AcMC^2: Accelerated Markov Chain Monte Carlo for Probabilistic Models

Subho S. Banerjee
University of Illinois at Urbana-Champaign
ssbaner2@illinois.edu

Zbigniew T. Kalbarczyk
University of Illinois at Urbana-Champaign
kalbarcz@illinois.edu

Ravishankar K. Iyer
University of Illinois at Urbana-Champaign
rkiyer@illinois.edu

Abstract

Probabilistic models (PMs) are ubiquitously used across a variety of machine learning applications. They have been shown to successfully integrate structural prior information about data and effectively quantify uncertainty to enable the development of more powerful, interpretable, and efficient learning algorithms. This paper presents AcMC^2, a compiler that transforms PMs into optimized hardware accelerators (for use in FPGAs or ASICs) that utilize Markov chain Monte Carlo methods to infer and query a distribution of posterior samples from the model. The compiler analyzes statistical dependencies in the PM to drive several optimizations to maximally exploit the parallelism and data locality available in the problem. We demonstrate the use of AcMC^2 to implement several learning and inference tasks on a Xilinx Virtex-7 FPGA. AcMC^2-generated accelerators provide a $47-100\times$ improvement in runtime performance over a 6-core IBM Power8 CPU and a $8-18\times$ improvement over an NVIDIA K80 GPU. This corresponds to a $753-1600\times$ improvement over the CPU and $248-463\times$ over the GPU in performance-per-watt terms.

CCS Concepts • **Computer systems organization → Parallel architectures**; • **Hardware → Hardware accelerators**; • **Software and its engineering → Compilers**; **Domain specific languages**.

Keywords Accelerator, Markov Chain Monte Carlo, Probabilistic Graphical Models, Probabilistic Programming

ACM Reference Format:
Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2019. AcMC^2: Accelerated Markov Chain Monte Carlo for Probabilistic Models. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304019

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304019

1 Introduction

Many statistical- and machine-learning (ML) applications automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision-making under uncertainty. Probabilistic models (PMs; e.g., Markov models or Bayesian networks) and inference techniques have been shown to successfully integrate prior and structural relationships to quantify this uncertainty [38]. This allows PMs to naturally complement many ML methods (like deep learning [26]; DL) that (1) do not quantify uncertainty in their outputs [23], (2) seldom produce interpretable results, and (3) do not generalize well from small datasets or in cases with *class imbalance*. In fact, there are ongoing efforts in the ML community to combine PMs and DL to produce a *Bayesian DL* paradigm that can take advantage of both the flexibility of PMs in encoding model-related information (e.g., uncertainty, interpretability) with the immense scalability of DL [26].

Creation of optimized accelerators for DL models is well-developed [1, 11, 35]. The creation of accelerators that can execute infernence on PMs in real-time is substantially nonexistent, or is done only on a very problem-specific, hand-optimized basis [6, 7, 15, 32, 34, 36, 42, 49]. Development of such accelerators is the focus of this paper. They will be fundamental not just to the addressing of PMs, but also to the integration PMs and DL.

Development of accelerators for execution of inference on PMs requires (1) a high-level language representation of PMs, and (2) a method to map this representation into an architecture and correspondingly synthesized hardware that meets the real-time constraints. To address (1) above, we leverage prior work that proposes *probabilistic programming languages* (PPLs) [28] as a way to represent complex PMs as programs (e.g., [13, 27, 30, 33, 44, 48, 56, 66]).

This paper addresses (2) above by proposing AcMC^2, a compiler that transforms general PMs expressed in a PPL into optimized hardware accelerators to infer query distributions (i.e., quantities of interest) over the posterior samples of a PM. Inference over PMs is analytically intractable in general [16]; therefore, we focus on methods that compute approximate answers, in particular the sampling-based Markov-Chain Monte Carlo (MCMC) methods. The crux of our approach is three fold. (1) We identify and accelerate common *computational kernels* used across multiple models. In the case of MCMC-based inference, that corresponds to the use of multiple types of random number generators. (2) We use marginal and conditional independences to maximally exploit

the parallelism and data locality available in the structure of a PM for its inference. (3) We integrate the above pieces with compositional MCMC techniques, i.e., where different variants of basic MCMC algorithms can be integrated together to solve an inference problem.[1] AcMC^2 then automatically generates HDL that corresponds to system-on-chip (SoC) components that can be integrated into CPU-based SoCs, FPGAs, or ASICs, which can then be used in both large servers and embedded devices.

Contributions. Our primary contributions are

1. We present a compiler workflow for generating hardware accelerators (both their architecture and implementation) from PMs described in PPLs. The compiler uses:
   a. Conditional statistical independences (captured using *Markov blankets*) in a PM to generate maximally parallel, deeply pipelined, problem-specific random number generators.
   b. Speculative execution to execute several independent MCMC chains in parallel on the generators above.
   c. Bounded approximation techniques that reduce off-chip bandwidth for storage of intermediate results.
2. We describe an FPGA-based prototype (on a Xilinx Virtex 7 FPGA) for AcMC^2-generated accelerators that communicate to host CPUs (IBM POWER8) by using the CAPI interface [58].
3. We demonstrate AcMC^2's performance using a set of PPL micro-benchmarks. The AcMC^2-generated accelerators provided an average 46.8× improvement in runtime and a 753.5× improvement in terms of performance-per-watt over CPU-based software implementations.
4. We illustrate the generality of AcMC^2 by solving two real-world problems that require real-time analytics.
   a. *Precision Medicine:* Identification of seizure-generating brain regions through analysis of electroencephalograms (EEGs) [63].
   b. *Datacenter Security:* Detecting data-center-scale security incidents by analysis of alerts generated from network- and host-level security monitoring tools [12].

   We demonstrate a $48.4 - 102.1\times$ improvement in performance over a CPU baseline; and a $8.6-18.2\times$ improvement in performance over a NVIDIA K80 GPU.

Placing AcMC^2 in Perspective. Traditional methods have been unsuccessful at addressing the challenge of accelerating the execution of inference in PMs. (1) Optimizing compilers and high-level synthesis engines [46] (HLS; C/C++ to HDL compilers) have used control and data dependences in programs to drive parallelism and SIMD optimizations [2, 50]. That approach is inherently limited because it analyzes only the inference procedure (i.e., the program that is executed), and not the dependence (and hence the parallelism) available in the PM. (2) General-purpose accelerators like GPUs have limited success with MCMC algorithms that are inherently sequential (i.e., compute as a chain of steps) and present significant *branch divergence* across multiple chains. (3) The use of domain-specific languages (DSLs) to describe parallel patterns [37, 53] that generate efficient code/accelerators have not shown much promise, as they do not offer the abstractions required to easily represent PMs. As a result, accelerated applications for PMs have generally required manual optimization on a problem-by-problem basis, e.g., [6, 7, 15, 32, 34, 36, 42, 49]. In contrast, AcMC^2 effectively analyzes statistical properties of the PM at compile time and is able to achieve significant parallelism that the traditional methods described above are not designed to accomplish.

[1] For example, we use *Gibbs sampling* [24] for discrete variables, and *Hamiltonian Monte Carlo* (HMC) [51] for continuous variables when gradient information is available.

2 Background

2.1 Bayesian Modeling

AcMC^2 considers PMs with joint distribution factorization

$$p(\theta, x_D) = p(\theta)p(x_D|\theta), \tag{1}$$

where $p(\theta)$ is a distribution over parameters $\theta = \{\theta_1, \ldots, \theta_m\}$ called the *prior*, and $p(x_D|\theta)$ is the conditional distribution of the dataset $x_D = \{[x_{0,0}, \ldots, x_{n,0}], \ldots, [x_{0,D}, \ldots, x_{n,D}]\}$ given the parameters θ. Given observed data point $x = [x_0, \ldots, x_n]$, the goal of an AcMC^2 accelerator is to compute the posterior distribution,

$$p(\theta|x) = \frac{p(\theta, x)}{p(x)} = \frac{p(\theta)p(x|\theta)}{\int p(\theta)p(x|\theta)d\theta}. \tag{2}$$

A user can then query this distribution (called an *inference*) to obtain required information about the model/data.

For example, consider the use of a commonly used PM to solve the problem of clustering a set of points into K clusters. Here, the PM models how we believe the observations are generated. For instance, one explanation might be 1. that there are K cluster centers chosen randomly according to some distribution, and 2. that each data point (independent of all other data points) is normally distributed around a randomly chosen cluster center. That explanation describes what is known as a *Gaussian mixture model (GMM)*. We can then query this model to ask, "What is the number of clusters for a given dataset under this model?" or "What is the most likely cluster assignment for a data point under the model?"[2]

Application characteristics often bring additional latent structure to the density factorization shown in (1). In the example above, the explanation of the generative process defines this latent structure. Prior work at the conjunction of graph theory and probability theory has developed a powerful formalism called *factor graphs (FGs)* [38]. An FG factorizes (1) into C sets of dependent variables:

$$p(x_1, \ldots, x_n, \theta_1, \ldots, \theta_m) = \frac{\prod_{c \in C} f_c(x_c)}{\int \cdots \int \prod_{c \in C} f_c(x_c) d\theta_1 \ldots d\theta_m},$$

where we use the shorthand $x_c = \{x_i | i \in C\}$, and f_c represents *factor functions* describing the statistical relationships between different x_cs. Overall, FGs provide an intuitive and compact representation to parse out the independences.

[2] Fig. 2 shows the AcMC^2 workflow of the model described above.

Algorithm 1: Generic Hastings sampler.

Input : Initial distribution D,
Proposal distribution q,
Number of samples N,
Number of burn-in samples b
Output : Samples from target distribution p

1 Initialize $X_0 = \{X_0^1, \ldots, X_0^n\}$ from some distribution D
2 **for** $i \leftarrow [1, N]$ **do**
3 Generate $X \sim q(X|X_{i-1})$
4 Generate α such that
$$\alpha = \min\left\{1, \frac{p(X)q(X_{i-1}|X)}{p(X_{i-1})q(X|X_{i-1})}\right\}$$
5 $$X_i \leftarrow \begin{cases} X & \text{with probability } \alpha \\ X_{i-1} & \text{with probability } 1-\alpha \end{cases}$$
6 **return** $(X_{b+1}, \ldots, X_N)$

Inference. Probabilistic inference is the task of deriving the probability that one or more random variables will take a specific value or set of values. Inference tasks are generally structured as in (2), where a set of variables θ are being queried over a PM described by $p(\theta, x)$. Inference is analytically intractable for general PMs [16]. Therefore, approximate Bayesian inference methods have become popular. These approximations can be divided into two categories: variational inference and Monte Carlo methods. In this paper, we focus on the second method.

2.2 MCMC Methods & Hastings Samplers

MCMC presents a direct method for simulating samples from the posterior distribution in (1) or estimating other properties of the distribution. The idea behind MCMC is to have a Markov chain whose stationary distribution is the target distribution; then, samples can be generated by simulating the chain until convergence. In practice, it is common to discard samples from the chain before it converges. This stage is referred to as the *burn-in* stage.

In particular, we consider the Hastings sampler [29] (described in Algorithm 1) that generates sample candidates from a proposal distribution q that is generally different from the target distribution p (above). The algorithm then decides whether to accept or reject candidates based on an acceptance test (α).

$$\alpha = \min\left\{1, \frac{p(x') \times q(x|x')}{p(x) \times q(x'|x)}\right\}$$

Here x and x' represent the current and proposed values, respectively. The choice q and the acceptance test produce a variety of MCMC methods, e.g., Gibbs sampling and HMC.

We consider three variants of the Hastings sampler. The simplest variant is the Metropolis-Hastings algorithm [29, 47], which combines a Gaussian random walk proposal with an accept-reject test as described above. In general this method scales poorly with increasing dimension and complexity of the target distribution. The Gibbs sampling variant [24] utilizes the structure of the target distribution by taking its element-wise conditional distribution as the transition proposal, forcing the conditionals (in Line 3 of Algorithm 1) to be analytically computable. The third variant, called *HMC* [51] uses Hamiltonian dynamics [41] (H) to define a continuous-time transition (i.e., $p(\theta, \rho|x)$) and the stationary distribution of the corresponding Markov chain. To sample from $p(\theta|x)$, HMC introduces an auxiliary momentum variable ρ with the same dimensionality as θ, and effectively samples from the joint distribution $p(\theta, \rho|x) = p(\theta|x)\exp\{-\frac{1}{2}\rho^T M^{-1}\rho\}$, where M is called the *mass matrix*. Samples are generated by simulating

$$\frac{\partial\theta}{\partial t} = \nabla_\rho H \text{ and } \frac{\partial\rho}{\partial t} = \nabla_\theta H. \quad (3)$$

3 Approach Overview

The §(3)–§(7) describe the AcMC^2 system. The key optimizations that drive the system are:

1. Identification of dependences between variables in a PM by constructing and identifying non-intersecting Markov blankets, and use of this information to build parallel multidimensional random number generators.
2. Use of the dependences from (1) and MCMC update strategies to enable concurrent speculative execution of multiple proposal distribution samples and acceptance tests, thereby creating a maximally parallel execution plan.
3. Finally, use of bounded approximation provided by counting bloom filters to optimally utilize the on-chip memory for staging intermediate results.

Fig. 1 illustrates the workflow that integrates the above optimizations. We briefly describe each of its components below.

Compiler Frontend (❶ in Fig. 1). The compiler frontend converts a high-level PPL program (in our case, PMs expressed in the BLOG programming language [48]) into an FG that is used in the following steps of the workflow. FGs [39] allow the description of general probability distributions and subsume all other probability-modeling formalisms [22]. AcMC^2 is decoupled from the choice of frontend PPL language through the use of this IR. We describe this stage further in §4.

Sampling Element Builder (❷ in Fig. 1). Using the IR generated from the PPL program, AcMC^2 first computes a partition of the input FG such that different MCMC update strategies (e.g., Gibbs sampling, HMC) are applied to different portions of the PM. The partition strategy is based on a heuristic approach, unless otherwise specified by the user. For each partition, we separately optimize an accelerated *sampling element* (SE) by identifying the parallelism that is available in the model through the identification of statistical independences. We describe this process in detail in §5.

Hardware Templates (❹ in Fig. 1). Several components of the hardware accelerators are reused across PMs. They include random number generators (RNGs; e.g., following uniform, Gaussian and exponential distributions); arithmetic operators (e.g., floating point adders and multipliers); interfaces to off-chip memory; and host memory. We call those components *templates*, and pre-design them to make optimal trade-offs between on-chip resource utilization and performance. In this paper, we consider optimizations of

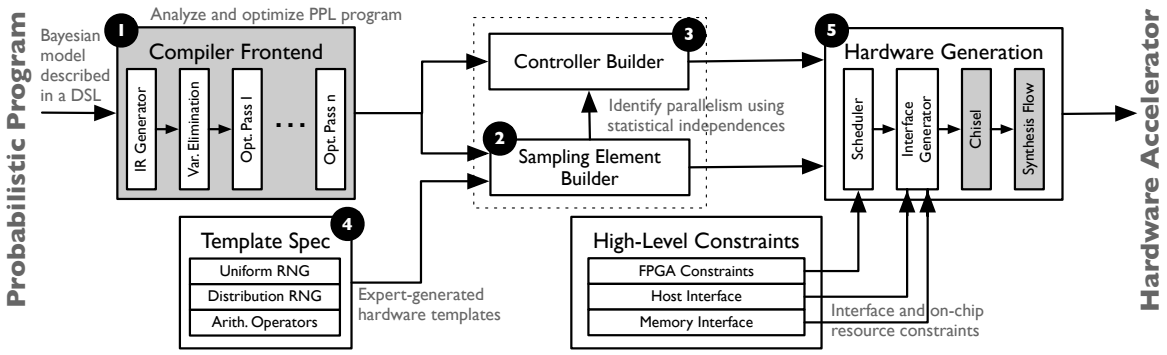

Figure 1. Overview of the AcMC^2 approach. Boxes shown in gray represent third-party components integrated into AcMC^2.

these templates for Xilinx FPGA devices. We describe these templates in §5.2.

Controller Builder (❸ in Fig. 1). We construct *Controllers* to synchronize the actions of SEs. For example, to ensure maximal parallelism in the sampling process, we execute several chains of MCMC samplers in parallel, as well as speculatively sample propsal distributions in a single MCMC chain. Aggregation (mixing) of these results identifies the probability distribution (i.e., a histogram) corresponding to a user's query. We describe these optimizations further in §6.

Hardware Generation (❺ in Fig. 1). In the final step of the AcMC^2 workflow, the above statistical relationships and hardware templates are combined together in an execution schedule for an accelerator. A statically generated schedule significantly simplifies the generated hardware. We then use the Chisel [5] to automatically generate HDL corresponding to the computed schedule. The Chisel-generated Verilog can then be fed into a traditional hardware synthesis workflow. We describe the hardware synthesis process in detail in §7.

4 Compiler Front-End

4.1 The BLOG Language

BLOG [48] represents a strongly typed first-order programming language that can be used to define probability distributions over worlds with unknown numbers of variables. Fig. 2 illustrates the mapping from a statistical model describing a Gaussian mixture model (GMM; used for clustering data) in BLOG to the underlying FG representation, and its correspondence to the final accelerator generated by AcMC^2.

Why BLOG? Research into PPLs has resulted in the development of several languages (e.g., [13, 27, 30, 44, 48, 66]) that allow users to describe PMs as programs. These PPLs can be categorized into two groups. The first is DSLs that are embedded in higher-level languages like Lisp (for Church [27]) or Python (for PyMC3 [56]). The second group (which includes BLOG) consists of essentially standalone languages (with their own interpreters and compilers). The first group of languages is unsuitable for hardware synthesis, as solutions to the PPLs' synthesis must necessarily include solutions to the "high-level synthesis" problems of the higher-level language in which they are embedded (e.g., dealing with unbounded recursion, unbounded loops, and library calls). That was the primary motivation for selection of BLOG as the front-end language for AcMC^2. Further, among the second group of languages, BLOG is one of the few PPLs that can represent PMs that contain both discrete and continuous random variables.

Extensions to BLOG. The BLOG language (and compiler [67]), however, has one drawback. It has no method for describing abstract inputs without binding them to particular values. For example, in Fig. 2, the `obs` keyword is used to describe both the input and its value. We have extended the language by adding an `input` keyword to define formally named inputs that will be made available at runtime. These inputs correspond to input ports on the AcMC^2-generated SEs. Outputs are defined using the `query` keyword.

4.2 FG Generation

AcMC^2 uses the lexer and parser of the BLOG compiler presented in [67] to generate an abstract syntax tree (AST) of the input BLOG program.[3] We then proceed as follows:

1. Identify `query` statements in the AST, replacing them with new variables. We will use these new variables to define named outputs in later steps of the workflow.
2. Search the AST for subtrees for arithmetic expressions that can be statically evaluated (i.e., deterministic code containing constant values) and evaluate them.
3. The AST is traversed to find a list of deterministic variables/functions (i.e., those which are not randomly generated) in the model. This identification is done based on the *type* of the variable or the *return type* of the function. AcMC^2 statically composes these deterministic variables/functions with other random functions, so as to build an FG with only random components.
4. Convert the AST into an FG by associating each BLOG function with a factor function, and its inputs and outputs with associated random variables.
5. Apply *variable elimination* [38] to the FG to reduce the size and complexity of the FG. This method is roughly equivalent to *static function execution* and *dead code elimination* in traditional compilers. Note that variable eliminations that lead to *marginal probability distributions* that cannot be directly sampled are dropped.

[3] It does so after adding the extension keyword `input` mentioned above.

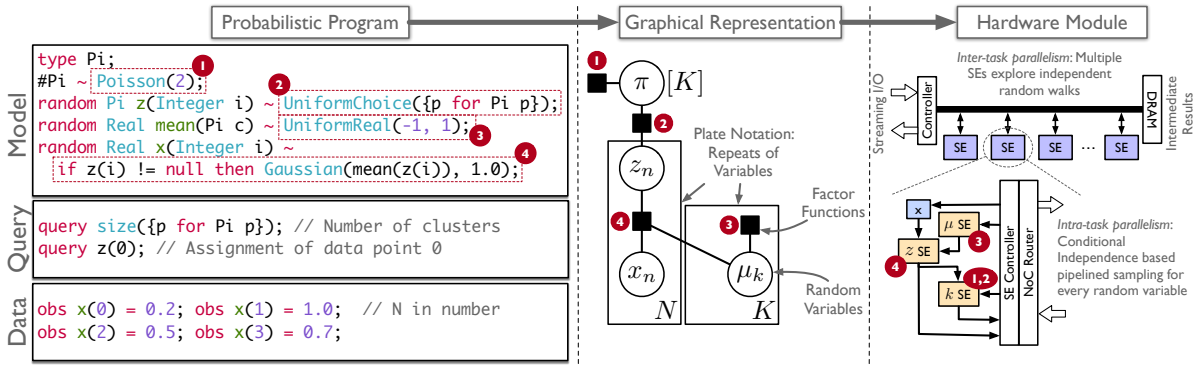

Figure 2. Example of the end-to-end workflow: A Bayesian Gaussian mixture model described as a as a BLOG PPL program, converted into an HDL-based hardware module that van be integrated as SoC components.

Note that the optimizations listed above resemble traditional compiler optimizations that most PPL compilers should perform. The FG conversion is specific to the problem at hand, as the downstream steps of AcMC^2 expect an FG as input. The inputs and outputs of the overall process are illustrated in Fig. 2. Note that we keep track of repetitions of variables and factor functions in the model that correspond to repeated or indexed variables in the original BLOG program. Dynamic PMs (i.e., which express time varying behavior of variables) are expressed in two parts: (1) the FG corresponding to one instant of time, and (2) factor functions corresponding to statistical relationships across timesteps.

The process described above (specifically, Step 4) converts a directed graphical model into an undirected one (i.e., an FG). Both of them provide a formalism for representing independences; however, each of them can represent independence constraints that the other cannot. The conversion process occurs by the construction of a *moral graph* [38] from the original directed acyclic graph. If the original PM is *moral*, then the converted PM is a perfect map. The *moralization* process can cause loss of conditional independence if it introduces new edges. In AcMC^2 such a loss does not change the accuracy of the MCMC, merely the effective amount of parallelism (described in §5).

5 Sampling Element Design

This section describes an algorithm ("SE Builder" from Fig. 1) for generating the design of a single SE (see Fig. 3) based on the input FG. The process has the following steps.

1. Depth-first traversal of the FG identifies a. variables that will be provided as runtime inputs, b. variables that will have to be generated by random sampling, and c. output variables corresponding to a user's query.
2. Variables that need to be generated through sampling are identified and partitioned into sets corresponding to their MCMC proposal and update strategy.
3. AcMC^2 then constructs samplers corresponding to the proposal distributions q. This step identifies conditional independences in FG that can be used to extract the maximal parallelism in each of the partitions.

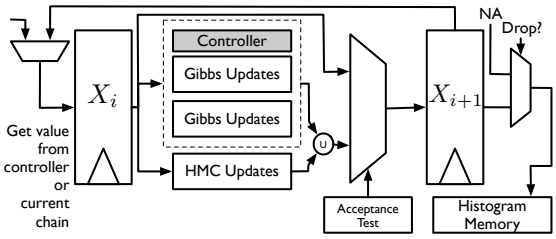

Figure 3. Architecture of sampling element: The SE design is based on a compositional Hastings sampler (see Algorithm 1) that uses Gibbs sampling and HMC updates. X_i is the state of the sampler in its i^{th} iteration.

4. Using the set of template components available to it, AcMC^2 generates samplers for each of the FG partitions.

After those samplers are executed for their burn-in phases, the values corresponding to the query variables are extracted, tabulated, and stored as histograms (as described in §5.3). Each SE generates a fixed number of samples, which represents one execution of an MCMC chain. The output of the "SE Builder" stage is a data-flow graph corresponding to the high-level schematic in Fig. 3. This data-flow graph does not incorporate timing information among the different blocks shown in the figure. Timing is described further in §7.

5.1 Compositional MCMC

Often the high dimensionality of the vectors x and density $q(x'|x)$ being estimated in a Hastings sampler make the sampling process difficult (and, in some cases, intractable). However, it has been shown that it is possible to find MCMC updates for x' that consist of several sub-steps, each of which updates one component or a group of components in x [10, 38]. Finding the optimal division of an FG into partitions for an arbitrary PM is still an open problem.

AcMC^2 relies on a straightforward heuristic to find those partitions. First, it identifies the variables on which it can perform Gibbs sampling. This set consists of discrete random variables in the model, along with continuous variables that exhibit conjugacy relationships. A conjugacy relation implies that a conditional distribution $p(\theta|x)$ takes the same (or an equivalent) functional form as $p(\theta)$. AcMC^2's list of

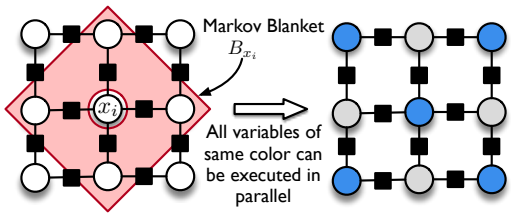

Figure 4. Conditional independences of an FG node can be utilized to identify strategies for parallel execution when a Gibbs-sampling-based Hastings sampler is used.

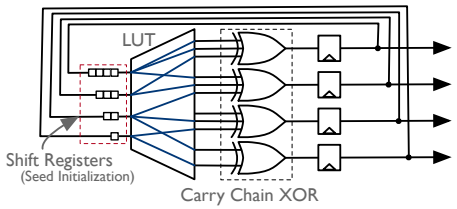

Figure 5. XOR-shift-based FPGA-optimized uniform random number generator.

conjugacy relationships are built based on [18]. The remaining continuous variables are sampled with HMC. We do not explicitly use the Metropolis-Hastings sampler, because of its bad scaling behavior in high-dimensional spaces. A user can override this heuristic and manually specify partitions.

Gibbs Sampling. Recall that a Gibbs sampler utilizes the target distribution as its proposal distribution q and also takes compositional steps, where each step targets the sampling of element-wise conditional distributions. Hence in each sub-step, where x_i is updated with x_i', the sampler uses $q(x_i'|x_i) = p(x_i|x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = p(x_i|x_{-i})$. That means that in an arbitrary PM, every sub-step must proceed sequentially. However, in the case of AcMC^2, since the FG already encodes latent structure in the distribution $p(x_i|x_{-1})$, we can extract conditional independences encoded in the model to relax the dependency of x_i on the set x_{-i}. We do so through the computation of a Markov blanket B_{x_i} on the FG. B_{x_i} defines a subset of x_{-i} such that x_i is conditionally independent of x_{-i} given B_{x_i} (see Fig. 4). Hence $q(x_i'|x_i) = p(x_i|B_{x_i})$, which implies that the sub-steps corresponding to x_i can be executed in parallel with $x_{-i} \setminus B_{x_i}$.

One can generalize that observation to all nodes in the FG by studying the graphical structure of the FG. Variables that are in each other's Markov blankets (i.e., that share a common factor function) cannot be sampled in parallel. Hence, computation of a k-coloring of the FG (i.e., solving a graph coloring problem on the FG), where variables that share a factor function are not given the same color, will give us the maximal parallelism available during the Gibbs sampling process. Here, k will represent the number of synchronization points in the sampling process. [25] provides a proof of correctness of the technique. Fig. 4 demonstrates the property on a factor graph. The coloring is synthesized into a state machine that drives the "Controller" in Fig. 3.

Hamiltonian Monte Carlo. AcMC^2 uses reverse-mode automatic differentiation [8] to automatically compute the gradients required in (3) from the joint distribution of the FG. The current implementation of AcMC^2 performs a source-to-source translation of the symbolic gradients to OpenCL for high-level synthesis through Xilinx SDAccel [21]. That allows us to generate optimized data-paths corresponding to the HMC proposal distribution.

5.2 Template-Based Elements

SE components that are reused across a range of PE models are provided to AcMC^2 as a library of manually designed template patterns that provide low latency, high throughput, and low on-chip resource utilization. In our implementation, all of the template-based components are specialized for FPGAs (which we use as a prototyping platform). However, AcMC^2 can also be used to generate ASICs by replacing the template components. We describe these components below.

Random Number Generators. AcMC^2's RNG library provides three types of generators: (1) uniform RNGs, (2) discrete RNGs corresponding to particular probability mass function definitions, and (3) RNGs for general probability distributions (e.g., Gaussian, Exponential). Our minimum requirement for these RNGs is that they be high-quality generators that pass common statistical tests; they are not required to be cryptographically secure. The composition of RNGs across complex PMs ensures that even though we might exhaust the period of a single RNG, we will never exceed the period of all the RNGs used in an SE. Further, large PMs require many RNGs, so they have to use on-chip resources optimally. Finally, we are interested in RNGs that have deterministic performance. For example, rejection-sampling-based RNGs [14], which might retry an indefinite number of times before generating a random number, are not suitable in our use case, because the hardware generation step in the AcMC^2 requires definite latency characteristics to produce a static schedule of SEs.

The Uniform RNG (see Fig. 5) is the simplest type of random number generator available in AcMC^2. Our implementation draws heavily from [61], and represents a modified XOR-shift [45] generator that is optimized for low resource usage on FPGAs. The 4-bit RNG completely utilizes a single logic cell available on an FPGA: it utilizes a 4-input look-up table (LUT), the XOR-gates from a carry chain adder, and the output registers to buffer output. Overall it produces a single 4-bit random number per clock cycle, utilizing only a single LUT and a single shift register. These RNGs are used as a source of randomness for the other types of RNGs.

The second type of generator in AcMC^2 uses an alias-table-based strategy [65] to generate discrete random numbers whose probability distribution is provided at compile time. Fig. 6 shows the schematic layout for this RNG. It reuses two uniform RNGs to generate addresses and store them into a locally stored alias table, which is accessed to retrieve the value of the random number. The address lookup happens in two clock cycles; hence, the memory element is duplicated to ensure a throughput of 1 operation per cycle. The dynamic range of the probability values in the alias table is set to a 32-bit floating-point number.

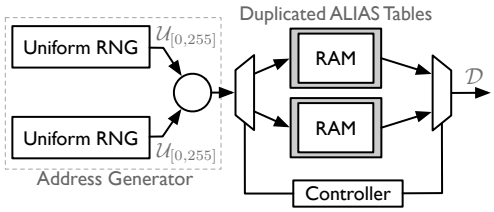

Figure 6. Alias-table-based RNG to generate arbitrary discrete random variables with static distributions.

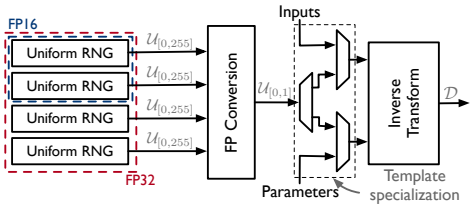

Figure 7. Inverse-transform-based RNGs.

The final type of RNG used in AcMC^2 generates values from well-known distributions by using the inverse transform method [55]. Fig. 7 shows a schematic of the generator. This method transforms integer uniform random numbers into fractional numbers in $[0, 1]$ and then uses the inverse of the target distribution's cumulative density function to generate the required random numbers. The method is computationally intensive, as computing the inverse transform often requires several floating-point operations, leading to higher latencies and lower throughputs than the other RNGs mentioned above. AcMC^2 provides support for exponential, Poisson, Gaussian, and binomial distributions.[4] One can add more RNGs to AcMC^2 using template implementations.

5.3 Storing Sampled Results

The final step of the SE pipeline (see Fig. 3) corresponds to saving the values generated by the SE into *Histogram Memory*. It is implemented using on-chip memory as follows.

- When output variables take a finite number of values (i.e., types corresponding to the output variable are defined over finite sets), AcMC^2 generates counters corresponding to each of the values. The counters are incremented when a corresponding sample is produced.
- When the output variables' domain corresponds to large sets, the user is required to annotate a binning criterion corresponding to each query.
- Binning provides only a partial solution, as there is a limited amount of on-chip memory for storing histogram information. Off-chip DRAM provides an attractive alternative for storing the histograms; however, write latencies to DRAM, as well as write contention across multiple SEs, make the use of DRAM intractable from the point of view of performance. In order to remove this bottleneck, we allow for an approximate solution by using counting Bloom filters [20] (see Fig. 8). The core idea is to allow for storage of the histograms in an approximate fashion whereby counts corresponding to some bins can be larger than their true values, in order to trade off the amount of memory required to store the values. Counting Bloom filters provide bounded approximations for storage that allow us to tune the parameters of the Bloom filters to stay within the noise margins of the MCMC simulations. A problem with the Bloom filter approach is that they eventually fill up over time when they must deal with a large stream of data. As a result, at some point the Bloom filter becomes unusable due to its high error rates. Hence we need to periodically checkpoint the state of the Bloom filter (by storing it to DRAM), and reset it to avoid such problems.

AcMC^2 requires a user to actively *opt in* to any of the above storage types, annotating an FG model with information about histogram binning and Bloom filter size. Our implementation of the counting Bloom filter uses MurmerHash [4].

5.4 Handling Infinite Models

Plate-based models, like the GMM example in Fig. 2, in which portions of the model get repeated multiple times are handled by synthesizing SEs that correspond to the plates in the PM. These SEs are then repeatedly executed based on the plate specification in the PM. This repetition is encoded into state machines (in the "Controller" block in Fig. 3), which control the execution of the proposal distributions across partitions. Broadly speaking, there can be two types of plates in a model: plates that repeat based on user input at runtime (e.g., a plate corresponding to N in Fig. 2), and plates that correspond to repetitions due to model variables (e.g., a plate corresponding to K in Fig. 2). AcMC^2 automatically handles the second type, and requires explicit human annotation of the number of repeats in the first type.

5.5 Importance of RNG Efficiency

RNG efficiency does indeed play a significant role in overall performance. However, the latency throughput characteristics of these RNGs have to be tuned with other components of the system to ensure the best performance. We describe these trade-offs below.

Throughput. Accelerators generated by AcMC^2 leverage data-flow between the RNGs and computational elements (e.g., adders, multipliers) and matches throughput between these elements. In most cases, Xilinx-provided IPs for computational elements execute at 1 op/cycle, which is matched by the template-based RNGs described in §5.2. Thus we can generate MCMC samples at throughputs close to 1 sample/cycle

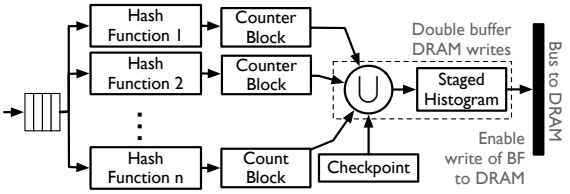

Figure 8. Counting Bloom-filter-based approximate memory for storing histogram outputs of an SE.

[4]The binomial distribution is generated through a look up-table-based approximation of the distribution's inverse density function.

for a single SE. Using worse RNGs could impact the overall throughput of the accelerator and result in SE stalls.

Latency. We prefer high-throughput high-latency RNGs. Note that there is a trade-off point after which the latency negatively affects performance: the assumption in this argument is that a single RNG latency is significantly lower than DRAM write latency. Guaranteeing high throughput (1 sample/cycle) often results in increased RNG latency. The trade-off between sample latency and overall performance has to be tuned to the performance of the memory system. For example, the time taken to generate the output histogram of an SE has to be greater than the time taken to write that histogram to onboard DRAM (i.e., the accelerator is not memory-bound). In fact, we use double buffering of histogram writes (see Fig. 8) to effectively hide the increased RNG latency. An improved memory system on the FPGA board could alleviate these problems.

6 Controller Design

AcMC^2 uses multiple independent instances of the Hastings sampler executed in parallel to generate several target distributions. The final results can then be aggregated from individual samplers. Further, multiple SEs can be used to speculatively execute future steps of Hastings chains in parallel. The "Controller Builder" block from Fig. 1 identifies the scope of that parallelism and constructs a controller that can coordinate the execution of SEs to enable the above optimizations. Fig. 9 illustrates the integration of the controller and SEs into a single accelerator.

Ensemble Samplers. Individual MCMC chains have internal serial dependence; however, multiple chains can be computed in parallel to generate several independent estimates of the target PM posterior distribution. The final result is calculated by pooling the results of these different chains (i.e., by aggregating the output sample histograms from all the chains). This corresponds to exploiting the *embarrassingly parallel* nature of the MCMC process. Overall, this optimization improves throughput and accuracy of the inference process. The only drawback is that each sampler has an independent burn-in phase, so that the number of redundant burn-in samples grows linearly with the number of ensemble samplers employed. We achieve this optimization in the generated accelerator by generating multiple instances of the single-Controller and multiple-SE block (see Fig. 9).

Speculative and Predictive Evaluation. Hastings samplers show branching behavior, i.e., possible random walks explored by the samplers form a *branch tree*. An MCMC chain will traverse several paths (corresponding to the proposal distributions) in this branch tree, but will eventually take only a single path (corresponding to a successful acceptance test). That provides a scope for exploiting parallelism through speculation. For example, in depth-first traversal, a generated sample is assumed to be accepted, and its subtree is generated speculatively. Similarly, in breadth-first traversal, a sample is assumed to be rejected, and other samples from the same level are generated speculatively. In contrast to prior work in statistics (e.g., [3, 57]) that adopted the depth-first approach presented above, AcMC^2 uses the breadth-first approach, as the generated hardware is much simpler. All SEs are set to start with one initial state; thereafter, each SE explores individual samples from the proposal distribution. When an SE generates a value that passes the acceptance test, it broadcasts the new state value to all other SEs, and proceeds with the next step in its own pipeline. The controller arbitrates the bus and ensures race-free executions. Some aspects of the depth-first approach are captured in each individual SE's pipeline, where, as soon as a proposal value is generated, the next level of the branch tree can start execution in the pipeline.

Other auxilliary functions of the controller include:

1. *Initialization*: The controller initializes all SEs with the seeds required to start random number generation, and computes the initial starting state of the Hastings sampler (which is generated from a Gaussian distribution).
2. *Bus Scheduling*: The controller acts as an arbiter to give individual SEs the ability to write data to the multicast bus (described further in §7.2).
3. *Batching*: If a dataset is being used that is larger than the available memory on the accelerator, the data have to be divided into batches, and the inference has to be run one batch at a time. The controller is responsible for copying input batches from the host memory to the accelerator.
4. *Moving Results to Host Memory*: The controller is responsible for moving the final outputs of the MCMC chains (i.e., histograms generated over the user query) from the on board DRAM on which it is stored to the host memory space. Doing so involves aggregating the counting Bloom filters from each of the ensemble samplers.

Communication between the controller and SEs is encapsulated in an AXI-Stream protocol. That protocol allows us to decouple the controller from the SEs and synthesize them separately, relying on the communication protocol between them to ensure synchronization properties.

7 Accelerator Synthesis

The final step of the AcMC^2 workflow converts the data flow graph corresponding to an SE into a synthesizable Chisel model, which can then be fed into traditional FPGA/ASIC synthesis flows.

7.1 Scheduling

The scheduler is responsible for converting the SE data-flow graph into a cycle-by-cycle schedule for a given set of resource constraints. AcMC^2 uses this schedule to generate synchronization between inputs and outputs of stages of the compositional MCMC SE (see Fig. 3). AcMC^2 uses the *As-Soon-As-Possible scheduling algorithm* (based on [40]) to compute such schedules. The scheduling algorithm works by scheduling an operation as soon as all of its predecessor operations (in the data-flow graph) have completed. The maximum number of parallel operations permitted are defined by resource constraint parameters fed in by the user.

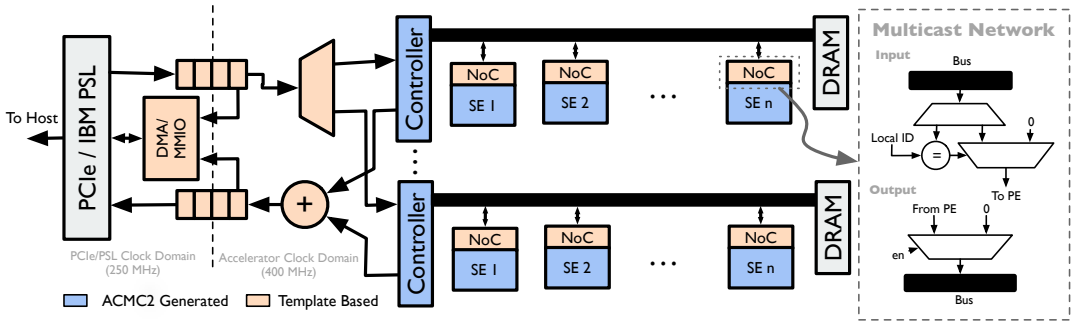

Figure 9. Architecture of the generated accelerator, including SEs, controllers, and host-accelerator communication.

AcMC^2 generates the design for a single SE that accommodates this schedule. Note that the scheduling procedure in AcMC^2 does not consider resource limitations or routing issues between SE components – these decisions are left to the subsequent synthesis flow (recall from §3).

7.2 Overall Accelerator Design

The SE data-flow graph, along with its static schedule, is converted into a Chisel HDL-based description of an SE pipeline. Through the use of the cycle counts for operations in the SE design, a Chisel HDL description of the *Controller* (described in §6) is also generated. These units then interface with off-chip DRAM and host-memory interfaces to produce the final probabilistic inference accelerator. The overall architecture of the accelerator is shown in Fig. 9. We describe the remaining components of the accelerator next.

Broadcast-Based NoC for SEs. AcMC^2 uses a template-based, bus-based network-on-chip (NoC) design to enable point-to-point and broadcast messaging between the Controller, SEs, and DRAM controllers. The bus uses an AXI-Stream-based communication protocol for data transfer. Fig. 9 shows the design of the routers used in the network. To read from the bus, a router matches its local identifier to that of the stream being sent on the bus and connects the SE to the bus if there is a match. Writing to the bus is mediated by the controller in a single-writer, multiple-reader protocol.

Host-Accelerator Communication. AcMC^2 uses the IBM Coherent Accelerator Processor Interface (CAPI) [58] for Power8 processors to facilitate host-accelerator communication. CAPI is layered over PCIe and provides low-latency, high-speed device-to-host memory transfers. In particular, CAPI simplifies the generated host CPU code, reduces dependency on DMA drivers, and eliminates the need for page-pinning and bounce buffering to extract high performance from the underlying PCIe bus. A DMA and MMIO controller for the IBM Power Service Layer (PSL; the IBM IP component that interfaces with the accelerator) is provided as a template for the accelerator. This interface is used to send inputs to the accelerator, receive inputs from the accelerator, and initialize the accelerator with *seed* values for RNGs. The PSL and DMA/MMIO interfaces are clocked at 250 MHz, and the remainder of the accelerator is clocked at 400 MHz.

Host-accelerator communication can be of the following types. 1. In the *batch data transfer* mode, the host loads the input dataset into host memory, from which the accelerator reads data in batches. Similarly, outputs are transferred to a host buffer. 2. In the *streaming data transfer* mode, the host and accelerator share circular buffers corresponding to inputs and outputs. Synchronization between host and accelerator is ensured using CAPI's atomic operations. In both cases, the accelerator is initialized with the addresses of the input/ouptut buffers. The accelerator actively prefetches new data (in cacheline-sized 128-byte chunks) and adds it into the input FIFOs (see Fig. 9) for further processing.

8 Evaluation and Discussion

Experimental Setup. AcMC^2 has been implemented in ~ 2k lines of Scala. The templates used in the compiler were developed in System Verilog and use IPs from Xilinx to implement single-precision floating-point math operators, BRAM blocks, an off-chip DRAM interface, and shift registers. The accelerator communicates with the host through the IBM CAPI interface [9, 58]. All generated accelerators (and CPU baselines) were evaluated on an IBM Power8 S824L system with an Alpha-Data ADM-PCIE-7V3 FPGA board (with Xilinx Virtex 7 XC7VX690T FPGA) and an NVIDIA K80 GPU.

8.1 Comparison With CPU-Based PPLs

As a first-level comparison, we compared the runtime of accelerators generated through AcMC^2 with common benchmarks that are used to evaluate PPLs. We used the set of benchmarks from [48, 67], which represent tests over a wide set of PPL features available in BLOG. They are indicative of performance, as they compare the performance of AcMC^2-generated accelerators to that of accelerators generated by CPU-based PPL compilers. However, these benchmarks do not represent complex the PMs that a user would encounter in the real world. We consider such real-world problems (and their implementations on GPUs and HLS compilers) in §8.2.

A performance comparison of the techniques is shown in Fig. 10, and a comparison of peak power usage is presented in Table 1. Power usage estimates of the generated accelerators are collected from the Xilinx Vivado tools and via the S824L's in-built "system-level" power measurement infrastructure.[5] Power estimates for CPU-based code were made based on

[5]The system reports power measurements averaged over 30s intervals. We measure a distribution of power consumption when the system was idling, and when the accelerator was being used.

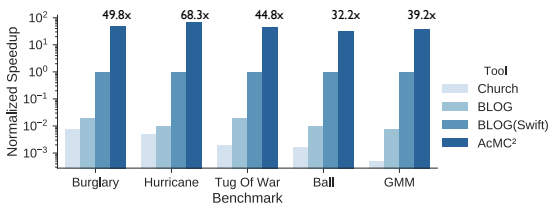

Figure 10. Comparison of runtime performance for AcMC^2-generated accelerators with PM inference on other PPLs (normalized to `Blog(Swift)` [67] compiler).

publicly available TDP values. In all cases, the programs were instrumented to run 2 million samples before stopping. Overall, we observed an average speedup of 46.8× and an average reduction in power of 16.1× (which corresponds to an overall improvement of 753× in terms of performance-per-watt). We observe that FPGA BRAM utilization was the dominant resource used, as seen in Table 1. Note that all performance measurements presented above were collected over 1000 runs of each of the programs, to amortize OS costs in process creation and setting up of communication through CAPI. The GMM benchmark uses the counting Bloom filter approximation described in §5.3. It infers the distribution of the number of mixture components, as well as the distribution of means and variances for each mixture component. The histograms corresponding to the means and variances are stored in the Bloom filter; in each case, a 1000-bin histogram was stored in 100 counters and checkpointed/refreshed every 10000 samples. All benchmarks used the batch data-transfer model for host-accelerator communications. The number of ensemble samplers was limited to 4 to ensure that each sampler was mapped to a single onboard memory DIMM.

Overall, Fig. 10 and Table 1 suggest that when there is no unbounded repetition in the PM, AcMC^2-generated accelerators fare better (with respect to both runtime performance and power usage) with discrete variable PMs. That is expected, because (1) the use of continuous distributions (even if they have conjugacy relationships) requires expensive floating-point computations to compute inverse transforms, which significantly increases the latency of the RNGs and results in higher resource cost; and (2) unbounded worlds require re-execution of the MCMC chain (i.e., the SE) with different (sampled) initial worlds. It is important to note that the performance comparison across Church, BLOG, and Swift also compares the overhead of the language runtimes: Church uses Lisp [27], BLOG uses Java [48], Swift uses C++ [67]. AcMC^2 does not have any of these overheads.

8.2 Real-World Case Studies

Case Study 1: Epilepsy SoZ Localization. We applied AcMC^2 to a PM [63] that is used to infer characteristics of human-brain activity by using electroencephalogram (EEG) sensors, with the goal of identifying brain regions responsible for the onset of epilepsy. [63] presents a generative FG model that estimates brain activity and localizes it to a particular EEG sensor, allowing clinicians to identify regions

Table 1. Power and resource utilization for benchmark BLOG programs. Numbers of SEs are described as $x \times y$: x is the number of controllers and y is the SEs per controller.

	Power (Watts)				
Benchmark	**CPU**	**Vivado**	**Measured**	**# of SEs**	**BRAM %**
Burglary	190	10.1	16.3 ± 2.1	4 × 4	35%
Hurricane	190	10.3	16.8 ± 1.9	4 × 4	31%
Tug of War	190	12.5	19.4 ± 3.1	4 × 4	38%
Ball	190	12.9	19.3 ± 2.6	4 × 4	41%
GMM	190	14.1	15.3 ± 3.4	4 × 4	46%
EEG-Graph	190	11.8	15.8 ± 3.8	4 × 8	64%

of the brain (called *seizure onset zones* or *SoZs*) that show pathological behavior like epileptic seizures. This application represents a typical use of PMs in the field of precision medicine, where data are obtained from medical sensors in a streaming fashion and used to make decisions in real time.

Case Study 2: Network Security. The second real-world problem for which we show the application of AcMC^2 is in the domain of *network security*. Here, a PM [12] is used to describe the relationships between user intent (i.e., whether a user is benign, suspicious, or malicious and represents a threat to the integrity of a networked computer system) and events observed by security monitors (e.g., network monitors like Bro [52]). Using these statistical relationships, [12] aims to infer the user state given real-time data from the security monitors. This application represents the typical use of PMs to build "intelligent" compute-embedded network devices like network interface cards (NICs) or switches that can automatically detect and preempt malicious intrusions.

For each of the case studies described above, we constructed (1) a BLOG program and (2) an OpenCL program corresponding to the model. The OpenCL program was hand-tuned to use the model-specific optimizations presented in §5 and §6. Two separate versions of this program were created, using compiler-specific attributes targeting NVIDIA's OpenCL and Xilinx's SDAccel compilers. In both cases, the CPU baseline corresponds to software obtained from the original authors. Note that the CPU baselines represent research software, which, as such, might not be perfectly tuned to the system architecture. However, that is a common situation for research software for which an underlying compiler is expected to perform meaningful performance optimizations.

Why these models? (1) These models represent real–world applications of PMs in performance-critical applications across varied application domains. (2) These models use a large subset of the PPL language features: discrete and continuous random variables, distribution conjugacy relationships, and unbounded dynamic models that obey the Markov property. (3) Further, they represent challenging situations for AcMC^2 because they use multivariate factor functions in relatively small FGs. The PM is relatively densely connected so that the generation of random samples in the Gibbs sampling-based SE is almost completely serialized.

Comparisons to CPU: Case Study 1. The generated accelerator can infer the query posterior distribution for a 3s chunk of input EEG data in an average of ∼ 41.8 ms compared

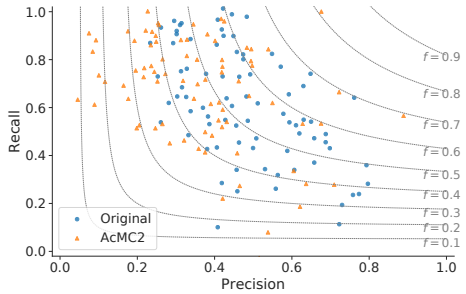

Figure 11. Accuracy (in terms of precision and recall) of the EEG-GRAPH energy minimization method (labeled `Original`) vs. the MCMC accelerator generated by AcMC^2.

to the CPU baseline, which uses 4.3s (i.e., it does not meet the 3s real time requirement of the applications). Following Little's Law, a single FPGA accelerator would be able process $71 \approx {}^{3s}/_{41.8ms}$ patients in parallel in real-time. That is the largest accelerator configuration we have successfully fit onto the FPGA. Fig. 11 shows the difference in accuracy between the original EEG-GRAPH technique and the AcMC^2 accelerator. Overall there is a drop in accuracy from an f-score[6] of 0.47 to one of 0.465. That can be attributed to the approximate nature of the MCMC procedure. The method proposed in [63] uses exact inference procedures based on energy minimization, which might produce marginally more accurate answers.

Comparisons to CPUs: Case Study 2. The AcMC^2-generated accelerator performed the inference at the rate of 0.4 ms/event. That is ~ 48.4× faster than the CPU-based implementation in [12]; here, an event corresponds to the output security monitor. The CPU implementation uses threading as well as high-performance math libraries that are optimized for the SIMD and memory locality. In this case, the approximate Bloom filter optimization did not affect the statistical correctness of the output (i.e., 74.2% true-positive rate, 98.5% true-negative rate, 1.5% false-positive rate, and 25.8% false negative rate).

All our experiments were setup to use the same number of accepted samples. The number of samples was chosen in each case to ensure that the MCMC procedure would be close to convergence. However, the approximate nature of MCMC implies that independent runs do not give the same answers. To verify the correctness of our generated acclerators, we performed a Kolmogorov-Smirnov test across the CPU, GPU, and AcMC^2 implementations to ensure that the sampled distributions were identical with high probability. Tables 2 and 3 show a comparison of power and FPGA resource utilization for the case studies presented above.

OpenCL Comparison: Code Complexity. Table 4 shows a comparison of the complexity of the AcMC^2 and OpenCL accelerators in terms of lines of code (LoC). For example, in Case Study 1, compared to the AcMC^2 accelerator, which is described in 183 LoC. The GPU and FPGA

[6]The f-score is two times the harmonic mean of precision and recall. An F-Score can take values between [0, 1], with 1 being the best possible score.

Table 2. Performance & power consumption improvements.

		Power (Watts)			
Benchmark	**Perf. CPU**	**CPU**	**Vivado**	**Measured**	**# of SEs**
Case Study 1	102.1×	190	11.8	19.3 ± 3.7	4 × 8
Case Study 2	48.4×	190	10.4	11.2 ± 0.8	4 × 8

Table 3. Summary of FPGA resource utilization.

	BRAM		DSP		FF		LUT	
	Av.	%	Av.	%	Av.	%	Av.	%
Case Study 1	1470	64%	3600	49%	866400	22%	43200	50%
Case Study 2	1470	83%	3600	29%	866400	34%	43200	61%

Av. = Available on FPGA

OpenCL require 622 and 961 LoC, respectively. Their added complexity is derived from writing memory and synchronization code on the GPU. In the case of the FPGA, explicit code annotation (e.g., `__attribute__` directives), statically bound loops and other code snippets is used to force effective pipelining. In addition to LoC, the expertise (of the underlying hardware system) required to construct the OpenCL version clearly emphasizes the superiority of AcMC^2.

OpenCL Comparison: Performance (GPUs & FPGAs). Table 4 further shows a comparison between performance and power requirements for the four configurations. For each case study, performance has been normalized to that of the GPU. We observe that the AcMC^2-generated accelerators performed 8 – 18× better than the K80 GPU and 248 – 462× better in performance-per-Watt terms. We speculate that the reduced performance resulted from (1) *control divergence* between threads that were exploring separate parts of the MCMC search space, and (2) *throughput-optimized* RNG libraries that perform better when they have to generate a batch of values rather than the single values used in MCMC's inherently sequential random walks. We drew these conclusions based on the rejection-sampling-based algorithms used in the GPU implementations; i.e., the kernels generate several RNs and accept/reject a fraction of them, which means that the control flow is different on different threads. Further, the GPU libraries for RNGs work by generating batches of RNs; they are later consumed to generate samples for the MCMC. Thereby maximizing the throughput of RNG but not the throughput of the MCMC computation. The exact measurement of divergence in this case is difficult to make as the mapping between PTX and SASS is unknown on NVIDIA devices. Measurement on a microarchitectural simulator might not lead to exact results.

The HLS-generated FPGA accelerator performs an order of magnitude worse than the AcMC^2 and GPU implementations, in terms of absolute performance terms. There are multiple reasons: (1) the accelerator can attain a maximum clock speed 163 MHz, while the AcMC^2-generated accelerator can achieve 400 MHz; and (2) the automatically generated XOR-shift RNGs are higher-latency and lower-throughput than the 1-RNG-per-cycle generators described in §5.

Choice of CPU Baseline. In our measurements Intel Xeon E5 CPUs performed less than 6 – 8% better than the

Table 4. Comparing complexity and performance of AcMC^2-generated accelerators with that of OpenCL accelerators.

	NVIDIA K80 GPU			FPGA		
Implementation	**LoC**	**Perf.**	**Power (W)**	**LoC**	**Perf.**	**Power (W)**
AcMC^2- CS 1	–	–	–	183	18.2×	11.8
OpenCL - CS 1	622	1×	300 (TDP)	961	0.2×	14.2
AcMC^2- CS 2	–	–	–	146	8.6×	10.4
OpenCL - CS 2	586	1×	300 (TDP)	8984	0.8×	15.6

CS = Case Study; Performance normalized to GPU implementation.

IBM Power8. Our choice of that baseline did not change our conclusions. However, the use of the CAPI-based interface for host-accelerator communication significantly simplified the implementation (recall §7.2). The Power8 and Xilinx FPGA use different process technologies, i.e., 22nm and 28nm respectively, and hence it might not be completely fair to compare their results. However, the decision to use CAPI limits us to using FPGA boards that are supported by IBM (with their PSL IP components). We believe that changing the FPGA technology will not change our performance (as the 250 Mhz clock should be replicable on even newer FPGA parts); however, performance-per-Watt measurements will change with the design of the FPGA routing network.

Performance Implications of CAPI. All the microbenchmarks correspond to transmision of $100 - 960$ KB of input data to the accelerator for computation. The total time for these transfers (in streaming mode) is included in the results presented in the paper and contribute $< 10\%$ of the runtime. We observe in Case Study 2 that the runtime in is bound by the PCIe messaging latency (which is on the order of 100 ns for the 128-byte cache-line transferred over PCIe in CAPI). A real deployment of this accelerator would involve at least 3 such messages over PCIe, i.e., network interface card (NIC) to CPU, CPU to the accelerator on the FPGA, and, finally, returning the result to the CPU. We will address this communication latency issue in future work.

9 Related Work

Parallelization of Probabilistic Inference. Low et al. [43] present a distributed computing framework for machine learning algorithms. They demonstrate the distributed parallelization of probabilistic inference using belief propagation [38]. Gonzales et al. [25] provide a proof of correctness for parallelization of Gibbs sampling through the use of conditional independences; this motivated our use of Markov blankets in §5. Recht et al. [54] suggest *asynchronous Gibbs sampling*, whereby conditional independences are not honored by the sampler; [17] provides bounds on the asymptotic correctness of such a sampler. [3, 57] provide methods for speculative execution (called *dynamic prefetch*) using biased proposal distributions. We use the approximation from [41] to generate samples in HMC. Homan and Gelman [31] present a further-optimized algorithm (which converges more quickly than traditional HMC with [41]) to generate samples from the proposal distribution of an HMC. [13] provides an implementation of [31] in a PPL.

Accelerated Probabilistic Inference. Several prior FPGA/ASIC efforts solve some form of probabilistic inference; however, they cannot be generalized to apply to all PMs. For example, [36] proposes an architecture for LDPC code encoding/decoding; [6, 7, 34] propose architectures for several bioinformatics applications; [15] proposes an architecture for image segmentation; and [49] proposes an architecture for inference on a class of state space models. In comparison, AcMC^2 has the ability to generate accelerators for general PMs expressed in the BLOG language. The use of GPUs in MCMC has been explored, but only for particular applications, e.g., in [19, 59, 60]. [42] comes closest to AcMC^2. It describes the design of an FPGA-based accelerator and a compiler to target the acceleration of belief propagation in Bayesian networks. In contrast, AcMC^2 can be applied to a much larger set of PMs. [64] explores the use of analog circuits to perform statistical inference.

Hardware Generators. AcMC^2 uses the Chisel HDL [5] to generate RTL corresponding to the accelerator. Other HDL generators, e.g., [37, 53], provide higher-level constructs that can be used to declare and annotate the parallelism available in a program; they cannot be used directly with PMs as PPLs.

Programming Language Design. [62] proposes Edward, a PPL for Bayesian neural networks (BNN) probabilistic inference that uses variational inference techniques with MCMC methods. It utilizes Tensorflow's [1] GPU-based tensor operations, and potentially Google's TPUs [35]. [11] provides an optimized architecture for inference (which includes both variational inference and MCMC) on BNNs with Gaussian priors. In comparison, AcMC^2 only targets MCMC applications and cannot handle variational inference.

10 Conclusion & Future Work

This paper presented the design and evaluation of AcMC^2, a compiler that can transform PMs expressed in the BLOG PPL into optimized accelerators that compute inference queries by using an ensemble of MCMC methods. We believe that AcMC^2 significantly simplifies the process of constructing workload-optimized accelerators for executing inference on PMs, making the benefits of such optimization available to a larger group of researchers. As a result, AcMC^2 forms the basis for creating future high-performance SoCs for AI applications that can be deployed as edge devices or in clouds.

Future Work. The design of AcMC^2-generated accelerators in this paper assumes that the PM models being compiled fit on a single FPGA. It is not impossible to conceive of a PM for which that assumption would fail. We believe a solution will involve the distributed execution of the ensemble samplers over multiple FPGAs and unreliable network links.

Acknowledgments

This research was supported by the National Science Foundation (NSF) under Grant Nos. CNS 13-37732 and CNS 16-24790. We thank P. Cao, Y. Varatharajah, J. Applequist and K. Atchley for their help in preparing the manuscript.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283.

[2] B. Andres, T. Beier, and J.H. Kappes. 2012. OpenGM: A C++ Library for Discrete Graphical Models. *CoRR* abs/1206.0111 (2012).

[3] Elaine Angelino, Eddie Kohler, Amos Waterland, Margo Seltzer, and Ryan P. Adams. 2014. Accelerating MCMC via Parallel Predictive Prefetching. In *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence (UAI'14)*. AUAI Press, Arlington, Virginia, United States, 22–31.

[4] Austin Appleby. 2008. MurmurHash. Retrieved Jan 27, 2019 from https://sites.google.com/site/murmurhash

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. ACM, New York, NY, USA, 1216–1225.

[6] S. S. Banerjee, M. el Hadedy, C. Y. Tan, Z. T. Kalbarczyk, S. Lumetta, and R. K. Iyer. 2017. On accelerating pair-HMM computations in programmable hardware. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8.

[7] Narges Bani Asadi, Christopher W. Fletcher, Greg Gibeling, Eric N. Glass, Karen Sachs, Daniel Burke, Zoey Zhou, John Wawrzynek, Wing H. Wong, and Garry P. Nolan. 2010. ParaLearn: A Massively Parallel, Scalable System for Learning Interaction Networks on FPGAs. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10)*. ACM, New York, NY, USA, 83–94.

[8] Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon. 2000. Automatic differentiation of algorithms. *J. Comput. Appl. Math.* 124, 1-2 (2000), 171–190.

[9] Matthijs Brobbel. 2015. capi-streaming-framework. Retrieved Jan 27, 2019 from https://github.com/mbrobbel/capi-streaming-framework

[10] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. 2011. *Handbook of Markov Chain Monte Carlo*. CRC press.

[11] Ruizhe Cai, Ao Ren, Ning Liu, Caiwen Ding, Luhao Wang, Xuehai Qian, Massoud Pedram, and Yanzhi Wang. 2018. VIBNN: Hardware Acceleration of Bayesian Neural Networks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 476–488.

[12] Phuong Cao, Eric Badger, Zbigniew Kalbarczyk, Ravishankar Iyer, and Adam Slagell. 2015. Preemptive Intrusion Detection: Theoretical Framework and Real-world Measurements. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security (HotSoS '15)*. ACM, New York, NY, USA, Article 5, 12 pages.

[13] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* 76, 1 (2017).

[14] George Casella, Christian P. Robert, and Martin T. Wells. 2004. Generalized Accept-Reject sampling schemes. In *Institute of Mathematical Statistics Lecture Notes - Monograph Series*. Institute of Mathematical Statistics, 342–347.

[15] J. Choi and R. A. Rutenbar. 2012. Hardware implementation of MRF map inference on an FPGA platform. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*. 209–216.

[16] Gregory F. Cooper. 1990. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence* 42, 2 (1990), 393 – 405.

[17] Christopher De Sa, Kunle Olukotun, and Christopher Ré. 2016. Ensuring Rapid Mixing and Low Bias for Asynchronous Gibbs Sampling. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16)*. JMLR.org, 1567–1576.

[18] Morris H DeGroot and Mark J Schervish. 2012. *Probability and statistics*. Pearson Education.

[19] Kenneth Esler, Jeongnim Kim, David Ceperley, and Luke Shulenburger. 2012. Accelerating quantum Monte Carlo simulations of real materials on GPU clusters. *Computing in Science & Engineering* 14, 1 (2012), 40–51.

[20] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. 2000. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking* 8, 3 (2000), 281–293.

[21] Jeff Fifield, Ronan Keryell, Hervé Ratigner, Henry Styles, and Jim Wu. 2016. Optimizing OpenCL Applications on Xilinx FPGA. In *Proceedings of the 4th International Workshop on OpenCL (IWOCL '16)*. ACM, New York, NY, USA, Article 5, 2 pages.

[22] Brendan J. Frey. 2003. Extending Factor Graphs So As to Unify Directed and Undirected Graphical Models. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI'03)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 257–264.

[23] Yarin Gal. 2016. Uncertainty in deep learning. *University of Cambridge* (2016).

[24] Stuart Geman and Donald Geman. 1987. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. In *Readings in Computer Vision*. Elsevier, 564–584.

[25] Joseph Gonzalez, Yucheng Low, Arthur Gretton, and Carlos Guestrin. 2011. Parallel Gibbs Sampling: From Colored Fields to Thin Junction Trees. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research)*, Geoffrey Gordon, David Dunson, and Miroslav Dudík (Eds.), Vol. 15. PMLR, Fort Lauderdale, FL, USA, 324–332.

[26] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. Vol. 1. MIT press Cambridge.

[27] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence (UAI'08)*. AUAI Press, Arlington, Virginia, United States, 220–229.

[28] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *Proceedings of the on Future of Software Engineering (FOSE 2014)*. ACM, New York, NY, USA, 167–181.

[29] W Keith Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* 57, 1 (1970), 97–109.

[30] Shawn Hershey, Jeff Bernstein, Bill Bradley, Andrew Schweitzer, Noah Stein, Theo Weber, and Ben Vigoda. 2012. Accelerating inference: towards a full language, compiler and hardware stack. *arXiv preprint arXiv:1212.2991* (2012).

[31] Matthew D. Homan and Andrew Gelman. 2014. The No-U-turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.* 15, 1 (Jan. 2014), 1593–1623.

[32] M. Hosseini, R. Islam, A. Kulkarni, and T. Mohsenin. 2017. A Scalable FPGA-Based Accelerator for High-Throughput MCMC Algorithms. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 201–201.

[33] Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. 2017. Compiling Markov Chain Monte Carlo Algorithms for Probabilistic Modeling. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 111–125.

[34] Sitao Huang, Gowthami Jayashri Manikandan, Anand Ramachandran, Kyle Rupnow, Wen-mei W. Hwu, and Deming Chen. 2017. Hardware Acceleration of the Pair-HMM Algorithm for DNA Variant Calling. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 275–284.

[35] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 1–12.

[36] M. Karkooti and J. R. Cavallaro. 2004. Semi-parallel reconfigurable architectures for real-time LDPC decoding. In *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.*, Vol. 1. 579–585 Vol.1.

[37] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. 2016. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 115–127.

[38] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT press.

[39] Frank R Kschischang, Brendan J Frey, and H-A Loeliger. 2001. Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory* 47, 2 (2001), 498–519.

[40] David C Ku and Giovanni DeMicheli. 2013. *High level synthesis of ASICs under timing and synchronization constraints*. Vol. 177. Springer Science & Business Media.

[41] Benedict Leimkuhler and Sebastian Reich. 2004. *Simulating hamiltonian dynamics*. Vol. 14. Cambridge university press.

[42] Mingjie Lin, Ilia Lebedev, and John Wawrzynek. 2010. High-throughput Bayesian Computing Machine with Reconfigurable Hardware. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '10)*. ACM, New York, NY, USA, 73–82.

[43] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8 (April 2012), 716–727.

[44] Vikash K. Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: A higher-order probabilistic programming platform with programmable inference. *arXiv preprint* arXiv:1404.0099 (2014).

[45] George Marsaglia. 2003. Xorshift RNGs. *Journal of Statistical Software* 8, 14 (2003).

[46] G. Martin and G. Smith. 2009. High-Level Synthesis: Past, Present, and Future. *IEEE Design Test of Computers* 26, 4 (July 2009), 18–25.

[47] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. 1953. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics* 21, 6 (1953), 1087–1092.

[48] Brian Milch, Bhaskara Marthi, and Stuart Russell. 2004. Blog: Relational modeling with unknown objects. In *ICML 2004 Workshop on Statistical Relational Learning and Its Connections*.

[49] Grigorios Mingas, Leonardo Bottolo, and Christos-Savvas Bouganis. 2017. Particle MCMC algorithms and architectures for accelerating inference in state-space models. *International Journal of Approximate Reasoning* 83 (2017), 413 – 433.

[50] Joris M. Mooij. 2010. libDAI: A Free and Open Source C++ Library for Discrete Approximate Inference in Graphical Models. *Journal of Machine Learning Research* 11 (Aug. 2010), 2169–2173.

[51] Radford M Neal et al. 2011. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo* 2, 11 (2011).

[52] Vern Paxson. 1999. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks* 31, 23-24 (1999), 2435–2463.

[53] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Paterns. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 389–402.

[54] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 693–701.

[55] Christian P Robert. 2004. *Monte Carlo Methods*. Wiley Online Library.

[56] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* 2 (April 2016), e55.

[57] Ingvar Strid. 2010. Efficient parallelisation of Metropolis–Hastings algorithms using a prefetching approach. *Computational Statistics & Data Analysis* 54, 11 (2010), 2814–2835.

[58] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. 2015. CAPI: A Coherent Accelerator Processor Interface. *IBM J. Research and Develop.* 59, 1 (Jan 2015), 7:1–7:7.

[59] Marc A Suchard, Quanli Wang, Cliburn Chan, Jacob Frelinger, Andrew Cron, and Mike West. 2010. Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures. *Journal of computational and graphical statistics* 19, 2 (2010), 419–438.

[60] Alexander Terenin, Shawfeng Dong, and David Draper. 2018. GPU-accelerated Gibbs sampling: a case study of the Horseshoe Probit model. *Statistics and Computing* (19 Mar 2018).

[61] D. B. Thomas and W. Luk. 2010. FPGA-Optimised Uniform Random Number Generators Using LUTs and Shift Registers. In *2010 International Conference on Field Programmable Logic and Applications*. 77–82.

[62] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep probabilistic programming. In *International Conference on Learning Representations*.

[63] Yogatheesan Varatharajah, Min Jin Chong, Krishnakant Saboo, Brent Berry, Benjamin Brinkmann, Gregory Worrell, and Ravishankar Iyer. 2017. EEG-GRAPH: A Factor-Graph-Based Model for Capturing Spatial, Temporal, and Observational Relationships in Electroencephalograms. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 5371–5380.

[64] Benjamin Vigoda. 2003. *Analog Logic: Continuous-Time Analog Circuits for Statistical Signal Processing*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[65] A.J. Walker. 1974. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters* 10, 8 (1974), 127.

[66] Frank Wood, Jan Willem Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research)*, Samuel Kaski and Jukka Corander (Eds.), Vol. 33. PMLR, Reykjavik, Iceland, 1024–1032.

[67] Yi Wu, Lei Li, Stuart Russell, and Rastislav Bodik. 2016. Swift: Compiled Inference for Probabilistic Programming Languages. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*. AAAI Press, 3637–3645.