner. As mentioned above, the exchange moves push MCMC samples from the high-temperature chains, which are closer to the uniform distribution, to the lower-temperature chains, which are closer to the “correct” target distribution. Eventually samples reach the first chain which samples from the “correct distribution” and help it escape from local modes. The exchange acceptance ratio between chains (q, r) is:

$$\begin{aligned} \tilde{e}_q &= \frac{p(\theta^r) \tilde{p}(\mathbf{Y}_{1:T}|\theta^r)^{\frac{1}{\text{Temp}_q}} p(\mathbf{X}_{1:T}^r|\mathbf{Y}_{1:T}, \theta^r) p(\theta^q) \tilde{p}(\mathbf{Y}_{1:T}|\theta^q)^{\frac{1}{\text{Temp}_r}} p(\mathbf{X}_{1:T}^q|\mathbf{Y}_{1:T}, \theta^q)}{p(\theta^q) \tilde{p}(\mathbf{Y}_{1:T}|\theta^q)^{\frac{1}{\text{Temp}_q}} p(\mathbf{X}_{1:T}^q|\mathbf{Y}_{1:T}, \theta^q) p(\theta^r) \tilde{p}(\mathbf{Y}_{1:T}|\theta^r)^{\frac{1}{\text{Temp}_r}} p(\mathbf{X}_{1:T}^r|\mathbf{Y}_{1:T}, \theta^r)} \\ &= \frac{\tilde{p}(\mathbf{Y}_{1:T}|\theta^r)^{\frac{1}{\text{Temp}_q}} \tilde{p}(\mathbf{Y}_{1:T}|\theta^q)^{\frac{1}{\text{Temp}_r}}}{\tilde{p}(\mathbf{Y}_{1:T}|\theta^q)^{\frac{1}{\text{Temp}_q}} \tilde{p}(\mathbf{Y}_{1:T}|\theta^r)^{\frac{1}{\text{Temp}_r}}} \end{aligned} \quad (4.12)$$

where again the PF likelihood estimates are used, $q \in \{1, \dots, M-1\}$, $r = q+1$ and $(\mathbf{X}_{1:T}^q, \theta^q)$ and $(\mathbf{X}_{1:T}^r, \theta^r)$ are the current samples of chains q and r respectively. The above equation shows why it is important to apply the tempering technique only to the likelihood $p(\mathbf{Y}_{1:T} | \theta)$ and not to the term $p(\mathbf{X}_{1:T} | \mathbf{Y}_{1:T}, \theta)$; it allows the latter to cancel out in the exchange acceptance ratio and leads to the simple form in the second line of the equation, which requires no additional PF runs (all the values are already known from the preceding update step).

It is important to justify why the above exchange move fulfils the requirements of the theory of pMCMC [45] with regards to maintaining the correct target distributions of the two chains. The exchange step is equivalent to a Metropolis update where the updated state is the joint state of both chains (with indexes q and r). According to Andrieu and Roberts [45], a Metropolis update maintains the target distribution as long as the numerator and denominator of the acceptance ratio are unbiased estimates of the target density. In the case of the exchange step (and focusing only on the numerator for simplicity), this means that the product $p(\theta^r) \tilde{p}(\mathbf{Y}_{1:T} | \theta^r)^{\frac{1}{\text{Temp}_q}} p(\mathbf{X}_{1:T}^r | \mathbf{Y}_{1:T}, \theta^r) p(\theta^q) \tilde{p}(\mathbf{Y}_{1:T} | \theta^q)^{\frac{1}{\text{Temp}_r}} p(\mathbf{X}_{1:T}^q | \mathbf{Y}_{1:T}, \theta^q)$ has to be an unbiased estimate of the product of the target densities of the two chains (which

were given in equation (4.11)). It is easy to show that this is the case, since:

$$\begin{aligned}
& \mathbb{E}[p(\theta^r) \tilde{p}(\mathbf{Y}_{1:T} | \theta^r)^{\frac{1}{T_{\text{temp}q}}} p(\mathbf{X}_{1:T}^r | \mathbf{Y}_{1:T}, \theta^r) p(\theta^q) \tilde{p}(\mathbf{Y}_{1:T} | \theta^q)^{\frac{1}{T_{\text{temp}r}}} p(\mathbf{X}_{1:T}^q | \mathbf{Y}_{1:T}, \theta^q)] \\
&= p(\theta^r) p(\mathbf{X}_{1:T}^r | \mathbf{Y}_{1:T}, \theta^r) p(\theta^q) p(\mathbf{X}_{1:T}^q | \mathbf{Y}_{1:T}, \theta^q) \mathbb{E}[\tilde{p}(\mathbf{Y}_{1:T} | \theta^r)^{\frac{1}{T_{\text{temp}q}}} \tilde{p}(\mathbf{Y}_{1:T} | \theta^q)^{\frac{1}{T_{\text{temp}r}}}] \\
&= p(\theta^r) p(\mathbf{X}_{1:T}^r | \mathbf{Y}_{1:T}, \theta^r) p(\theta^q) p(\mathbf{X}_{1:T}^q | \mathbf{Y}_{1:T}, \theta^q) \mathbb{E}[\tilde{p}(\mathbf{Y}_{1:T} | \theta^r)^{\frac{1}{T_{\text{temp}q}}}] \mathbb{E}[\tilde{p}(\mathbf{Y}_{1:T} | \theta^q)^{\frac{1}{T_{\text{temp}r}}}] \\
&= p_q(\mathbf{X}_{1:T}, \theta^r | \mathbf{Y}_{1:T}) p_r(\mathbf{X}_{1:T}, \theta^q | \mathbf{Y}_{1:T})
\end{aligned} \tag{4.13}$$

The first equality is true because the first four terms in the second line of the equation are zero-variance estimators. The second equality is true because the two estimates $\tilde{p}(\mathbf{Y}_{1:T} | \theta^r)^{\frac{1}{T_{\text{temp}q}}}$ and $\tilde{p}(\mathbf{Y}_{1:T} | \theta^q)^{\frac{1}{T_{\text{temp}r}}}$ are independent estimators (since they are generated by two independent PFs, each assigned to its own MCMC chain) and therefore the expectation of their product is equal to the product of their expectations. The final equality is true due to equation (4.11).

The ppMCMC algorithm

The pseudo-code of ppMCMC is shown in Algorithm 9. All the differences compared to Algorithm 8 are included, i.e. multiple chains, different proposals and updates, exchange moves. They are all based on the equations presented previously. Note that the update and exchange ratios in the pseudo-code use slightly different notation compared to the previously derived equations for ease of presentation. Apart from the inputs of pMCMC, ppMCMC requires also the number of chains, the initial samples of each chain and the temperature of each chain. The output includes the samples and the posterior density values of the first chain.

4.4 FPGA architectures for pMCMC and ppMCMC

This section presents novel FPGA architectures for the pMCMC and ppMCMC algorithms. For the ppMCMC architecture, a performance model is described for the case when coarse-grain pipelining of the multi-chain computations is used to enhance throughput.

4.4.1 Parallelism in the algorithms

The available parallelism of pMCMC is P , since all particles inside the PF can be processed in parallel. The N iterations of pMCMC and the T states of the SSM are strictly sequential. In ppMCMC the

Algorithm 9 Population-based Particle MCMC

```

1: procedure PPMCMC( $P, T, \mathbf{Y}_{1:T}, N, M, \theta_{1:M}^{init}, Temp_{1:M}$ ) - Inputs:  $P$  (number of particles),  $T$ 
   (number of SSM states),  $\mathbf{Y}_{1:T}$  (observations),  $N$  (number of MCMC samples),  $M$  (number of
   chains),  $\theta_{1:M}^{init}$  (initial MCMC samples for all chains),  $Temp_{1:M}$  (chain temperatures)
2: First iteration ( $i = 1$ ):
3:   for  $j = 1, \dots, M$  do
4:      $(\tilde{p}(\mathbf{Y}_{1:T} | \theta_j^{init}), \mathbf{X}_{1:T}^{1:P}) \leftarrow \text{BootstrapPF}(P, T, \theta_j^{init}, \mathbf{Y}_{1:T})$  // get lik. & states (chain  $j$ )
5:     Randomly select an index  $p$  from  $\{1, \dots, P\}$  and set  $\mathbf{X}_{1:T}^{init} \leftarrow \mathbf{X}_{1:T}^p$ 
6:      $Sample[j][1] \leftarrow (\theta_j^{init}, \mathbf{X}_{1:T}^{init})$  // save initial sample (chain  $j$ )
7:      $Posterior[j][1] \leftarrow p(\theta_j^{init}) \left( \tilde{p}(\mathbf{Y}_{1:T} | \theta_j^{init})^{\frac{1}{Temp_j}} \right)$  // compute and save posterior (chain  $j$ )
8:      $\theta_j \leftarrow \theta_j^{init}$  // temporary variable (chain  $j$ )
9: Remaining iterations:
10:  for  $i = 2, \dots, N$  do
11:    Chain updates:
12:    for  $j = 1, \dots, M$  do
13:       $\theta^* \sim q_j(\theta^* | \theta_j)$  // propose new  $\theta$  (chain  $j$ )
14:       $(\tilde{p}(\mathbf{Y}_{1:T} | \theta^*), \mathbf{X}_{1:T}^{1:P}) \leftarrow \text{BootstrapPF}(P, T, \theta^*, \mathbf{Y}_{1:T})$  // get lik. and states (chain  $j$ )
15:      Randomly select an index  $p$  from  $\{1, \dots, P\}$  and set  $\mathbf{X}_{1:T}^* = \mathbf{X}_{1:T}^p$ 
16:      Accept proposed sample  $(\theta^*, \mathbf{X}_{1:T}^*)$  with probability
      
$$\min\left(1, \frac{p(\theta^*) \tilde{p}(\mathbf{Y}_{1:T} | \theta^*)^{\frac{1}{Temp_j}} q_j(\theta_j | \theta^*)}{p(\theta_j) \tilde{p}(\mathbf{Y}_{1:T} | \theta_j)^{\frac{1}{Temp_j}} q_j(\theta^* | \theta_j)}\right)$$

17:      if accepted then
18:         $Sample[j][i] = (\theta^*, \mathbf{X}_{1:T}^*)$  // save proposed sample (chain  $j$ )
19:         $Posterior[j][i] = p(\theta^*) \left( \tilde{p}(\mathbf{Y}_{1:T} | \theta^*)^{\frac{1}{Temp_j}} \right)$  //comp. and save posterior (chain  $j$ )
20:         $\theta_j = \theta^*$  // temporary variable (chain  $j$ )
21:      else
22:         $Sample[j][i] = Sample[j][i-1]$  // replicate previous sample (chain  $j$ )
23:         $Posterior[j][i] = Posterior[j][i-1]$  // replicate previous posterior (chain  $j$ )
24:    Chain exchanges:
25:    for  $(q, r) = (1, 2), (3, 4), \dots$  OR  $(q, r) = (2, 3), (4, 5), \dots$  (in turn order) do
26:      Exchange  $Sample[q][i] \leftrightarrow Sample[r][i]$ ,  $Posterior[q][i] \leftrightarrow Posterior[r][i]$  and  $\theta_q \leftrightarrow \theta_r$ 
      with probability  $\min\left(1, \frac{\tilde{p}(\mathbf{Y}_{1:T} | \theta_r)^{\frac{1}{Temp_{pq}}} \tilde{p}(\mathbf{Y}_{1:T} | \theta_q)^{\frac{1}{Temp_{pr}}}}{\tilde{p}(\mathbf{Y}_{1:T} | \theta_q)^{\frac{1}{Temp_{pq}}} \tilde{p}(\mathbf{Y}_{1:T} | \theta_r)^{\frac{1}{Temp_{pr}}}}\right)$ 
27: return ( $Sample[1][1 : N]$ ,  $Posterior[1][1 : N]$ ) ( $N$  sets of MCMC samples and posterior values
   from the first chain)

```

available parallelism increases to $M \cdot P$ due to the existence of M MCMC chains which can be updated independently.

4.4.2 pMCMC architecture

The architecture of pMCMC is illustrated in Figure 4.1. Table 4.1 lists all the constant and variable parameters related to the architecture and the pMCMC algorithm. Table 4.2 lists the sizes of all the memory blocks of the system. The architecture consists of a pMCMC block and some extra logic which is responsible for initialization and communication with the off-chip memory and the host PC. The pMCMC block comprises all the necessary parts to implement a Metropolis MCMC sampler plus a Bootstrap PF block which estimates the likelihood of the proposed sample and generates posterior state samples. The PF block consists of three main stages; Transition & Weight, Partial sums and Resampling stage. These stages perform the tasks described in section 2.3.3 for the PF (transition, weight, resample) but they are grouped in a different way for implementation reasons (which will be explained shortly).

In more detail (using the notation and line numbering of Algorithm 8): For iteration i of the MCMC loop in line 9, the system reads the θ component of the current (latest) MCMC sample from the Current theta memory and sends it to the Sample Proposal block. The block samples from the proposal $q(\theta^* | \theta)$ in line 10, which is model-specific and thus defined by the user. Any necessary random numbers, which depend on the form of $q(\theta^* | \theta)$, are generated using Random Number Generators (RNGs). The proposed value θ^* is written to the Proposed theta memory and then forwarded to the PF block. The latter implements lines 11-12 of Algorithm 8, returning the estimated likelihood of the proposed MCMC sample ($\tilde{p}(\mathbf{Y}_{1:T} | \theta^*)$) and a randomly selected state sample ($X_{1:T}^p$). These are written to the respective memories. The proposed θ^* is also fed to the Prior Evaluation block, which computes the prior value $p(\theta^*)$ (also a model-specific computation). The Update block is responsible for accepting or rejecting the proposed sample (lines 13-20), based on the acceptance ratio of equation (4.7). To do this, it needs the proposed likelihood emitted by the PF block ($\tilde{p}(\mathbf{Y}_{1:T} | \theta^*)$), the proposed prior emitted by the Prior Evaluation block $p(\theta^*)$, the current MCMC sample's likelihood and prior ($\tilde{p}(\mathbf{Y}_{1:T} | \theta)$ and $p(\theta)$) which are read from the respective memory, as well as the values $q(\theta | \theta^*)$ and $q(\theta^* | \theta)$. These two values are computed inside the Update block using θ and θ^* . Finally, the Update block needs a uniform random number from an RNG to implement the acceptance step. If the update step is successful, the block writes the proposed MCMC sample and the proposed likelihood and prior



Table 4.1: Constant and variable parameters in the algorithms and architectures of this chapter

Symbol	Description	Range (used in Section 4.7)
T	Number of SSM states	[100, 16000]
n_X	Dimension of each SSM state $\mathbf{X}_t$	1
n_Y	Dimension of each SSM observation $\mathbf{Y}_t$	8
n_θ	Dimension of SSM's unknown parameter θ	2
n_{tr}	Dimension of unknown parameters related to the transition density ($\leq n_\theta$)	1
n_{obs}	Dimension of unknown parameters related to the observation density ($\leq n_\theta$)	1
n_Z	Dimension of each (known) constant vector $\mathbf{Z}_t$ used in transition density	1
P	Number of particles used in PF	[128, 16000]
N	Number of MCMC iterations in pMCMC or ppMCMC	10000
M	Number of Markov chains in ppMCMC	[1, 5]
B	Degree of parallelism in pMCMC and ppMCMC architectures (applies to many parts of the PF datapath)	2
d_{rng}	Number of RNG modules in pMCMC and ppMCMC architectures	[8, 32]
P_{max}	Maximum number of particles in pMCMC and ppMCMC architectures	[4096, 16384]
T_{max}	Maximum number of SSM states in pMCMC and ppMCMC architectures	[4096, 16384]
M_{max}	Maximum number of chains in ppMCMC architecture	[1, 8]

values to the respective current memories (which are the equivalent of $Sample[i]$ and $Posterior[i]$ in Algorithm 8). Otherwise, the current memories remain unchanged (keeping the values $Sample[i - 1]$ and $Posterior[i - 1]$). After the update is completed, the current memories are read and their values are sent to the off-chip memory (using a DMA transfer). The biggest part of the transfer is due to the current state sample, since its dimension (T) is typically much larger than the dimension of θ , while the likelihood and prior are scalars. The above sequence of operations is repeated for all pMCMC loop iterations until N samples and likelihoods/priors have been sent to the off-chip memory. The first iteration of pMCMC (lines 3-7 in Algorithm 8) is done in a similar way but the proposal and update steps are absent.

The memories outside the PF block have pre-defined sizes at compile time. These sizes are defined according to the constant parameters T_{max} , n_θ and n_X of Table 4.1. Table 4.2 (upper part) lists all the memories and their sizes.

Table 4.2: Memories of the pMCMC/ppMCMC architectures

Memory name	Depth (entries)	Width (bits)	Partitioning degree	Replication in ppMCMC (M_{max} times)
Current theta	1	$32 \cdot n_\theta$	n/a	Yes
Current states	T_{max}	$32 \cdot n_X$	n/a	Yes
Current likelihood and prior	1	64 (2 x 32)	n/a	Yes
Proposed theta	1	$32 \cdot n_\theta$	n/a	Yes
Proposed states	T_{max}	$32 \cdot n_X$	n/a	Yes
Proposed likelihood	1	32	n/a	Yes
Particle (1 & 2)	P_{max}	$32 \cdot n_X$	B	Yes
Transition data	T_{max}	$32 \cdot n_Z$	n/a	No
Observation data	T_{max}	$32 \cdot n_Y$	n/a	No
Log-weight	P_{max}	32	B	Yes
Weight	P_{max}	32	B	Yes
Partial sums	B	32	B	Yes
Reduction counts	P_{max}	$\log_2(P_{max})$	B	Yes
Saved particles	T_{max}	$32 \cdot n_X$	n/a	Yes
Estimated likelihood	1	32	n/a	Yes

Particle Filter block

Within the PF block of the architecture, which is activated once per pMCMC iteration, the three main operations of the PF (transition, weight and resampling) are repeated T times (once for every PF time step, including the initial state). The PF transition and weight operations are implemented inside the Transition & Weight stage of the architecture. The PF resampling operation is implemented by the Partial sums and Resampling stages. Also, with reference to Algorithm 7, the loop of line 8 is implemented with the order of operations changed; the sample step happens first, followed by the weight step and finally the resampling step. Each step has to process P particles (see the three loops in lines 9, 12 and 14). Computations for each particle are independent and therefore they are parallelized and pipelined. The degree of parallelism (number of parallel modules) is the same for all steps and it is denoted by B (see also Table 4.1). This means that each parallel module processes $\frac{P}{B}$ particles. In order to feed the modules with data at every cycle, each memory that is connected to the modules is split into B separate sub-memories and each sub-memory is assigned to one module.

As shown in Figure 4.2 (again using the notation and line numbering of Algorithm 7), at each time step t the particles $\mathbf{X}_{t-1}^{1:P}$ are read from the Particle memories and fed to the Transition & Weight Stage block

(which implements lines 12-15). The B state transition modules sample from the transition density of equation (4.2) in order to propagate the particles to the next time step, i.e. they generate $\tilde{\mathbf{X}}_t^{1:P}$. Random numbers are provided by suitable RNGs. The proposed θ^* (more specifically, its sub-component θ_{tr}^*) is used as the parameter of the transition density. Any model-specific constants which are used inside the transition density ($\mathbf{Z}_{1:T}$, mentioned in Section 2.3.3) are stored into the Transition data memories and read by the state transition modules. The propagated particles are written to Particle memories 2 and also passed to the observation density modules to compute the weights $W_t^{1:P}$. The weights are not computed directly; the log-weights $\log(W_t^{1:P})$ are computed instead, they are then renormalized using the maximum log-weight and finally exponentiated (part of these computations are done in the Partial sums stage). Using log-weights helps avoid floating point numerical issues. It is the standard approach in PF literature, especially when the number of particles is large [54]. The B observation density modules compute the log-weight of each particle using the logarithm of the observation density of equation (4.3). The proposed θ^* (more specifically, its sub-component θ_{obs}) is used as the parameter of the observation density. Also, the data $\mathbf{Y}_{1:T}$ are stored in the Observation data memories and they are read by the modules. When the log-weights are computed, they are written to the Log-weight memories and also passed to a comparator tree. The maximum log-weight is found when all weights have passed through the tree.

The following two stages of the architecture perform the resampling operation (equivalent to lines 9-11 of Algorithm 7). Among the various resampling algorithms that have been proposed, the Residual Sytematic Resampling (RSR) method of Athalye et al. [111] is used here. RSR does not perform resampling in the way shown in lines 9-11 of Algorithm 7. One difference of RSR is that it does not produce ancestor indices (a^k in line 10). Instead, it produces replication counts r^k . This is the number of times that particle k is replicated during resampling. Also, in contrast to other resampling methods [54], RSR does not need to compute a cumulative sum before resampling starts. Instead, it only requires partial sums of weights for every parallel processing module. In other words, processing module $l \in \{1, B\}$ requires the partial sum $\sum_{k=1}^{(l-1) \frac{P}{B}} W_t^k$. An FPGA implementation of the RSR method (modified to improve throughput) was proposed by Liu et al. [1] and it is adopted here. The implementation makes use of fixed point arithmetic in some of its parts in order to reduce latency and allow more efficient pipelining. For a detailed description of the implementation, see [1].

In the Partial Sums stage, the largest log-weight (which was found previously) is used for renormalization of all log-weights. The renormalized log-weights are exponentiated to find the renormalized

weights of the particles, which are written to the Weight memories. Also, a set of B parallel accumulator modules receive the renormalized weights and produce the sums of weights of each module $\sum_{k=(l-1)\frac{P}{B}+1}^{l\frac{P}{B}} W_t^k, l \in \{1, \dots, B\}$. These are then used to generate the partial sums and the sum of all weights and they write them to the respective memories.

In the Resampling stage, B parallel datapaths are responsible for computing the replication counts for each particle. The datapath operates on a number of inputs: 1) A set of B values $U_{1:b}$, which are initialized using the partial sums and then updated as particles stream through the data path (for details on the utility of these values, see [111] and [1]), 2) The weights, 3) The number of particles, 4) The total sum of weights.

Finally, the replication counts are read from the replication memories one by one (not in parallel) and based on their values, the particle set is regenerated. Each particle that has a replication count above zero is copied from Particle memories 2 to Particle memories 1. The number of copies created in Particle memories 2 is equal to the particle's replication count. Two particle memories are required in order to avoid overwriting a particle before its replication count is examined (which would be possible if only one particle memory was used). The replication process cannot be parallelized by replicating each bunch of $\frac{P}{B}$ particles separately, since the total number of replications of a bunch might exceed the memory partition assigned to it (which is again $\frac{1}{B}$ of the total memory). For instance, if $P = 10$ and $B = 2$, each parallel block would process 5 replication counts/particles and have a particle memory of length 5 to its disposal. If particle $k = 1$ has a replication count of 6, this means that the first parallel block does not have enough memory space to replicate the particle and needs to access the second memory block.

In parallel to the resampling operation, the mean of the current weights $\frac{1}{P} \sum_{k=1}^P W_t^k$ is computed, its logarithm is found and the maximum log-weight is used to invert the renormalization mentioned above. The generated value is fed to an accumulator (lower right part of Figure 4.1). After the above PF stages have been repeated T times, the accumulator's output is the estimate of the log-likelihood $\log(\tilde{p}(\mathbf{Y}_{1:T} | \theta^*)) = \log(L) = \sum_{t=1}^T \log(\frac{1}{P} \sum_{k=1}^P W_t^k)$ (line 17). This estimate is written to the Proposed likelihood memory outside the PF block. Also, at the end of the T time steps, the Saved particles memory has stored a randomly selected set of particles, i.e. the proposed state $\mathbf{X}_{1:T}^*$ in Algorithm 8. This is copied to the Proposed states memory outside the PF block.

The processing of the initial PF state in Lines 3-6 of Algorithm 7 is done in the same way but in this

case the initial density of equation (4.1) is used to initialize the particles (Particle initialization logic block). The initial particles are written to Particle memories 1. The resampling step is performed for the initial state without any effect (it is not necessary) to simplify the implementation.

The memories within the PF all have pre-defined sizes at compile time. These sizes are defined according to the constant parameters P_{max} , T_{max} , n_{θ} , n_X , n_Y and n_Z of Table 4.1, which are all given by the user. Table 4.2 (lower part) lists all the memories and their sizes (along with the degree of partitioning for each memory).

4.4.3 ppMCMC architecture

The ppMCMC architecture is shown in Figure 4.2. The differences compared to the pMCMC architecture are that: 1) Many chains need to be processed at each MCMC step and 2) Exchange moves need to be performed. In order to process multiple chains, the Proposed theta, Proposed states, Proposed likelihood, Current theta, Current states, Current likelihood and prior memories are replicated M times (equal to the number of chains). This way, each chain has its separate memory and can be updated and exchanged without interfering with the other chains. The size of each of the M memories is the same as in the pMCMC architecture (see Table 4.2).

The architecture performs the same steps as the pMCMC architecture but for all chains. With reference to Algorithm 9, for MCMC iteration $i \in \{2, \dots, N\}$ (loop in line 10) and for chain iteration $j \in \{1, \dots, M\}$ (loop in line 12), all the steps described previously for pMCMC (sample proposal, PF run, prior evaluation and update) are performed using the respective blocks in the ppMCMC architecture. The Proposal, Update, Exchange and Prior blocks are pipelined, which means they can process one chain per clock cycle. The Proposal block proposes θ^* values for all chains (one chain per cycle) and when it finishes with all M chains the PF blocks starts (which is described later). Only one PF block is instantiated (as in pMCMC); all chains are processed by this one block. After the PF block has computed likelihoods and sampled states for all chains, the remaining blocks (Update, Exchange) are fed with one chain per cycle and they perform the respective operations. The Update block has an extra input compared to the pMCMC architecture: The temperature of the chain which is currently processed ($Temp_j$).

As shown in the loop of line 25, at each MCMC iteration, exchanges are attempted between specific pairs of chains (either (1,2),(3,4)... or (2,3),(4,5),...). As soon as a pair of chains which is designated

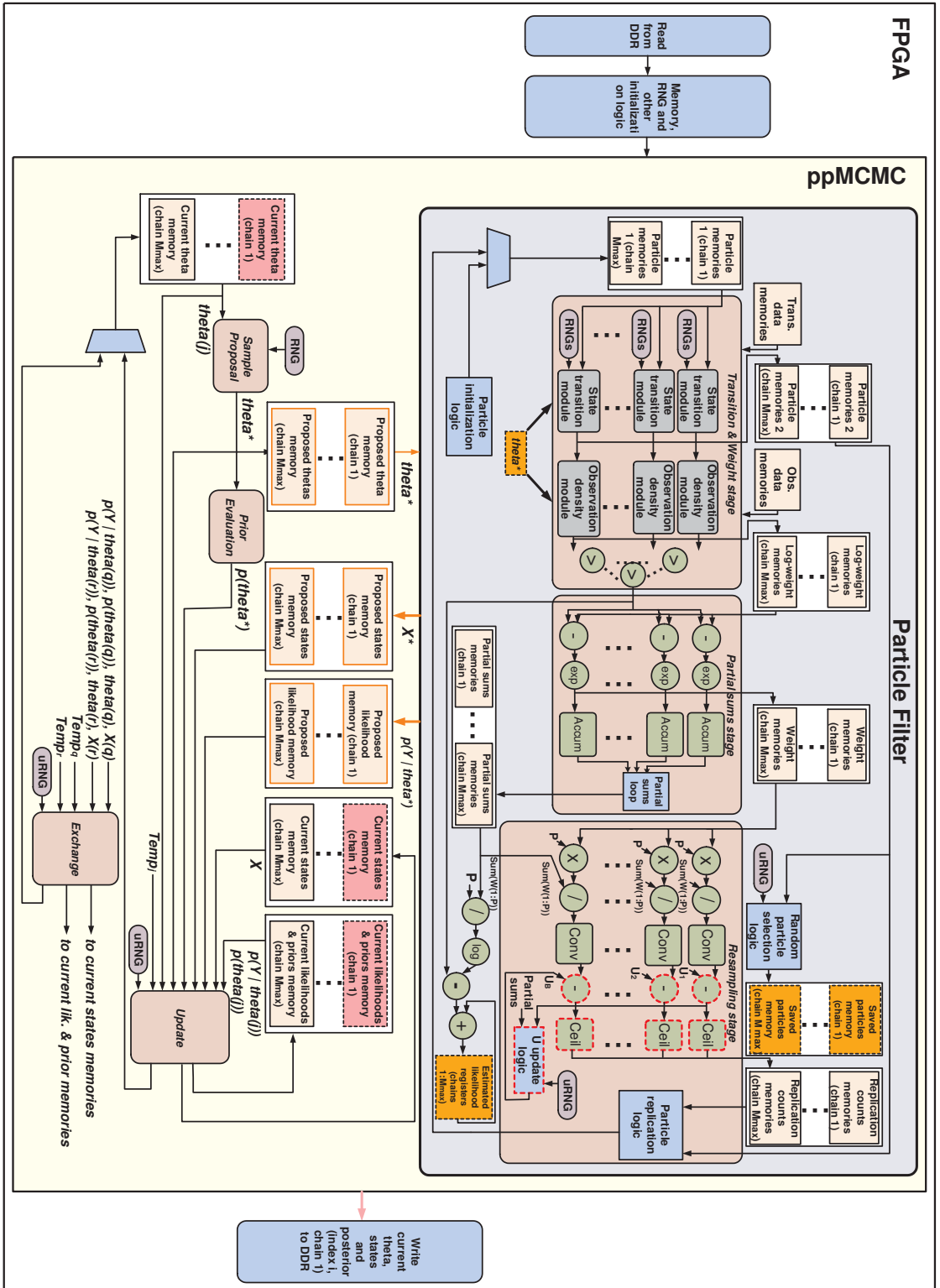


Figure 4.2: FPGA architecture for ppMCMC. Blocks with red dotted lines operate in fixed point arithmetic (see [1]).

for exchange in the current MCMC iteration (chains q and r in Figure 4.2) has been updated, the exchange blocks of the ppMCMC architecture starts processing the exchange move between those chains. Its inputs include the current θ , $\mathbf{X}_{1:T}$, $\tilde{p}(\mathbf{Y}_{1:T} | \theta)$ and $p(\theta)$ of chains q and r , as well as the temperatures of the chains and a uniform random number. These are used to compute the ratio in line 26 of Algorithm 9 and exchange the MCMC samples, likelihoods and priors (line 26) between the respective memories if the exchange is successful. The prior density evaluation overlaps with the PF run as in pMCMC.

Coarse-grain pipelining in the PF

The above modifications to the pMCMC architecture are enough to acquire a baseline ppMCMC implementation. However, it is possible to improve the performance of this baseline implementation by exploiting coarse grain pipelining inside the PF. During a PF run of a single chain, the PF datapath is traversed T times (once per time step). One traversal comprises the Transition & Weight, Partial sums and Resampling stages and each of these stages can only start after the previous one has finished. This means that at each moment in time, only one of the three stages is utilized in the FPGA architecture.

Let the latencies of the three stages be Lat_{tw} , Lat_{ps} , Lat_{re} (for the P particles to be processed at each stage). Then the total latency for one time step is $Lat_{tw} + Lat_{ps} + Lat_{re}$. Without applying coarse-grain pipelining, the order with which tasks are fed into the pipeline is the following: Starting from the first chain $j = 1$, the first time step $t = 1$ traverses the PF datapath. When it finishes, the second time step $t = 2$ traverses the datapath, etc. When all T time steps of the first chain have finished, the PF run of the first chain is complete. The next chain $j = 2$ then begins to be processed, starting from time step $t = 1$, etc. This goes on until all chains have completed their PF runs. The total latency for M chains with T time steps each is $M \cdot T \cdot (Lat_{tw} + Lat_{ps} + Lat_{re})$.

In order to increase the utilization of the PF datapath, the proposed ppMCMC architecture exploits the fact that each chain's PF run in ppMCMC is completely independent; a PF run does not need to wait for the PF run of the previous chain to finish. Although the PF block of ppMCMC is the same as in pMCMC (i.e. a single block), it is configured to exploit the independence of ppMCMC chains in order to increase datapath utilization. The architecture feeds the PF datapath with multiple chains at the same time. This happens by changing the order with which the tasks are fed to the PF datapath: The first task is time step $t = 1$ of chain $j = 1$. This is followed by time step $t = 1$ of chain $j = 2$, then time step $t = 1$ of chain $j = 3$, etc. When all the first time steps have been processed, the system

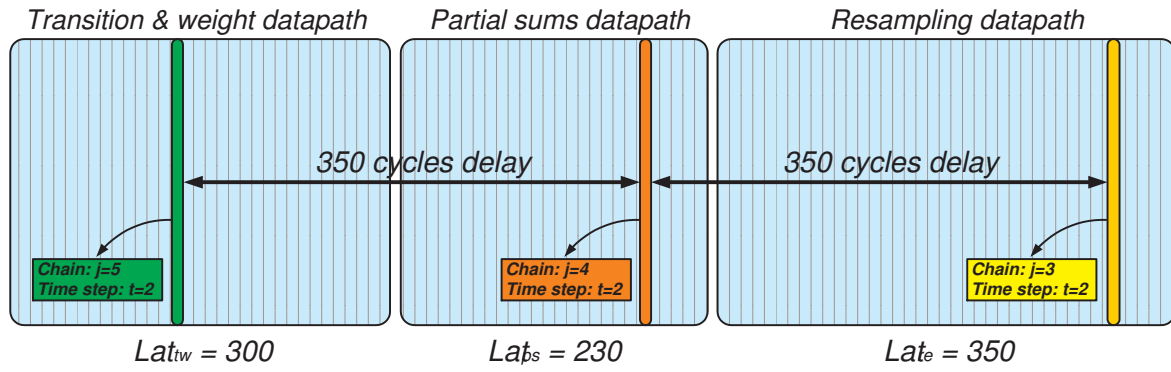


Figure 4.3: Coarse-grain pipelining in ppMCMC architecture for a specific SSM model and specific parameter configuration (P, T, M) . The Resampling stage in this example has latency $Lat_{re} = 350$, which is the largest latency among the three PF stages. Thus a new chain can be fed to the PF datapath every $Lat_{re} = 350$ clock cycles. Here, three chains ($j = 3, j = 4, j = 5$) are inside the datapath, all of them at the same time step $t = 2$. Chain $j = 4$ will enter the Resampling stage one cycle after chain $j = 3$ exits the Resampling stage.

starts processing the second time steps ($t = 2$) for all chains, etc. By changing the order of operations, the available parallelism is exposed and it is possible to pipeline the tasks. In other words, since all chains are independent, the tasks with index $t = 1$ of all chains can be pipelined easily. Of course, pipelining can only be coarse-grain, since the three stages of the PF cannot be shared by two chains at a time. Each chain needs to have enough time to finish a PF stage before the next chains reaches the same stage (no two chains are in the same stage at the same time). Using this technique, it is possible to introduce a new chain to the PF datapath (i.e. the Transition stage) every $\max(Lat_{tw}, Lat_{ps}, Lat_{re})$ clock cycles. This means that the total latency is reduced to $M \cdot T \cdot \max(Lat_{tw}, Lat_{ps}, Lat_{re})$. Exact formulas for the above quantities are given in the next section. Figure 4.3 depicts how the coarse-grain pipelining strategy is used to process multiple chains simultaneously and increase datapath utilization in the PF block of ppMCMC.

In order to implement the above pipelining strategy, it is necessary for many “virtual” PFs to work simultaneously using the same PF hardware block. This requires the use of multiple memories (one for each chain) for all the variables accessed by the PF. The ppMCMC architecture uses M Particle, Log-weight, Weight, Saved particles, Replication counts and Partial sums memories, instead of one memory of each type in pMCMC (in fact, M_{max} memories exist in the architecture as shown in Table 4.2 and explained in the next paragraph). Exactly as in the pMCMC architecture, each of the M memories is partitioned into B independent memory blocks in order to feed the B processing modules in parallel. The Transition data and Observation data memories do not need to be replicated, since the data are common for all chains. This is shown in Figure 4.2.

Resource overheads compared to pMCMC architecture

The resource overheads of the ppMCMC architecture are mainly the extra memories. M_{max} memories of each kind need to be instantiated (see Table 4.2, last column), instead of 1 (M_{max} is the upper limit of chains that can be used and it is set at compile time). The Transition data and Observation data memories do not need to be replicated. The Exchange block is an additional overhead which takes up relatively few resources (see Section 4.7), since the exchange ratio is simple to compute.

4.4.4 Performance models

This section gives exact formulas for the latency and throughput of the various blocks in the architectures. These models are later used to break down the total clock cycles consumed by the pMCMC architecture into the clock cycles of the various blocks (see Section 4.7.3). The correctness of the pMCMC model is also validated using real pMCMC runs on the FPGA. The ppMCMC performance model is used to estimate the performance of the proposed ppMCMC architecture (which is not run on the FPGA). The results of Section 4.7.5 (including the design space exploration to find the parameter combinations that maximize performance) are based on these estimates.

pMCMC

The latency of the Transition & Weight stage is:

$$Lat_{tw} = C_{st} + C_{od} + Lat_{tree} + \left\lceil \frac{P}{B} \right\rceil \quad (4.14)$$

where C_{st} is the latency of one state transition module (depends on the targeted SSM), C_{od} is the latency of one observation density module (depends on the targeted SSM), $Lat_{tree} = \lceil \log(B) \rceil \cdot Lat_{comp}$ is the latency of the comparator tree and the term $\left\lceil \frac{P}{B} \right\rceil$ is the latency for passing all the P particles through the transition and observation modules when the degree of parallelism of the architecture is equal to B (the degree of parallelism applies to transition and observation modules as well as the comparator tree). All parallel modules are pipelined and can receive one input per cycle.

The latency of the Partial sums stage is:

$$Lat_{ps} = Lat_{sub} + Lat_{exp} + Lat_{accum} + \left\lceil \frac{P}{B} \right\rceil \cdot Lat_{add} + Lat_{psl} \quad (4.15)$$

where Lat_{sub} and Lat_{exp} are the latencies of the subtraction and exponent operators, $Lat_{accum} = Lat_{add}$ is the latency of the accumulator, $Lat_{psl} = B \cdot Lat_{add}$ is the latency of the partial sums loop (which forms B partial sums) and the term $\lceil \frac{P}{B} \rceil \cdot Lat_{add}$ is the latency for processing all P particles when the degree of parallelism of the architecture is equal to B (the degree applies to the subtracters, exponents and to the accumulator modules). All operators are pipelined but they can receive one input per Lat_{add} cycles because of the latency of the floating point adder.

The latency of the Resampling stage is:

$$Lat_{re} = Lat_{initU} + Lat_{mult} + Lat_{div} + Lat_{conv} + Lat_{subF} + Lat_{ceilF} + \lceil \frac{P}{B} \rceil + P \cdot Lat_{rep} \quad (4.16)$$

where Lat_{initU} is the latency required to initialize the U values using the partial sums and Lat_{mult} , Lat_{div} , Lat_{conv} , Lat_{subF} and Lat_{ceilF} are the latencies of the multiplication, division, float-to-fixed point conversion, fixed-point subtracter and fixed-point ceiling operators. The term $\lceil \frac{P}{B} \rceil$ is the latency for processing all P particles when the degree of parallelism of the architecture is equal to B (the degree applies to all above operators). All operators are pipelined and can receive one input per cycle. Finally, the term $P \cdot Lat_{rep}$ is due to the particle replication loop, which replicates each particle a number of times equal to its replication factor. The number of replications changes randomly at runtime. Lat_{rep} is the mean number of cycles needed for the replication of each particle.

Given the latencies of the three stages, the latency of the PF critical path for one time step (one iteration of the loop in line 8 of Algorithm 7) is:

$$Lat_{ts_pmcmc} = C_1 + Lat_{tw} + Lat_{ps} + Lat_{re} \quad (4.17)$$

where C_1 is some extra constant latency incurred by minor computations before and after the three PF stages (to initialize variables, read/process proposed θ^* parameter fed from the blocks outside the PF, select and save random particle). For a specific implementation of the architecture in Xilinx Vivado HLS 2014.1 on a Xilinx Zynq ZC706 board, using single precision floating point arithmetic, this constant takes the value $C_1 = 127$.

The total latency for a complete run of the PF (equivalent to one call of Algorithm 7 or to line 11 in Algorithm 8) is:

$$Lat_{pf_pmcmc} = C_2 + \lceil \frac{P}{B} \rceil + T \cdot Lat_{ts_pmcmc} \quad (4.18)$$

where the term $T \cdot Lat_{ts}$ is due to the T time steps which are processed sequentially, the term $\lceil \frac{P}{B} \rceil$ is due to the particle initialization before each PF run and C_2 is a constant latency due to minor computations before each iteration. For a specific implementation of the architecture in Xilinx Vivado HLS 2014.1 on a Xilinx Zynq ZC706 board, using single precision floating point arithmetic, this constant takes the value $C_2 = 21$.

The latency of the critical path of each pMCMC iteration (one iteration of the loop in line 9 of Algorithm 8) is:

$$Lat_{iter_pmcmc} = C_3 + Lat_{prop} + Lat_{pf_pmcmc} + Lat_{acc} + (T + 1) + Lat_{mem_cpy} \quad (4.19)$$

where Lat_{prop} is the latency of sampling from the MCMC proposal $q(\theta^* | \theta)$ in line 10 of Algorithm 8 (depends on the choice of proposal) and $Lat_{acc} = 2 \cdot Lat_{add} + Lat_{sub} + Lat_{comp}$ is the latency of computing the acceptance ratio (4.7) and comparing with a uniform number (using log-values). The term $T + 1$ is due to the transfer of the proposed SSM states and the estimated likelihood (generated by the PF) from the memories within the PF block to the respective memories outside the PF block (when the update is successful, lines 15-16 in Algorithm 8). The term C_3 is a constant latency due to other computations within the pMCMC loop, before and after the PF run (initializations, copying of variables between registers). For a specific implementation of the architecture in Xilinx Vivado HLS 2014.1 on a Xilinx Zynq ZC706 board, using single precision floating point arithmetic, this constant takes the value $C_3 = 42$. The term Lat_{mem_cpy} is the latency needed to transfer the new MCMC sample (updated or not) to the off-chip memory after the MCMC iteration. Vivado HLS estimates this latency to be $Lat_{mem_cpy} = n_\theta + 1 + T$. Note that the prior density and proposal density latencies are omitted from (4.19) because their datapaths are not in the critical path of the MCMC iteration; both are computed at the same time that the PF is running and their latencies are much smaller than the PF latency Lat_{pf_pmcmc} .

The total latency of the pMCMC algorithm (all N iterations) is:

$$Lat_{pmcmc} = Lat_{init_pmcmc} + Lat_{first_pmcmc} + (N - 1) \cdot Lat_{iter_pmcmc} \quad (4.20)$$

where $Lat_{first_pmcmc} = C_3 + Lat_{pf_pmcmc} + (T + 1) + Lat_{mem_cpy}$ is the latency of the first pMCMC iteration (lines 3-7 in Algorithm 8) and Lat_{init_pmcmc} is the latency for initializing the system (copying initial values from the off-chip memory and initializing the RNGs). It is equal to $Lat_{init_pmcmc} =$

$21 + 2 \cdot n_\theta + T \cdot (n_Z) + T \cdot n_Y + d_{rng} + 1000 \cdot d_{rng}$, where all the terms are related to data transfer latencies, except for the term $1000 \cdot d_{rng}$ which is the latency for initializing the RNGs (using 1000 cycles of initialization time for each RNG). All the symbols are explained in Table 4.1.

Finally, the total runtime of the pMCMC system (including the time to transfer data from/to the host PC) is:

$$Time_{total_pmcmc} = \frac{Lat_{pmcmc}}{freq} + Time_{I/O_pmcmc} \quad (4.21)$$

where $freq$ is the clock frequency of the pMCMC IP in Hz and $Time_{I/O}$ is the time needed for communication with the host PC (including time spent on parts of the FPGA system other than the pMCMC IP, e.g. a CPU on the FPGA). $Time_{I/O}$ cannot be modelled exactly. A calibration procedure based on actual runtimes of the system is used to estimate this quantity.

ppMCMC

For pMCMC, the latencies of the three PF stages are the same as the ones shown in Equations (4.14), (4.15) and (4.16). The latency of processing one time step for all chains (the same for all chains, e.g. $t = 1$) is:

$$Lat_{ts_ppmcmc} = C_1 + (Lat_{tw} + Lat_{ps} + Lat_{re}) + (M - 1) \cdot \max(Lat_{tw}, Lat_{ps}, Lat_{re}) \quad (4.22)$$

Since one chain is inserted into the pipeline every $\max(Lat_{tw}, Lat_{ps}, Lat_{re})$ cycles, the cost of processing all chains is $(M - 1) \cdot \max(Lat_{tw}, Lat_{ps}, Lat_{re})$ plus the latency of the first chain (term inside the brackets) plus C_1 . The most cycle-expensive stage of the PF data path can be any of the three stages mentioned above.

The latency of processing all T time steps of all M chains (i.e. completing M PF runs) is:

$$Lat_{pf_ppmcmc} = C_2 + M \cdot \left\lceil \frac{P}{B} \right\rceil + T \cdot Lat_{ts_ppmcmc} \quad (4.23)$$

where the term $M \cdot \left\lceil \frac{P}{B} \right\rceil$ represents the cycles needed for particle initialization for all chains and $T \cdot Lat_{ts_ppmcmc}$ are the cycles needed to complete all time steps for all chains.

The latency of one ppMCMC iteration (one iteration of the loop in line 10 of Algorithm 9, which

comprises updates and exchanges for all chains) is:

$$Lat_{iter_ppmcmc} = C_3 + (Lat_{prop} + M) + Lat_{pf_ppmcmc} + (Lat_{acc} + Lat_{ex} + (T + 1) + M) + Lat_{mem_cpy} \quad (4.24)$$

where the term $(Lat_{prop} + M)$ is the latency needed to pass all chains through the pipelined Sample Proposal block, $Lat_{ex} = Lat_{mult} + Lat_{add} + Lat_{sub} + Lat_{comp}$ is the latency for computing the exchange ratio (4.12) and comparing with a uniform number (using log-values), the term $(Lat_{acc} + Lat_{ex} + (T + 1) + M)$ is the latency to pass all chains through the pipelined Update and Exchange blocks (plus updating the current sample memories) and the term Lat_{mem_cpy} is the latency to copy the MCMC sample of the first chain to the off-chip memory (only samples of the first chain are kept).

The total latency of the ppMCMC algorithm (all N iterations) is:

$$Lat_{ppmcmc} = Lat_{init_ppmcmc} + Lat_{first_ppmcmc} + (N - 1) \cdot Lat_{iter_ppmcmc} \quad (4.25)$$

where $Lat_{first_ppmcmc} = C_3 + Lat_{pf_ppmcmc} + (T + 1 + M) + Lat_{mem_cpy}$ is the latency of the first ppMCMC iteration (lines 3-8 in Algorithm 9) and Lat_{init_ppmcmc} is the latency for initializing the system (copying initial values from the off-chip memory and initializing the RNGs). It is equal to $Lat_{init_ppmcmc} = 21 + 2 \cdot M \cdot n_\theta + M + T \cdot (n_Z) + T \cdot n_Y + d_{rng} + 1000 \cdot d_{rng}$, where all the terms are related to data transfer latencies from off-chip memory, except for the term $1000 \cdot d_{rng}$ which is the latency for initializing the RNGs (using 1000 cycles of initialization time for each RNG). All the symbols are explained in Table 4.1.

Finally, the total runtime of the ppMCMC system (including the time to transfer data from/to the host PC) is:

$$Time_{total_ppmcmc} = \frac{Lat_{ppmcmc}}{freq} + Time_{I/O_ppmcmc} \quad (4.26)$$

where $freq$ is the clock frequency of the pMCMC IP in Hz and $Time_{I/O}$ is the time needed for communication with the host PC (including time spent on parts of the FPGA system other than the ppMCMC IP, e.g. a CPU on the FPGA).

4.5 Case Study

Statistical genetics is a representative example of a complex Bayesian application which typically handles large-scale data. SSMs are suitable to model many phenomena in genetics. Here, an SSM which models DNA methylation profiles is used as a case study. DNA methylation is a biochemical process which happens naturally in specific positions of the genome. During methylation, a methyl group is added to a cytosine or adenine nucleotide. Methylation plays a key role in normal development but it is also associated with a number of diseases [152, 14].

The goal of methylation analysis is to discover which positions of the genome contribute to the appearance of some disease through the mechanism of methylation. Methylation analysis can be done either using single-tissue DNA (e.g. DNA from cells of the heart) or using multiple-tissue DNA (e.g. whole-blood samples which contain several distinct tissues) [14]. In both cases, it is challenging to infer a methylation profile because the current technology used to detect methylation in each DNA base within a DNA sequence generates a lot of noise and also because the length of the DNA sequences that need to be analysed is large (up to 10^7 bases). Here, the two cases (single- and multiple- tissue methylation) are examined separately, since they lead to different kinds of posterior distributions.

Single-tissue DNA - Uni-modal posterior

In the single-tissue DNA case, a methylation data set from rats is used in order to discover the genetic causes of glomerulonephritis (GN), which is a very common cause of kidney disease. In the UK, Europe and the US, GN is the third most common cause of end-stage kidney disease, accounting for 10-15% of patients. In order to understand which positions in the genome play a role in the appearance of GN through methylation, single-tissue data from rats are used. DNA samples from two rat strains ($i \in \{1, 2\}$) are employed: 1) Wistar-Kyoto (WKY) rats which demonstrate significant reproducible susceptibility to GN when inoculated with nephrotoxic serum, 2) Lewis (LEW) rats which are resistant to GN. DNA from four biological replicates ($j \in \{1, \dots, 4\}$) from each strain is collected and sodium bisulfite treatment is applied to it. This is a typical setting of comparing affected and control subjects to understand the differences in methylation profiles.

Sodium bisulfate treatment is a process that allows the detection of methylated DNA bases through a chemical reaction that affects only the bases that are not methylated. A number of bisulfate experiments are performed for each base. The resulting data from applying the treatment are: 1) the total

number of bisulfate experiments at each DNA position t (n_{ijt}) and 2) the number of successful detections of methylation at each DNA position t (Y_{jt}^i), with $t \in \{1, \dots, T\}$. All data are generated using the tool proposed in [14]. The goal is to discover the probability that position t is methylated (for each t), given the number of experiments and the number of successful detections.

The problem's complexity originates not only in the size of the sequences but also in the fact that a lot of noise is typically present in the bisulfate experiment results. For example, it is common for a small number of "spikes" of successful detections to appear in an area of otherwise un-methylated DNA. This should be treated as measurement noise. This is where the use of SSMs (whose states are correlated with each other) proves useful, as will be shown in the following paragraphs.

In order to analyse the data, the following model is built: The data can be described as a series of binomial experiments (one experiment per strain $i \in \{1, 2\}$, per biological replicate $j \in \{1, \dots, 4\}$ and per DNA position $t \in \{1, \dots, T\}$). The number of DNA positions T can range up to 10^7 for a whole chromosome. Here, DNA chunks of up to 16384 positions are used. Formally:

$$x_{ijt} \sim \text{Bin}(n_{ijt}, p_{ijt}) \quad (4.27)$$

Here, $\text{Bin}(n_{ijt}, p_{ijt})$ is a binomial distribution, x_{ijt} represents the number of successes, n_{ijt} represents the number of trials and p_{ijt} represents the probability that this DNA position is methylated. This model corresponds well with the way the bisulfate treatment experiments are performed (a series of experiments per base, with an approximately equal probability of success for a certain replicate). The equation assigns a different probability for each one of the four replicates at the same DNA position. This helps improve accuracy, since the different replicates might react differently to the treatment and/or their genome might be less/more affected by methylation. Nevertheless, the probabilities of the replicates at a particular position cannot be fully independent, since the affected replicates must have some degree of correlation between their methylation profiles (although these profiles are not identical). To enforce this correlation, the model assumes that there is a common mean for the probabilities of the replicates at the same position. The common mean of the probabilities of all the replicates for strain i and for position t is denoted μ_{it} . It is related to the probabilities p_{ijt} using the following equation:

$$\text{logit}(p_{ijt}) \sim \text{Normal}(\mu_{it}, \sigma_i^2) \quad (4.28)$$

where σ_i^2 is the common observation variance for strain i (which is unknown). The above equation

is called a random effect. It enforces a common mean (μ_{it}) but also adds the necessary randomness between replicates through the variance σ_t^2 .

Another key part of the model is the definition of a spatial dependence between the above means, which corresponds to a reality in methylation analyses; methylated nucleotides are correlated and the appearance of large consecutive areas of methylated or non-methylated nucleotides is common. In order to model this correlation, the model is extended by specifying a non-observable Markov dependence:

$$\mu_{it} \sim \text{Normal}(\mu_{i,t-1}, \sigma_t^2) \quad (4.29)$$

where σ_t^2 is the variance due to the DNA position, which depends on a common (unknown) variance σ and the DNA physical position (δ_t) as follows: $\sigma_t^2 = \sigma^2 |\delta_t - \delta_{t-1}|$. Introducing this spatial dependence in the model, helps tackle the issue of noise in the bisulfate experiments (e.g. sudden “spikes” of methylated based in an otherwise un-methylated region). For more information on spatial dependence in methylation profiles, see [14].

The above model can be translated into an SSM with unknown parameters. A separate SSM is defined for each strain i and the hidden state of SSM i at time step t is $\mathbf{X}_t^i = \mu_{it}$ (therefore, $n_X = 1$). The transition equation (which corresponds to (4.29)) is:

$$\mathbf{X}_t^i \sim \text{Normal}(\mathbf{X}_{t-1}^i, \sigma_t^2), t > 1 \quad (4.30)$$

The physical positions of the DNA are the known constant parameters of the transition density $\mathbf{Z}_t = \delta_t$, with $n_Z = 1$. The observations of SSM i at time step t are $\mathbf{Y}_t^i = Y_{1:4,t}^i = x_{i,1:4,t}$. Therefore, the observation equations (which correspond to (4.27)) are:

$$\mathbf{Y}_{jt}^i \sim \text{Bin}(n_{ijt}, p_{ijt}), j \in \{1, \dots, 4\}, t > 0 \quad (4.31)$$

The dimension of the data per time step is $n_Y = 8$ (both $x_{i,1:4,t}$ and $n_{i,1:4,t}$ are included). Using the above observation equation, the likelihood/weight of particle $k \in \{1, \dots, P\}$ inside the PF (see Section 2.3.3) is:

$$W_t^k = \prod_{j=1}^4 \text{BinPDF}(x_{ijt}, n_{ijt}, p_{ijt}) \quad (4.32)$$

where $\text{BinPDF}(x_{ijt}, n_{ijt}, p_{ijt})$ is the binomial density and the three arguments are the successes, trials and probability of success. The product is converted to a sum when using log-densities.

The intermediate random effect equation (4.28) is used to connect the transition and observation densities. Finally, the prior state equation (representing the state of the first DNA base) is:

$$\mathbf{X}_1^i \sim \text{Normal}(0, 1) \quad (4.33)$$

The SSM's unknown parameters (θ in Section 4.2) are the variances σ_i^2 and σ^2 , i.e. $\theta = [\sigma_i^2, \sigma^2]$. In other words, the goal of SSM inference in this case is to use the collected bisulfate experiment data in order to:

1. Estimate the mean probabilities of methylation (μ_{it}) for each base t and for each replicate i (this is not part of θ but it is estimated by the PF inside pMCMC). The probabilities p_{ijt} of each replicate j are not necessary but they are also estimated.
2. Infer the parameter σ_i , which shows how much variance exists between the probabilities of different replicates.
3. Infer the parameter σ , which shows how much spatial dependence exists between the mean probabilities of a sequence of DNA bases.

pMCMC is able to infer a probability density for all of the above quantities. The posterior of θ in this case is uni-modal (although the true position of the mode is unknown). The dimensions of the unknown parameters are $n_\theta = 2$, $n_{tr} = 1$, $n_{obs} = 1$. The data for experiments are generated setting $\theta = [1, 0.1]$.

Multiple-tissue DNA - Multi-modal posterior

In the multiple-tissue DNA case, the DNA comes from various tissues. In this case, the methylation profile is a mixture of different profiles. The exact same strains, replicates and DNA lengths are used as in the single-tissue case but each set of data comes from a mixture of two tissues. The only difference in the multiple-tissue SSM model that results from this change is that the state transition has to be modelled as a mixture of two distributions, i.e.:

$$\mathbf{X}_t^i \sim \sum_{c=1}^2 [w_c \cdot \text{Normal}(\mathbf{X}_{t-1}^i, \sigma_{tc}^2)], \quad t > 1 \quad (4.34)$$

where $w_c = 0.5$ is the weight of the mixture component with index $c \in \{1, 2\}$ and $\sigma_{tc}^2 = \sigma_c^2 |\delta_t - \delta_{t-1}|$ is the unknown variance of component with index c .

All the other equations are the same as in the single-tissue case. The SSM's unknown parameters (θ in Section 4.2) are the variances σ_i^2 , σ_1^2 and σ_2^2 , i.e. $\theta = [\sigma_i^2, \sigma_1^2, \sigma_2^2]$. Therefore, the model has one extra parameter compared to the single-tissue model. pMCMC estimates the SSM states (mean probabilities of methylation), the parameter σ_i^2 (variance between replicates) and the parameters σ_1^2 and σ_2^2 (spatial dependences between a sequence of bases). The data for experiments are generated setting $\theta = [0.2, 10, 0.02]$. The posterior of θ in this case is multi-modal with modes in $\theta = [0.2, 10, 0.02]$ and $\theta = [10, 0.2, 0.02]$. This makes the exploration of the posterior challenging, as will be shown in Section 4.7.

4.6 Implementation

4.6.1 IP implementation and FPGA system integration

pMCMC

The pMCMC sampler was implemented in C++ using Xilinx Vivado HLS 2014.1. Single precision floating point was used for all datapaths. Single precision is one of the two default precision configurations in MCMC literature. This chapter uses single instead of double precision in order to achieve better performance for the case study presented above.

The PROTOIP framework (available in Xilinx Tcl store [153]) was used for prototyping. The system was mapped on a Xilinx ZC706 Zynq board and connected to the host PC via an Ethernet cable. The system clock (which is also used in the IPs) was set to $freq = 144$ MHz. The full system on the board is shown in Figure 4.4. It consists of the pMCMC IP, an AXI bus, an ARM Cortex processor and a DMA controller. The ARM processor acts as a UDP/IP server over Ethernet and communicates with a client Matlab application in the host. The DMA controller is used by both the ARM processor and the pMCMC IP to access the off-chip DDR memory of the ZC706 board.

The sequence of operations for a complete run of the pMCMC sampler are the following: The Matlab application in the host allows the user to select the parameters of the pMCMC sampler (e.g. the constants N, P, T and several initialization values like the initial pMCMC sample). The parameters are sent to the ARM processor and forwarded to the DDR memory. Then a start signal is sent from the host to the ARM processor, which is passed to the pMCMC IP. The IP performs the pMCMC run and writes the output MCMC samples and likelihoods to the DDR directly. The ARM processor constantly

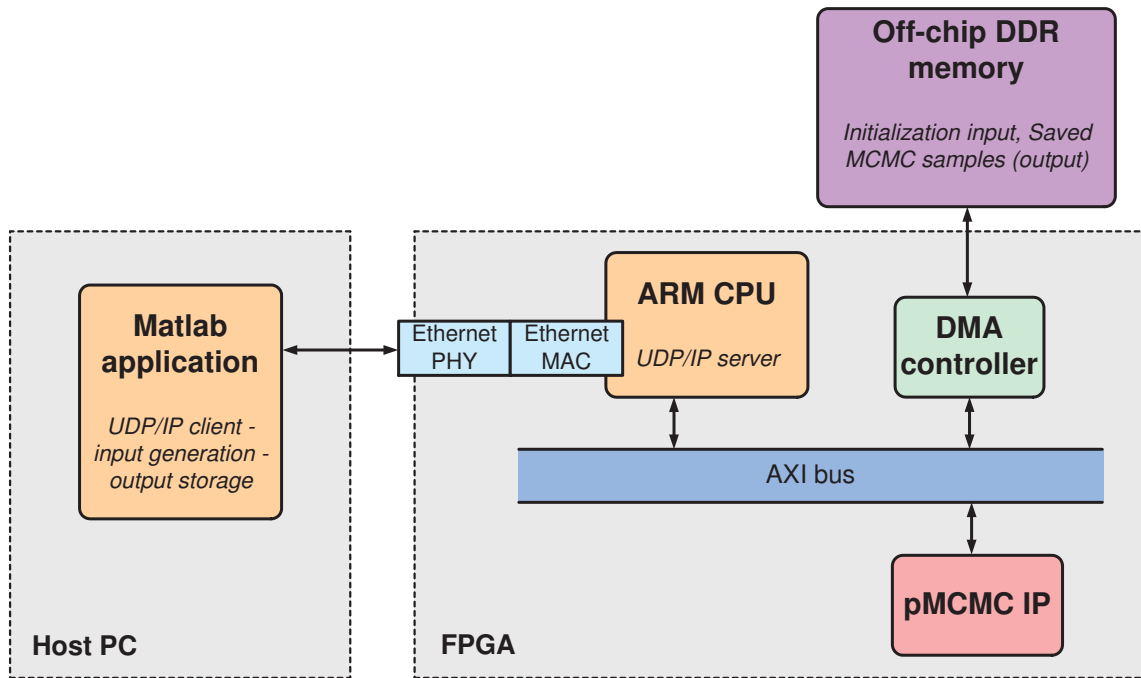


Figure 4.4: FPGA/host system prototype.

queries the state of the IP. When the IP terminates, the processor copies the MCMC output from the DDR to the host.

ppMCMC

The ppMCMC sampler was only implemented in software (Matlab). It was not implemented in Vivado HLS. In order to assess the throughput of the ppMCMC hardware architecture, the performance model presented in Section 4.4.4 was used. The differences between the pMCMC and ppMCMC architectures are limited, since the same single PF block is used in both of them with the only modification in the PF datapath being the use of coarse-grain pipelining in ppMCMC. The effect of this difference in the latency and the performance is incorporated in the model, as shown in Section 4.4.4. The results of the software runs were used to assess the effect of the number of chains and the number of particles on the mixing of ppMCMC (see Section 4.7). It has to be noted that the Matlab implementation is only used to compare pMCMC and ppMCMC in the algorithmic level, assuming sequential implementations. The FPGA and GPU versions are compared against an optimized CPU version in C++ (not in Matlab).

4.6.2 Random number generators

RNGs are needed in many parts of the pMCMC and ppMCMC architectures. Uniform RNGs are needed for the Update and Exchange blocks, as well as to select random particles and perform Resampling inside the PF. Depending on the MCMC proposal, various types of RNGs might be needed to feed the Sample Proposal block. The same applies to the State transition and Observation density blocks inside the PF, which require different RNGs depending on the transition and observation densities of the SSM. When targeting the case study of Section 4.5, only Gaussian RNGs are needed for these blocks.

The Tausworthe Uniform RNG described by L'Ecuyer [154] was implemented in Vivado HLS and used throughout the system. For Gaussian random numbers, a CDF inversion method which uses a polynomial approximation (Thomas et al. [155]) was employed.

4.7 Investigation and results

In this section, the two FPGA accelerators for pMCMC and ppMCMC are compared to state-of-the-art pMCMC implementations on a multi-core CPU and a GPU. The LibBi framework [2] for SSM inference was used to create the CPU and GPU implementations of pMCMC. Both the uni-modal and multi-modal SSM of Section 4.5 were defined in LibBi using the framework's domain-specific modelling language. LibBi was then configured to run joint state estimation and parameter inference on the SSM using pMCMC. Also, sequential implementations of pMCMC and ppMCMC in Matlab are used to compare the two algorithms when no hardware acceleration is employed.

The complete list of comparisons is the following: For the uni-modal SSM, comparisons were performed between the FPGA pMCMC sampler and LibBi's pMCMC samplers in the CPU and GPU. For the multi-modal SSM, comparisons were made between the pMCMC and ppMCMC algorithms in Matlab and in an FPGA (including an exploration of the chains/particles design space). Also, the two FPGA samplers were compared to LibBi's pMCMC samplers in the CPU and GPU.

4.7.1 Platforms and devices

As mentioned previously, the pMCMC architecture were mapped on a Xilinx ZC706 Zynq board, which contains a Z-7045 FPGA. The board was connected to a host PC with a quad-core Intel Core i7-2600 CPU (frequency 3.4 GHz) and 16 GB of RAM through an Ethernet cable. All implementations were done in single precision floating point arithmetic.

The CPU pMCMC sampler (compiled automatically using LibBi) ran on a server with a quad-core Intel Core 2 Q9550 CPU (frequency 2.83 GHz) and 8 GB of RAM. LibBi was configured to make use of OpenMP multithreading with 4 threads and SSE vector parallelism. Intel C++ compiler 2011 was used by LibBi to compile the code (LibBi uses a C++ template library) and all implementations ran in single precision. Every effort was made to select the LibBi compilation parameters that maximize performance.

The GPU pMCMC sampler (compiled using LibBi) ran on an Nvidia Tesla C2050 GPU. The host server had a quad-core Intel Core 2 Q9550 CPU (frequency 2.83 GHz) and 8 GB of RAM. LibBi was configured to make use of all the optimizations of the CPU version in combination with the use of CUDA. The implementation ran in single precision. Every effort was made to select the LibBi compilation parameters that maximize performance.

The (sequential) Matlab pMCMC/ppMCMC code ran on the same server used for the CPU pMCMC sampler without any exploitation of parallelism.

4.7.2 Resource utilization

Table 4.3 contains the FPGA resource utilization results for the pMCMC and ppMCMC samplers. The chosen architecture parameters are shown above the table. The numbers for pMCMC are post-place and route results from Xilinx Vivado 2014.1. The numbers for ppMCMC are estimates based on the extra modules and memories which were described in Section 4.4.3. More specifically, the ppMCMC IP takes up slightly more LUTs, FFs and DSPs than the pMCMC IP due to the Exchange module. The extra resources needed for this module were estimated by synthesizing a stand-alone Exchange block in Vivado HLS. Also, the BRAM utilization of ppMCMC is significantly larger due to the instantiation of separate memory replicates for each chain (the maximum number of chains was set to $M_{max} = 5$). The memories which are replicated are listed in 4.2 (see last column). The estimation of the extra

Table 4.3: Resource utilization of the pMCMC/ppMCMC IPs and the respective FPGA systems. Architecture parameters were set to $B = 2$, $d_{rng} = 30$, $P_{max} = 8192$, $T_{max} = 8192$ (for both samplers), $M_{max} = 5$ (for ppMCMC). The numbers in the parentheses show what percentage of the available Z-7045 resources is needed for the given block.

Block name	LUTs	FFs	DSPs	BRAMs
pMCMC IP	85593 (39.1%)	109017 (24.9%)	710 (78.8%)	223 (40.9%)
ppMCMC IP (estimate)	87122 (39.8%)	110543 (25.2%)	752 (83.5%)	496 (91.0%)
Others (AXI bus, DMA, CPU)	729 (0.3%)	926 (0.2%)	0 (0%)	0 (0%)
pMCMC system total	86322 (39.4%)	109943 (25.1%)	710 (78.8%)	223 (40.9%)
ppMCMC system total (estimate)	87851 (40.1%)	111469 (25.5%)	752 (83.5%)	496 (91.0%)
Xilinx Z-7045 total resources	218600 (100%)	437200 (100%)	900 (100%)	545 (100%)

BRAMs was done based on the memory sizes of Table 4.2 and the post-place and route results of pMCMC.

There are two resource types which limit the implementation in different ways: 1) The critical computational resource for both samplers (i.e. the one that limits B - the degree of parallelism) is the number of available DSPs in the device. The chosen FPGA device has 900 DSP and this limits the degree of parallelism to $B = 2$. LUT and FF utilization is significantly lower. 2) The number of available BRAMs in the device forces an upper limit on the problem sizes that can be addressed in pMCMC/ppMCMC (T_{max} and P_{max}) and the number of chains that can be used in ppMCMC (M_{max}). The maximum number of SSM states and particle for pMCMC using the particular FPGA device is $T_{max} = 16384$ and $P_{max} = 16384$ (although the results in Table 4.3 were generated using the combination $T_{max} = 8192$ and $P_{max} = 8192$). Other combination can also be applied, e.g. $T_{max} = 8192$ and $P_{max} = 32768$. For ppMCMC, an additional limitation is the maximum number of chains M_{max} , which has been set to $M_{max} = 5$ here. In order to run more chains, either a larger FPGA device with more BRAMs has to be used or T_{max} and P_{max} need to be reduced.

4.7.3 pMCMC: Hardware comparison (uni-modal posterior)

In order to compare the performance of the CPU, GPU and FPGA samplers, the single-tissue model of Section 4.5 was used with various data sizes (i.e. number of SSM states or time steps or T). The number of particles in the PF (P) was also set to different values to investigate how the performance of the samplers scales as the problem size increases. For each combination of T and P , the pMCMC samplers were used to generate $N = 10000$ MCMC samples from the uni-modal posterior of the single-

tissue SSM. This section compares the samplers' performance in two different ways:

1. Comparison based only on the runtime necessary to generate N MCMC samples, i.e. $Time_{total}$, for various values of P and T .
2. Comparison based on both the runtime as well as the effect that P has on the mixing of the sampler.

The performance metric for the first comparison is the samples per second that the sampler can generate:

$$Samples/sec = \frac{N}{Time_{total}} \quad (4.35)$$

This metric is also called “raw performance” in this section and the speedup that refers to it is called “raw speedup”. The performance metric for the second comparison is the effective samples per second that the sampler can generate:

$$ES/sec = \frac{ESS}{Time_{total}} \quad (4.36)$$

where ESS is the effective sample size of the N MCMC samples. Effective sample size (due to autocorrelation) [156] is the most common metric of MCMC mixing in the literature (see also Chapter 2); it estimates how many independent (effective) samples the dependent MCMC samples are equivalent to, i.e. it quantifies the “exploration value” of the samples. This is necessary for assessing pMCMC performance, since the number of particles (P) influences mixing: A sample from pMCMC with $P = 512$ has a different “exploration value” compared to a sample from pMCMC with $P = 256$. ES/sec simultaneously considers raw speed ($Time_{total}$) and mixing speed (ESS); it is the metric that ultimately interests a practitioner. The same metric was used in Chapter 3 to quantify the effect of custom precision on the mixing of the Parallel Tempering algorithm. This metric is also called “effective performance” in this section and the speedup that refers to it is called “effective speedup”.

Raw performance comparison

Figures 4.5 and 4.6 compare the various samplers using the metric (4.35). They show how the raw speedups of the GPU and FPGA samplers over the multi-core CPU sampler scale with the number of particles when $T = 1000$ and when $T = 16000$ respectively. All speedups refer to a fixed P , i.e. comparisons are made between different platforms with the same P . The actual runtimes of the CPU sampler for $T = 1000$ ranged from 58 minutes (for $P = 256$) to 2.4 days (for $P = 16384$). For $T =$

16000, the actual runtimes ranged from 13.6 hours (for $P = 256$) to 37.2 days (for $P = 16384$). The largest runs (which were impractical to perform) were terminated early (e.g. for $N = 100$ or $N = 10$) and the measured runtimes were multiplied by the necessary factor. This does not affect the accuracy of the comparisons since MCMC iterations are purely sequential. It is worth noting that typically more than 10000 samples are needed in pMCMC applications and many independent runs have to be performed. Therefore, the runtimes of LibBi's multi-core CPU implementation are clearly prohibitive.

The achieved raw speedups of each device are very similar between Figures 4.5 and 4.6, showing that the number of states does not affect the performance of the samplers (which is expected, since states are processed sequentially and not in parallel). In contrast, the number of particles affects the raw speedups of the GPU and FPGA, since more particles offer more parallelism which can be exploited by the devices. The GPU is slower than the CPU by up to 4.1x for small numbers of particles but becomes faster by up to 3.9x for $P \geq 2048$. This significant increase in the GPU's raw speedup shows that the GPU is under-utilized when only a small amount of parallel computations are performed. For 8192-16384 particles the raw speedup of the GPU flattens out because the GPU is fully utilized.

The FPGA's raw speedup over the CPU is 6.4x-14.9x. Its raw speedup also increases with P but it reaches values close to its peak value much earlier than the GPU. This is due to the FPGA architecture's ability to exploit even modest amounts of parallelism in the algorithm. The speedup of the FPGA is 3.8x-30.8x higher than the raw speedup of the GPU. The speedup over the GPU decreases with larger P as the GPU gradually reaches its peak performance (the FPGA is close to its peak performance even for $P = 1024$).

Regarding the multi-core CPU sampler, it has to be noted that its runtime grows proportionately to the number of particles (although this is not shown in the figure). This reveals that even 256 particles are enough to fully utilize the CPU's resources.

Figure 4.7 examines the raw performance of the FPGA sampler (i.e. without taking into account the mixing of the samples) in a different way. It shows the number of FPGA clock cycles spent for each step of the pMCMC architecture for different numbers of particles. The numbers come from the performance model of Section 4.4.4. In other words, the figure shows how the latency functions of Section 4.4.4 vary with the parameter P (number of particles) when T , B and N are fixed to the values shown in the figure caption. The figure also shows the actual (from a real run) runtime of the pMCMC architecture (taken from Figure 4.5) converted to FPGA clock cycles.

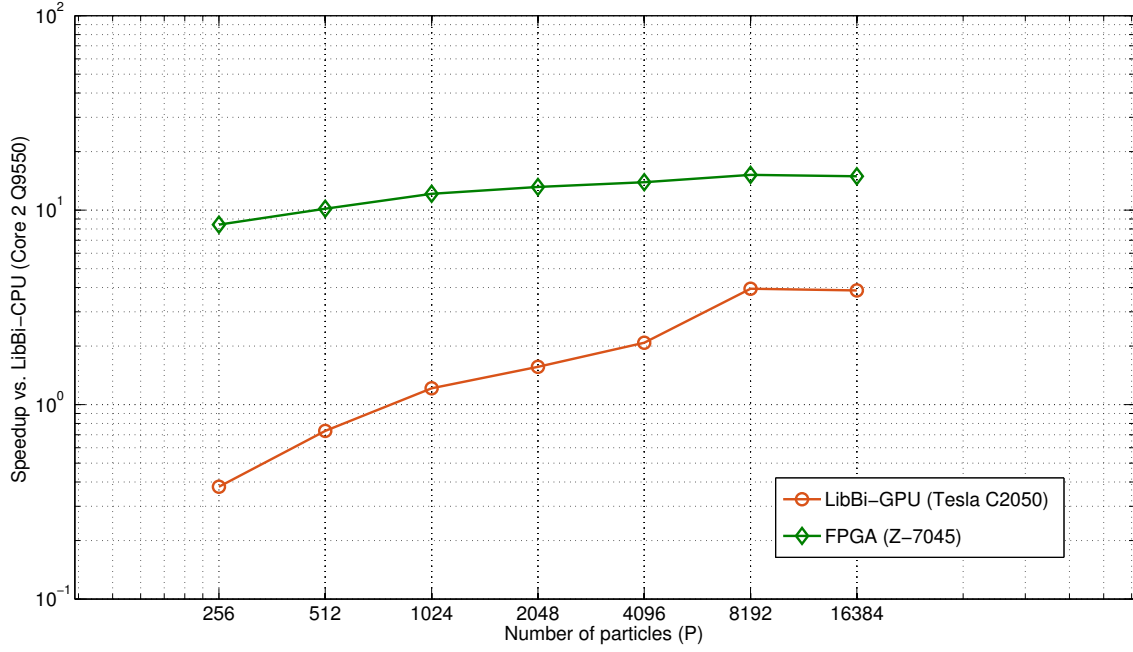


Figure 4.5: Raw speedup of GPU [2] and FPGA vs. multi-core CPU [2] implementation of pMCMC. The number of SSM states/time steps is set to $T = 1000$. Measured runtimes include time to transfer data between the devices and the hosts.

From comparing the purple and black lines in the figure, it is clear that the proposed performance model is accurate, with relative error between 0.7% and 7.5%. The larger error is observed for $P = 256$ due to the larger relative effect of the CPU and I/O clock cycles (which cannot be modelled precisely) on the total clock cycles.

Moreover, the figure reveals how the total clock cycles are broken down into the three stages of the PF. It is clear that, for the particular SSM case study, the Transition and Weight stage is the one that consumes the fewest cycles among the three PF stages. The resampling stage is the most expensive stage. Also, the PF clock cycles increasingly dominate the total clock cycles of the system as more particles are used; for $P = 256$ the PF takes 54.2% of the total clock cycles but this increases to 98.5% for $P = 16384$. Similar results were observed for other values of T . The CPU and I/O clock cycles constitute a large part of the total cycles for $P = 256$ (45.6%) but become negligible for large P (1.4% of the total for $P = 16384$).

Figure 4.8 illustrates the way the clock cycles of the various steps of pMCMC change when the degree of parallelism of the architecture (B), i.e. the number of parallel modules in each PF stage, scales. This is equivalent to investigating how the clock cycles (or the time) of each step change when more resources are available to implement the computations of the step. All the numbers in Figure 4.8 are

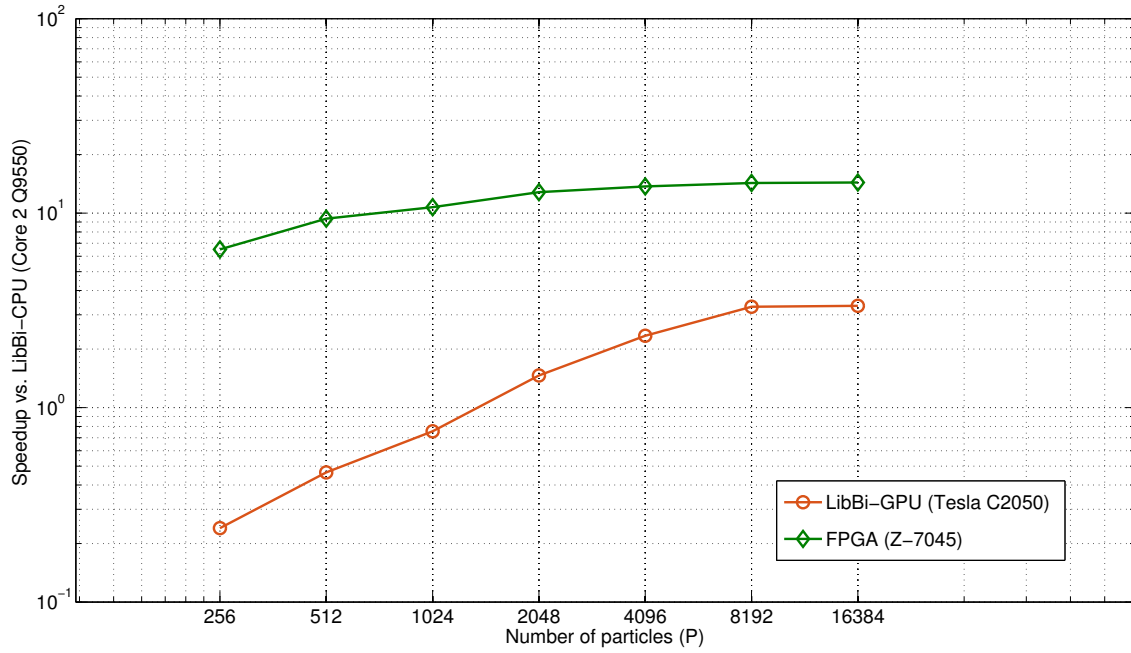


Figure 4.6: Raw speedup of GPU [2] and FPGA vs. multi-core CPU [2] implementation of pMCMC. The number of SSM states/time steps is set to $T = 16000$. Measured runtimes include time to transfer data between the devices and the hosts.

based on the performance model of Section 4.4.4, using a fixed problem size ($T = 1000$, $P = 16384$). In other words, the figure shows how the functions of Section 4.4.4 vary with the parameter B when T , P and N are fixed.

The figure reveals that the Resampling stage of the PF becomes the bottleneck computation when B increases. For $B = 1$, the Resampling stage accounts for 46.5% of the total clock cycles. For $B = 16$, the Resampling stage accounts for 90.1% of the total clock cycles. The number of Resampling clock cycles decreases only slightly when instantiating more parallel processing modules. This happens because of the particle replication step at the end of the Resampling stage, which is represented by the term $P \cdot Lat_{rep}$ in Equation (4.16). It is clear from the equation that this is the only non-constant term in all three PF stages (Equations (4.14)-(4.16)) which does not have a denominator B . This is due to the reasons mentioned in Section 4.4.2; the particles replication cannot be parallelized because memory access conflicts have to be avoided. The clock cycles of the remaining PF stages (Transition and Weight, Partial Sums) decrease almost proportionately with B (which is expected from looking at Equations (4.14) and (4.15)). The total number of clock cycles decreases with larger B but the speedup gains are diminishing. The figure also shows what degree of parallelism is accessible given the resources of five real Xilinx devices. As mentioned previously, Zynq Z-7045 was used as an

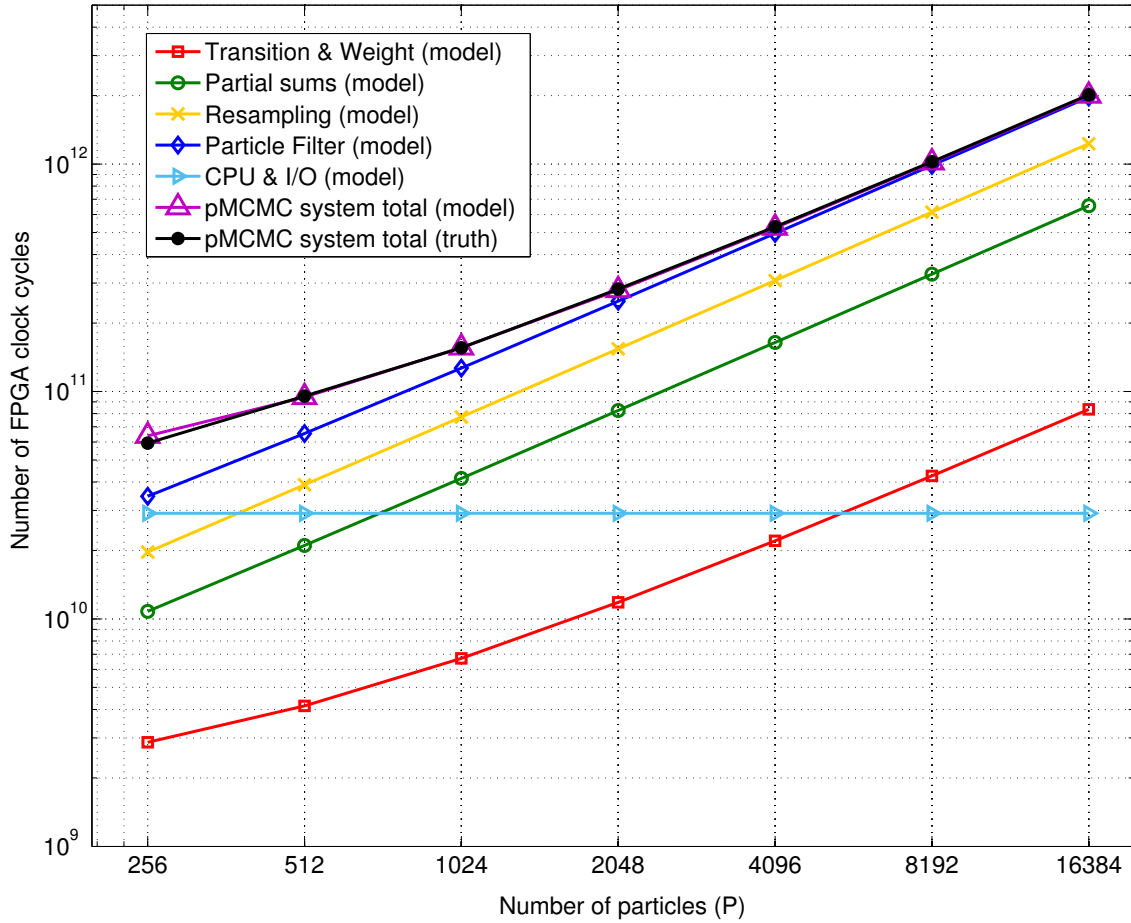


Figure 4.7: Total clock cycles consumed by each stage of the pMCMC architecture as the number of particles (P) changes. The remaining parameters are set to $T = 1000$, $B = 2$, $N = 10000$, $T_{max} = 16384$, $P_{max} = 16384$.

implementation platform in this chapter. The other four devices' parallelism degrees were estimated based on their available resources and the resource utilization results from Z-7045.

Effective performance comparison

The above comparison of the pMCMC accelerators using the first metric (*Samples/sec*) provides information on how the relative speed of the accelerators scales with the number of particles. This is useful for assessing raw speed and choosing an accelerator when the number of particles is known (e.g. it has already been chosen by the practitioner). Nevertheless, when the number of particles has to be tuned, the second performance metric (*ES/sec*) has to be used instead, since it takes into account both the raw speed and the effect of P on mixing.

Figure 4.9 shows the *ESS* of the pMCMC sampler (which is the same for all implementations - CPU, GPU, FPGA) when P varies. It is clear that the gains in mixing are significant when P ranges from

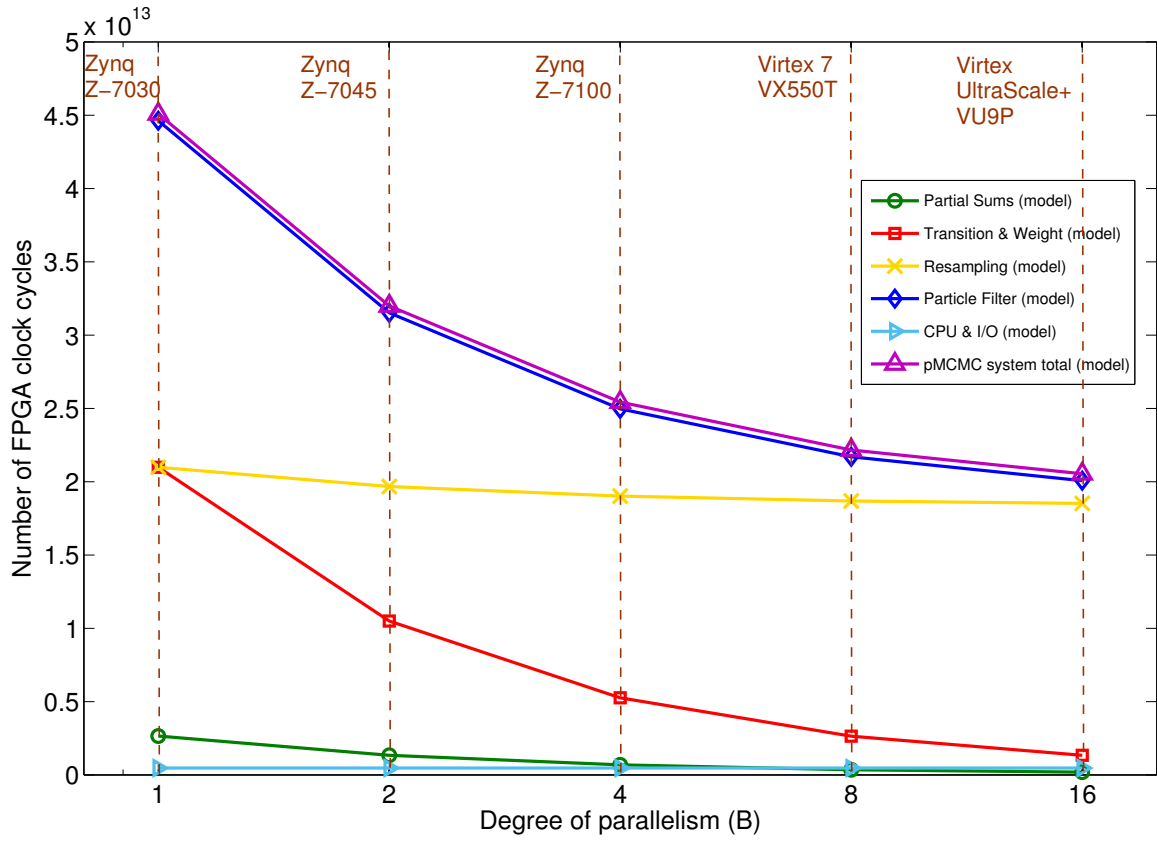


Figure 4.8: Total clock cycles consumed by each stage of the pMCMC architecture as the degree of parallelism (B) changes. The remaining parameters are set to $T = 1000$, $P = 16384$, $N = 10000$, $T_{max} = 16384$, $P_{max} = 16384$. Five FPGA devices are shown in the top horizontal axis in order to demonstrate what degree of parallelism is possible with each device's available resources (assuming full resource utilization). An actual implementation was done only for Zynq Z-7045. The other four devices were placed in the graph based on projections.

256 to a 1024 but grow at a slower rate for larger P . The fluctuation in the figure is due to the variance of the ESS estimator.

Figure 4.10 shows the ES/sec (effective performance) of the multi-core CPU, GPU and FPGA pMCMC samplers for varying P . Absolute performances (instead of speedups) are shown in order to make it possible to compare configurations which use different values of P . The results give a significantly different image of how the performance changes with P compared to the previously presented results. The ES/sec of all samplers increases with P until $P = 1024$ and then drops for larger P . This is because the ESS grows faster than the runtime for $P \leq 1024$ (for all platforms). This is clear by observing the steep increase of ESS in Figure 4.9 for $P \leq 1024$. In contrast, for $P > 1024$, ESS improvement from using larger P is outweighed by the longer runtimes. For the CPU sampler, runtime increases proportionately to P for all P and ES/sec peaks for $P = 1024$. The GPU sampler's performance also peaks for $P = 1024$. It then drops at a slower rate than the CPU sampler's performance,

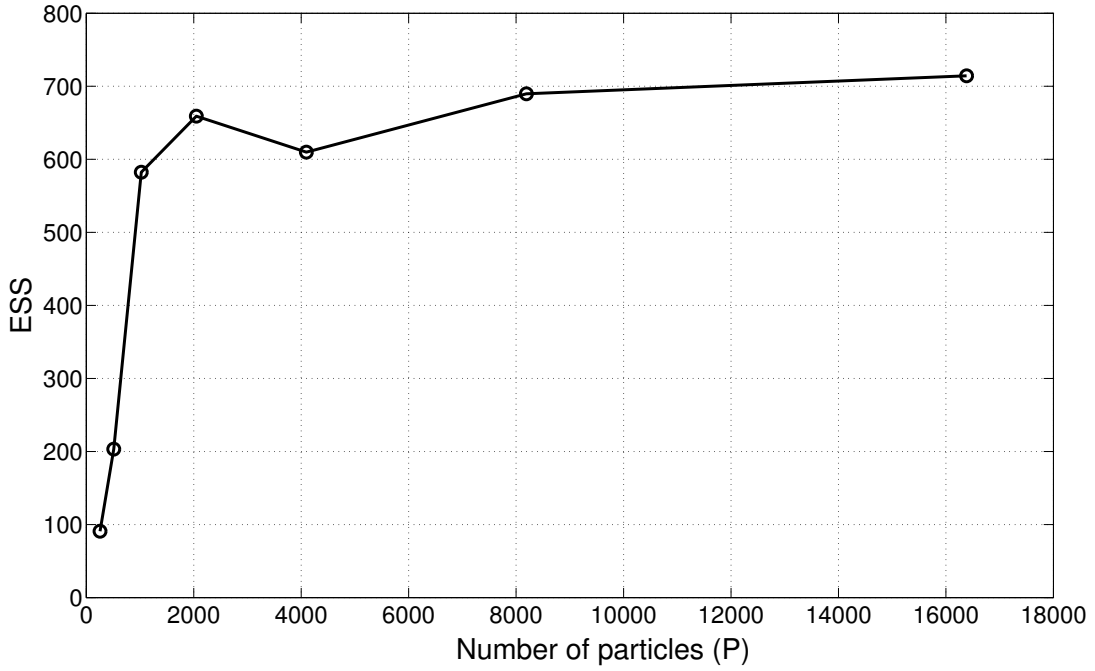


Figure 4.9: *ESS* achieved by pMCMC sampler for various P when sampling from the uni-modal SSM posterior. The other parameters are set to $T = 1000$ and $N = 10000$. The fluctuations in the graph are due to the fact that the *ESS* value is approximated and thus variance is present.

staying close to the peak value until $P = 16384$. This is because the GPU's massive parallel resources allow it to increase P without paying a large penalty in runtime (runtime grows slower than proportionately with P). This small penalty allows the GPU to be relatively efficient (in terms of *ES/sec*) for large P . The peak GPU performance is 1.2x higher than the peak CPU performance. The FPGA's *ES/sec* also peaks at $P = 1024$. The peak FPGA performance is 12.1x higher than the peak CPU performance and 10.0x higher than the peak GPU performance. Larger P leads to reduced *ES/sec*. The reduction happens at a faster rate than in the GPU case because the FPGA achieves close-to-full resource utilization earlier than the GPU (as demonstrated in Figures 4.5 and 4.6) and thus the FPGA's runtime grows almost linearly after $P = 1024$. This leads the performance gap between FPGA and GPU to decline with larger P . Finally, notice that for $P \geq 8192$ the performance of all devices drops at a similar rate, since all of them are fully utilized (and thus runtime grows proportionately to P).

The above results (although coming from applying pMCMC to a specific SSM posterior) demonstrate that a larger number of particles is not always preferable for high sampling efficiency (when mixing is taken in to account). This has already been shown in Pitt et al. [130], Sherlock et al. [132] and Doucet et al. [131] but with the assumption that pMCMC runtime grows proportionately to P . In contrast, the above comparison considers the runtimes of parallel pMCMC accelerators, which grow slower than

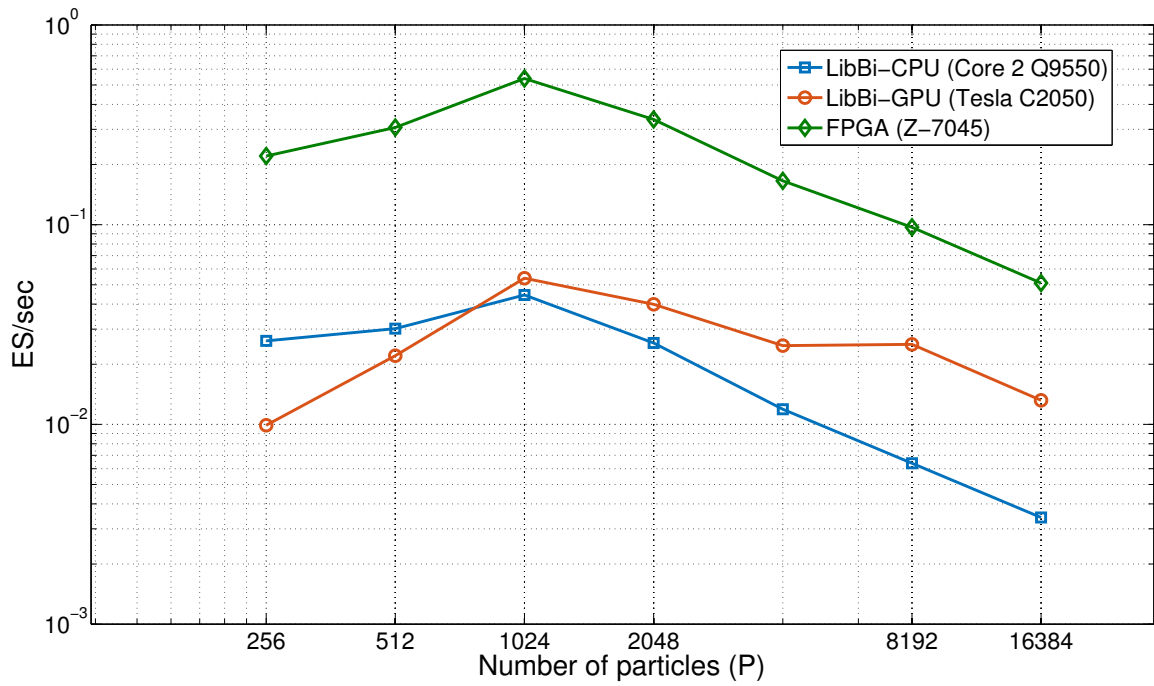


Figure 4.10: ES/sec achieved by the multi-core CPU, GPU and FPGA pMCMC samplers for various P when sampling from the uni-modal SSM posterior. The other parameters are set to $T = 1000$ and $N = 10000$.

proportionately to P until parallel resources are fully utilized (apart from the CPU sampler where full utilization is reached even for $P = 256$).

It has to be noted that, in order to perform the above optimization, MCMC runs are necessary for all P values and all devices. These runs can be short if the algorithm converges quickly, i.e. if few samples are required to get an accurate estimate of ESS . The examined case study belongs to this category; accurate ESS estimates are produced even with $N = 1000$ pMCMC samples. In cases where pMCMC needs a large number of samples to converge, the overhead of performing runs for all P settings can be significant. Nevertheless, even in this case, the typical scenario is that a practitioner performs the few runs required to optimize P , as well as runs to optimize other parameters of the algorithm and then uses the optimal parameters to perform a set of final runs. This set typically consists of dozens of independent runs which are longer than the optimization runs. The final runs are used to get the output estimate (2.9) and its variance. Therefore, the set of long final runs typically takes much more time than the few (relatively short) runs needed for parameter optimization.

Note that no ppMCMC results are included in this section. This is because the ppMCMC algorithm is unsuitable for this problem; using extra chains does not improve mixing for uni-modal posteriors. Therefore, the extra area and extra computations of the ppMCMC FPGA architecture make it less

efficient than the pMCMC FPGA architecture in all scenarios mentioned above.

4.7.4 ppMCMC vs. pMCMC: Algorithm comparison and trade-offs (multi-modal posterior)

In this section, the performance of the proposed ppMCMC algorithm when sampling from a multi-modal distribution is evaluated, without the use of any hardware acceleration. The multiple-tissue posterior of Section 4.5 is used as the target distribution and sequential software implementations (in Matlab) of pMCMC and ppMCMC are compared in order to reveal what sampling efficiency gains are possible by using different combinations of MCMC chains and PF particles and how the two algorithms' performances compare. Matlab is used instead of a LibBi CPU implementation because LibBi does not support ppMCMC. The temperatures of ppMCMC were chosen based on manual tuning as follows: For $M = 2$ the setting $Temp_{1:2} = [1, 2.5]$ was used, for $M = 3$ the setting $Temp_{1:3} = [1, 2.5, 5]$ was used, for $M = 4$ the setting $Temp_{1:4} = [1, 2.5, 5, 10]$ was used and for $M = 5$ the setting $Temp_{1:5} = [1, 2.5, 5, 10, 15]$ was used.

The performance metric used in this section is ES/sec (described in the previous section). This metric is necessary for the same reason described previously, i.e. to take into account the effect of mixing in total efficiency. Moreover, in the case of ppMCMC, not only P but also M (the number of ppMCMC chains) affects mixing. For example, a sample from ppMCMC when using $M = 3$ MCMC chains has a different “exploration value” compared to a sample from ppMCMC when using $M = 2$ MCMC chains or a sample from pMCMC which is generated with only one MCMC chain.

Figure 4.11 shows the ESS and ES/sec of the Matlab implementation of ppMCMC when the number of chains ranges from $M = 2$ to $M = 5$ and the number of particles ranges from $P = 150$ to $P = 8200$. The ESS and ES/sec of the Matlab implementation of pMCMC for the same P range is also shown. The number of SSM states is fixed to $T = 200$. $N = 10000$ MCMC samples are generated for each combination. Larger values of M (for ppMCMC) and P (for both methods) improve mixing (ESS in (4.36)) but at the same time they increase runtime ($Time_{total}$ in (4.36)). The latter is not shown here but it grows proportionately to P and M (since the implementation is sequential). By exploring the above trade-off between ESS and $Time_{total}$ it is possible to find the optimal combination of M and P for ppMCMC, as well as the optimal P for pMCMC (the latter optimization was also done for the single-tissue posterior in the previous section).

It is clear from the figure that mixing improves with both M and P but most of the gains come with moderate numbers of chains and particles, e.g. up to $M = 3$ and $P = 1200$. Increasing both values above those limits offers small gains. The above design space exploration is done for the first time in the pMCMC literature. Previous works have only addressed the way the mixing of pMCMC changes with the size of the particle set.

Regarding ES/sec , in pMCMC it is maximized for $P = 300$, while in ppMCMC it is maximized for $(P = 300, M = 3)$. The latter is 1.96x faster than the former. For the same P , using multiple chains is more efficient (in terms of ES/sec) than single-chain pMCMC by up to 2.8x (despite the extra computational cost from running multiple chains). The only ppMCMC configuration which is slower than pMCMC is $(P = 150, M = 2)$. The figure also indicates that increasing P above a certain number decreases ES/sec for any M . This happens because the gains in ESS are small for very large P , while $Time_{total}$ increases proportionately to P . The optimal ES/sec values when fixing M are achieved either for $P = 300$ or $P = 600$, depending on M . The gains from using ppMCMC are expected to increase when applied to SSM posteriors with larger dimensions (n_θ) and more modes (the example used here has $n_\theta = 3$ and 2 modes).

Note that the above optimization of P and M also requires runs to be performed for all candidate parameter combinations, as described in the end of Section 4.7.3 for pMCMC. The same comments on the overhead of these runs apply here (and in the following section).

4.7.5 ppMCMC vs. pMCMC: Hardware comparison and trade-offs (multi-modal posterior)

In this section, the (estimated) performance of the proposed ppMCMC FPGA architecture is evaluated using the multi-tissue model. The same chains/particles design space exploration is performed, as in the software-based comparison of the previous section. Moreover, the FPGA ppMCMC performance is compared to the CPU and GPU implementations of pMCMC (which use LibBi [2]). The runtimes of the FPGA sampler were estimated based on the model of Section 4.4.4.

The ES/sec metric for the FPGA implementation is affected by two factors, as mentioned previously; the runtime ($Time_{total}$) and the mixing (ESS). The former factor is illustrated in Figure 4.12, which shows the raw speedup of FPGA ppMCMC compared to Matlab ppMCMC when considering only the runtime $Time_{total}$ and not the effect of mixing. This is equivalent to the $Samples/sec$ (raw

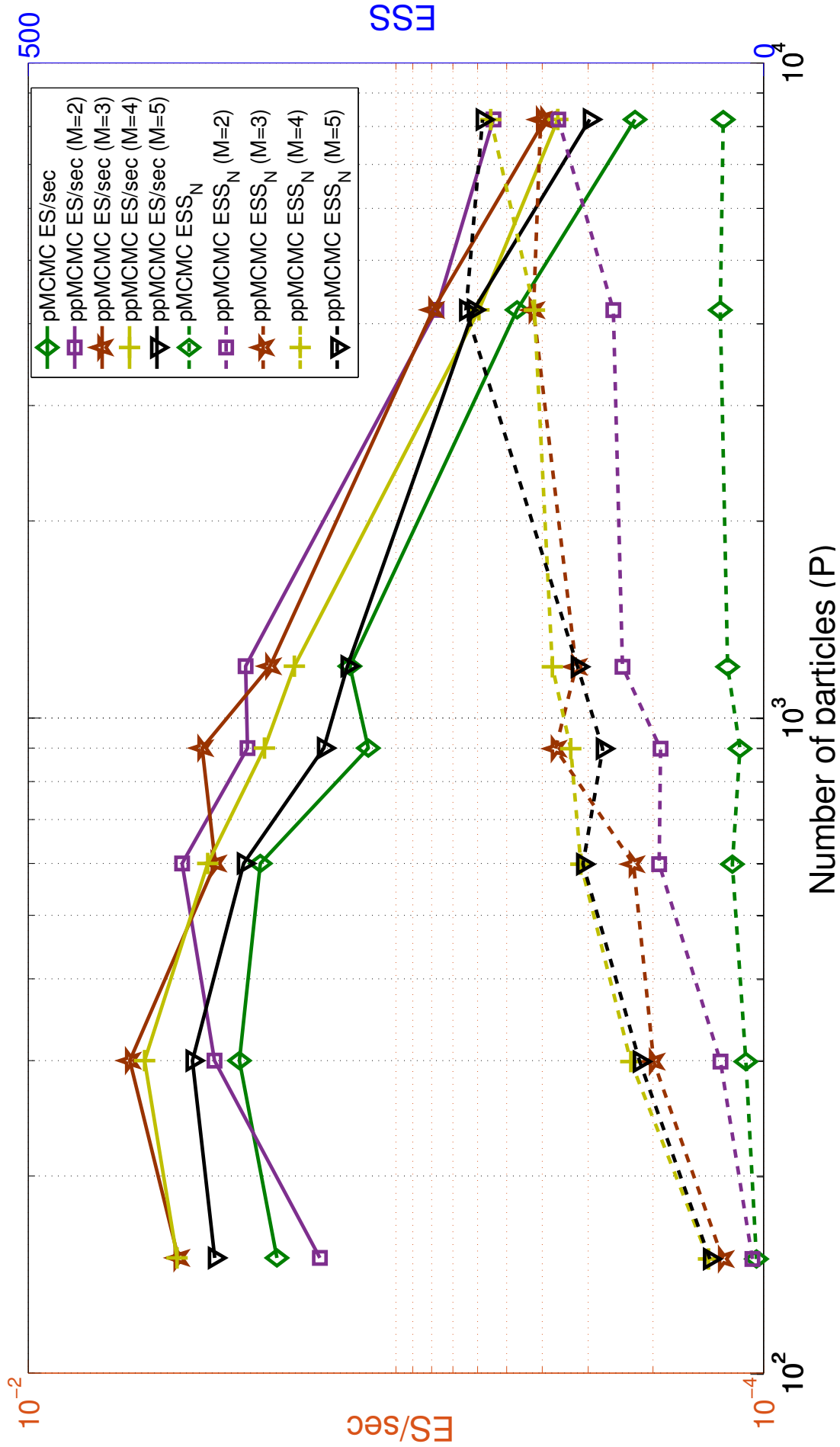


Figure 4.11: ESS and ES/sec of pMCMC and ppMCMC in sequential Matlab implementations when the number of chains (M) and the number of particles (P) change. The multi-modal, multi-tissue SSM of section 4.5 is used as the target distribution. The SSM time steps are fixed to $T = 200$. $N = 10000$ samples are generated for each combination and the ESS and runtime are measured in order to compute ES/sec .

performance) metric described in Section 4.7.3 for pMCMC. The same parameters as in the Matlab evaluation are used, i.e. $T = 200$ and $N = 10000$. The number of chains and particles varies. The actual runtimes of the Matlab sampler range from 76 minutes for $P = 150$ and $M = 2$ to 7.3 days for $P = 8200$ and $M = 5$. The raw speedup of the FPGA pMCMC sampler over the Matlab pMCMC sampler is also shown.

From observing the figure, it is clear that the use of multiple chains improves the efficiency of the FPGA architecture, i.e. the raw speedup over Matlab improves with larger M . There is a gain of up to 2.6x in raw speedup when moving from $M = 1$ (pMCMC) to $M = 5$ when P is fixed. This is due to the use of coarse-grain pipelining, which was described in Section 4.4.3 and which increases the number of MCMC chains that the PF can process per second. Moreover, the figure shows an improvement of raw speedup with the number of particles (which is also supported by the earlier results for pMCMC, see Figures 4.5 and 4.5). This is due to increased utilization of FPGA resources when P is large. Overall, the raw speedup in Figure 4.12 ranges from 31.8x to 132.9x.

The second factor that affects the ES/sec metric for the FPGA implementation is mixing (measured by ESS). The effect of this factor has already been explained and presented in Figure 4.11. The ESS of the FPGA implementations is the same as the ESS of the Matlab implementations when the same P and M are used.

A comparison of the two factors that affect ES/sec (mixing and runtime/speedup over Matlab) reveals that mixing is more important because ESS varies significantly (from $ESS = 4.8$ to $ESS = 202.4$ as shown in Figure 4.11) while speedup over Matlab varies from 31.8x to 132.9x (Figure 4.11).

The ES/sec of the FPGA ppMCMC sampler is shown in Figure 4.13 for all M , along with the ES/sec of the FPGA pMCMC, GPU pMCMC and CPU pMCMC samplers. The same parameters as in the Matlab evaluation, i.e. $T = 200$, $N = 10000$, are used. The ES/sec of the CPU pMCMC peaks for $P = 300$ and drops for larger P because the extra computational cost outweighs the ESS benefit (the latter is equal to the ESS in Figure 4.11). The same behaviour was observed when applying the CPU pMCMC to the single-tissue, uni-modal posterior in Section 4.7.3. The ES/sec of the GPU pMCMC peaks at $P = 600$ and remains close to its peak value for larger P because the computational cost here is smaller than in the CPU case (the GPU has more parallel resources to utilize and pays small $Time_{total}$ penalties for larger P , as already explained earlier). The ES/sec of the FPGA pMCMC peaks at $P = 600$. This peak performance is 10.7x and 12.8x higher than the peak CPU and GPU

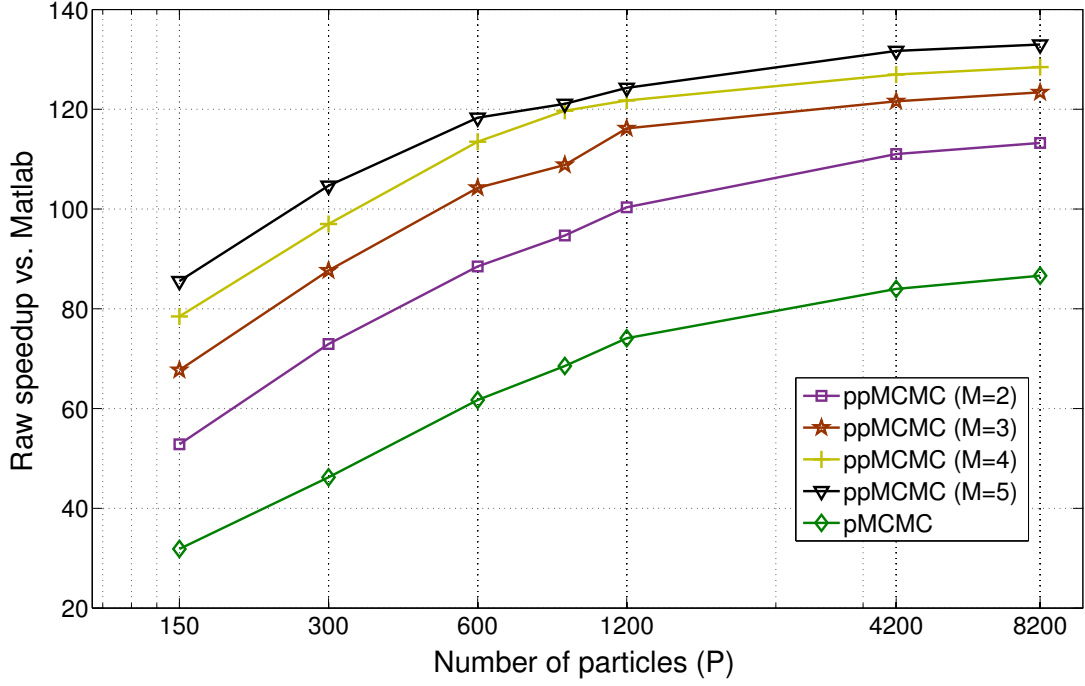


Figure 4.12: Raw speedup of FPGA ppMCMC sampler compared to sequential Matlab implementation of ppMCMC and raw speedup of FPGA pMCMC sampler compared to sequential Matlab implementation of pMCMC. the number of chains (M) and the number of particles (P) change. The multi-modal, multi-tissue SSM of section 4.5 is used as the target distribution. The SSM time steps are fixed to $T = 200$.

performances respectively. The fastest ppMCMC configuration ($P = 300$, $M = 4$) is 34.9x, 41.8x and 3.24x faster than the fastest CPU, GPU and FPGA pMCMC configurations. Notice that the optimal ppMCMC configuration changes from Matlab to FPGA. By comparing Figures 4.11 and 4.13, it can be observed that combinations with large P and (especially) M are more “favoured” in the FPGA than they are in Matlab. This happens because 1) FPGA runtime does not increase proportionately to P and M as in Matlab (it increases at a slower rate) and 2) chain pipelining in the ppMCMC architecture improves efficiency for $M > 1$. For constant P , adding chains improves ES/sec by up to 3.96x vs. pMCMC, while in Matlab the equivalent improvement is 2.8x.

4.8 Conclusions

This chapter introduced a novel particle-based MCMC algorithm called ppMCMC, which combines pMCMC with the principles of population-based MCMC methods in order to achieve higher mixing speed in multi-modal target distributions. Moreover, two FPGA architectures (one for the basic pMCMC and one for ppMCMC) were proposed, taking advantage of the flexibility of FPGAs to optimize

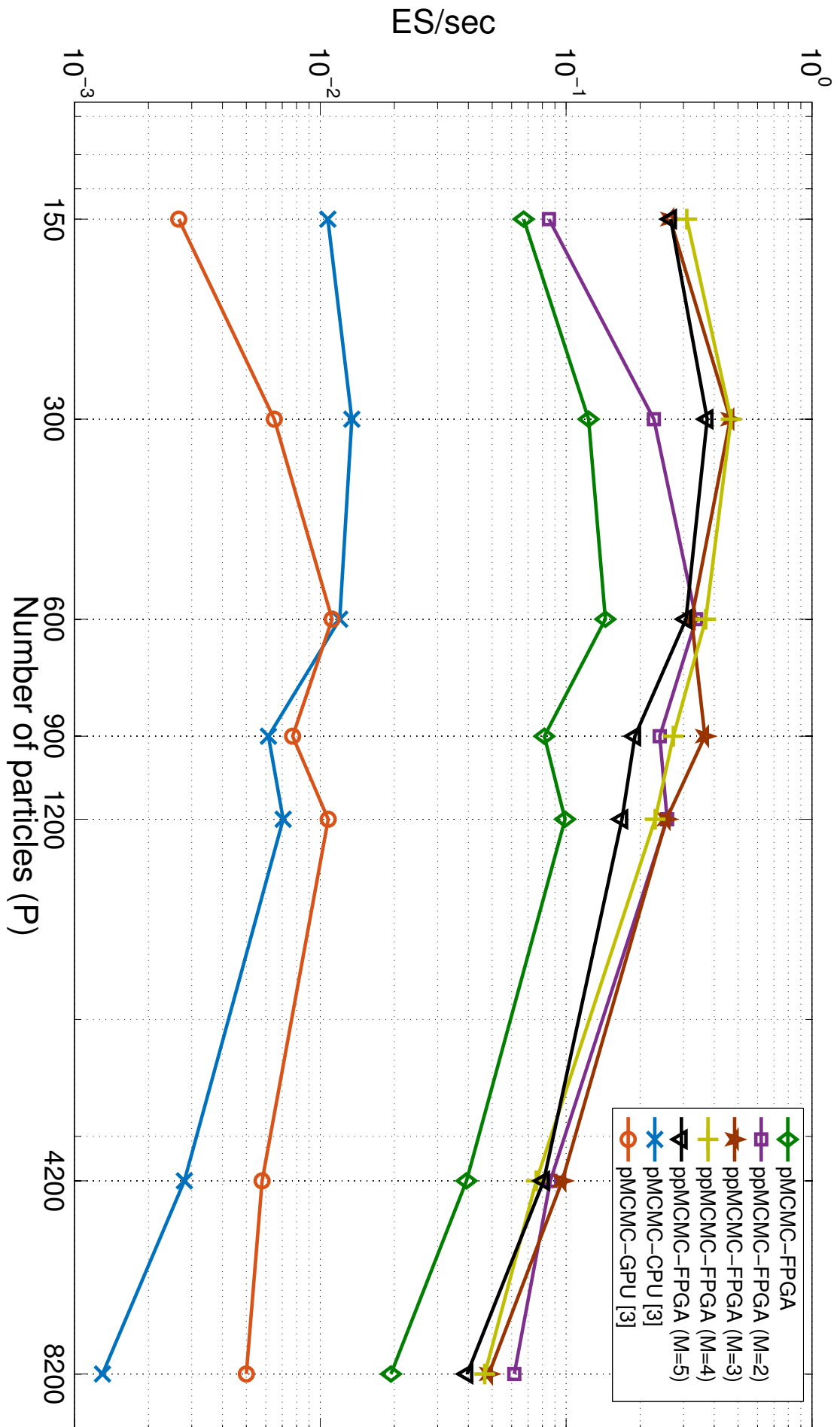


Figure 4.13: *ES/sec* of FPGA pmMCMC, FPGA pmMCMC, GPU pmMCMC and CPU pmMCMC samplers when sampling from the multi-modal, multi-tissue SSM of section 4.5. The number of chains (M) and the number of particles (P) change. The SSM time steps are fixed to $T = 200$ and $N = 10000$.

the speed of the samplers.

The evaluation of the algorithm and the architectures showed that the FPGA samplers are significantly faster in terms of sampling speed compared to state-of-the-art CPU and GPU samplers when applied to a SSM inference for DNA methylation problems. Results showed that the FPGA pMCMC architecture reaches its peak performance earlier than the GPU. The performance of the FPGA pMCMC sampler does not increase proportionately to the FPGA device's size, because the resampling step involves a non-parallelizable operation (generation of new particle population by replicating particles from the previous population). This issue is the subject of future work. It could be addressed by the use of distributed resampling algorithms which do not require all the parallel nodes to have access to the whole particle memory, although the effects of this strategy on resampling performance need to be examined.

Regarding the ppMCMC FPGA sampler, results showed that by exploiting the design space trade-offs of the architecture (i.e. tuning the number of particles and the number of chains), it is possible to find the parameter combination which maximizes effective sampling performance. The speed gains due to the ppMCMC algorithm (vs. pMCMC) without hardware acceleration, as well as the gains due to the ppMCMC FPGA architecture vs. the pMCMC FPGA architecture and pMCMC samplers in other platforms were quantified. The FPGA ppMCMC sampler was shown to be an order of magnitude faster than the CPU and GPU pMCMC samplers and up to 3.24x faster than the FPGA pMCMC sampler when sampling from a multi-modal distribution. The above results confirm that the combination of ppMCMC and a specialized FPGA architecture offers significant efficiency gains over existing algorithms and accelerators when the posterior is multi-modal.

It is worth noting that ppMCMC is an unbiased algorithm; in all experiments that were performed, ppMCMC converged to the “true” posterior densities (confirmed either by comparing to pMCMC's results on the same target distribution or by comparing to the known “true” model), as theoretically expected.

The design space explorations of Section 4.7.3-4.7.5 are performed for specific SSM case studies. Therefore, the optimal parameter configurations found cannot be generalized to other problems. Depending on the posterior's shape, dimension, number of modes, distance between modes, etc, different parameter configurations will lead to the best mixing and to the best effective performance (ES/sec). However, it is expected that for all problems the ES/sec metric will not increase monotonically with

P and M (for ppMCMC), either in software or in hardware. There will always be some intermediate combination of P and M that maximizes ES/sec . Also, any performance gap between FPGAs and GPUs is expected to gradually reduce with P and M and GPU samplers will generally reach peak effective performance for larger values of P and M compared to FPGAs and CPUs.

Regarding the raw speedup of FPGAs vs. CPUs and GPUs, the results presented in Figures 4.5 and 4.12 can be generalized more easily than effective speedup results. The FPGA should reach similar speedups over other platforms for different problems, as long as the same numbers of particles are used. Transition and observation densities in SSMs are typically well-known densities (e.g. Gaussian, Binomial, Poisson) which require similar arithmetic operators to implement. Therefore, the implementation costs and thus the performance differences between devices are likely to be maintained for other SSMs. The critical factor that defines raw performance differences between devices is the chosen parameters of the algorithm (P and M); the performance of each device scales differently with these parameters, as shown in Figures 4.5 and 4.12.

Overall, the pMCMC and ppMCMC algorithms can benefit a lot from FPGA acceleration despite the complications related to the resampling step of the PF. The following chapter will approach the problem of MCMC acceleration from a different perspective compared to the rest of this thesis. It will focus on generic MCMC acceleration (independent of the targeted MCMC algorithm) and it will propose a precision optimization method to maximize the speedup offered by an FPGA implementation.

Chapter 5

Arithmetic precision optimization for generic MCMC

5.1 Introduction

The two preceding chapters examined specific classes of MCMC algorithms and exploited their inherent parallelism in order to construct parallel implementations. They also proposed novel modifications to these algorithms, which combined knowledge about the characteristics of the algorithms and the features of FPGAs, achieving an increase in sampling efficiency. Although the gains from this approach were shown to be significant, each algorithmic modification and FPGA mapping presented in the previous chapters is specific to the targeted MCMC algorithm or to algorithms from the same MCMC family (e.g. population-based MCMC).

This chapter follows a different path; instead of focusing on a specific MCMC class, it targets the entirety of MCMC methods and seeks a general way to improve sampling efficiency in an FPGA setting. This general approach comprises using custom (reduced) arithmetic precision when evaluating the probability density $p(\theta)$ inside MCMC, instead of the double floating-point precision which is the default approach in MCMC literature. All other parts of the MCMC algorithm operate in double precision. Custom precision is a unique capability of FPGAs (see Chapter 2). By reducing precision, arithmetic operators become cheaper in terms of FPGA area and thus more parallel operators can be instantiated. At the same time though, error is introduced in calculations. In the case of MCMC, error translates to bias in the estimation of the integral of Equation (2.8). This bias in the output estimate of

MCMC has already been defined in equation (2.26) of Chapter 2.

Since all MCMC methods require the computation of a probability density, the above approach has potential as a general method which can increase MCMC's computational efficiency in any problem. What is more, the presence of large-scale data in modern Bayesian application has made the evaluation of the probability density the main computational bottleneck in most MCMC methods. Therefore, focusing on decreasing the computational cost of this particular part of MCMC is a priority.

In order to use custom precision in practice for the above task, it is necessary to figure out what precision configuration is optimal for the probability density evaluations inside MCMC, taking into account the effect of precision on performance and output bias. This chapter poses the following questions:

- “Assuming that the probability density in MCMC is evaluated using custom precision (resulting in a biased output estimate), what precision configuration should be selected to minimize MCMC runtime in the FPGA while simultaneously bounding the bias in the estimation of (2.8)?”
- “Is it possible to discover this configuration without performing full MCMC runs in all precisions?”

The methods and results presented in this chapter show that it is possible to select an optimized precision which satisfies a bias threshold by performing short MCMC pre-runs in multiple precisions on the FPGA and using an efficient bias estimator (which is proposed here).

Chapter outline

Section 5.2 introduces some basic concepts on the use of custom arithmetic in an MCMC setting, including its effect on the convergence of the algorithms and on the output bias. It repeats some of the background of Chapter 2 for easier reference. The following sections constitute the main part of the chapter and include the following contributions:

1. The introduction of an efficient estimator of the bias that appears in the output estimate of MCMC when evaluating the probability density $p(\theta)$ in custom precision (Section 5.3). The estimator requires MCMC samples from a sampler which operates in the custom precision of

interest. It is able to produce a bias estimate using few MCMC samples (compared to the ones needed to get the output estimate, i.e. to get an estimate of the integral (2.8) using (2.9)).

2. A precision optimization method for FPGA-mapped MCMC samplers, which exploits the proposed bias estimator and the reconfigurability of FPGAs to automatically choose a minimized precision from within a pre-defined set of precision configurations, while satisfying the user's bias tolerance requirements (Section 5.4). Due to the stochasticity of MCMC, the output estimate is always approximated within some standard deviation. MCMC practitioners set a target standard deviation and run the algorithm until this target is reached. Given a *specific output estimate* (e.g. the mean of the probability distribution), a target standard deviation for the output estimate, a tolerable bias and a few other user parameters, the automated precision optimization method can find a minimized precision whose bias is within tolerance (with a certain probability). The optimization trades off output accuracy for resource savings (due to reduced precision). Resource savings lead to higher parallelization factors and thus shorter runtimes. The process involves pre-runs on the FPGA (1st bitstream) to estimate the bias of each precision and choose the optimized one for the final, long FPGA run (2nd bitstream). The pre-runs add only a small runtime overhead, owing to the efficiency of the bias estimator and the use of a termination criterion.

The proposed optimization method is automated, can tailor the precision to user requirements (most importantly the kind of output estimate and the bias tolerance) and can be applied to any MCMC algorithm (with the assumption that the likelihood computation inside MCMC is parallelizable in some way but without any assumption on the form that this parallelism takes). The chosen precision is not necessarily the minimum precision that satisfies the bias tolerance (this is why the term *optimized* is used, instead of the term *optimal*), since: 1) The candidate precisions are limited, 2) A smaller precision than the chosen one might satisfy the bias threshold but the available time for pre-runs might not be enough to know this with enough certainty.

The proposed methodology is evaluated using two Bayesian inference problems (Section 5.7): Mixture model inference and neural network regression. The PT algorithm is used to sample from the models. The performance of the methodology is compared with the baseline PT FPGA accelerator of Chapter 3 on the same FPGA. Results show that the optimized-precision designs are 2.85x-4.90x faster than double-precision designs (including the optimization overhead), depending on the type of estimate and the user parameters. Results are also presented on the trade-off between bias in the output estimate

and speedup. Finally, a comparison with an existing unbiased precision optimization methodology [3] is included.

5.2 MCMC estimates and the effect of custom precision

5.2.1 MCMC in infinite and double precision

As mentioned in previous chapters, the typical problem that MCMC aims to solve is the estimation of the integral:

$$I = E_p[f(\theta)] = \int f(\theta)p(\theta)d\theta \quad (5.1)$$

where θ is a random vector, $p(\theta)$ is the probability density of θ (the posterior in Bayesian inference), $f(\theta)$ is a function of interest (e.g. mean, moment) and $E_p[f(\theta)]$ denotes the expectation of $f(\theta)$ when θ is distributed according to $p(\theta)$. The above integral is the same as the one given in Equation (2.8).

MCMC draws samples $\theta^{(i)}$, $i \in \{1, \dots, N\}$ from $p(\theta)$ (i.e. the target distribution) and uses them to evaluate:

$$\tilde{I} = \frac{1}{N} \sum_{i=1}^N f(\theta^{(i)}) \approx I \quad (5.2)$$

which is an unbiased estimator of the integral (5.1). (5.2) is called the output estimate (also given in Equation (2.9)). The method presented in this chapter performs precision optimization for a specific output estimate, i.e. specific $p(\theta)$ and $f(\theta)$, given by the user.

For finite N , (5.2) differs from (5.1) by some Normal-distributed random number with mean zero and variance $\sigma_{\tilde{I}}^2$, called the variance of the output estimate:

$$\sigma_{\tilde{I}}^2 = \frac{\sigma_f^2}{N} \quad (5.3)$$

where σ_f^2 is the variance of the function of interest $f(\theta)$ under $p(\theta)$. The variance σ_f^2 is constant for fixed $f(\theta)$ and $p(\theta)$ but it is unknown. The variance $\sigma_{\tilde{I}}^2$ can be reduced arbitrarily by increasing N but this requires more computational time. In this chapter, $\sigma_{\tilde{I}}^2$, σ_f^2 and all other variances that follow are approximated by running the MCMC sampler multiple times (typically 30-100) with different random seeds. In the rest of the chapter a run refers to such a set of parallel runs.

As already explained in Section 2.6.3, the above equations are theoretical; they assume that all compu-

tations inside MCMC are performed in infinite precision and therefore MCMC samples are distributed according to the true distribution and the output estimate (5.2) converges to the true value (5.1) for $N \rightarrow \infty$. Nevertheless, in practical MCMC implementations on digital devices, infinite precision is impossible to achieve. As a result, almost all work in MCMC literature uses double or single precision floating point, which are considered adequate for existing problems. In this chapter, double precision is considered equivalent to infinite precision in all implementations. The reason for this choice is the following: This chapter deals with precision optimization and this optimization is done using a certain precision as reference, i.e. the reference precision is considered equivalent to infinite precision. Therefore, the highest of the two default precisions in literature (double precision) is used here to resemble infinite precision as closely as possible.

5.2.2 Parts of MCMC - precision domains

Each MCMC algorithm consist of two parts:

1. The evaluation of the probability of each proposed sample according to the probability density (target density) $p(\theta)$. This part is specific to the targeted problem.
2. The generic operations, which mainly include proposing samples and accepting/rejecting them according to their probability density value. This part is generic for a given MCMC algorithm, i.e. it is the same regardless of the targeted problem (although it changes for each different MCMC algorithm).

The probability evaluations (first part) take up the bulk of the computation time, especially when complex models and large-scale data are employed. The generic MCMC operations are much less computationally demanding. Therefore, the crucial task to achieve higher sampling throughput in MCMC is the minimization of the cost of implementing $p(\theta)$.

As already explained in Section 2.6.3, if the two parts of MCMC are treated as separate precision domains, it is possible to acquire four different precision combinations, shown in Table 2.1. In this chapter, Combination D/C is used. This combination performs all generic operations in double precision and uses custom precision only for density evaluations. It thus guarantees convergence to *some* probability distribution. Due to the use of custom precision for probability density evaluations, this stationary distribution is not the “true” distribution ($p(\theta)$) but an approximation of it (denoted $p_c(\theta)$),

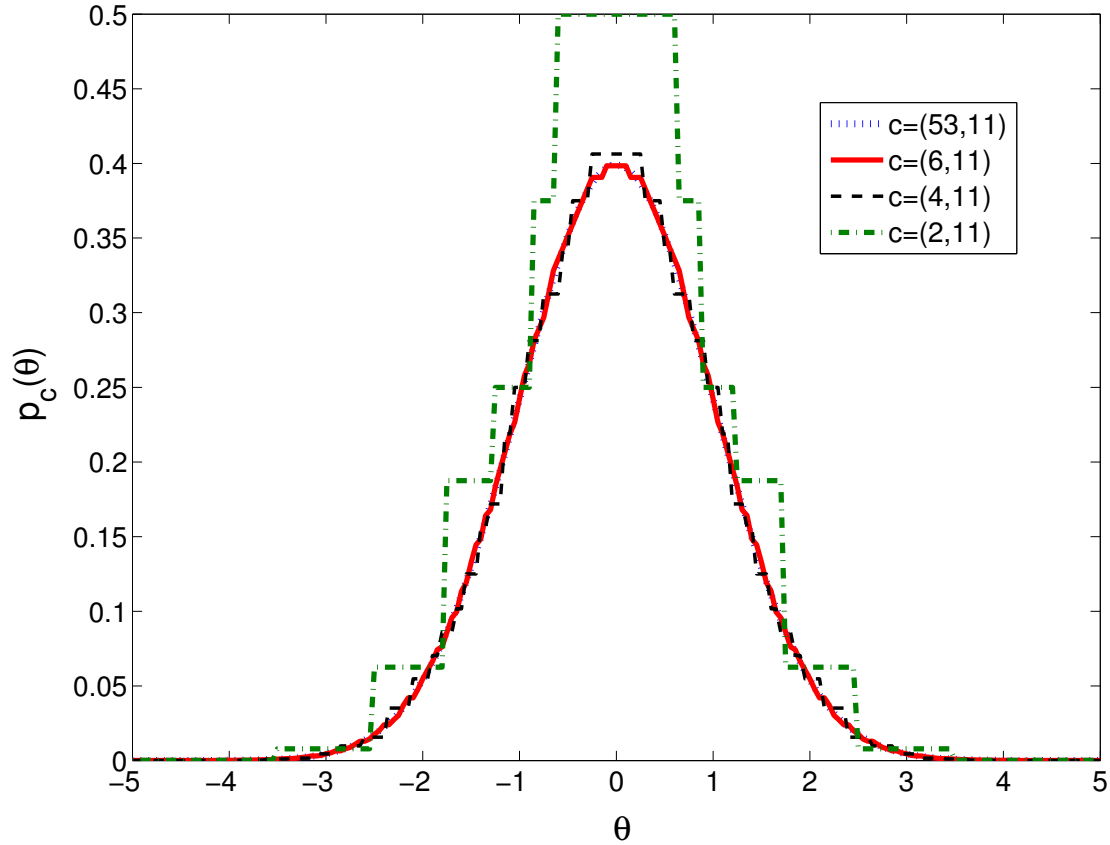


Figure 5.1: The same Normal target density (Mean = 0, Variance = 1) for different precision configurations.

where $c = (\text{mantissa bits}, \text{exponent bits})$ is the precision configuration used). See Figure 5.1 (recalled from Chapter 2) for an example of what a custom precision density looks like for different c .

It is worth noting that, in Chapter 3, the two proposed custom precision algorithms for PT used a mix of Combinations D/D and D/C. WPT performed all probability computations in custom precision and generic operations in double precision (like Combination D/C) but used double precision to correct the output estimate. Therefore, the use of custom precision had no effect on the output, which converged to the “true” value (like Combination D/D). MPPT used custom precision for probability evaluations in auxiliary chains (like Combination D/C) and double precision for probability evaluations in the first chain (like Combination D/D). All generic operations were performed in double precision. The effect of this mixed strategy was the same as in WPT; no bias in the output. The approach presented in this chapter differs in that it permits bias in the output estimate but controls it.

FPGA-mapped MCMC samplers contain one hardware block for the generic MCMC operations and one hardware block which implements $p(\theta)$ (which can consist of multiple parallel sub-blocks). For

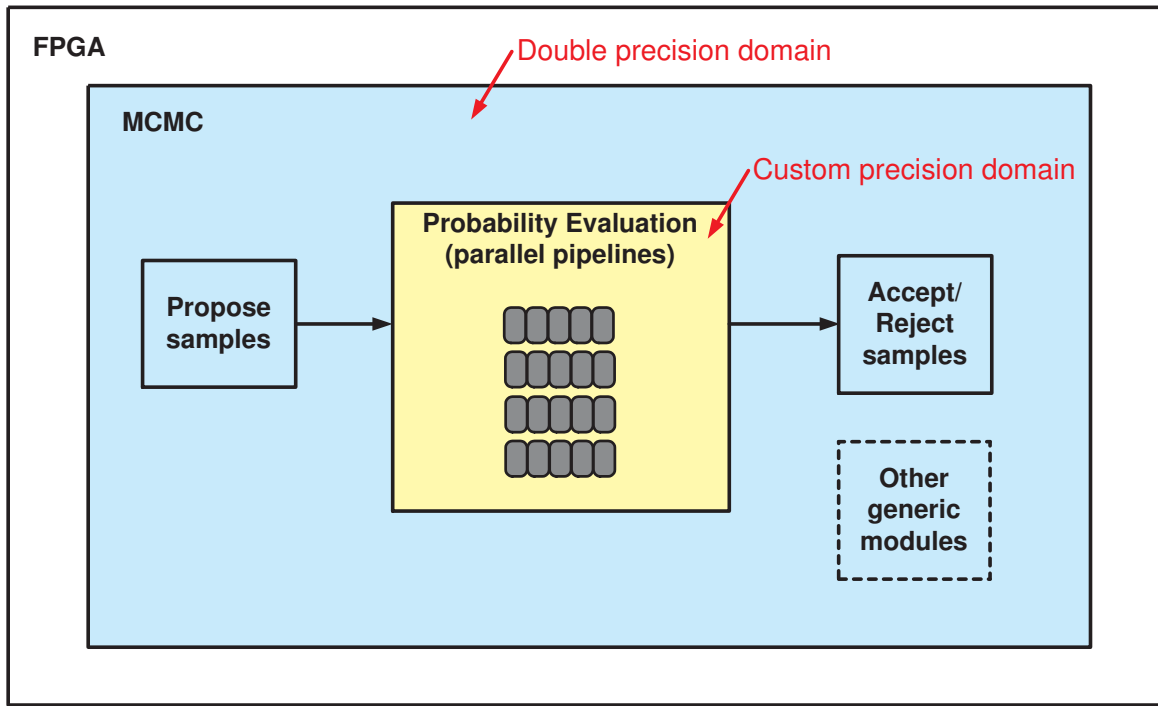


Figure 5.2: Double and custom precision domains in an FPGA implementation of MCMC (Combination D/C in Table 2.1). The double precision domain corresponds to the generic MCMC operations. The custom precision domain corresponds to the Probability Evaluation block.

example, in Chapter 3, the Parallel Tempering FPGA architecture contains a Probability Evaluation block responsible for computing the densities (possibly using multiple parallel pipelines) and several other blocks which perform the generic operations. Because $p(\theta)$ is the bottleneck computation in any MCMC algorithm with large-scale data, sampling throughput increases when more parallel blocks are instantiated to compute $p(\theta)$. Following Combination D/C, this chapter focuses on how to minimize the cost of implementing these blocks by automatically optimizing the employed precision in the block that evaluates $p(\theta)$. Figure 5.2 shows a high-level description of an FPGA implementation which follows Combination D/C.

5.2.3 MCMC estimate when using custom precision probability densities

When custom floating point precision is used to compute $p(\theta)$, MCMC samples are distributed according to $p_c(\theta)$, where $c = (\text{mantissa bits}, \text{exponent bits})$ is the precision configuration used. All custom precisions in this chapter are lower than double precision and use 11 exponent bits.

Recalling Chapter 2, the approximated integral is no longer given by Equation (2.8). It is now the

following:

$$I_c = E_{p_c}[f(x)] = \int f(\theta) p_c(\theta) d\theta \quad (5.4)$$

The value of the custom precision output estimate is:

$$\tilde{I}_c = \tilde{E}_{p_c}[f(x)] = \frac{1}{N} \sum_{i=1}^N f(\theta^{(i)}) \approx I_c \quad (5.5)$$

where $\theta^{(i)}, i \in \{1, \dots, N\}$ are samples drawn from $p_c(\theta)$. The variance of this estimator $\sigma_{\tilde{I}_c}^2$ is found in the same way as the variance of the double-precision estimator (see Equations (5.3)), only now σ_f^2 is different since samples are drawn from p_c .

5.3 Proposed bias estimator

This section proposes an efficient estimator of the bias in the MCMC output estimate when the probability distribution is evaluated in custom precision. Section 5.4 uses this bias estimator to develop an automated precision optimization method.

5.3.1 Bias estimator

Combining Equations (5.1) and (5.4), it is easy to find the following expression, which gives the bias in the output estimate due to the use of custom precision (same as equation (2.26) in Chapter 2):

$$b_c = I - I_c = \int f(\theta) p(\theta) d\theta - \int f(\theta) p_c(\theta) d\theta \quad (5.6)$$

This quantity needs to be approximated for all candidate precision configurations (all values of c) in order to choose which configuration to use (i.e. in order to optimize precision). In this chapter, it is assumed that b_c is a scalar, although the proposed optimization methodology can be easily extended to cases where b_c is a vector.

Figure 5.3 shows a simple example of a double- and a custom-precision MCMC estimator. The estimated quantity is the mean of a Gaussian distribution. The true value is 4. The bias is visible near the end of the runs. Standard deviations are also visible. The bias is independent of the estimator's standard deviation and cannot be avoided when sampling with reduced precision.

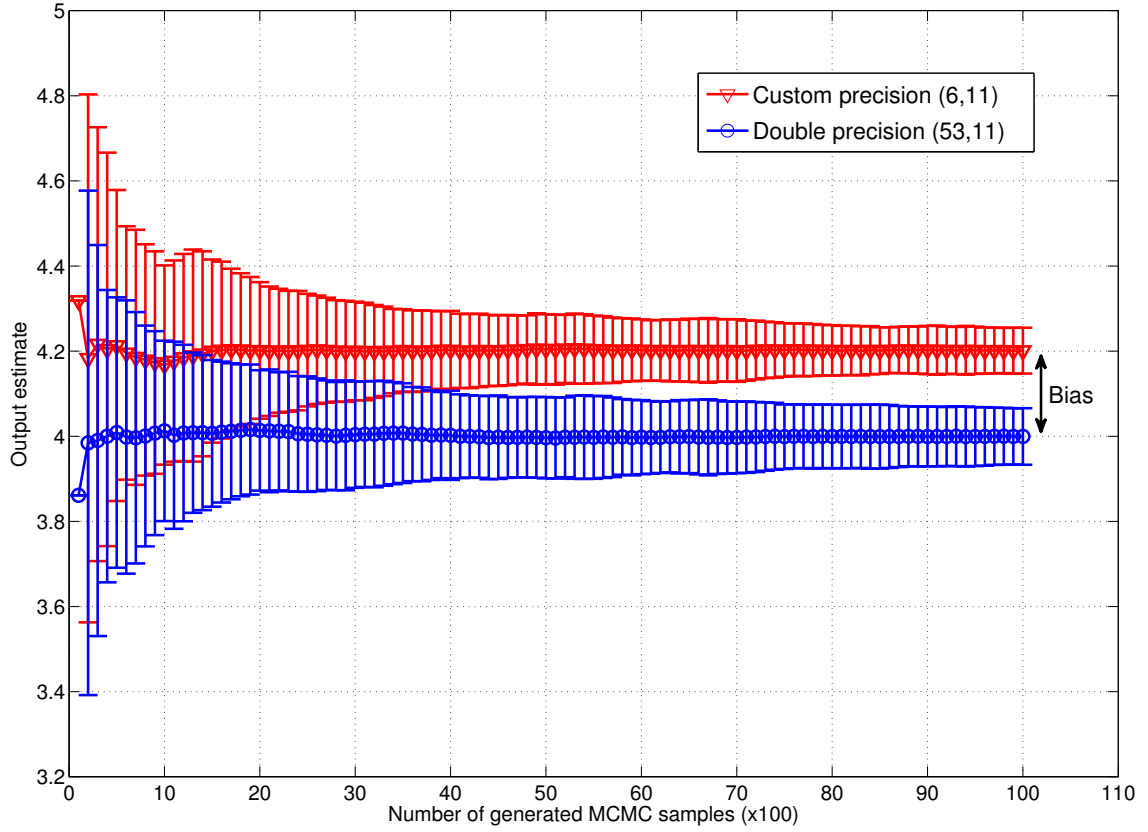


Figure 5.3: Double- and custom-precision ($c = (6, 11)$) sampling for the same output estimate $\tilde{I}$. The error bars are equal to two times the standard deviation of the estimate ($\pm 2 * \sigma_{\tilde{I}}$ for double and $\pm 2 * \sigma_{\tilde{I}_c}$ for custom precision). The true value of the estimated integral is $I = 4$

Straightforward estimator: Estimating b_c based on (5.6) would require two separate MCMC runs (targeting first $p(\theta)$ and then $p_c(\theta)$) in order to estimate the two integrals and then subtract them. The variance σ_{str}^2 of this straightforward bias estimator would be equal to the sum of the variances of the two output estimates: $\sigma_{str}^2 = \sigma_I^2 + \sigma_{I_c}^2$. Thus, this bias estimator has a larger variance than any of the two output estimators for an equal number of samples. Figure 5.5 shows (among other things) the ratio $\frac{\sigma_{str}^2}{\sigma_{I_c}^2}$ for all precisions for one of the examples of Section 5.5. σ_{str}^2 is around two times larger than $\sigma_{I_c}^2$. Optimizing the precision with this straightforward method would require a long MCMC run for each candidate precision configuration to estimate its bias with sufficient accuracy, making the total runtime larger than the time for a double-precision run (the default choice).

Proposed estimator: Here, a more efficient bias estimator is proposed. It achieves significantly smaller variance compared to the straightforward approach, as will be demonstrated in Section 5.3.2. It thus enables precision optimization by using only a set of short MCMC pre-runs (one for each precision configuration).

The bias can be rewritten as:

$$\begin{aligned}
 b_c &= \int f(\theta)p(\theta)d\theta - \int f(\theta)p_c(\theta)d\theta = \int f(\theta)(p(\theta) - p_c(\theta))d\theta = \int f(\theta)\frac{p(\theta)-p_c(\theta)}{p_c(\theta)}p_c(\theta)d\theta \\
 &= \int f(\theta)\left(\frac{p(\theta)}{p_c(\theta)} - 1\right)p_c(\theta)d\theta = \int f(\theta)(w_c(\theta) - 1)p_c(\theta)d\theta \\
 &= \int f_{b_c}(\theta)p_c(\theta)d\theta = E_{p_c}[f_{b_c}(\theta)]
 \end{aligned} \tag{5.7}$$

where $w_c(\theta) = \frac{p(\theta)}{p_c(\theta)}$ and $f_{b_c}(\theta) = f(\theta)(w_c(\theta) - 1)$. The weight $w_c(\theta^{(i)})$ is the ratio of the probability density of sample $\theta^{(i)}$ computed using double precision over the probability density of sample $\theta^{(i)}$ computed using custom precision. The proposed bias estimator approximates the integral in the last line of (5.7) by taking MCMC samples from $p_c(\theta)$ and computing the following estimate (similarly to (5.5)):

$$\tilde{b}_c = \frac{1}{N} \sum_{i=1}^N f_{b_c}(\theta^{(i)}) \approx b_c \tag{5.8}$$

Nevertheless, to get this estimate, it is also necessary to compute the probability density of each sample in double precision ($p(\theta^{(i)})$) to get the weights $w_c(\theta)$. This requires extra computations (i.e. a double precision density evaluation module in the FPGA) but it will be shown in the following sections that this overhead is small compared to the runtime savings from using custom precision.

Moreover, another issue needs to be addressed before using the proposed estimator: In Bayesian inference, the distributions $p(\theta)$ and $p_c(\theta)$ are typically known only up to a normalizing constant:

$$p(\theta) = \frac{p^u(\theta)}{C_p} \tag{5.9}$$

$$p_c(\theta) = \frac{p_c^u(\theta)}{C_{p_c}} \tag{5.10}$$

where C_p and C_{p_c} are the normalizing constants and only the unnormalized densities $p^u(\theta)$ and $p_c^u(\theta)$ can be evaluated. This means that the weights

$$w_c(\theta) = \frac{p^u(\theta)}{p_c^u(\theta)} \frac{1}{(\frac{C_p}{C_{p_c}})} = w_c^u(\theta) \frac{1}{(\frac{C_p}{C_{p_c}})} \tag{5.11}$$

(where $w_c^u(\theta) = \frac{p^u(\theta)}{p_c^u(\theta)}$ is the unnormalized weight) cannot be evaluated exactly because the ratio of constants $\frac{C_p}{C_{p_c}}$ is unknown.

In order to address this problem, an estimator of the ratio of constants is devised here. The ratio can be estimated by the mean of the unnormalized weights $w_c^u(\theta)$ based on (5.12). Since $p(\theta)$ is a probability density, its integral over all θ is equal to one, which leads to the following sequence of

equations (using Equations (5.9)-(5.11)):

$$\begin{aligned}
\int p(\theta) d\theta &= 1 \\
\Leftrightarrow \int \frac{p(\theta)}{p_c(\theta)} p_c(\theta) d\theta &= 1 \\
\Leftrightarrow \int \frac{\frac{C_p}{p_c^u(\theta)}}{\frac{C_p}{p_c^u(\theta)}} p_c(\theta) d\theta &= 1 \\
\Leftrightarrow \int \frac{p^u(\theta)}{p_c^u(\theta)} p_c(\theta) d\theta &= \frac{C_p}{C_{p_c}} \\
\Leftrightarrow E_{p_c}[w_c^u(\theta)] &= \frac{C_p}{C_{p_c}}
\end{aligned} \tag{5.12}$$

Therefore, samples from an MCMC run in precision c can be used not only to estimate the bias b_c (as described above) but also to estimate the ratio $\frac{C_p}{C_{p_c}}$. At the end of the MCMC run, the mean of the unnormalized weights is computed and then used to normalize the weights by dividing each weight with it. The result is the set of normalized weights $w_c(\theta_{1:N})$. These can then be used in the estimator (5.8). The variance of (5.8) is found in the same way as the previous variances, substituting f_{b_c} for f :

$$\sigma_{b_c}^2 = \frac{\sigma_{f_{b_c}}^2}{N} \tag{5.13}$$

It is worth noting that the above weight normalization technique was not applied in the bias estimator proposed by Chow et al. [3], since that work focused on non-MCMC Monte Carlo simulations, where the sampled distributions are fully known (in contrast to MCMC where distributions are known only up to a normalizing constant). Moreover, in Chow et al. [3] the computational bottleneck was the computation of the function $f(\theta)$ and not the generation of samples from $p(\theta)$. Therefore, custom precision was applied only to the computation of $f(\theta)$ after the samples had been generated.

The variables and constants mentioned in the above sections are summarized in Table 5.1 for easier reference.

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

The above section described the proposed estimator but did not explain why it is preferable over the straightforward estimator and how their efficiencies compare. This section investigates these questions.

As shown in Figure 5.1, custom precision densities $p_c(\theta)$ are generally close to the “true” double

Table 5.1: Variables and constants in this chapter.

Symbol	Description
c	Precision configuration $c = (\text{mantissa bits}, \text{exponent bits})$
$p(\theta), p_c(\theta)$	Target probability distributions in double precision and custom precision c respectively
I, I_c	True values of integrals in double precision and custom precision c respectively
$\tilde{I}, \tilde{I}_c$	Output estimates for double precision and custom precision c respectively
$f(\theta)$	Function of interest in (5.1)
$f_{b_c}(\theta)$	Bias function for precision c in (5.7)
$\tilde{b}_c$	True value of bias for custom precision c
$\hat{b}_c$	Bias estimate for custom precision c
$\sigma_{\tilde{I}}^2, \sigma_{\tilde{I}_c}^2$	Variance of $\tilde{I}$ and $\tilde{I}_c$
$\sigma_{\tilde{b}_c}^2$	Variance of $\tilde{b}_c$
$p^u(\theta), p_c^u(\theta)$	Unnormalized target probability distributions in double precision and custom precision c
$\frac{C_p}{C_{p_c}}$	Ratio of normalizing constants of $p(\theta)$ and $p_c(\theta)$
$w_c(\theta), w_c^u(\theta)$	Weights and unnormalized weights for custom precision c

precision densities $p(\theta)$ unless the precision configuration c has few mantissa bits (e.g. 4 mantissa bits). Because of this behaviour, the weights of the bias estimator $w_c(\theta) = \frac{p(\theta)}{p_c(\theta)}$ take values close to one when c is not very low. Therefore, the function $f_{b_c}(\theta) = f(\theta)(w_c(\theta) - 1)$ takes values close to zero and much smaller than $f(\theta)$ when $\theta \sim p_c(\theta)$. Due to the small values it takes, the function $f_{b_c}(\theta)$ also has smaller variance than $f(\theta)$ ($\sigma_{f_{b_c}}^2 < \sigma_f^2$). Figure 5.4 illustrates an example of this behaviour by comparing the histograms of $f(\theta) = \theta$ (mean estimate) and $f_{b_c}(\theta) = \theta(w_c(\theta) - 1)$ under the same distribution $p_c(\theta)$ ($c = (13, 11)$), taken from one of the case studies of Section 5.5.

The small variance of function $f_{b_c}(\theta)$ translates into small variance $\sigma_{\tilde{b}_c}^2$ for the bias estimator (5.5) (due to Equation 5.3). Figure 5.5 shows the ratio $\frac{\sigma_{\tilde{b}_c}^2}{\sigma_{\tilde{I}_c}^2}$ for different precisions c , when $N = 10^6$ MCMC samples are used for both estimators. $\sigma_{\tilde{b}_c}^2$ is orders of magnitude smaller than $\sigma_{\tilde{I}_c}^2$ for any configuration with more than 8 to 9 mantissa bits. Therefore, for these precisions, the bias estimator (5.8) needs much fewer samples than the custom-precision output estimator (5.5) to achieve the same variance.

This property is crucial for the proposed method. It means that, using the proposed bias estimator, it is possible to perform short MCMC pre-runs for a set of candidate precisions to estimate the biases for a given output estimate, without posing a large overhead to the final run (which is used to get the output estimate (5.4)). Figure 5.5 also depicts the ratio $\frac{\sigma_{\tilde{b}_c}^2}{\sigma_{\tilde{I}_c}^2}$ (which involves the straightforward estimator). This ratio is always above one, showing that performing pre-runs in all candidate precisions is impractical using this approach (as mentioned previously).

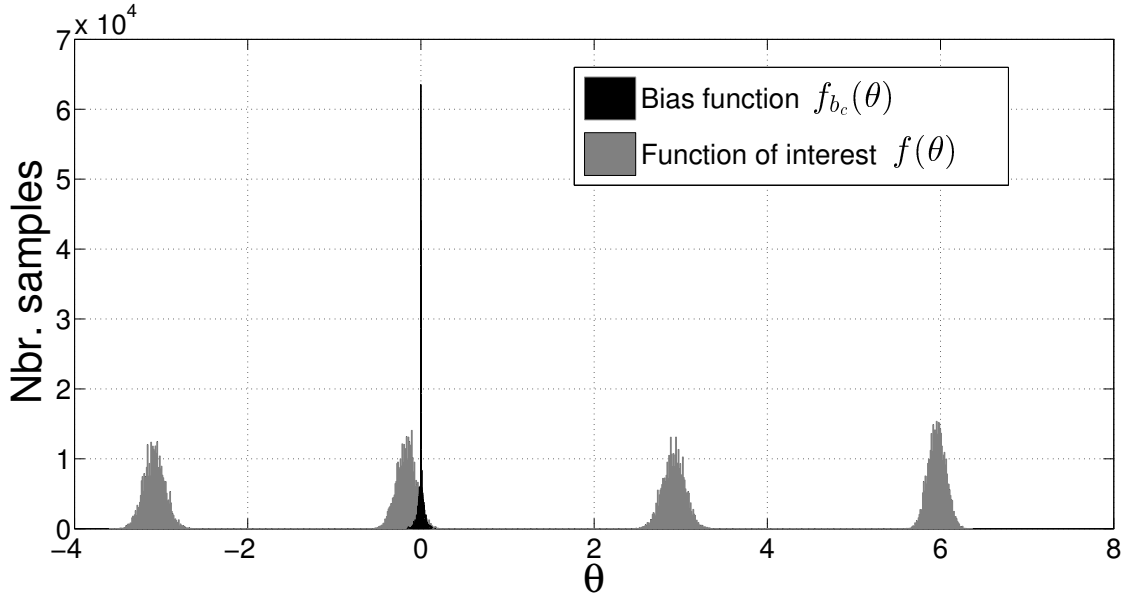


Figure 5.4: Histograms of $f(\theta)$ and $f_{b_c}(\theta)$ for the mean estimator under $p_c(\theta)$ with $c = (13, 11)$, where $p_c(\theta)$ is a Gaussian mixture distribution (Section 5.5). The variance of $f(\theta)$ is significantly larger.

5.4 Optimization method

The goal of the optimization method is to choose the most resource-economical (with the fewest bits) precision configuration c or a near-optimal configuration out of a user-defined set of candidate configurations, while probabilistically guaranteeing that the custom-precision output estimator (5.5) introduces bias b_c within a user-specified range. To do this, short FPGA pre-runs are performed for all candidate precisions and the bias is estimated using the estimator of the previous section.

The steps of the optimization process are shown in Figure 5.6. The process requires some input from the user and then follows four steps: The double- and custom- precision pre-runs (Steps 1 and 2), the precision selection (Step 3) and the final run (Step 4). The method exploits the reconfiguration property of FPGAs to optimize precision. Two different configuration bitstreams are used in the process. Steps 1 and 2 use the same mixed-precision bitstream, which is able to draw MCMC samples in any of the candidate precisions and in double precision. Step 4 uses a different, optimized-precision bitstream, which draws samples only in the optimized precision (which was selected by the previous steps). Step 3 runs in software on the host PC. Both bitstreams are taken from a library of pre-compiled bitstreams (Section 5.4.5). The flow of the method is described in detail in Sections 5.4.1-5.4.4.

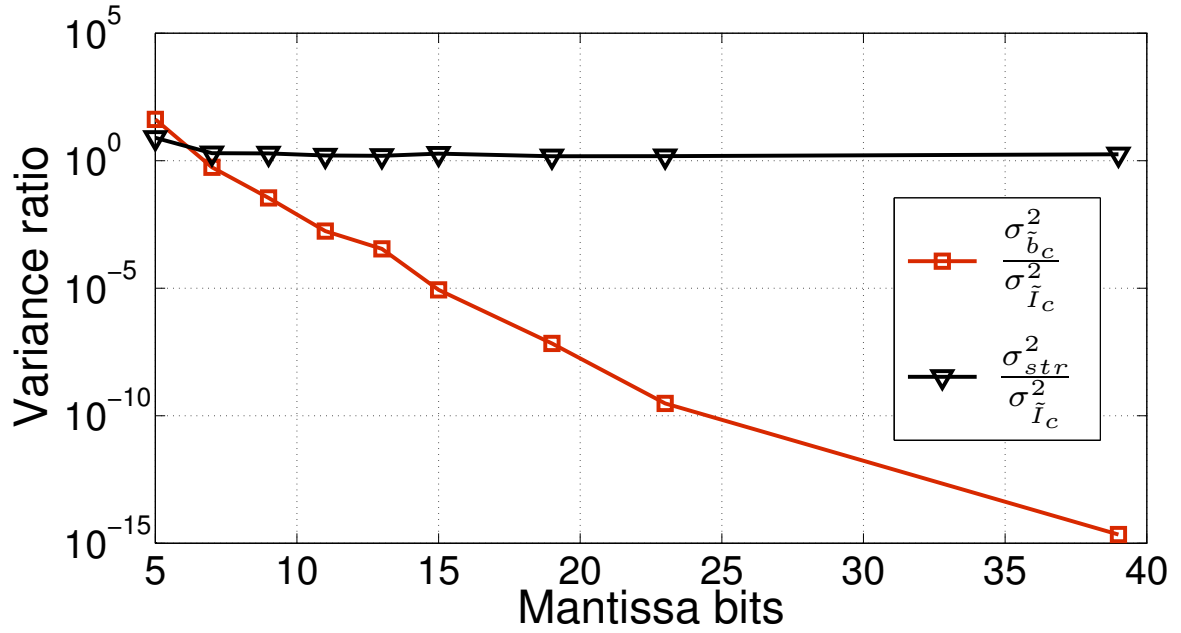


Figure 5.5: Ratios $\frac{\sigma_{\tilde{b}_c}^2}{\sigma_{\tilde{I}_c}^2}$ (variance of proposed bias estimator over variance of custom precision output estimator) and $\frac{\sigma_{str}^2}{\sigma_{\tilde{I}_c}^2}$ (variance of “straightforward” bias estimator over variance of custom precision output estimator) for various mantissa bit configurations (exponents bits=11). $p_c(\theta)$ is a Gaussian mixture distribution (Section 5.5) and $f(\theta)$ is the mean estimator. $N = 10^6$ samples are used for all estimators. The proposed estimator’s variance ($\sigma_{\tilde{b}_c}^2$) is orders of magnitude smaller than that of the output estimator ($\sigma_{\tilde{I}_c}^2$) for most precisions. The straightforward estimator’s variance (σ_{str}^2) is always larger than $\sigma_{\tilde{I}_c}^2$.

5.4.1 User input

Table 5.2 summarizes the input parameters. The user defines a target SD_T for the standard deviation $\sigma_{\tilde{I}_c}$ of the output estimate (5.5). This means that the final run will continue until this standard deviation is reached. This is a common user input in MCMC literature. The standard deviation is used instead of the variance because it is easier to interpret (same unit as the estimate). All standard deviations are approximated by performing multiple MCMC runs (typically 30-100) with different random seeds (a run refers to such a set of multiple runs). Apart from SD_T , the user also gives a threshold T_{bias} for the bias which can be tolerated. T_{bias} is given as a percentage of SD_T (e.g. setting $T_{bias} = 0.5$ translates to a bias tolerance equal to 50% of SD_T). Together, SD_T and T_{bias} determine the bias threshold $SD_T \cdot T_{bias}$, which is the maximum absolute bias that can be tolerated. By changing these parameters, the user trades off accuracy in the output estimate for lower precision (and thus higher performance through increased parallelization). The bias threshold can alternatively be defined as an absolute number without affecting the presented optimization process but this is not pursued here. The

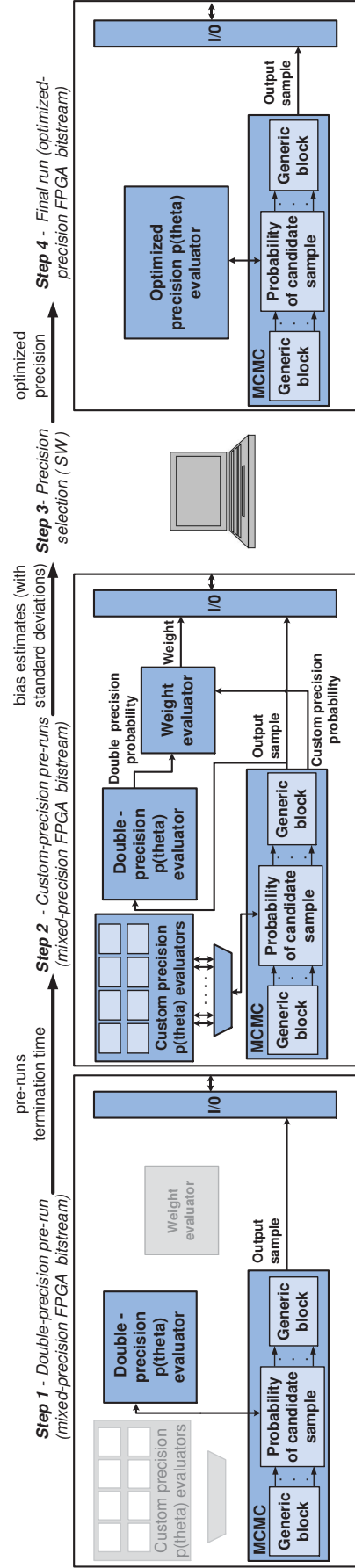


Figure 5.6: Optimization method flow: Steps 1,2 (mixed-precision FPGA bitstream), 3 (host PC software) and 4 (optimized-precision FPGA bitstream) are shown. During Step 1, the double-precision module is used to draw MCMC samples. During Step 2, it evaluates the probabilities of all generated samples, which are used for weight evaluations. Grey-coloured blocks (Step 1) are unused. The top part of the figure shows the information passed by each step to the next.

Table 5.2: User parameters in the optimization method

SD_T	Target standard deviation in output estimate.
T_{bias}	Bias threshold as a percentage of SD_T .
S	Set of candidate precision configurations (each precision is specified as a pair of integers: $c = (mantissa\ bits, exponent\ bits)$).
$Term\%$	Time available for Steps 1 and 2 as a percentage of t_{est} .
Pr_{min}	Minimum probability to accept a precision. If the probability of the event “the absolute bias is smaller than $SD_T \cdot T_{bias}$ ” is higher than Pr_{min} , the precision is acceptable.

user also defines the set of candidate precisions (S) and the parameters $Term\%$ and Pr_{min} , which will be explained shortly.

5.4.2 Steps 1 and 2: Pre-runs (mixed-precision bitstream)

The goal of the pre-runs is to estimate the bias for every precision. A mixed-precision bitstream is loaded on the FPGA (Steps 1 and 2 in Figure 5.6). It contains probability evaluation blocks in all candidate precisions (S) and in double precision, a block which implements the generic operations of the MCMC algorithm in double precision (see Section 5.2.2 for more details) and a weight evaluation module in double precision. Sampling with precision c is done by plugging the respective probability evaluation block to the generic block (which can easily be done at runtime).

Step 1 consists of a single, short, double-precision MCMC run, which estimates how much time a double-precision FPGA sampler would need to give an output estimate with standard deviation equal to SD_T . This estimated time is then used to find how long the pre-runs can last without introducing a large overhead to the final run, i.e. specify a reasonable time to be given for pre-runs.

During Step 1, the double precision probability evaluation block is plugged in the generic MCMC modules. The system generates MCMC samples and sends them to the host PC in DMA batches. The host PC uses a generalized version of the Gelman-Rubin diagnostic [35] (popular in MCMC literature) in order to detect if the sampler has converged after each new batch (convergence is necessary for the samples to be usable). When convergence is achieved, sampling stops. At that point, the standard deviation in the output estimate (SD_{pre}) is evaluated based on the multiple independent MCMC runs that comprise the pre-run. Also, the number of generated samples of each independent run (N_{pre}) is noted. These two numbers are then used to estimate the number of samples that a double precision

FPGA sampler would need to reach a standard deviation of SD_T (the target standard deviation):

$$N_T = N_{pre} \frac{SD_{pre}^2}{SD_T^2} \quad (5.14)$$

The equation holds because the standard deviation of MCMC-based estimates drops with $\frac{1}{\sqrt{N}}$ (see equation (5.3)). Using the above equation, it can be inferred that a double-precision FPGA run would need t_{est} seconds to reach SD_T :

$$t_{est} = \frac{N_T}{TH_{dp}} \quad (5.15)$$

Here, TH_{dp} is the throughput (in samples/sec) of a double-precision FPGA sampler which utilizes the whole device (i.e. uses as many parallel pipelines in the probability evaluation block as can fit in the device). This quantity is known, since it is assumed that bitstreams which utilize the whole FPGA are available in all precisions and their sampling throughputs are known. For more information on the assumption made about the available bitstreams, the probability evaluation blocks and their parallelism see Section 5.4.5.

After calculating t_{est} , the system decides how much time should be devoted to Steps 1 and 2 in order to limit the overhead to the total runtime. This decision is taken based on the user parameter $Term\%$ which expresses what percentage of t_{est} should be given to Steps 1 and 2. Reasonable values are 5% or 10% ($Term\% = 0.05$ or $Term\% = 0.1$ respectively) but the user is free to set this parameter. Although (5.15) might be a rough estimate because of the short double-precision pre-run, high accuracy is not needed to ensure a small overhead.

After the available time for pre-runs is known, Step 2 starts. For each candidate precision configuration, it estimates the bias $\tilde{b}_c$ introduced in the output estimate, as well as the standard deviation of the bias $\sigma_{\tilde{b}_c}$. The two quantities are given by (5.8) and (5.13) respectively. In order to generate the necessary custom precision MCMC samples inside the FPGA (i.e. sample from $p_c(\theta)$ for all $c \in S$), the custom-precision probability evaluation blocks (unused in Step 1) are sequentially plugged to the generic block of the system. An equal number of samples is taken for each c . The double-precision block is now used for a different purpose compared to Step 1. It computes all sample probabilities in double precision and sends them to the weight evaluator module to get the weights $w_c(\theta)$. The samples and weights are sent to the host in batches until $Term\% \cdot t_{est}$ seconds have passed (since the start of Step 1) and $\tilde{b}_c$ and $\sigma_{\tilde{b}_c}$ are computed in the host PC for all precisions.

5.4.3 Step 3: Selection (software)

Step 3 chooses the optimized precision based on the user's accuracy requirements. It runs in the host PC (Figure 5.6). Each bias estimate is a Normal random variable (see Section 5.2.1) with mean $\tilde{b}_c$ and standard deviation $\sigma_{\tilde{b}_c}$. The criterion to accept a configuration c is the probability that the absolute value of its bias b_c is smaller than the user-defined tolerance:

$$p(-SD_T \cdot T_{bias} < b_c < SD_T \cdot T_{bias}) = N_{CDF}(SD_T \cdot T_{bias}, \tilde{b}_c, \sigma_{\tilde{b}_c}) - N_{CDF}(-SD_T \cdot T_{bias}, \tilde{b}_c, \sigma_{\tilde{b}_c}) \quad (5.16)$$

where $N_{CDF}(x, \mu, \sigma)$ is the value in x of a Normal cumulative density function with mean μ and standard deviation σ . If the probability in (5.16) is larger than a user-defined value Pr_{min} (e.g. $Pr_{min} = 0.95$), the bias is very likely to be within tolerance and thus the configuration is accepted. Otherwise, the bias is considered too large. The definition of “very likely” rest in the user. The lowest precision to pass this test is chosen as the optimized one. If no precision passes the test, double-precision is chosen. The *optimized* precision might not always be the *optimal* one, as will be demonstrated in Section 5.7. Moreover, it has to be noted that the optimization method guarantees the bound on the bias probabilistically, i.e. with some probability smaller than one, and not deterministically.

5.4.4 Step 4: Final runs (optimized-precision bitstream)

After Step 3, a second bitstream is selected from the pre-compiled library (which contains bitstream in all candidate precisions) and loaded on the FPGA. It contains probability evaluation blocks only in the optimized precision and a generic MCMC block (Figure 5.6 (right)). The throughput of this sampler is higher than the throughput of Step 2 samplers, since the FPGA resources that were previously used to instantiate blocks in the various candidate precisions are now available to be used exclusively for implementing blocks in the selected precision. Also, the auxiliary double precision block found in the first bitstream is not required. The final MCMC run lasts until SD_T is reached. The resulting estimate is biased within tolerance (guaranteed with probability Pr_{min}).

5.4.5 Assumptions

The two main parts in an MCMC implementation on an FPGA are the probability evaluation block and the generic block (which includes all the remaining operations). Figure 5.2 illustrates a high level

description of the two domains. Here, it is assumed that hardware blocks which implement probability density functions $p_c(\theta)$ are available for all precisions c . Also a hardware block which implements the generic parts of MCMC in double precision is assumed to be available. These can be similar to the blocks presented in Chapter 3 or in other FPGA implementation in the literature [25].

The speedup offered by the optimization method is based on the assumption that the first bitstream (which includes probability evaluation blocks in all candidate precisions) is used to select a precision and then the second bitstream is able to use this precision to accelerate sampling (compared to the sampling speed achieved by a double precision implementation). In order for this acceleration to be significant, it is necessary that the probability density can be evaluated in parallel. In other words, it is necessary that the probability evaluation block is constructed in a way that allows the instantiation of more parallel blocks to increase throughput (for example, see the way the i.i.d. probability density is implemented in Chapter 3, although other forms of parallelism are also suitable). If this is the case, it is possible to increase the amount of parallelism in the FPGA by reducing precision, since reduced precision leads to lower resource consumption by a single block. If the probability density is not parallelizable, reducing precision offers savings in terms of FPGA area but this cannot be exploited to increase parallelism. The only gains in terms of sampling speed come from the drop in the latency of the probability evaluation block, which is usually limited. Therefore, the optimization method can be applied regardless of the form of the probability density but significant gains in sampling speed are expected only when the probability density is parallelizable.

As mentioned above, the method assumes that a library of pre-compiled bitstreams has been generated off-line. This is a realistic assumption, since the bitstreams are re-usable (i.e. the same $p(\theta)$ is often sampled in different problems and settings). The library includes versions of the first bitstream (used in Steps 1 and 2) for various sets (S) of candidate precisions, as well as versions of the second bitstream (used in Step 4) in all precisions. Full utilization of the targeted FPGA's resources is assumed for Step 4 bitstreams. This means that these bitstreams use the maximum amount of parallelism allowed by the device in order to maximize throughput. Bitstreams used in Steps 1 and 2 can also use parallelism in the probability evaluation blocks if enough FPGA resources are available, although this approach is not explored here (each custom precision probability evaluation block comprises a single datapath/pipeline). Because it is practically difficult to create versions of the Step 1/2 bitstream for every possible combination of candidate precisions S , an alternative is to include only some common combinations (or even one combination) in the library.

Since Step 4 bitstreams in all candidate precision are available, the throughput of the MCMC sampler with full FPGA utilization is known for all precisions (including double precision). This information is used in Step 1 of the method to decide how much time will be allocated for pre-runs, as described in Section 5.4.2.

Finally, it is assumed that the MCMC method is properly tuned in order to avoid large convergence times. Correct tuning is outside of the scope of this work. In order to detect convergence of the double-precision sampler during Step 1, a generalized version of the Gelman-Rubin diagnostic [35] is used (as mentioned earlier). This diagnostic compares the multiple independent MCMC runs and produces a metric (called the potential scale reduction factor and denoted $\hat{R}$ in [35]) which suggests convergence if its value is close to one (e.g. below 1.1). Although no guarantee of convergence can be provided by such metrics, they are a standard approach in literature and this is why they are used here.

5.5 Case studies: Models and MCMC method

Two Bayesian inference models are used for evaluation. Both case studies involve the handling of i.i.d. data, which permits the exploitation of parallelism in the probability evaluation block, as mentioned in Section 5.4.5. Parallelism is exploited in the same way as in Chapter 3, i.e. by instantiating multiple parallel pipelines inside the block. Other forms of parallelism in the probability density can be exploited, although this is not investigated here.

5.5.1 Mixture model

Mixture models (MMs) are a family of models heavily used in machine learning [9]. Here, a Gaussian MM defined in [16] is used. The same mixture model case study was used in Chapter 3 but it is summarized here for easier reference.

A set of i.i.d. data $d_{1:n}$, where $d_l \in \Re$ for $l \in \{1, \dots, n\}$, is given (see [16]). In this chapter, the number of data is fixed to $n = 100$. Each observation is distributed according to:

$$p(d_l | \mu_{1:4}, \sigma_{1:4}, a_{1:4}) = \sum_{i=1}^4 a_i N_{PDF}(d_l | \mu_i, \sigma_i) \quad (5.17)$$

Here, the number of mixture model components is fixed to 4, N_{PDF} denotes the density of a univari-

ate Gaussian distribution, $\mu_{1:4}$, $\sigma_{1:4}$ and $a_{1:4}$ are the parameters of the model (means, variances and weights of components respectively). The parameters are fixed to $\sigma_i = \sigma = 0.55$ and $a_i = a = 1/4$ for $i \in \{1, \dots, 4\}$. The parameter $\mu_{1:4}$ is the unknown parameter which is sampled by MCMC, i.e. $\theta = \mu_{1:4}$ (same as in [16]). The prior distribution on $\mu_{1:4}$ is a four-dimensional uniform. In order to perform inference, a data set $d_{1:n}$ is simulated using $\mu_{1:4} = (-3, 0, 3, 6)$. The goal is to infer μ by sampling from its posterior distribution.

Due to the i.i.d. assumption, the likelihood is a product of sub-densities:

$$p(d_{1:100}|\mu_{1:4}) = \prod_{j=1}^{100} p(d_j|\mu_{1:4}, \sigma_{1:4}, a_{1:4}) \quad (5.18)$$

Let $p(\mu_{1:4})$ be the density of a uniform prior. Then the log-posterior density of $\mu_{1:4}$ is given by:

$$\log(p(\mu_{1:4}|d_{1:100})) \propto \sum_{l=1}^{100} \log(p(d_l|\mu_{1:4})) + \log(p(\mu_{1:4})) \quad (5.19)$$

5.5.2 Neural network model

Neural networks (NNs) are universal function approximators [6]. Bayesian inference is one of the methods used to train NNs. Here, an example from [6] is used: Input data $d_{in} = \mathbf{x}_{1:200}$ with $\mathbf{x}_t = (1, -10 + t * .1)$ and output data $d_{out} = y_{1:200}$ with $y_t = g(\mathbf{x}_t) + \varepsilon_t$ are generated, where $\varepsilon_t \sim N(0, \sigma^2)$ for $t \in \{1, \dots, 200\}$ is noise. The function $g(\cdot)$ is unknown and is approximated by a 2-2-1 NN:

$$\tilde{g}(\mathbf{x}_t) = \sum_{j=1}^2 \beta_j \psi(\mathbf{x}_t' \gamma_j) \quad (5.20)$$

where $\beta_j \in \Re$ and $\gamma_j \in \Re^2$ are the connection weights of the network. The activation function $\psi(z)$ is a *tanh* function. The model can be viewed as a non-linear regression model, where the aim is to infer the unknown parameters $\beta_{1:2}$ and $\gamma_{1:2}$ and the unknown variance of the noise σ^2 . In this chapter, the unknowns are packed into a single variable $\theta = \{\beta_{1:2}, \gamma_{1:2}, \sigma^2\}$ and samples from the posterior of this variable are generated using MCMC. The prior distributions are described in [6]. The data are simulated using $\gamma_1 = (2, -1)$, $\gamma_2 = (1, 1.5)$, $\beta_1 = 20$, $\beta_2 = 10$, $\sigma = .1$. The log-posterior distribution is given by:

$$\begin{aligned} \log(p(\beta_{1:2}, \gamma_{1:2}, \sigma^{-2}|d_{in}, d_{out})) \propto \\ -(100 + \nu - 1) \log(\sigma^2) - \frac{1}{2\sigma^2} \left\{ 2\delta + \sum_{t=1}^{200} [y_t - \sum_{j=1}^2 \beta_j \psi(\mathbf{x}_t' \gamma_j)]^2 \right\} - \sum_{j=1}^2 \frac{\beta_j^2}{2\sigma_\beta^2} - \sum_{j=1}^2 \sum_{i=1}^2 \frac{\gamma_{ij}^2}{2\sigma_\gamma^2} \end{aligned} \quad (5.21)$$

where the prior constant parameters are set to $\nu = 0.01$, $\delta = 0.01$, $\sigma_\beta = 20$ and $\sigma_\gamma = 5$ [6].

5.5.3 MCMC algorithm

The optimization method proposed in this chapter can be applied to any MCMC algorithm. Nevertheless, a particular MCMC algorithm is used here to evaluate performance; the Parallel Tempering (PT) MCMC algorithm [6] is employed to sample from the posterior of the two models. The number of chains is set to $m = 32$ for both case studies and the remaining tuning parameters of PT are identical to the ones used in Chapter 3. Both the Bayesian case studies presented above lead to multi-modal distributions. These are the problems that PT is suitable for, although this is not related to the ideas introduced in this chapter.

5.6 Implementation

5.6.1 IP implementation and FPGA system integration

The FPGA systems which process the pre-runs stage and the final run stage (for all precisions) were implemented in VHDL, using floating point operators generated by FloPoCo [97] and the Xilinx Core Generator. The generic parts of the PT sampler are the same as the generic parts of the baseline PT implementation of Chapter 3.

The same system setup as in Chapter 3 was used to test the optimization method: It is based on the RIFFA prototyping framework (version 1.0) [143]. RIFFA wraps the PT IP and uses a PCI-express connection to transfer between the FPGA and the host PC. All the I/O on the hardware side and the software drivers on the host side are handled by the framework. A small piece of C code was written for the host side in order to initialize the FPGA, start the run and receive the outputs. Moreover, a double buffering architecture was designed on top of RIFFA in order to be able to send output data (MCMC samples and weights) to the host PC through DMA, right after they are generated by the IP.

All FPGA samplers were synthesized, placed and routed using Xilinx XPS 13.1. The bitstreams were downloaded to a Xilinx ML605 board, which contains a Xilinx Virtex-6 LX240T FPGA. The board was connected to a host PC containing an Intel i5-650 CPU and running Linux. The IP clock of all designs was set to $freq = 210$ MHz.

The double precision FPGA implementation of PT which serves as the reference for comparison in Section 5.7.2 is the baseline PT sampler of Chapter 3 (with all tuning parameters identical to the ones used in this chapter).

5.6.2 Transferring data between the host and the FPGA

The sequence of operations for generating samples using the FPGA is the same as in Chapter 3, i.e. a C application in the host uses RIFFA driver functions to send the initialization data directly to the FPGA IP and order the IP to start, the IP generates the samples and writes them to the double buffer within the FPGA and simultaneously the data are sent back to the host, where the RIFFA driver receives them and stores them in a file.

Nevertheless, extra operations are needed to perform the four steps of the optimization method which are described in the previous section. During Step 1, the double precision FPGA sampler needs to generate a set of independent MCMC pre-runs. These runs need to be sent to the host in DMA batches (chunks). The size of the batch is set to $N = 5000$, i.e. batch 1 contains the first $N = 5000$ MCMC samples for each independent run, batch 2 contains the next $N = 5000$ MCMC samples for each independent run, etc. Therefore, the C function in the host requests batches of $N = 5000$ new samples for every independent run and waits to receive the results (the size of the batch can be customized). This is done so that a check for convergence can be performed frequently to avoid generating more MCMC samples than necessary. After a batch is received, the $\hat{R}$ metric of Brooks and Gelman [35] is evaluated (using all independent runs) to check whether convergence has occurred with the batches received so far. If convergence has not occurred, the next batch is requested for all independent runs. If convergence has occurred, the available time for Steps 1 and 2 is evaluated as described in Section 5.4.2. The same process is followed during Step 2 for all precisions until the available time for pre-runs ($Term_{\%} \cdot t_{est}$) expires. Also, a weight is generated for every sample. The result of Step 2 is that an equal number of samples is generated for every precision. The host is then able to compute biases and standard deviations for each bias and decide on which precision to use. This is followed by the final runs, where the optimized bitstream is loaded on the FPGA and a set of independent runs is performed. In this case, after every batch of samples arrives at the host, the standard deviation of the output estimate is found and compared to the target standard deviation SD_T . When SD_T is reached, the final run terminates and the output estimate is evaluated.

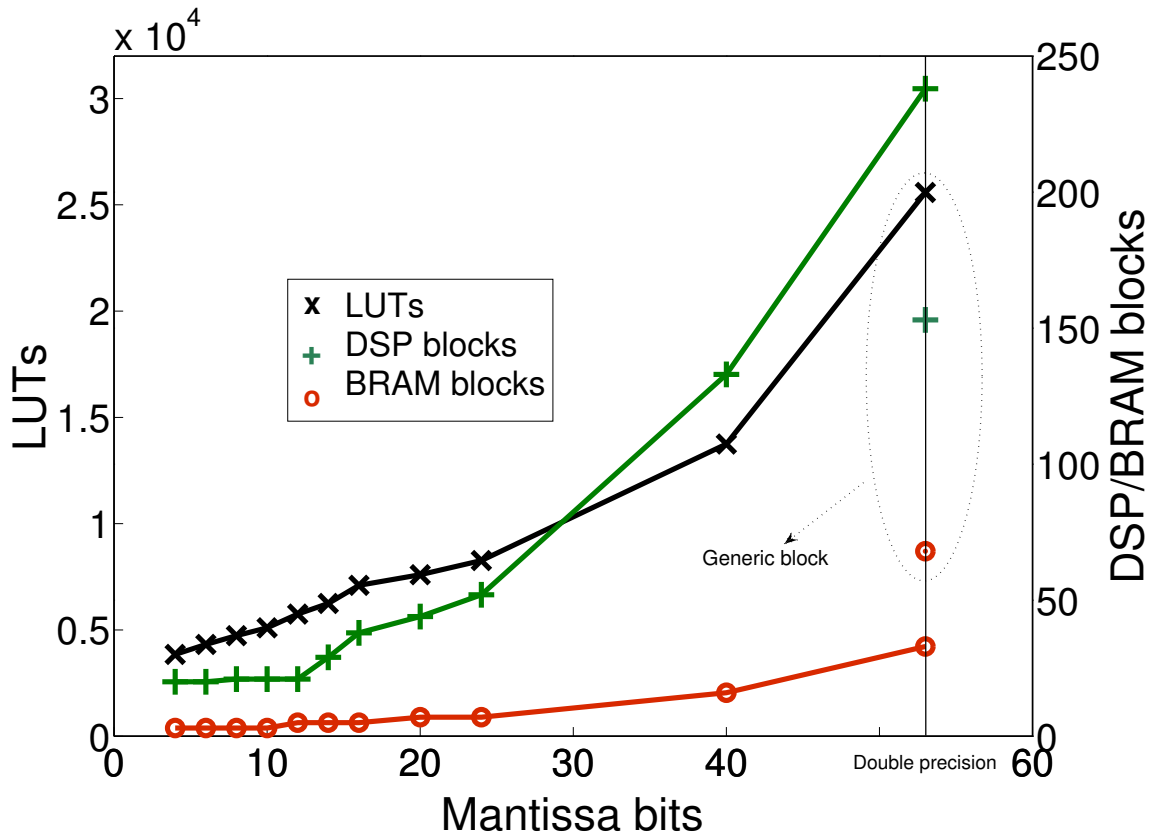


Figure 5.7: Resource utilization of a single probability evaluation pipeline (many of which can exist in a probability evaluation block). The pipeline implements the MM posterior using precisions with various mantissa sizes (exponent bits are always 11). The generic MCMC block’s resources (which always operate in double precision and are not part of the precision optimization) are also shown.

5.7 Investigation and results

This section presents resource utilization results for the bitstreams and blocks described above, an evaluation of the optimization method using the two case studies and a comparison with an unbiased precision optimization method found in the literature [3].

5.7.1 Resource utilization

Figure 5.7 shows the resource utilization of a single probability evaluation pipeline (many of which can be instantiated within a probability evaluation block) for different precisions (MM case study). Large savings are possible by reducing precision, allowing for the instantiation of more parallel blocks. The resource utilization of a generic MCMC block is also shown.

5.7.2 Performance evaluation - trade-off between speed and bias

The optimization methodology and the proposed FPGA-based system were applied to both case studies for a variety of output estimates (combination of some posterior variable and some function of interest $f(\theta)$) and for various user parameters combinations.

Demonstration of the method's steps

This section demonstrates the results of each step of the optimization method using a particular case study and parameter setting: The 2nd moment of the unknown variable μ_1 from the MM case study was set as the output estimate. The target standard deviation was set to $SD_T = 0.02$ and the bias tolerance to $T_{bias} = 0.5$. Six candidate precisions were used in the first (mixed-precision) bitstream ($S = \{(23, 11), (19, 11), (15, 11), (13, 11), (11, 11), (9, 11)\}$). The minimum acceptance probability was set to $Pr_{min} = 0.95$ and the termination parameter was set to $Term_{\%} = 0.05$. The number of independent MCMC runs in each set was set to 30 (applies to all steps and all precisions).

The double precision pre-run of Step 1 was terminated after the first batch of samples was sent from the FPGA to the host, since the $\hat{R}$ metric suggested convergence when the batch arrived ($\hat{R} = 1.012$). This shows that the MCMC sampler is well-tuned and converges rapidly. The standard deviation of the output estimate at that point (SD_{pre}) was evaluated and used (along with the number of samples $N_{pre} = 5000$, the user parameter SD_T and the throughput TH_{dp}) to compute t_{est} , using Equations (5.14) and (5.15).

After Step 1 finished, Step 2 began using the same bitstream. Figure 5.8 shows the results of Step 2 for three out of the six candidate precisions. The generated samples were used to evaluate the bias estimates at various times from the beginning of Step 2 to its termination (although this is done to provide details on the evolution of the bias - the method only needs to compute the biases and standard deviations at the termination of Step 2). Both the bias and its standard deviation increase with lower precisions, as shown in the figure. The figure also shows that Step 2 terminates after around $8.5 \cdot 10^5$ samples to limit the pre-runs' overhead. The time of termination is decided based on t_{est} (evaluated by the preceding Step 1) and the user parameter $Term_{\%} = 0.05$, as described in Section 5.4.2.

Based on the output of Step 2, Step 3 runs on the host in order to decide which precision to use. Figure 5.9 shows the precision selection procedure (with $Pr_{min} = 0.95$). The probability that a bias is within tolerance (see Equation (5.16)) converges to a certain value for each precision as more samples

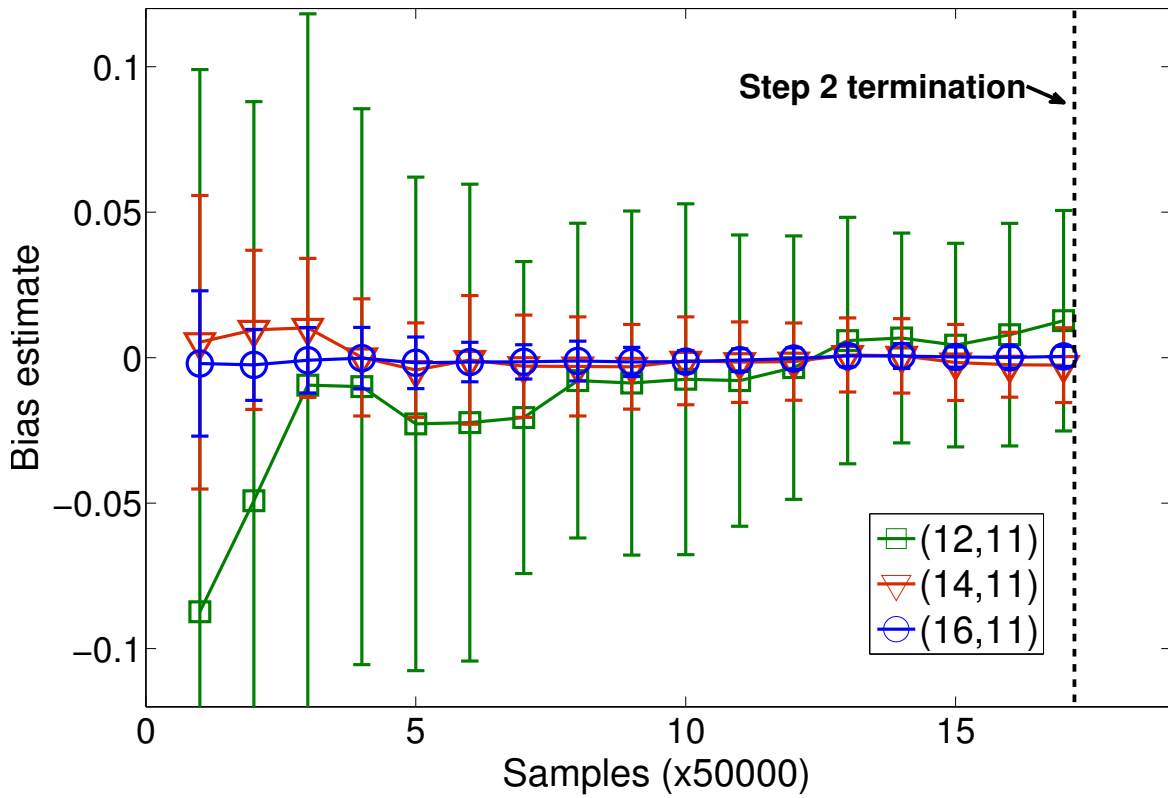


Figure 5.8: Step 2: Bias estimates for 3 out of the six 6 candidate precisions. The MM case study is targeted and the output estimate is the 2nd moment of μ_1 . Error bars represent $\pm 2\sigma_{b_c}$. The vertical line shows when the pre-runs need to stop to avoid large overheads to the final run (the line is set based on the output of Step 1).

are drawn. Few samples suffice to show that the probability is acceptable for high precisions (e.g. (20, 11)). More samples are needed for some lower precisions, e.g. (16, 11) (this is due to the higher variance in the bias estimates for lower precisions, see Figures 5.5 and 5.8). The fluctuations of the probability values appear because the values are estimates and thus come with some variance. At termination, the lowest precision which gives tolerable bias with 95% probability (or above) is $c = (16, 11)$. This is chosen as the optimized precision. Note that the host needs to compute the probabilities only after all samples have been generated (end of Step 2), although Figure 5.9 shows the probability values at various time instances. In other words, Step 3 computes the probabilities only at the right-most point of Figure 5.9.

When the optimized precision has been chosen, the final bitstream which corresponds to this precision (i.e. includes probability evaluation blocks only in the chosen precision) is loaded and the final run (Step 4) starts. It has to be noted that the *optimized* precision is not always the *optimal* one in the set S . If the pre-runs terminate too early because of a large target SD_T or small $Term_{\%}$, a sub-optimal precision can be chosen (e.g. if Step 2 terminates after 10^5 samples in Figure 5.9). This means a sub-

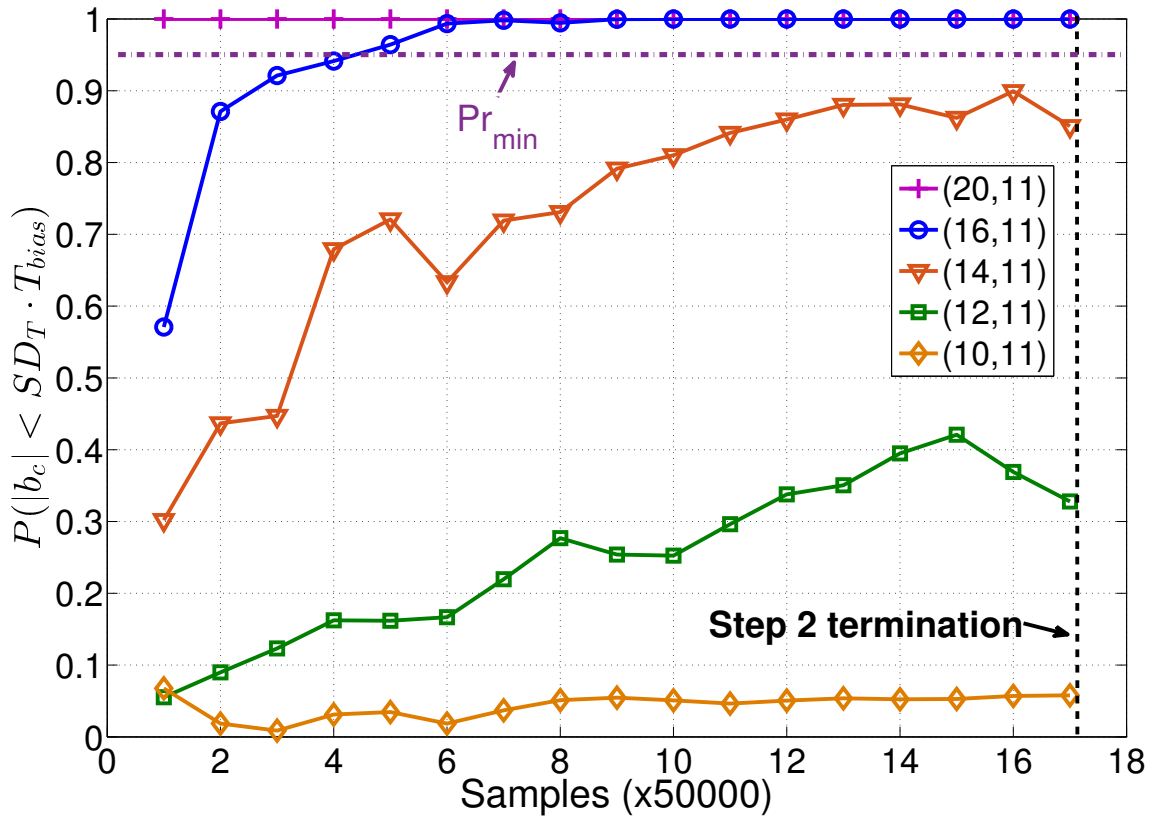


Figure 5.9: Step 3: Probability that the bias is within tolerance for five out of the six candidate precisions (Precision $c = (24, 11)$ is omitted). The output estimate is the 2nd moment of μ_1 (MM case study). Parameters are set to $SD_T = 0.02$, $T_{bias} = 0.5$, $Pr_{min} = 0.95$, $Term_{\%} = 0.05$. The optimized precision is $c = (16, 11)$.

optimal sampling throughput in the final run. Nevertheless, the bias threshold is still probabilistically guaranteed. Moreover, as mentioned in the introduction, a sub-optimal precision might be chosen due to the limited number of candidate precisions in the set S .

Optimization results - comparison to double precision FPGA samplers

Table 5.3 contains the results of applying the optimization method to the two case studies for different functions of interest and parameter combinations. The parameters S , $Term_{\%}$ and Pr_{min} remain the same as above. The SD_T values were chosen to represent one to three decimal digits of accuracy in the MCMC estimate and the chosen T_{bias} values give bias thresholds smaller or equal to one SD_T . All final runs lasted until SD_T was reached. The last column of Table 5.3 shows the speedups that the optimization system achieves against a reference double precision FPGA sampler (baseline PT sampler in Chapter 3) on the same FPGA and using the same tuning parameters. The speedup is defined as follows:

$$Speedup = \frac{Time_{dp}}{Time_{opt} + Time_{op}} \quad (5.22)$$

Table 5.3: Optimization results for various estimates and user parameters from the Mixture Model (MM) and the Neural Network (NN) case studies. In all cases: $Pr_{min} = 0.95$, $Term_{\%} = 0.05$ and $S = \{(24, 11), (20, 11), (16, 11), (14, 11), (12, 11), (10, 11)\}$.

Output estimate	SD_T	T_{bias}	Optimized precision	Optimization runtime $Time_{opt}$ (Steps 1-3)	Final run runtime $Time_{OP}$ (Step 4)	Speedup vs. DP FPGA
Mean of μ_1 (MM)	$5 \cdot 10^{-2}$	1	(14,11)	112.5 sec	102.1 sec	3.96x
	$2 \cdot 10^{-2}$	0.5	(16,11)	461.74 sec	705.3 sec	4.31x
	$7 \cdot 10^{-3}$	0.05	(20,11)	3427.9 sec	6571.7 sec	4.10x
2nd moment of μ_1 (MM)	$5 \cdot 10^{-2}$	0.5	(16,11)	302.7 sec	399.2 sec	4.05x
	$5 \cdot 10^{-2}$	0.1	(20,11)	331.0 sec	455.1 sec	3.76x
Mean of β_1 (NN)	$2 \cdot 10^{-2}$	0.5	(16,11)	15312.4 sec	47184.0 sec	4.90x
	$5 \cdot 10^{-2}$	0.01	(24,11)	2465.8 sec	14119.2 sec	2.95x
2nd moment of γ_{10} (NN)	$5 \cdot 10^{-3}$	0.5	(20,11)	1855.8 sec	6166.8 sec	3.44x
	$2 \cdot 10^{-3}$	0.5	(24,11)	11073.5 sec	49358.7 sec	2.85x

where $Time_{DP}$ is the time that the reference double-precision FPGA sampler needs to achieve the target standard deviation SD_T in the output estimate, $Time_{opt}$ is the time spent for optimizing precision using the proposed methodology (i.e. the time consumed by the first FPGA bitstream to complete Steps 1-2 plus the time spent by the host PC to complete Step 3) and $Time_{OP}$ is the time that the optimized-precision FPGA sampler needs to achieve the target standard deviation SD_T in the output estimate (i.e. the time consumed by the second FPGA bitstream to perform the final run - Step 4). The times $Time_{opt}$ and $Time_{OP}$ are shown in Table 5.3. In all cases, the chosen precisions are lower than double precision. Depending on the output estimate and the parameters, optimized precisions range from (14, 11) to (24, 11). The speedups range from 2.85x to 4.90x. All of this speedup is due to the use of lower precision (which translates into more parallel probability evaluation pipelines in the final run compared to the ones used by the reference double precision sampler). All other parts of the two PT implementations are identical.

By looking at the $Time_{opt}$ and $Time_{OP}$ values in Table 5.3, it can be observed that larger SD_T leads to the optimization process (pre-runs - $Time_{opt}$) taking up a larger percentage of the total runtime (e.g. compare $Time_{opt}$ and $Time_{OP}$ in lines 1-3). This can be explained as follows: The runtime of the first bitstream (which dominates $Time_{opt}$) is a fixed percentage of the theoretical runtime t_{est} , which is the estimated runtime of a double-precision sampler aiming to reach SD_T . Therefore, the runtime of the first bitstream changes proportionately to the estimated runtime of the double precision sampler (which is inversely proportional to SD_T^2). The runtime of the second bitstream ($Time_{OP}$) is related to SD_T in the same way but it is also affected by the chosen (optimized) precision. This precision

becomes smaller with larger SD_T (since the error tolerance is less strict and smaller precision can be used). The result of using smaller precisions is that less time is needed to complete the final run using the second bitstream. Therefore, when SD_T is increased, the runtime of the second bitstream ($Time_{OP}$) decreases both due to its inversely proportional relationship with SD_T^2 and due to the use of smaller optimized precisions. This is a faster rate of decrease compared to the runtime of the first bitstream (most of $Time_{opt}$), which simply drops inversely proportionally to SD_T^2 . The faster rate of decrease in $Time_{OP}$, leads $Time_{opt}$ to take a larger percentage of the total runtime as SD_T grows.

Comparing the first three lines of the table, it is also clear that the speedup in the first line is lower than the speedup in the other two lines, in spite of the lower optimized precision that the method achieves in the first line. This is due to the increased SD_T setting which increased the pre-runs (first bitstream) overhead (as explained in the previous paragraph). The increased pre-runs overhead cancels out the speedup gains offered by the low precision in the second bitstream. Moreover, large SD_T values can have a negative effect on speedup in another way: Large SD_T means that Step 2 can become too short (due to small t_{est}), resulting in a sub-optimal precision being chosen (for example a sub-optimal choice would be made if the run in Figure 5.9 stopped after 50000 samples).

On the other edge of the bias tolerance spectrum (e.g. for the strict SD_T and T_{bias} configuration in the last line of the table), the speedups also decrease but in this case it is a result of the high selected precision ($c = (24, 11)$) which results in a slow final run (higher precisions offer smaller sampling throughputs). The peak speedups vs. the double precision sampler are achieved with intermediate SD_T values (which also lead to intermediate precisions). Finally, it is worth noting that the pre-runs runtime is only affected by SD_T and $Term\%$, not by T_{bias} . The latter only affects precision selection, i.e. higher precisions are needed in order to guarantee smaller T_{bias} .

Figure 5.10 shows how the speedup over the double precision FPGA sampler changes with the choice of accuracy parameters SD_T and T_{bias} for a fixed output estimate (mean of μ_1 in the MM case study). Speedup is defined in the same way as above (see equation (5.22)). The optimized precisions are between (12, 11) and (24, 11) and a speedup between 2.99x and 4.61x is achieved. There is a tendency for higher speedups as the user requirements become less strict (larger SD_T and T_{bias}). This is due to the selection of lower precisions. Nevertheless, the speedup drops with higher SD_T for $SD_T > 0.02$ (instead of increasing). This is due to the effect of the pre-runs overhead as well as the choice of suboptimal precisions in some cases, as described in the previous paragraph. The relatively sudden drop in acceleration for $SD_T = 0.005$ and $SD_T = 0.0025$ when $T_{bias} = 0.01$ is due to the use of pre-

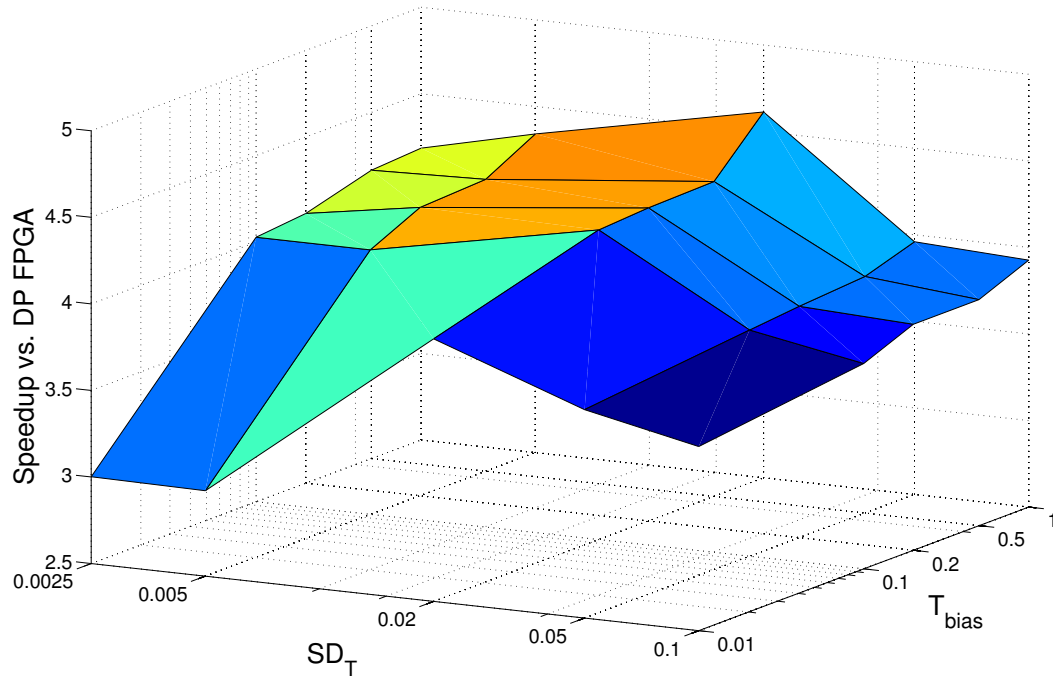


Figure 5.10: Trade-off between tolerable bias and speedup (vs. double precision FPGA sampler). Bias tolerance is represented by the two parameters SD_T and T_{bias} . The output estimate is the mean of μ_1 (MM case study). The other user parameters are set to $Pr_{min} = 0.95$, $Term_{\%} = 0.05$ and $S = \{(24, 11), (20, 11), (16, 11), (14, 11), (12, 11), (10, 11)\}$.

cision configuration $c = (24, 11)$, which requires significantly more FPGA resources to implement a probability density pipeline compared to lower precisions.

One of the key ideas of the presented method is that it targets a specific output estimate each time. It does not attempt to optimize for all possible estimates (functions $f(\theta)$) related to the target distribution $p(\theta)$. By doing this, the precision is highly optimized for the problem under investigation only. i.e. the method can exploit information about the shape of $f(\theta)$ in order to optimize precision.

Moreover, through the manipulation of input parameters, the choice of precision can be adapted to the specific accuracy requirements of the user. The user can exploit the bias/standard deviation tradeoff, which is common to statisticians. Another possibility is to adapt the pre-run overhead according to how many final runs are going to be performed. For example, it is common for the distribution to be sampled many times (e.g. 20 final runs of 100 independent MCMC runs each) to investigate MCMC tuning parameters. In this case, the pre-runs only need to be done once (i.e. 1 set of 100 independent runs). This means that the pre-runs overhead is now very small compared to the total runtime (which is dominated by the 20 final runs of 100 MCMC runs each). This is a scenario where the user could choose to increase the time available for pre-runs in the hope that a better (lower) precision will be

chosen; since the pre-runs will have more time to reduce the variance of the bias estimates, it is possible that a precision which was considered inadequate with shorter pre-runs will now prove to be adequate based on the probabilistic criterion of Equation (5.16). In general, the assumption that only one set of final runs will be performed (which is used throughout the chapter) is strict.

5.7.3 Comparison to an unbiased precision optimization approach

The methodology presented in this chapter focuses on MCMC samplers and aims to reduce precision as much as possible while tolerating a custom amount of bias in the output. It is interesting to investigate the relative advantages of this approach compared to an *unbiased* precision optimization approach. The only existing unbiased precision optimization method in literature, which is applicable to Monte Carlo algorithms, is described by Chow et al. [3]. The method focuses on non-MCMC Monte Carlo sampling, where the generated samples come from simple distributions (e.g. Normal) and they are independent (not correlated as in MCMC). In order to perform precision optimization, Chow et al. [3] propose the use of mixed precision sampling where: 1) A low precision sampler generates samples from the target distribution, 2) A mixed precision sampler (consisting of high- and low-precision modules) generates samples from the distribution of the bias. The output estimates of the two samplers are added in order to correct the bias of the low precision sampler and get an unbiased result. Therefore, Chow et al. [3] do not introduce any bias in the output. The standard deviation of the output is equal to the sum of the standard deviations of the low- and mixed-precision estimates. The sampler runs until the target standard deviation (SD_T) is reached.

Here, the method of Chow et al. [3] is ported to the MCMC domain in order to create an unbiased precision optimization method that targets MCMC. This unbiased method is compared with the method presented in this chapter. Some necessary changes and assumptions are made in order to make the comparison possible:

- 1) Chow et al. [3] assigned the double-precision part of the mixed-precision computations to a CPU. In contrast, the unbiased method of this section implements both the low-precision and mixed-precision computational modules of the method inside the FPGA.
- 2) In contrast with the bias estimator proposed in [3], the bias estimator of Section 5.3 (which is suitable for MCMC samplers) is used in the unbiased method of this section.
- 3) As in [3], for each target SD_T a particular combination of low-precision and mixed-precision mod-

ules are instantiated in order to minimize the runtime needed to reach the target standard deviation (SD_T). The process described in [3] to perform this optimization is followed here, although the mixed precision part is substituted by the bias estimator introduced in this chapter.

4) It is assumed that the pre-runs procedure (Steps 1 and 2) described in this chapter is used in order to estimate σ_f^2 and the various $\sigma_{f_{bc}}^2$ in the unbiased method (for all $c \in S$). These quantities are needed in [3] (and in the unbiased method of this section) in order to optimize the number of low- and mixed-precision modules for a given SD_T . This overhead is not included in the results of [3], where it is assumed that the variances are known from beforehand. Nevertheless, it is reasonable to assume that pre-runs are needed every time a new target distribution is examined (as done in this chapter). It is assumed that the pre-runs termination criterion introduced earlier in this chapter is used to terminate the pre-runs in the unbiased method and therefore the same runtime overhead is added.

5) It is assumed that, after estimating the variances, the unbiased method decides which precision is optimal (equivalent to Step 3 in the previous sections) and then performs the final run using the low- and mixed-precision modules (equivalent to Step 4 in the previous sections).

6) The unbiased method was not implemented on a real FPGA. The throughput and runtime presented in this section for the unbiased method were estimated based on: 1) The resource utilization of the probability evaluation modules in the various precisions, 2) The available resources of the ML605 FPGA, 3) The bias and variance of the bias results from the real runs of the previous section, 4) Equations (17) and (18) of [3].

Figure 5.11 shows the speedup of the method proposed in this chapter against the unbiased method when targeting the mean of μ_1 (MM case study) with $SD_T = 0.007$. The figure shows how the speedup changes when adjusting the bias “knob” T_{bias} . The pre-runs runtime overhead is included in all measurements. The unbiased method uses a combination of 1 mixed precision module and 9 low precision modules for the final run. The low precision is set to $c = (16, 11)$, which gives the minimum runtime compared to other configurations. The method proposed in this chapter uses a varying number of low precision modules for the final run, depending on T_{bias} (since T_{bias} affects the chosen precision as explained earlier in the chapter). The chosen precisions range from $c = (14, 11)$ to $c = (24, 11)$.

It is clear from Figure 5.11 that the method proposed in this chapter is faster than the unbiased method by up to 1.45x. This is a result of being able to instantiate more parallel probability evaluation modules inside the second bitstream, compared to the modules instantiated in the second bitstream of the

unbiased method. The reason for this difference is that the unbiased method needs a double precision module in the second bitstream to correct bias. This module consumes a significant percentage of the FPGA's resources. The penalty that the method proposed in this chapter pays is the addition of (controllable) bias in the output estimate. As the bias tolerance converges to zero, the speedup decreases, since larger precisions are chosen by the proposed method, which lead to more time-consuming final runs. In contrast, the unbiased method keeps using the same precision configuration (since it has no T_{bias} parameter). Setting the bias tolerance to zero ($T_{bias} = 0$) makes the proposed method 3.3x slower than the unbiased method (not shown in Figure 5.11). This happens because, with $T_{bias} = 0$, the chosen (optimized) precision is always double precision. Double precision probability evaluation modules consume a lot of area in the FPGA, which means that only 2 of these modules fit in the final bitstream (in contrast to the 9 low precision modules of the unbiased method). Therefore, the (biased) method proposed in this chapter is worth using in this case study only when some bias can be tolerated. Similar results are achieved for other functions $f(\theta)$.

5.8 Discussion

This chapter presents a method that exploits the reconfigurable nature of FPGAs to minimize the arithmetic precision of any FPGA-mapped MCMC sampler, while probabilistically guaranteeing a user-constrained bias in the output. The method uses an efficient bias estimator, which is proposed here. Results show that significant speedups over double-precision designs are achieved and that the method allows the user freedom to adjust the amount of bias according to requirements, trading off speed for accuracy or vice versa. A comparison with a ported version of the unbiased method presented in [3] shows that a speedup of up to 1.45x is possible when bias can be tolerated in the estimate. A potential drawback of the method is the requirement to perform pre-runs in all candidate precisions, although this can be a negligible overhead in many case (as explained in the preceding sections).

Due to the explosion in the amount of data processed by modern MCMC samplers, the computational cost of evaluating the probability density is growing at a fast rate. Minimizing this cost in any way is highly desirable to tackle large-scale analyses. The method presented in this chapter manages to do this without making any assumption on the form of the probability density, apart from the assumption that it is parallelizable in some way. This technique can thus prove valuable for many problems in Bayesian inference, in contrast to parallelization techniques which are optimized for the computational

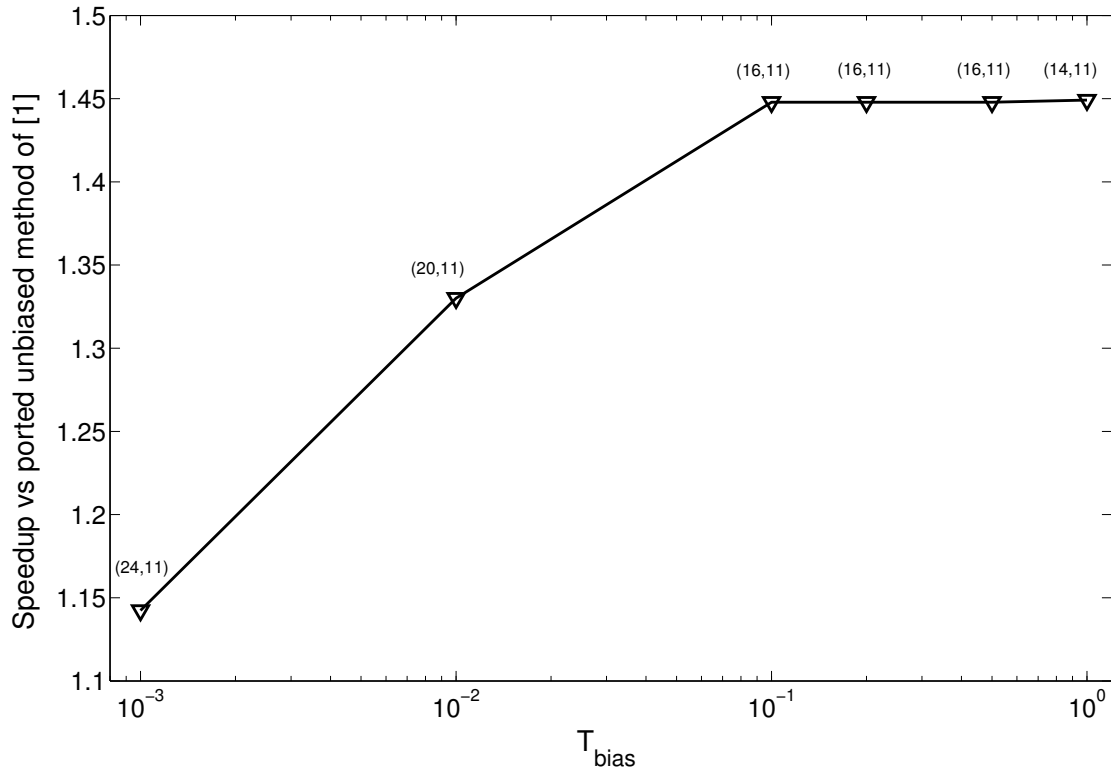


Figure 5.11: Speedup of the method proposed in this chapter against ported unbiased method of Chow et al. [3] for various values of T_{bias} . The target standard deviation is $SD_T = 0.007$. The output estimate is the mean of μ_1 (MM case study). The other user parameters are set to $Pr_{min} = 0.95$, $Term\% = 0.05$ and $S = \{(24, 11), (20, 11), (16, 11), (14, 11), (12, 11), (10, 11)\}$. The chosen (optimized) precisions of the method proposed in this chapter are shown above each point in the graph. The low precision of the ported unbiased method is always $c = (16, 11)$ (optimized using the process described in [3]). The runtimes of both methods include the time spent for pre-runs, as well as the time spent for the final runs.

structures found in particular classes of densities.

Moreover, a lot of stochastic methods in recent years have made use of bias “knobs” similar to the one used in this chapter (e.g. [79] and [80]). This allows the user to sacrifice accuracy in favour of speed, which can prove crucial for computationally demanding inference tasks. Therefore, the focus of this chapter on biased inference is on par with a wider trend in the Bayesian and Monte Carlo literatures, which emphasizes the need to cut computational costs even if this leads to some degree of inaccuracy in the result. The following chapter, which concludes this thesis, provides more information on the current state of the MCMC and MCMC acceleration fields, as well as a discussion of possible future developments.

Chapter 6

Conclusion

The previous chapters propose novel MCMC algorithms, modifications to existing MCMC algorithms, FPGA architectures for MCMC and precision optimization methodologies for MCMC on FPGAs, with the aim of improving the efficiency of MCMC sampling for large Bayesian problems. This chapter gives an overview of the current state of research on MCMC acceleration, summarizes the main achievements of the thesis, discusses some important questions that were posed in the previous chapters, and proposes future work directions to improve on the results presented in this thesis.

6.1 Overview of MCMC acceleration research

One of the major scientific challenges of recent years has been the pressing demand for ways to analyse and extract knowledge from massive data sets, particularly using advanced statistical methods. This trend, which is the main motivation for the rise and evolution of Data science, has also influenced developments in the MCMC literature. Since MCMC is one of the mainstream methods used for extracting the knowledge from data and the computational burden of large-scale MCMC-based analysis has naturally exploded, the interest in exploiting parallel hardware platforms for tackling this burden has gained momentum recently. Although accelerating MCMC through the use of parallel hardware is still a relatively under-mined research field, it has already had an effect on the way statisticians understand MCMC-based inference. Practitioners now realize that MCMC computation is not synonymous to MCMC methodology, i.e. the construction of MCMC algorithms to process the data. It is rather a term that represents many design aspects, such as how the algorithm will be mapped to a hardware

device, how the data will be managed, etc. All of these aspects need to be taken into account when designing and implementing MCMC. It has also become clear that the requirements of modern applied statistics in fields such as machine learning, medicine, biology and environmental modelling can no longer be addressed simply by improving MCMC algorithms (which was the assumption in the first 15 years of “modern” MCMC history) [157]. The use of parallel platforms and High Performance Computing (HPC) infrastructure is a necessity and a shift has to be made towards embracing them as a powerful tool for performing MCMC-based analysis [158, 159].

The above realisations are mirrored in the way Data science is being understood and defined: A field that combines *statistics* and *computing*, aiming to use data (in many cases “big” data) in order to extract useful knowledge and help make decisions [157, 160, 158]. MCMC is one of the algorithms used in Data science, particularly when Bayesian models are employed. Thus this combination of statistics and computing is naturally becoming a trend in MCMC literature [16, 2, 26, 67, 66, 76].

As discussed in Chapter 2, multi-core CPUs, GPUs and FPGAs have all been employed to achieve higher sampling throughputs during the last few years. Currently, most of the fundamental MCMC methods which are widely used by the statistics community (Hamiltonian Monte Carlo, Parallel Tempering, Particle MCMC, Slice sampling), have been mapped to parallel hardware, offering speedups of one to two orders of magnitude compared to desktop CPUs. Nevertheless, several popular MCMC methods have not been implemented in hardware, although they are amenable to parallelization (e.g. MCMC with multiple proposals [56]). This is also true for newer methods; despite the fact that some of them have been designed with parallel hardware in mind [77, 78], there is a lack of real-world, massively parallel implementations. In addition to work focusing on specific MCMC methods, there are several efforts towards accelerating model-specific computations (i.e. computation of the likelihood) using parallel hardware. Nevertheless, these efforts lack general applicability.

Despite this increasing relevance of parallel hardware for MCMC computation, the MCMC community is still a long way from widely adopting parallel hardware implementations for everyday inference tasks. This problem is more intense in academia than it is in industry, since in the latter multi-disciplinary teams (including data engineers) are typically employed to exploit parallel hardware infrastructure. In contrast, academic statistics researchers are accustomed to implementing their algorithms in R and Matlab or using some high-level tool like WinBUGS [161]. Even C/C++ implementations are rare. Knowledge of MPI, CUDA, OpenCL or hardware description languages is extremely scarce. Cooperation between statisticians and computer scientists/engineers in academia

has increased but it is far from common.

Moreover, while CPUs and GPUs have proved popular for MCMC acceleration, FPGAs have been limited to specific application areas (e.g. phylogenetics [25, 63]). In academia, this can be attributed to the almost non-existent availability of FPGA boards outside engineering departments and the fact that writing code for them is difficult and requires specialized skills (even when using High Level Synthesis tools). Nevertheless, FPGAs combine a series of features that make them attractive for MCMC computation; fully custom architecture, custom precision, fast on-chip communication, low power consumption. In addition, the latest FPGAs provide native support for floating point computations (using dedicated hardened circuitry similar to the well-known fixed point multipliers). Given that MCMC algorithms are implemented almost exclusively in floating point, this new capability is promising.

Finally, almost no attention has been given on the idea of co-designing MCMC algorithms and MCMC hardware. This comment applies to all hardware platforms. Existing literature has focused on naively mapping MCMC methods/likelihood computations to hardware. This approach can be successful in many cases (especially when the algorithm is embarrassingly parallel) but fails to fully exploit the power of modern computing platforms.

6.2 Summary and discussion of thesis achievements

This thesis contains several contributions towards the use of FPGA as MCMC accelerators. It enriches the toolset of hardware-accelerated MCMC samplers, paying particular attention to how algorithm design and hardware design can be done jointly.

Chapter 3 focused on popMCMC (and more specifically on Parallel Tempering). It investigated ways in which popMCMC can be accelerated using FPGAs, moving along two directions. Firstly, it presented an FPGA architecture for popMCMC, which achieved significant speedups over sequential software and optimized CPU and GPU samplers (for most problem sizes). This architecture made extensive use of pipelining and parallelism, exploiting the structure of the Global updates and exchanges of the popMCMC algorithm. The impressive speedups over sequential software are a result of the suitability of the algorithm for parallelization. Secondly, the chapter looked into ways in which the efficiency of this baseline FPGA sampler can be further improved by jointly enhancing the algorithm

and the underlying hardware. Two novel modifications to the popMCMC algorithm were proposed, along with accompanying FPGA architectures. Both modifications were based on using custom precision in an algorithm-aware fashion. The first modification used low precision for all popMCMC chains and corrected sampling errors using importance weights for the first chain, which proved to be a small overhead compared to the speed gains from reducing precision. The second modification showed that it is possible to use low precisions for all auxiliary chains and high precision only in the first chain with no cost in sampling accuracy. Speedups of up to 6.5x over the baseline FPGA sampler were achieved using the above two modifications on an FPGA.

The absence of sampling errors in both custom precision methods is important. Although biased sampling methods are common nowadays, most MCMC practitioners are still used to working with unbiased methods. Also, avoiding trade-offs between accuracy and speed (which require tuning) improves ease of use. Moreover, in applications that require accurate sampling, unbiased methods have an obvious advantage. Furthermore, both custom precision techniques are good examples of the fusion of algorithm and hardware design mentioned above. This fusion was successful because popMCMC has the distinctive characteristic that only the samples from the first chain are actually kept. The samples of auxiliary chains are thrown away and this can be exploited in the FPGA platform, which can use very low precisions. The two modified methods were also mapped to CPUs and GPUs, where the gains are smaller compared to FPGAs since precision cannot be fully customized. This reveals that FPGAs are an extremely promising platform for MCMC if their relative advantages are exploited in a way that suits the algorithm. The speedups of FPGAs over the other platforms when custom precision is used range up to one order of magnitude.

The main limitation of the two custom precision techniques is that their effective speedups (i.e. speedups that take into account the effect of mixing) over software drop when large data sizes are used in the likelihood due to the accumulation of precision-related error in computations (which has a negative effect on mixing). Tackling this disadvantage is an open research question. Moreover, although both methods guarantee unbiased sampling with any custom precision configuration, their performance is maximized at specific precision configurations which are application-dependent. Finding the optimal precision configuration requires pre-runs in many different precisions. These pre-runs can be short if the sampler converges quickly to the target distribution, since in this case few samples are required to get an accurate estimate of the necessary *ESS* metrics. Nevertheless, if convergence is slow, pre-runs have to be long to make sure that the samplers converge before they give

an accurate estimate of ESS . Whether long pre-runs are a problem depends on the way the practitioner uses the sampler. Fortunately, MCMC practitioners typically perform a few pre-runs to tune algorithm parameters (e.g. the variance of the proposal distribution) and then perform multiple final runs (usually dozens) to get the final estimate of the integral of interest and estimate the variance of the estimate. In this scenario, the optimization of precision can be seen as an extra tuning step. The overhead from performing a few custom precision runs is small compared to the runtime needed for the final runs (since the final runs are more in number) and the gain from performing the final runs in reduced precision is significant.

In Chapter 4, the focus was transferred to another popular and computationally intensive MCMC variant, the pMCMC algorithm. pMCMC is commonly used on SSMs with unknown parameters, a class of Bayesian models which is used in many real applications and often needs to process large data sets. The chapter introduced the first FPGA-based accelerator for pMCMC. The tailored FPGA architecture focused mostly on parallelizing the PF operations inside each pMCMC iteration. The achieved speedups over state-of-the-art CPU-based and GPU-based pMCMC samplers ranged up to one order of magnitude. The performance of the FPGA was limited by the resampling step, which is not trivially parallelizable (due to the particle memory update after replication indexes are found). The results also showed that larger numbers of PF particles in pMCMC do not always lead to higher effective performance. This has already been shown in previous literature [132] but Chapter 4 extended the analysis to take into account the effect of parallel hardware implementations.

Continuing the approach of fusing hardware and algorithmic design, the chapter also introduced a novel pMCMC algorithm which aims at increasing sampling efficiency for multi-modal posterior distributions in SSMs, along with a tailored architecture which maps the algorithm to an FPGA. The new algorithm, denoted ppMCMC, combined the features of pMCMC and popMCMC, effectively transforming pMCMC to a multiple-chain method. The FPGA architecture made use of coarse-grain pipelining in the PF datapath to increase resource utilization, taking advantage of the independence of the PF runs of different ppMCMC chains. Results indicated that the new algorithm and its FPGA architecture provide significant speedups over pMCMC implementations in CPUs and GPUs when multi-modal posteriors are targeted. These speedups can be maximized by tuning the number of particles and the number of chains. The results again confirmed that the co-design of algorithms and hardware is beneficial when knowledge about the functionality and purpose of the targeted method is exploited.

The proposed hardware-supported samplers enable pMCMC to compete with approximate methods for SSM inference, such as [83], which are faster than software-based pMCMC. This is critical for applications where exact results are required. Until recently, pMCMC's applicability was limited by long runtimes. Practitioners are more likely to use pMCMC when knowing that sampling can finish in a reasonable time frame through the help of hardware acceleration. Demonstration of the benefits of pMCMC over [83] for difficult problems is the subject of future work. Regarding ppMCMC, finding the optimal combination of chains and particles requires pre-runs, with the comments made previously on popMCMC also applying in this case. One possible limitation of ppMCMC is that it requires replicating most of the memories in the system so that each chain has its own set of memories. Although the numbers of chains are unlikely to range to more than a few dozens in real scenarios, replicating memories can be costly when massive numbers of particles are used.

Finally, Chapter 5 followed a different approach compared to the previous two chapters. It proposed a generic methodology to optimize the precision used for probability density computations inside each MCMC step. The methodology is applicable to any FPGA-based MCMC sampler and is based on performing short pre-runs to estimate the bias introduced by each precision configuration within a pre-defined set of candidate configurations. The bias refers to the estimate of the integral of interest. The main contributions of the chapter are a novel bias estimator which is based on combining high- and low-precision samples and a methodology which uses the bias estimates to find the minimum precision which satisfies a user-defined bias tolerance. The methodology was implemented on an FPGA system, where the reconfigurability of FPGAs was exploited by successively loading mixed-precision and optimized-precision bitstreams on the device. The achieved speedups over a reference double precision FPGA sampler were up to 4.9x (including the optimization overhead from the first bitstream). This speedup was achieved with some cost in terms of bias but the amount of bias can be controlled by the user.

By design, the proposed methodology targets a specific type of estimate (i.e. specific function $f(\theta)$ in equation (2.8)). It is thus able to optimize precision specifically for this estimate and for the bias that the user selects, which is more effective than precision optimization for all estimates. Due to the recent explosion in the size of inference tasks in real world problems, the idea of accepting some bias in the result in exchange for faster inference has gained ground [79], despite the inclination of most practitioners to use unbiased methods (as mentioned above). The method presented here is an extension of approximate MCMC inference ideas into the field of FPGA computation, which has not

been attempted before. The ability to control the bias is a plus, since most approximate inference methods do not give any guarantee on the level of introduced bias.

The speedups that the presented method achieves are similar to the ones achieved by the two custom precision popMCMC samplers in Chapter 3. This demonstrates that this is the range of speedups that should be expected when lowering precision in FPGA implementations of MCMC (given that the introduced bias is not significant). The slightly higher speedup in Chapter 3 is explained by the fact that the methods proposed there are tailored for the specific algorithm, while the methodology in Chapter 5 is generic.

The main caveat of the method is the requirement to perform pre-runs in all candidate precisions. This can create similar issues as the ones described for the methods in Chapter 3, although a long final run consisting of multiple independent iterations will, again, make the pre-run overhead small. In addition, in the case of Chapter 5, the pre-runs' MCMC samples have potential for reuse: If a different type of estimate (function $f(\theta)$) is needed, the same pre-run samples can be used to find a new optimal precision for the new estimate without repeating the pre-runs (i.e. Steps 1 and 2 of the methodology are avoided). The new optimal precision is then used to perform the new final run. Having said that, the most desirable scenario for a precision optimization methodology would be to avoid pre-runs entirely (i.e. avoid simulation-based optimization). Such an “off-line” precision optimization approach (i.e. without taking samples) would remove the complexity of implementing and compiling a separate bitstream for pre-runs.

General comments

As a general conclusion, this thesis has shown that FPGAs are a promising platform for MCMC acceleration. The reported speedups for two popular MCMC families are encouraging and outperform state-of-the-art samplers in CPUs and GPUs, while the generic precision methodology showed that any MCMC method can enjoy efficiency gains when using an FPGA. The most unique contribution of the thesis is the idea of co-designing MCMC algorithms and MCMC hardware, which led to significant performance improvements over straightforward implementations. Particularly, the exploration of the various ways in which custom precision can benefit MCMC samplers (depending on the algorithm and the user requirements) shows the large potential of FPGAs as MCMC accelerators and encourages further research on the topic as well as on the wider domain of approximate inference.

The main limitations of the presented work are related to the pre-runs that are necessary to optimize

arithmetic precision and/or tune other parameters of the architectures; although typically short, pre-runs can present problems in the scenarios mentioned above. Nevertheless, even the baseline versions of the architectures (without tuning) manage to outperform competing platforms. Furthermore, the fact that the PT architecture is implemented in VHDL can hinder its wide adoption, since specifying the MCMC posterior in VHDL is beyond the typical skill set of practitioners. The pMCMC architecture is not limited by this, since it is written in C++. Finally, it is important to note that the thesis focused on optimizing the core computations (main compute kernels) of various methods on a single FPGA device. This work can have significant impact on MCMC-based inference in several sectors of academia (where single-device computation is common), as well as in industry. Nevertheless, the extension and enhancement of the presented systems in order to be used in distributed, multi-device environments is necessary in the future to cover the increased computational demands of big data inference.

Comparison of computing platforms

Regarding the suitability of the various computing platforms (multi-core CPUs, GPUs and FPGAs) for MCMC inference, the choice depends on several factors. The multi-core CPU platform is by far the easiest to program, making it ideal for users with no parallel programming background. It is also available in practically every PC. Moreover, a lot of application-specific software libraries for multi-cores can be found, which makes MCMC implementation easier for practitioners. Nevertheless, the limited parallelism of multi-core CPUs and the fact that their frequencies have stopped increasing mean that they cannot satisfy the needs of demanding statistical analyses. Multi-core CPUs can be faster and should be preferred over the other two platforms when the implemented algorithm is non-parallel and/or is dominated by conditional operations and/or accesses the memory randomly. The mainstream Gibbs and Reversible-Jump MCMC samplers [9] are good examples. The former is purely sequential, while the latter contains operations which do not map well to parallel platforms.

GPUs have massive computational power and are relatively easy to program. Someone with no parallel programming experience can learn to write efficient CUDA code in a few months. Moreover, GPUs are widely accessible, as they are embedded in most PCs, and they are relatively cheap. GPUs offer significant speedups if the characteristics of the implemented algorithm match their architecture (i.e. if the algorithm is SIMD). Even for non-SIMD algorithms, it is possible to modify the way the computations happen in order to achieve better mapping to the GPU architecture, although this requires effort and does not always help. For popMCMC and pMCMC algorithms, GPUs are a good

match, since they can easily exploit the existing parallelism. Unfortunately, there are only small gains (or no gains) in MCMC mixing when increasing the number of chains and the number of particles respectively above a certain limit. This means that it is likely that the GPU device will remain under-utilized in many real use cases where small or medium numbers of chains and particles are used. This can change if the likelihood computations are SIMD, since more parallel tasks are available. GPUs are also suitable for other parallel MCMC methods, e.g. Multiple-Try Metropolis [56] and related methods.

Finally, FPGAs are capable of delivering high speedups but they are harder to program. Nevertheless, some of their attributes make them strong competitors in the statistical computing field. They have the advantage of a completely customizable architecture, translating to significant performance gains (as demonstrated in this thesis). More specifically, they are not limited to exploiting embarrassingly parallel algorithms, like GPUs are; provided the designer is familiar with the structure and characteristics of the targeted methods, FPGAs can efficiently exploit non-obvious and/or limited parallelism, maximize pipelining effectiveness and adapt the memory architecture to the access pattern of the algorithm. This is particularly important for Bayesian inference, since the types of computations in the likelihood and the prior change from application to application; they can range from easily parallelizable Gaussian evaluations to complex linear algebra operations [10] or specialized algorithms, e.g. Felsenstein's pruning algorithm for phylogenetic likelihoods [25]. An architecture tailored for the targeted computations can perform much better than a generic one, although the development time is relatively long. In addition to the tailored architecture, FPGAs offer the ability to experiment with custom precision, which is one of the main themes of this thesis. MCMC algorithms which are robust to precision reduction are ideal candidates for FPGA implementation. Finally, FPGAs provide significantly higher on-chip memory bandwidth compared to GPUs and CPUs, which can pay off when implementing MCMC methods with frequent communication between parallel processes.

Unfortunately, FPGAs also suffer from limitations that can hinder their wide adoption by the statistics community. They require programmers with a specialized skill set, they require long development times, they are generally not available in data centres and university clusters (with a few exceptions) and they are usually more expensive than GPUs and CPUs. The first two limitations can be mitigated by the fact that, during the last few years, several high level synthesis tools have been developed for FPGAs, allowing developers to write code in C, C++, OpenCL, etc [93]. This removes most of the barriers that software programmers faced until today and also accelerates development. Nevertheless,

the wider availability of FPGAs remains an issue. Recent developments in the CPU/FPGA field [162] might contribute to change this situation, and further extend the use of FPGAs in data centre and HPC environments.

6.3 Future work

Long term predictions on where the MCMC acceleration field is heading are hard. Nevertheless, it is likely that, as GPUs, FPGAs and heterogeneous devices become more common in big data centres [162] and big data analyses become mainstream for a wide range of businesses and universities, the demand for optimized libraries to accelerate MCMC (as well as other data analysis algorithms) will grow significantly. New algorithms, suitable for the above environment, are also likely to be proposed.

In more detail, algorithms and accelerators will seek to exploit distributed computing principles (as single-device processing is not enough for massive analyses), will leverage the power and flexibility of heterogeneous computing and will look to trade accuracy for speed and vice versa. Furthermore, research on how these algorithms can efficiently access memory [163] is likely to prove crucial, since memory access is one of the main bottlenecks in modern large-scale analyses. Finally, in order for accelerators to be widely accessible, they need to closely integrate with existing high-level MCMC tools.

Based on the above trends and predictions, as well as the work presented in this thesis, this section suggests possible directions for future research.

6.3.1 Targeting other MCMC methods

There are numerous computationally intensive MCMC algorithms which are often employed in demanding Bayesian applications and which could benefit from hardware acceleration. Of particular interest are some recently proposed samplers which are based on data partitioning [78, 77, 75]. These are specifically designed for implementation on parallel platforms since they allow many independent MCMC samplers to tackle separate chunks of data. Another extremely promising recent algorithm is Firefly Monte Carlo (FMC) [62], which is designed to tackle large data sets. FMC subsamples the data in each iteration; a full sub-likelihood is computed only for a few data (“bright data”) and an approximation of the sub-likelihood is used for the rest of the data (“dark data”). In order to avoid

sampling bias, the approximation has to be a lower bound on the true sub-likelihood. This algorithm is promising for FPGA implementation because it can be combined with custom precision techniques; instead of using a function approximation for the dark data, a custom precision approximation can be used, leading to area savings. The lower bound requirement can be satisfied by using formal verification tools like Gappa++ [164]; Gappa++ can give the maximum precision-related error when the sub-likelihood is evaluated in a certain precision (vs. full precision). The maximum error can then be subtracted from the sub-likelihood expression to guarantee that the expression is a lower bound on the true (full precision) sub-likelihood. The algorithm's mixing behaviour when precision changes has to be investigated. It is possible that this algorithm scales better with large data sets compared to the WPT and MPPT methods of Chapter 3.

Moreover, some older but popular methods like Hamiltonian Monte Carlo, Slice sampling and Multiple-Try Metropolis are promising candidates for parallelization. Although some work exists in the literature on how to map these methods to GPUs, FPGAs have not been employed. It would be interesting to explore ways in which custom precision can be used within these methods. For example the computation of the gradient in Hamiltonian Monte Carlo helps the algorithm propose better samples and mix faster but it is not critical for accuracy in a similar way that auxiliary chains do not affect the accuracy of popMCMC. Therefore, custom precision could be used to compute the gradient and the effects on mixing could be investigated.

There are also several methods that do not strictly belong to the MCMC family but they are used in the same problems, share a lot of characteristics with MCMC and are amenable to hardware acceleration. One example is the recently proposed SMC^2 method [134], which is a competitor of pMCMC. Like pMCMC, SMC^2 uses a PF for state estimation. Nevertheless, instead of the MCMC algorithm that is used by pMCMC for parameter estimation, SMC^2 employs a second PF on top of the first one. This feature increases the available parallelism of the method from $O(P)$ to $O(P^2)$, which is ideal for massively parallel platforms. Another algorithm which offers potential for hardware acceleration is ABC [81]. The main idea of this method is to completely avoid likelihood computations when deciding whether to keep a proposed sample. Instead, ABC plugs the proposed sample to the model/likelihood and simulates data from it, i.e. it generates random data assuming that the unknown parameter of the model is equal to the proposed sample. This shifts the computational burden from computing the likelihood to generating random samples from a known model. Given that most Bayesian likelihoods can be decomposed into simple distributions (e.g. Gaussian distributions), this boils down to generating

random samples from these simple distributions. FPGAs are known for their efficiency in generating Gaussian and Uniform random numbers [5, 94]. This could prove a significant advantage when implementing ABC.

6.3.2 Extension to distributed and heterogeneous platforms

With the computational demands of modern Bayesian inference problems increasing rapidly, it is crucial that the various existing MCMC computational kernels are extended to run on distributed platforms and that an increased adoption of distributed MCMC methods is pursued. A good example is the pMCMC architecture of Chapter 4. The bottleneck computation in this architecture is the resampling operation, whose wall clock time does not drop significantly when larger device sizes are used. This scaling problem limits the applicability of the system to SSMs with massive amounts of particles. In order to be able to tackle these models, distributed resampling is the most promising approach [112]. Distributed resampling partitions the particle set into blocks and runs independent resampling operations for each block. Several ways have been proposed to avoid bias in the resampling results. Another case of distributed MCMC computation is the class of data partitioning methods mentioned previously [78, 77, 75]. Research on hardware accelerators needs to focus more on mapping methods like these to distributed platforms (multi-FPGA and multi-GPU systems, HPC), as well as extending other methods to be amenable to distributed computation. Issues that need to be addressed include communication between devices and correctness of results.

Moreover, the exploitation of heterogeneous computing techniques for MCMC is a completely unexplored research avenue. There are several opportunities in the MCMC toolset to optimally allocate tasks to different devices, e.g. use CPUs/GPUs for double precision computations and FPGAs for custom precision computations or allocate matrix computations (commonly required to evaluate likelihoods in Bayesian models) to different devices depending on the size and shape of the matrix. Adoption of cross-platform programming languages like OpenCL [165] could prove valuable in this setting.

6.3.3 Tools to improve accessibility of hardware for MCMC

As mentioned above, the adoption of hardware accelerators by the MCMC community has been limited. The main reason behind this slow adoption is the lack of user-friendly tools to allow practitioners

to use parallel hardware without having to write code. This comment also applies to the accelerators presented in this thesis. Although the generic modules of the popMCMC and pMCMC architectures are reusable, the probability density modules inside the system need to be written every time a new inference task is targeted.

Most of the available tools in the MCMC community are limited to using CPUs and do not provide any support for parallel computation. The most popular ones are WinBUGS [161], which uses Gibbs sampling and Slice sampling to tackle Bayesian hierarchical models, and Stan [166], which uses Hamiltonian Monte Carlo as its default sampler. The development of tools to automate the process of implementing MCMC in hardware is a relatively new idea. Several R packages provide parallel CPU support to R routines (including MCMC), namely the packages *parallel*, *snow*, *rparallel* and *gputools* [167]. These packages cannot bridge the gap between MCMC practitioners and parallel devices. The only MCMC package which explicitly focuses on parallel platforms (CPUs, GPUs) is the recent LibBi framework [2] which is designed for targeting SSMs and supports pMCMC and SMC².

A promising path towards creating hardware-based MCMC tools is to concentrate on a specific class of Bayesian models or a group of related classes and also on a group of MCMC methods that are suitable for these classes. This limits the space of possible problems that the user can address (i.e. the flexibility of the tool) but also allows the tool designer to build optimized compute kernels for this particular problem, which are easy to parameterize and can be reused. This is particularly relevant for FPGAs, since compilation of new bitstreams should be avoided due to long compile times. The above approach has been followed by LibBi [2] for SSMs-pMCMC but could also be applied to other combinations of models and methods, e.g. Gaussian process inference (where the main kernels perform linear algebra computations) combined with Metropolis-Hastings or Hamiltonian Monte Carlo.

In the case of FPGAs, a tool based on the above principles should provide the user with a domain-specific language in which the model of interest can be described. The exact form of the description depends on the model class. Based on the description, the model-specific parts of the system should be compiled automatically, possibly using floating point operator libraries such as FloPoCo [97]. These parts typically include the probability density evaluators, e.g. the probability density pipelines used in the PT architectures of Chapter 3. These evaluator blocks should then be plugged-in to the generic MCMC architecture. The tool should be able to detect parallel tasks and automatically map them to hardware using a library of reusable compute kernels. Some of the above features have already been embedded in the pMCMC accelerator of Chapter 4. A mature FPGA-based MCMC tool would be

able to automatically optimize the various system parameters, i.e. simultaneously optimize MCMC algorithm parameters, architecture parallelism and precision, given some information from the user.

An extra step towards making FPGA-based MCMC tools more accessible is to create an interface between FPGA synthesis software and one of the mainstream software tools used to perform MCMC-based inference (BUGS, Stan). These tools have a large user base which is in need of hardware acceleration capabilities.

6.3.4 Off-line precision optimization

As mentioned above, all precision optimization methods found in this thesis (in Chapters 3 and 5) are based on performing pre-runs in a set of candidate precisions and then choosing the “best” precision based on some criterion.

Nevertheless, an “off-line” precision optimization approach, i.e. optimization without performing pre-runs and collecting samples, could also be investigated in future work. This would greatly simplify the optimization process and would remove the need for generating multiple FPGA bitstreams. Nevertheless, off-line optimization is a challenging problem, since the decision about which precision to use has to be taken without using samples. A metric has to be devised that is easy to compute in software and can give useful information on the accuracy of each precision.

A possible solution could employ a formal verification tool like Gappa++ [164] to place an upper bound on the error of each precision compared to double precision when computing the posterior distribution. This upper bound would then be used to bound the error in the estimate (2.9) of integral (2.8) (i.e. the output estimate). This would allow for a similar bias tolerance criterion to be applied as in Chapter 5. Techniques to tighten the output bias bound (which is likely to be loose for complex densities) should be investigated.

For discrete state spaces, the results of [136] could also be used to connect the precision-related error (perturbation) in the transition matrix of the chain to some metric of divergence between the approximate and true target distributions (e.g. total variation). For continuous state spaces, similar results exist but they are difficult to apply due to computational issues. Ways to approximate the connection between the perturbation in the transition kernel and the divergence metric could be devised for these cases.

A third way to implement off-line precision optimization is to follow an approach like the one used by the ABC method [81]. In a similar way that ABC simulates data from the candidate model and compares them to the real data in order to accept or reject the model, an off-line precision optimization method could simulate data from the model in all candidate precisions (which does not require likelihood evaluations) and then compare the data to double precision data in order to decide which precisions provide adequate accuracy. A potential issue is the choice of the summary statistic which will be used to compare the data. Many statistics have been proposed [82] but no universally best recipe exists.

All the above approaches aim at controlling the *error* in the output when custom precision is used, i.e. they are similar to the approach of Chapter 5. Nevertheless, they could be extended to cover cases where the *mixing* of the sampler needs to be controlled and optimized (i.e. similar to the approach of Chapter 3). This would require analytical results on how precision is related to mixing for specific MCMC methods.

Bibliography

- [1] S. Liu, G. Mingas, and C.-S. Bouganis, “Parallel resampling for particle filters on FPGAs,” in *Field-Programmable Technology (FPT), 2014 International Conference on*, Dec 2014, pp. 191–198.
- [2] L. Murray, “Bayesian State-Space Modelling on High-Performance Hardware Using LibBi,” *Journal of Statistical Software*, vol. 67, no. 1, pp. 1–36, 2015.
- [3] 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*. New York, NY, USA: ACM, 2012, pp. 57–66.
- [4] C. P. Robert and G. Casella, *Monte Carlo statistical methods*. Springer Texts in Statistics, 2004, vol. 319.
- [5] D. B. Thomas, L. Howes, and W. Luk, “A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation,” in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’09. New York, NY, USA: ACM, 2009, pp. 63–72.
- [6] J. S. Liu, *Monte Carlo strategies in scientific computing*. Springer, 2001.
- [7] A. Jasra, D. A. Stephens, and C. C. Holmes, “On population-based simulation for static inference,” *Statistics and Computing*, pp. 263–279, 2007.
- [8] M. Tibbits, M. Haran, and J. Liechty, “Parallel multivariate slice sampling,” *Statistics and Computing*, vol. 21, no. 3, pp. 415–430, 2011.
- [9] *Handbook of Markov Chain Monte Carlo*, 1st ed. Chapman and Hall/CRC, May 2011.

- [10] L. Bottolo and S. Richardson, “Evolutionary stochastic search for Bayesian model exploration,” *Bayesian Analysis*, vol. 5, no. 3, pp. 583–618, 09 2010.
- [11] A. U. Asuncion, P. Smyth, and M. Welling, “Asynchronous distributed estimation of topic models for document analysis,” *Statistical Methodology - Advances in Data Mining and Statistical Learning*, vol. 8, no. 1, pp. 3 – 17, 2011.
- [12] G. W. Peters, G. R. Hosack, and K. R. Hayes, “Ecological non-linear state space model selection via adaptive particle Markov chain Monte Carlo (AdPMCMC),” *ArXiv e-prints 1005.2238*, May 2010.
- [13] L. Bottolo, M. Chadeau-Hyam, D. I. Hastie, T. Zeller, B. Lique, P. Newcombe, L. Yengo, P. S. Wild, A. Schillert, A. Ziegler, S. F. Nielsen, A. S. Butterworth, W. K. Ho, R. Castagn, T. Munzel, D. Tregouet, M. Falchi, F. Cambien, B. G. Nordestgaard, F. Fumeron, A. Tybjrg-Hansen, P. Froguel, J. Danesh, E. Petretto, S. Blankenberg, L. Tired, and S. Richardson, “GUESS-ing Polygenic Associations with Multiple Phenotypes Using a GPU-Based Evolutionary Stochastic Search Algorithm,” *PLoS Genet*, vol. 9, no. 8, p. e1003657, 08 2013.
- [14] O. J. L. Rackham et al., “WGBSSuite: simulating whole-genome bisulphite sequencing data and benchmarking differential DNA methylation analysis tools,” *Bioinformatics*, vol. 31(14), 2015.
- [15] Y. Li, M. Mascagni, and A. Gorin, “A decentralized parallel implementation for parallel tempering algorithm,” *Parallel Comput.*, vol. 35, pp. 269–283, May 2009.
- [16] A. Lee, C. Yau, M. B. Giles, A. Doucet, and C. C. Holmes, “On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods,” *Journal of Computational and Graphical Statistics*, vol. 19, no. 4, pp. 769–789, 2010.
- [17] I. Murray, “Advances in Markov chain Monte Carlo methods,” PhD thesis, Gatsby computational neuroscience unit, University College London, 2007.
- [18] 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*, ser. FPGA ’08. New York, NY, USA: ACM, 2008, pp. 203–211.
- [19] F. Belletti, M. Cotallo, A. Cruz, L. Fernandez, A. Gordillo-Guerrero, M. Guidetti, A. Maiorano, F. Mantovani, E. Marinari, V. Martin-Mayor, A. Muoz-Sudupe, D. Navarro, G. Parisi, S. Perez-

- Gaviro, M. Rossi, J. Ruiz-Lorenzo, S. Schifano, D. Sciretti, A. Tarancon, R. Tripiccion, J. Velasco, D. Yllanes, and G. Zanier, “Janus: An FPGA-Based System for High-Performance Scientific Computing,” *Computing in Science Engineering*, vol. 11, no. 1, pp. 48–58, 2009.
- [20] Z. Ghahramani, “Bayesian non-parametrics and the probabilistic approach to modelling,” *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 371, no. 1984, 2012.
- [21] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin, *Bayesian data analysis*. Taylor & Francis, 2014, vol. 2.
- [22] R. Salakhutdinov, A. Mnih, and G. Hinton, “Restricted Boltzmann Machines for Collaborative Filtering,” in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07. New York, NY, USA: ACM, 2007, pp. 791–798.
- [23] T. Flury and N. Shephard, “Bayesian inference based only on simulated likelihood: Particle filter analysis of dynamic economic models,” *Econometric Theory*, vol. 27, pp. 933–956, 10 2011.
- [24] J. Yan, M. Cowles, S. Wang, and M. Armstrong, “Parallelizing MCMC for Bayesian spatiotemporal geostatistical models,” *Statistics and Computing*, vol. 17, no. 4, pp. 323–335, 2007.
- [25] S. Zierke and J. Bakos, “FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods,” *BMC Bioinformatics*, vol. 11, no. 1, pp. 184+, 2010.
- [26] J. Gross, W. Janke, and M. Bachmann, “Massively parallelized replica-exchange simulations of polymers on GPUs,” *Computer Physics Communications*, vol. 182, no. 8, pp. 1638–1644, 2011.
- [27] D. J. Earl and M. W. Deem, “Parallel tempering: Theory, applications, and new perspectives,” *Phys. Chem. Chem. Phys.*, vol. 7, pp. 3910–3916, 2005.
- [28] J. G. Propp and D. B. Wilson, “Exact sampling with coupled Markov chains and applications to statistical mechanics,” *Random Structures and Algorithms*, vol. 9, no. 1-2, pp. 223–252, 1996.
- [29] A. Doucet and A. M. Johansen, “A tutorial on particle filtering and smoothing: Fifteen years later,” *Handbook of Nonlinear Filtering*, vol. 12, pp. 656–704, 2009.

- [30] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.
- [31] S. Geman and D. Geman, "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. PAMI-6, no. 6, pp. 721–741, nov. 1984.
- [32] R. E. Caflisch, "Monte Carlo and quasi-Monte Carlo methods," *Acta Numerica*, vol. 7, pp. 1–49, 1998.
- [33] C. Andrieu, A. Doucet, and R. Holenstein, "Particle Markov chain Monte Carlo methods," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 72, no. 3, pp. 269–342, 2010.
- [34] W. Gilks and D. Spiegelhalter, *Markov chain Monte Carlo in practice*. Chapman & Hall/CRC, 1996.
- [35] S. P. Brooks and A. Gelman, "General Methods for Monitoring Convergence of Iterative Simulations," *Journal of Computational and Graphical Statistics*, vol. 7, no. 4, pp. 434–455, 1998.
- [36] R. E. Kass, B. P. Carlin, A. Gelman, and R. M. Neal, "Markov Chain Monte Carlo in Practice: A Roundtable Discussion," *The American Statistician*, vol. 52, no. 2, pp. 93–100, 1998.
- [37] A. E. Gelfand and A. F. M. Smith, "Sampling-Based Approaches to Calculating Marginal Densities," *Journal of the American Statistical Association*, vol. 85, no. 410, pp. 398–409, 1990.
- [38] C. Robert and G. Casella, "A Short History of Markov Chain Monte Carlo: Subjective Recollections from Incomplete Data," *Statist. Sci.*, vol. 26, no. 1, pp. 102–115, 02 2011.
- [39] W. K. Hastings, "Monte Carlo Sampling Methods Using Markov Chains and Their Applications," *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.
- [40] G. McLachlan and D. Peel, *Finite mixture models*. Wiley Series in Probability and Statistics, 2004.
- [41] C. J. Geyer, "Markov Chain Monte Carlo Maximum Likelihood," in *Computing Science and Statistics, Proceedings of the 23rd Symposium on the Interface*, 1991, pp. 156–163.

- [42] J. Owen, D. Wilkinson, and C. Gillespie, “Scalable inference for markov processes with intractable likelihoods,” *Statistics and Computing*, vol. 25, no. 1, pp. 145–156, 2015.
- [43] R. G. Everitt, “Bayesian parameter estimation for latent markov random fields and social networks,” *Journal of Computational and Graphical Statistics*, vol. 21, no. 4, pp. 940–960, 2012.
- [44] M. A. Beaumont, “Estimation of Population Growth or Decline in Genetically Monitored Populations,” *Genetics*, vol. 164, no. 3, pp. 1139–1160, 2003.
- [45] C. Andrieu and G. O. Roberts, “The pseudo-marginal approach for efficient Monte Carlo computations,” *Ann. Statist.*, vol. 37, no. 2, pp. 697–725, 04 2009.
- [46] P. E. Jacob and A. H. Thiery, “On nonnegative unbiased estimators,” *The Annals of Statistics*, vol. 43, no. 2, pp. 769–784, 04 2015.
- [47] I. Nevat, G. W. Peters, A. Doucet, and J. Yuan, “Channel Tracking for Relay Networks via Adaptive Particle MCMC,” *ArXiv e-prints 1006.3151*, Jun. 2010.
- [48] M. Dowd, E. Jones, and J. Parslow, “A statistical overview and perspectives on data assimilation for marine biogeochemical models,” *Environmetrics*, vol. 25, no. 4, pp. 203–213, 2014.
- [49] R. E. Kalman, “A new approach to linear filtering and prediction problems,” *Journal of Fluids Engineering*, vol. 82, no. 1, pp. 35–45, 1960.
- [50] N. Kantas, A. Doucet, S. S. Singh, J. M. Maciejowski, and N. Chopin, “On Particle Methods for Parameter Estimation in State-Space Models,” *ArXiv e-prints 1412.8695*, Dec. 2014.
- [51] O. Cappe, S. Godsill, and E. Moulines, “An Overview of Existing Methods and Recent Advances in Sequential Monte Carlo,” *Proceedings of the IEEE*, vol. 95, no. 5, pp. 899–924, May 2007.
- [52] N. Gordon, D. Salmond, and A. Smith, “Novel approach to nonlinear/non-Gaussian Bayesian state estimation,” *Radar and Signal Processing, IEE Proceedings F*, vol. 140, no. 2, pp. 107–113, Apr 1993.
- [53] R. Douc and O. Cappe, “Comparison of resampling schemes for particle filtering,” in *Image and Signal Processing and Analysis, 2005. ISPA 2005. Proceedings of the 4th International Symposium on*, Sept 2005, pp. 64–69.

- [54] L. M. Murray, A. Lee, and P. E. Jacob, "Parallel resampling in the particle filter," *Journal of Computational and Graphical Statistics*, vol. 0, no. ja, pp. 0–0, 0.
- [55] R. Neal, "Slice Sampling," *Annals of Statistics*, vol. 31, pp. 705–767, 2000.
- [56] J. S. Liu, F. Liang, and W. H. Wong, "The Multiple-Try Method and Local Optimization in Metropolis Sampling," *Journal of the American Statistical Association*, vol. 95, no. 449, pp. pp. 121–134, 2000.
- [57] S. Duane, A. Kennedy, B. J. Pendleton, and D. Roweth, "Hybrid Monte Carlo," *Physics Letters B*, vol. 195, no. 2, pp. 216 – 222, 1987.
- [58] M. Girolami and B. Calderhead, "Riemann manifold Langevin and Hamiltonian Monte Carlo methods," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 73, no. 2, pp. 123–214, 2011.
- [59] P. J. Green, "Reversible Jump Markov Chain Monte Carlo Computation and Bayesian Model Determination," *Biometrika*, vol. 82, no. 4, pp. pp. 711–732, 1995.
- [60] A. Jasra, C. C. Holmes, and D. A. Stephens, "Markov chain monte carlo methods and the label switching problem in bayesian mixture modeling," *Statistical Science*, vol. 20, no. 1, pp. pp. 50–67, 2005.
- [61] C. Andrieu and J. Thoms, "A tutorial on adaptive MCMC," *Statistics and Computing*, vol. 18, pp. 343–373, 2008.
- [62] D. Maclaurin and R. P. Adams, "Firefly Monte Carlo: Exact MCMC with Subsets of Data," *ArXiv e-prints 1403.5693*, Mar. 2014.
- [63] N. Alachiotis, E. Sotiriades, A. Dollas, and A. Stamatakis, "Exploring FPGAs for accelerating the phylogenetic likelihood function," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, may 2009, pp. 1 –8.
- [64] M. Whitley and S. Wilson, "Parallel algorithms for Markov chain Monte Carlo methods in latent spatial Gaussian models," *Statistics and Computing*, vol. 14, no. 3, pp. 171–179, 2004.
- [65] I. Lebedev, C. Fletcher, S. Cheng, J. Martin, A. Doupnik, D. Burke, M. Lin, and J. Wawrzynek, "Exploring Many-Core Design Templates for FPGAs and ASICs," *International Journal of Reconfigurable Computing*, Article ID 439141, 15 pages, vol. 2012, 2012.

- [66] M. Suchard, Q. Wang, C. Chan, J. Frelinger, A. Cron, and M. West, “Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures,” *Journal of Computational and Graphical Statistics*, vol. 19, no. 2, p. 419438, 2010.
- [67] W. Zhu, A. Yaseen, and Y. Li, “DECMCMC-GPU: An Efficient Multi-Objective Optimization Method with GPU Acceleration on the Fermi Architecture,” *New Generation Computing*, vol. 29, no. 2, pp. 163–184, 2011.
- [68] “Assessing the Reliability of Complex Models: Mathematical and Statistical Foundations of Verification, Validation, and Uncertainty Quantification,” *The National Academies Press*, 2012.
- [69] M. L. Stein, J. Chen, and M. Anitescu, “Stochastic approximation of score functions for Gaussian processes,” *Ann. Appl. Stat.*, vol. 7, no. 2, pp. 1162–1191, 06 2013.
- [70] J. Rougier, S. Guillas, A. Maute, and A. D. Richmond, “Expert Knowledge and Multivariate Emulation: The ThermosphereIonosphere Electrodynamics General Circulation Model (TIE-GCM),” *Technometrics*, vol. 51, no. 4, pp. 414–424, 2009.
- [71] P. Torma, A. György, and C. Szepesvári, “A Markov-Chain Monte Carlo Approach to Simultaneous Localization and Mapping,” in *International Conference on Artificial Intelligence and Statistics*, 2010, pp. 852–859.
- [72] G. Hendeby, R. Karlsson, and F. Gustafsson, “Particle Filtering: The Need for Speed,” *EURASIP Journal on Advances in Signal Processing*, vol. 2010, no. 1, p. 181403, 2010.
- [73] J. Brown and D. Capson, “A Framework for 3D Model-Based Visual Tracking Using a GPU-Accelerated Particle Filter,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, no. 1, pp. 68–80, Jan 2012.
- [74] S. Sutharsan, T. Kirubarajan, T. Lang, and M. McDonald, “An Optimization-Based Parallel Particle Filter for Multitarget Tracking,” *Aerospace and Electronic Systems, IEEE Transactions on*, vol. 48, no. 2, pp. 1601–1618, APRIL 2012.
- [75] S. L. Scott, A. W. Blocker, and F. V. Bonassi, “Bayes and Big Data: The Consensus Monte Carlo Algorithm,” in *Bayes 250*, 2013.
- [76] W. Neiswanger, C. Wang, and E. Xing, “Asymptotically Exact, Embarrassingly Parallel MCMC,” *ArXiv e-prints 1311.4780*, Nov. 2013.

- [77] S. Minsker, S. Srivastava, L. Lin, and D. Dunson, “Scalable and Robust Bayesian Inference via the Median Posterior,” in *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, T. Jebara and E. P. Xing, Eds. JMLR Workshop and Conference Proceedings, 2014, pp. 1656–1664.
- [78] X. Wang and D. B. Dunson, “Parallelizing MCMC via Weierstrass Sampler,” *ArXiv e-prints 1312.4605*, Dec. 2013.
- [79] A. Korattikara, Y. Chen, and M. Welling, “Austerity in MCMC Land: Cutting the Metropolis-Hastings Budget,” *ArXiv e-prints 1304.5299*, Apr. 2013.
- [80] 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)*, T. Jebara and E. P. Xing, Eds. JMLR Workshop and Conference Proceedings, 2014, pp. 405–413.
- [81] T. Toni, D. Welch, N. Strelkowa, A. Ipsen, and M. P. Stumpf, “Approximate Bayesian computation scheme for parameter inference and model selection in dynamical systems,” *Journal of The Royal Society Interface*, vol. 6, no. 31, pp. 187–202, 2009.
- [82] P. Fearnhead and D. Prangle, “Constructing summary statistics for approximate Bayesian computation: semi-automatic approximate Bayesian computation,” *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 74, no. 3, pp. 419–474, 2012.
- [83] H. Rue, S. Martino, and N. Chopin, “Approximate Bayesian inference for latent Gaussian models by using integrated nested Laplace approximations,” *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 71, no. 2, pp. 319–392, 2009.
- [84] M. Jordan, Z. Ghahramani, T. Jaakkola, and L. Saul, “An Introduction to Variational Methods for Graphical Models,” *Machine Learning*, vol. 37, no. 2, pp. 183–233, 1999.
- [85] J. v. Newmann, “First Draft of a Report on the EDVAC,” Tech. Rep., 1945.
- [86] G. E. Moore, “Cramming More Components onto Integrated Circuits,” *Electronics*, vol. 38, no. 8, pp. 114–117, Apr. 1965.
- [87] Ross, Philip E., “Why CPU Frequency Stalled,” 2008. [Online]. Available: <http://spectrum.ieee.org/computing/hardware/why-cpu-frequency-stalled>

- [88] Flynn, Laurie J., "Intel halts development of 2 new microprocessors," 2004. [Online]. Available: <http://www.nytimes.com/2004/05/08/business/08chip.html?ex=1399348800&en=98cc44ca97b1a562&ei=5007>
- [89] T. J. Todman, G. A. Constantinides, S. J. E. Wilton, O. Mencer, W. Luk, and P. Y. K. Cheung, "Reconfigurable computing: architectures and design methods," in *IEEE Proceedings - Computers and Digital Techniques*, 2005, pp. 193–207.
- [90] Altera, "The industrys first floating-point fpga," 2014. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/po/bg-floating-point-fpga.pdf
- [91] K. Vipin, S. Shreejith, D. Gunasekera, S. A. Fahmy, and N. Kapre, "System-level FPGA device driver with high-level synthesis support," in *Field-Programmable Technology (FPT), 2013 International Conference on*, Dec 2013, pp. 128–135.
- [92] A. Rafique, N. Kapre, and G. Constantinides, "Enhancing performance of Tall-Skinny QR factorization using FPGAs," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, Aug 2012, pp. 443–450.
- [93] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, 2011.
- [94] D. Thomas, "FPGA Gaussian Random Number Generators with Guaranteed Statistical Accuracy," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, May 2014, pp. 149–156.
- [95] B. Betkaoui, D. Thomas, and W. Luk, "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing," in *Field-Programmable Technology (FPT), 2010 International Conference on*, 2010, pp. 94–101.
- [96] Nvidia, "New Features in CUDA 7.5," 2015. [Online]. Available: <http://devblogs.nvidia.com/parallelforall/new-features-cuda-7-5>
- [97] F. de Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," *IEEE Design and Test of Computers*, vol. 28, pp. 18–27, 2011.

- [98] J. Detrey and F. de Dinechin, "Parameterized floating-point logarithm and exponential functions for FPGAs," *Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing*, vol. 31, no. 8, pp. 537–545, 2007.
- [99] C. P. R. Olivier Cappe, "Markov chain monte carlo: 10 years and still running!" *Journal of the American Statistical Association*, vol. 95, no. 452, pp. 1282–1286, 2000.
- [100] L. Murray, "Distributed Markov chain Monte Carlo," *Proceedings of Neural Information Processing Systems, Workshop on Learning on Cores, Clusters and Clouds*, vol. 11, 2010.
- [101] J. S. Rosenthal, "Parallel computing and Monte Carlo algorithms," *Far East Journal of Theoretical Statistics*, vol. 4, pp. 207–236, 1999.
- [102] D. J. Wilkinson, "Parallel Bayesian Computation," *Statistics Textbooks and Monographs*, vol. 184, p. 477, 2006.
- [103] V. Gopal and G. Casella, "Running Regenerative Markov Chains in Parallel," 2011.
- [104] A. L. Beam, S. K. Ghosh, and J. Doyle, "Fast Hamiltonian Monte Carlo Using GPU Computing," *Journal of Computational and Graphical Statistics*, vol. 0, no. ja, pp. 00–00, 0.
- [105] J. Byrd, S. Jarvis, and A. Bhalerao, "Reducing the run-time of MCMC programs by multi-threading on SMP architectures," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1 –8.
- [106] I. Strid, "Efficient parallelisation of MetropolisHastings algorithms using a prefetching approach," *Computational Statistics and Data Analysis*, vol. 54, no. 11, pp. 2814 – 2835, 2010.
- [107] E. Angelino, E. Kohler, A. Waterland, M. Seltzer, and R. P. Adams, "Accelerating MCMC via Parallel Predictive Prefetching," *ArXiv e-prints 1403.7265*, Mar. 2014.
- [108] V. K. Mansinghka, E. M. Jonas, and J. B. Tenenbaum, "Stochastic Digital Circuits for Probabilistic Inference," Massachusetts Institute of Technology, Technical Report MIT-CSAIL-TR-2008-069, 2008.
- [109] D. J. Earl and M. W. Deem, "Optimal Allocation of Replicas to Processors in Parallel Tempering Simulations," *The Journal of Physical Chemistry B*, vol. 108, no. 21, pp. 6844–6849, 2004.

- [110] M. Bolic, "Architectures for Efficient Implementation of Particle Filters," Ph.D. dissertation, Stony Brook, NY, USA, 2004, aAI3149104.
- [111] A. Athalye, M. Bolic, S. Hong, and P. M. Djuric, "Generic Hardware Architectures for Sampling and Resampling in Particle Filters," *EURASIP Journal on Advances in Signal Processing*, vol. 2005, no. 17, p. 476167, 2005.
- [112] M. Bolic, P. Djuric, and S. Hong, "Resampling algorithms and architectures for distributed particle filters," *Signal Processing, IEEE Transactions on*, vol. 53, no. 7, pp. 2442–2450, July 2005.
- [113] S. Saha, N. Bambha, and S. Bhattacharyya, "Parameterized design framework for hardware implementation of particle filters," in *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, March 2008, pp. 1449–1452.
- [114] M. Happe, E. Lubbers, and M. Platzner, "An adaptive Sequential Monte Carlo framework with runtime HW/SW repartitioning," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, Dec 2009, pp. 175–182.
- [115] T. C. Chau, M. Kurek, J. S. Targett, J. Humphrey, G. Skouroupathis, A. Eele, J. Maciejowski, B. Cope, K. Cobden, P. Leong, P. Y. Cheung, and W. Luk, "SMCGen: Generating Reconfigurable Design for Sequential Monte Carlo Applications," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, May 2014, pp. 141–148.
- [116] T. Chau, W. Luk, P. Cheung, A. Eele, and J. Maciejowski, "Adaptive Sequential Monte Carlo approach for real-time applications," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, Aug 2012, pp. 527–530.
- [117] B. Ye and Y. Zhang, "Improved FPGA implementation of particle filter for radar tracking applications," in *Synthetic Aperture Radar, 2009. APSAR 2009. 2nd Asian-Pacific Conference on*, Oct 2009, pp. 943–946.
- [118] J. U. Cho, S. H. Jin, X. D. Pham, J. W. Jeon, J. E. Byun, and H. Kang, "A Real-Time Object Tracking System Using a Particle Filter," in *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, Oct 2006, pp. 2822–2827.

- [119] J. Mountney, I. Obeid, and D. Silage, "Modular particle filtering FPGA hardware architecture for brain machine interfaces," in *Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE*, Aug 2011, pp. 4617–4620.
- [120] K. Hwang and W. Sung, "Load Balanced Resampling for Real-Time Particle Filtering on Graphics Processing Units," *Signal Processing, IEEE Transactions on*, vol. 61, no. 2, pp. 411–419, Jan 2013.
- [121] M.-A. Chao, C.-Y. Chu, C.-H. Chao, and A.-Y. Wu, "Efficient parallelized particle filter design on CUDA," in *Signal Processing Systems (SIPS), 2010 IEEE Workshop on*, Oct 2010, pp. 299–304.
- [122] M. Chitchian, A. Simonetto, A. S. van Amesfoort, and T. Keviczky, "Distributed Computation Particle Filters on GPU-architectures for Real-time Control Applications," *IEEE Transactions on Control Systems Technology*, vol. 21, no. 6, pp. 2224–2238, November 2013.
- [123] M. Chitchian, A. van Amesfoort, A. Simonetto, T. Keviczky, and H. Sips, "Adapting Particle Filter Algorithms to Many-Core Architectures," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, May 2013, pp. 427–438.
- [124] K. Par and O. Tosun, "Parallelization of particle filter based localization and map matching algorithms on multicore/manycore architectures," in *Intelligent Vehicles Symposium (IV), 2011 IEEE*, June 2011, pp. 820–826.
- [125] R. Cabido, A. Montemayor, and J. Pantrigo, "High performance memetic algorithm particle filter for multiple object tracking on modern GPUs," *Soft Computing*, vol. 16, no. 2, pp. 217–230, 2012.
- [126] A. Gelencsr-Horvth, G. Tornai, A. Horvth, and G. Cserey, "Fast, parallel implementation of particle filtering on the GPU architecture," *EURASIP Journal on Advances in Signal Processing*, vol. 2013, no. 1, 2013.
- [127] O. Mateo Lozano and K. Otsuka, "Real-time Visual Tracker by Stream Processing," *Journal of Signal Processing Systems*, vol. 57, no. 2, pp. 285–295, 2009.
- [128] O. Rosen, A. Medvedev, and M. Ekman, "Speedup and tracking accuracy evaluation of parallel particle filter algorithms implemented on a multicore architecture," in *Control Applications (CCA), 2010 IEEE International Conference on*, Sept 2010, pp. 440–445.

- [129] S. Henriksen, A. G. Wills, T. B. Schon, and B. Ninness, "Parallel Implementation of Particle MCMC Methods on a GPU," *System Identification, 16th IFAC Symposium on*, pp. 1143–1148, 2012.
- [130] M. K. Pitt, R. dos Santos Silva, P. Giordani, and R. Kohn, "On some properties of Markov chain Monte Carlo simulation methods based on the particle filter," *Journal of Econometrics*, vol. 171, no. 2, pp. 134 – 151, 2012, bayesian Models, Methods and Applications.
- [131] A. Doucet, M. K. Pitt, G. Deligiannidis, and R. Kohn, "Efficient implementation of Markov chain Monte Carlo when using an unbiased likelihood estimator," *Biometrika*, 2015.
- [132] C. Sherlock, A. H. Thiery, G. O. Roberts, and J. S. Rosenthal, "On the efficiency of pseudo-marginal random walk Metropolis algorithms," *The Annals of Statistics*, vol. 43, no. 1, pp. 238–275, 02 2015.
- [133] J.-C. Duan and A. Fulop, "Density-Tempered Marginalized Sequential Monte Carlo Samplers," *Journal of Business & Economic Statistics*, vol. 0, no. ja, pp. 00–00, 2014.
- [134] N. Chopin, P. E. Jacob, and O. Papaspiliopoulos, "SMC2: an efficient algorithm for sequential analysis of state space models," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 75, no. 3, pp. 397–426, 2013.
- [135] L. Breyer, G. O. Roberts, and J. S. Rosenthal, "A note on geometric ergodicity and floating-point roundoff error," *Statistics & Probability Letters*, vol. 53, no. 2, pp. 123–127, June 2001.
- [136] A. Y. Mitrophanov, "Sensitivity and Convergence of Uniformly Ergodic Markov Chains," *Journal of Applied Probability*, vol. 42, no. 4, pp. pp. 1003–1014, 2005.
- [137] X. Tian and C.-S. Bouganis, "A Run-Time Adaptive FPGA Architecture for Monte Carlo Simulations," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, September 2011, pp. 116 –122.
- [138] M. Fielding, D. J. Nott, and S.-Y. Liong, "Efficient MCMC Schemes for Computationally Expensive Posterior Distributions," *Technometrics*, vol. 53, no. 1, pp. 16–28, 2011.
- [139] R. Gramacy, R. Samworth, and R. King, "Importance tempering," *Statistics and Computing*, vol. 20, pp. 1–7, 2010.
- [140] "Intel Cilk Plus," <http://software.intel.com/en-us/intel-cilk-plus>.

- [141] “Intel C and C++ Compilers,” <http://software.intel.com/en-us/c-compilers>.
- [142] “Optimizing Parallel Reduction in CUDA,” <http://docs.nvidia.com/cuda/cuda-samples/index.html>.
- [143] M. Jacobsen, Y. Freund, and R. Kastner, “RIFFA: A Reusable Integration Framework for FPGA Accelerators,” in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, 2012, pp. 216–219.
- [144] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, “Analyzing CUDA workloads using a detailed GPU simulator,” in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, 2009, pp. 163–174.
- [145] Nvidia, “Nvidia CUDA Compiler Driver NVCC.” [Online]. Available: <http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc>
- [146] “Xilinx Power Estimator,” <http://www.xilinx.com/products/technology/power/xpe.html>.
- [147] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélicier, and P. Zimmermann, “MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding,” *ACM Transactions on Mathematical Software*, vol. 33, no. 2, pp. 13:1–13:15, Jun. 2007.
- [148] Y. F. Atchadé, G. O. Roberts, and J. S. Rosenthal, “Towards optimal scaling of metropolis-coupled Markov chain Monte Carlo,” *Statistics and Computing*, vol. 21, no. 4, pp. 555–568, 2011.
- [149] A. B. Owen, *Monte Carlo theory, methods and examples*, 2013.
- [150] “Maxeler MAX3 Card,” <http://www.maxeler.com/content/hardware>.
- [151] J. Jensen, “Sur les fonctions convexes et les inégalités entre les valeurs moyennes,” *Acta Mathematica*, vol. 30, no. 1, pp. 175–193, 1906.
- [152] K. D. Robertson, “DNA methylation and human disease,” *Nature Reviews Genetics*, vol. 6, no. 8, pp. 597–610, 2005.
- [153] A. Suardi, E. C. Kerrigan, and G. A. Constantinides, “Fast FPGA prototyping toolbox for embedded optimization,” in *Control Conference (ECC), 2015 European*. IEEE, 2015, pp. 2589–2594.

- [154] P. L'Ecuyer, "Maximally Equidistributed Combined Tausworthe Generators," *Mathematics of Computation*, vol. 65, no. 213, pp. pp. 203–213, 1996.
- [155] D. B. Thomas, W. Luk, P. H. Leong, and J. D. Villasenor, "Gaussian Random Number Generators," *ACM Comput. Surv.*, vol. 39, no. 4, Nov. 2007.
- [156] C. J. Geyer, "Practical Markov Chain Monte Carlo," *Statistical Science*, vol. 7, no. 4, pp. pp. 473–483, 1992.
- [157] "Wikipedia article: Data science," https://en.wikipedia.org/wiki/Data_science.
- [158] N. Matloff, *Parallel Computing for Data Science: With Examples in R, C++ and CUDA*, ser. Chapman & Hall/CRC The R Series. Taylor & Francis, 2015.
- [159] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf," *Procedia Computer Science*, vol. 53, pp. 121 – 130, 2015, {INNS} Conference on Big Data 2015 Program San Francisco, CA, {USA} 8-10 August 2015.
- [160] "Wikipedia article: DNA methylation," http://en.wikipedia.org/wiki/DNA_methylation.
- [161] D. Lunn, D. Spiegelhalter, A. Thomas, and N. Best, "The BUGS project: Evolution, critique and future directions," *Statistics in Medicine*, vol. 28, no. 25, pp. 3049–3067, 2009.
- [162] Clark, Don, "Intel Completes Acquisition of Altera," 2015. [Online]. Available: <http://www.wsj.com/articles/intel-completes-acquisition-of-altera-1451338307>
- [163] P. E. Jacob, L. M. Murray, and S. Rubenthaler, "Path storage in the particle filter," *Statistics and Computing*, vol. 25, no. 2, pp. 487–496, 2013.
- [164] 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 *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 230–237.
- [165] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010.
- [166] Stan Development Team, "Stan: A C++ Library for Probability and Sampling, Version 2.5.0," 2014. [Online]. Available: <http://mc-stan.org/>

- [167] “The Comprehensive R Archive Network,” <http://cran.r-project.org>, <http://cran.r-project.org>.