IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 3, 2021, accepted April 22, 2021, date of publication April 28, 2021, date of current version May 6, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3076207

PMBA: A Parallel MCMC Bayesian Computing Accelerator

YUFEI NI[1], YANGDONG DENG[2], (Senior Member, IEEE), AND SONGLIN LI[2]
[1]Institute of Microelectronics, Tsinghua University, Beijing 100084, China
[2]School of Software, Tsinghua University, Beijing 100084, China

Corresponding author: Yangdong Deng (dengyd@mail.tsinghua.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1702600, and in part by the NSFC Key Scientific Instrument and Equipment Development Project under Grant 20151310834.

ABSTRACT Bayesian computing, including sampling probability distributions, learning graphic model, and Bayesian reasoning, is a powerful class of machine learning algorithms with such wide applications as biologic computing, financial analysis, natural language processing, autonomous driving, and robotics. The central pattern of Bayesian computing is the Markov Chain Monte Carlo (MCMC) computing, which is compute-intensive and lacks explicit parallelism. In this work, we propose a parallel MCMC Bayesian computing accelerator (PMBA) architecture. Designed as a probabilistic computing platform with native support for efficient single-chain parallel Metropolis-Hastings based MCMC sampling, PMBA boosts the performance of probabilistic programs with a massive-parallelism microarchitecture. PMBA is equipped with on-chip random number generators as the built-in source of randomness. The sampling units of PMBA are designed for parallel random sampling through a customized SIMD pipeline supporting data synchronization every iteration. A respective computing framework supporting automatic parallelization and mapping of probabilistic programs is also developed. Evaluation results demonstrate that PMBA enables a 17-21 folds speedup over a TITAN X GPU on MCMC sampling workload. On probabilistic benchmarks, PMBA outperforms prior best solutions by factor of 3.6 to 10.3. An exemplar based visual category learning algorithm is implemented on PMBA to demonstrate its efficiency and effectiveness for complex statistical learning problems.

INDEX TERMS Accelerator architectures, Bayesian methods, FPGA, MCMC, parallel machines.

I. INTRODUCTION

As a foundation of modern statistics, Bayesian learning and inference theory provides efficient tools to evaluate and update beliefs in the presence of new observations. Due to its advantages in learning with small samples, Bayesian reasoning has received significant successes in the past a few decades [1]. Bayesian models have been built for a wide range of machine leaning applications including computer vision [2], document classification [3], object perception [4], word learning [5], working memory [6], and sensorimotor integration [7]. For instance, Lake *et al.* [8] realized a Bayesian model that could learn from sparse data and successfully recognize and write characters that are indistinguishable from those written by human. In addition, an increasing amount of evidence suggests that Bayesian reasoning account for many essential cognitive processes of human beings [9], [10] due to its natural advantages in handling uncertainties and providing interpretable results. Recently, a notable direction attracting considerable interests is to combine Bayesian reasoning with deep neural networks [11].

At the core of Bayesian computing is the computation of probabilistic distributions. As in many cases it is infeasible to directly derive the analytical formulation of a complex joint distribution over a large number of random variables, the Markov Chain Monte Carlo (MCMC) algorithm was proposed to approximate an arbitrary distribution for further calculation (e.g. likelihood function). However, the inherent sequential nature and computation intensity of MCMC pose essential challenges to perform Bayesian computations in an efficient manner [12]. To mitigate the computing pressure,

The associate editor coordinating the review of this manuscript and approving it for publication was Yanbo Chen.

different approaches, such as splitting data into sub-sets for parallel calculating [13], sampling multiple candidates simultaneously for a single-chain [14], and sampling a group of new states in parallel [15], have been proposed to parallelizing MCMC algorithms. These parallel MCMC algorithms, nevertheless, perform poor on current parallel platforms due to the frequent interactions among threads during sampling. The situation is even worse on GPU platforms because of the high synchronization cost. As a result, it is still infeasible to deploy large scale Bayesian computing applications, not to mention real-time applications like [16], on existing platforms. With the fast-growing demand for Bayesian computations, there is an urgent need to design dedicated hardware platform for efficient MCMC computations.

There are three major concerns for the design of MCMC hardware design. First, the sampling unit has to be flexible enough to support general statistical computing patterns. A design bound to a single distribution cannot meet the requirements of varying applications. Second, MCMC allows two different parallelization patterns, 1) parallel sampling multiple chains and 2) parallel sampling multiple results in a single chain. The former is limited to scenarios that require a large number of sampling tasks or data sets that can be spilt into individual sub-sets. In other words, the single-chain performance cannot be improved in this approach. The latter, in spite of being more general, has not found effective hardware solutions. Third, random numbers are indispensable in MCMC processing and thus dedicated random number generators are essential to promote the computing efficiency. Previous works, which will be reviewed later in sub-section 2.3, only partially solve the above problems, while a systematic treatment of general accelerator for Bayesian computing is still missing.

In this work, we propose the parallel MCMC Bayesian computing accelerator (PMBA), a computing platform supporting efficient Bayesian probabilistic computations. Motivated by an effective parallel Metropolis-Hastings (MH) algorithm [16], PMBA leverages the paradigm of massive parallel execution to effectively accelerate the processing of single-chain MCMC sampling. The microarchitecture consists of a set of sampling cores, with each handling a single thread of random sampling, a set of computing cores for general purpose probabilistic computing, and a merge engine to generate final sampling results. The sampling cores are packed as a sampling unit to sample MCMC states for varying levels of parallelism and multiple tasks. All the sampling units are integrated with one computing unit and work in a SIMD manner. Final sampling results are derived with a sequential merge engine.

The remainder of this paper is organized as follows. Section II explains the background related to this work. In Section III, we introduce the overall architecture of PMBA. Section IV covers the microarchitectural design details, while Section V reports the performance evaluation results. Section VI concludes the paper and discusses future work.

II. BACKGROUND

In this Section, we first review the basic pattern of Bayesian computation and MCMC algorithm. Then we introduce various parallel MCMC algorithms and explain our choice for PMBA. A brief survey of Bayesian computing hardware is given in subsection *C*.

A. BAYESIAN COMPUTATIONS

The Bayesian computing is based on the Bayes' Theorem, which offers a general way to derive the posterior distribution of a random variable. It indicates that statistical inference can be made as follows:

$$P(\theta \mid D) = \frac{P(D|\theta)P(\theta)}{P(D)} \tag{1}$$

where D denotes the observed data and θ denotes a hypothesis (e.g. an underlying cause leading to D or a parameter) we want to infer from D. The concrete form of the posterior probability explaining a given problem is determined by both the likelihood and priori distributions. For example, in order to learn to write words as humans (i.e. deriving the posterior distribution of an alphabet through given sample letters), prior distributions related to stokes (e.g. stroke features, stroke spatial relations, and token control points) and likelihood distributions related to characters with given strokes can be built to model handwritten characters [8]. Note that such probabilistic models can be readily expressed as a probabilistic graphic model or Bayesian networks. The structure of this model can be either manually defined or learned from data.

The Bayesian inference is a powerful tool developed in the 18th century, but it is severely limited due to the frequent intractability of deriving a close-form analytical solution to the posterior distribution [17]. It became a widely used tool to the science community and industry until the invention of the Markov chain Monte Carlo (MCMC) method, which can approximate the target distribution. MCMC can be performed with many algorithms like importance sampling, Metropolis-Hastings (MH) algorithm, and Gibbs algorithm [18]. Figure 1 is an illustration of the MH algorithm. Samples of a given probabilistic distribution $p(x)$ are generated by a Markov chain. Starting from an original state X_0, the Markov chain transits to next state according to a transition probability

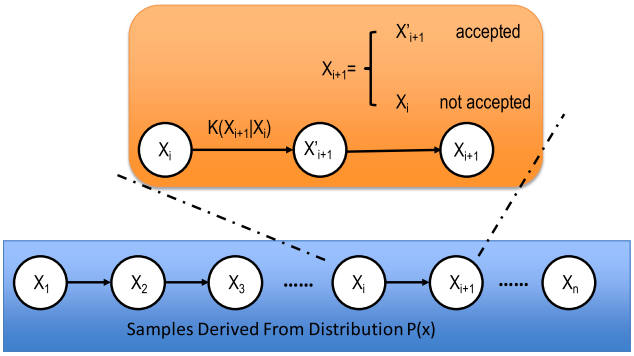


FIGURE 1. Metropolis-Hastings algorithm.

$K(X'_{i+1}|X_i)$. In other words, the state transitions happen with a chance as follow.

$$\alpha\left(X_i, X'_{i+1}\right) = min\left\{\frac{p(X'_{i+1})K(X_i|X'_{i+1})}{p(X_i)K(X'_{i+1}|X_i)}, 1\right\} \quad (2)$$

When a new state X'_{i+1} is generated from X_i, it is accepted with the rate $\alpha(X_i, X'_{i+1})$.

The Markov chain converges to $p(x)$ after a sufficient number of iterations under a probability of 1. The procedure can be highly time consuming with complicated probabilistic distributions. However, the original formulation of MCMC is inherently sequential. In fact, the fundamental data dependency is because every state of a Markov chain has to be determined by considering its last state. Significant research efforts have been dedicated to finding new algorithms better supporting parallel processing.

B. PARALLEL MCMC ALGORITHM

To improving computing performance, different approaches have been proposed to parallelize MCMC. One approach is to sampling multiple individual chains in parallel [19]. But it does not improve the performance of single-chain sampling. Moreover, multiple chains do not always follow the same instruction flows and thus the resultant load balance can be a concern. Another approach attempts to extract parallelization by sampling multiple states simultaneously [14], [20], while only one state is actually picked per iteration. Wang *et al.* [13] partitioned the data into subsets for parallelization sampling. This method only applies to a limited set of applications which have individual subset of data.

Recently, an efficient solution developed by Calderhead [15] is design to generate a joint set of new states in parallel for a joint probabilistic density. As listed in the pseudo code of Algorithm 1, N proposals are generated in parallel as the first step in each iteration using the transition function $K(x_i, x_{\setminus I})$. An extra random variable I is introduced as the index for choosing from proposals. The main task in this step is sampling new states for Markov chain based on random seeds. Given the transition probability of Markov chain, the new states are randomly generated from x_I according to inputs of random numbers. The second step is to calculate the stationary distribution of I based on the N proposals [15], given

$$P(I = k) \propto p(x_k)^* K(x_k, x_{\setminus k}) \quad (3)$$

This step can be put into parallel execution. The final step is to sample N values of I based on the stationary distribution calculated in the previous step. Then N new points are generated from proposals with the index. This algorithm makes it possible to sample multiple points for a single-chain in a single iteration. The number of proposals, N, can be arbitrarily assigned for a varying level of performance. The experimental results show that this algorithm achieve an effective sample size (ESS) rate of more than 80% and greatly increased speed given a task of generating 5000 samples [15]. Although the parallel paradigm is conceptually suitable for SIMD architectures, its performance is not satisfying on modern GPU platforms due to the frequent data synchronization among threads. During the execution on GPU, data transfer among the threads will harm the efficiency a lot for the reason that all the threads have to wait until such transfer ends. However, three times of synchronization occurred for each iteration of MCMC in Calderhead's algorithm. In this work, we choose this algorithm as the underlying MCMC algorithm for its potential to enable highly parallel hardware microarchitecture.

Algorithm 1

1: **INPUT**: state x_1, $I = 1$, $n = 0$ and N
2: For each iteration
3: **begin**:
4: **Parallel** $j = 1$: N
5: $x'_1 = x_I$
6: sample x'_{j+1} from $p(x'_{j+1}|x'_1)$
7: **Parallel** $k = 1$: N +1
8: calculate stationary distribution of I $p(I = k)$
9: normalize $p(I = j)$
10: **Parallel** $m = 1$: N
11 sample I_m from $p(I = I_m)$
12: $x_{n+m} = x'_{Im}$
13: $n = n + N$
14: $I = I_N$
15: **end**

C. HARDWARE ACCELERATOR FOR BAYESIAN COMPUTING

Different hardware microarchitectures have been proposed to improve the performance of MCMC based Bayesian computing. Mansinghka *et al.* implemented a Gibbs MCMC sampler by developing stochastic logic circuit that manipulates random bits with combinational gates for various distributions [21]. Mc3a [22] is a parallel ASIC accelerator for MCMC by extending the previous works [16], [23]. Mc3a adopts a Multiple Parallel Tempering algorithm to sample multiple chains simultaneously. A customized uniform Random Number Generator is installed in the ASIC. The goal of Mc3a is to realize high throughput sampling. but the performance of a single-chain is not changed. Moreover, only uniform random numbers are used in sampling, but it should be noted that in some cases random number of other distributions can remarkably increase the sampling efficiency. Causal-Learn [24] is a FPGA based architecture for probabilistic model learning by parallel sampling Gaussian processes with a Hamiltonian Markov Chain Monte Carlo algorithm. The single-chain MCMC sampling is not accelerated either. The algorithm proposed by Liu *et al.* [25], [26] mainly focuses on customized computing precision as well as accelerating likelihood evaluation in parallel, but does not accelerate the sampling itself. Mingas and Bouganis [27] developed a parallel tempering MCMC accelerator on FPGA where multiple chains are processed in parallel. In [12], a SIMD architecture

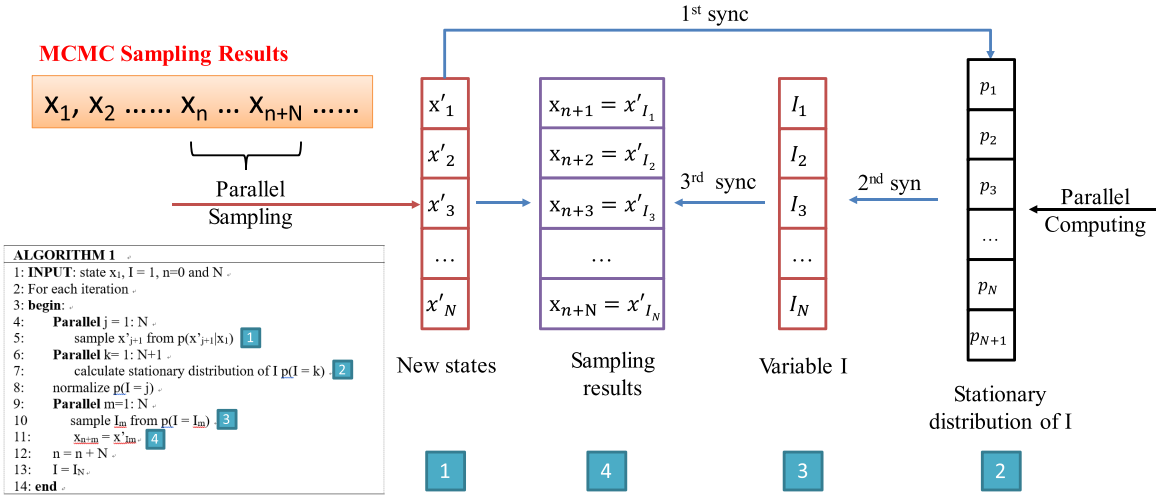


FIGURE 2. Execution flow of a single iteration in parallel MH MCMC.

is introduced for accelerating probabilistic models. Similar to CausalLearn, it is based on data-level parallelism and the sampling process is not accelerated. AcMC2 [28] presented a probabilistic model compiler to extract the parallelism in MCMC and implemented a hardware prototype accelerating sampling. The sampling task was partitioned into several parallel sub-chains through the Markov blanket concept that is used in AcMC2.

Notice that, none of the abovementioned works implement parallel sampling for a single MCMC chain. Instead, they leverage multiple chains or independent data for parallel execution. Besides, some works are limited to handle specific probabilistic distributions and cannot meet the need of general-purpose Bayesian computations.

III. THE MCMC COMPUTING FRAMEWORK BASED ON PMBA ARCHITECTURE

As the basis of a flexible parallel MCMC framework, PMBA is designed to efficiently generate sampling results for an arbitrary probabilistic distribution as required by a probabilistic program. A PMBA frontend first extracts MCMC models and parameters from a probabilistic program and then transforms the MCMC sampling process into codelets that can be regarded as a single instance of Metropolis-Hastings algorithm. These codelets are mapped to PMBA hardware for parallel execution.

In this section we will introduce the computing framework, the PMBA frontend and the overall architecture of PMBA respectively.

A. COMPUTING FRAMEWORK

PMBA is deliberately designed for high-performance MCMC with parallel execution of single-chain sampling. We adopt the parallel Metropolis-Hasting algorithm [15] introduced in Section II as the computing framework of PMBA. Figure 2 illustrates the fundamental execution flow of one round of processing. The first step is to concurrently sample n new states based on the results from the last iteration. Next, we calculate the stationary distribution of I with the $n+1$ values derived by synchronizing data from step 1. Then we sample I in parallel in step 3. Finally, we get the sampling results by combing the output from steps 1 and 3. These steps can be processed in parallel, but three synchronizations are required within a single iteration. Considering the fact that tens of thousands iterations are needed before converging to the target probabilistic distribution, the synchronization overhead has to be carefully handled.

In the overall computing framework, we split the four steps as four macroblocks so that the synchronization always happens at the boundary of macroblocks and no threads need to wait before reaching the end of a macroblock. In addition, Step 2, namely the computation of a stationary distribution, has a different degree of parallelism from other steps. The introduction of the macroblock makes it possible to process each step with a different hardware configuration.

Figure 3 shows the execution flow in terms of PMBA's hardware instructions for a single iteration of MCMC sampling. The core computing patterns within a single iteration consists of three kernels, sample kernel, computing kernel, and merge kernel. First, the sample kernel is called to generate N new states. An instruction of choosing random number source is introduced to select random numbers from either a uniform distribution or other specific distributions. The uniformly distributed random numbers can be produced directly by a random number generator, while other random numbers of distributions are derived with the on-chip stochastic gates similar to those proposed by Mansinghka *et al.* in [21]. Compared with the work in [21], PMBA extracts random numbers utilizing on-chip random source instead of using pre-stored random bit-stream. Second, the computing kernel to calculate $N+1$ stationary distribution parameters of I. Third, the sample kernel is called to generate N samples of I.

HIGH LEVEL MCMC INSTRUCTIONS FOR PMBA

1. Sample kernel (N, Generate New States)
   Load new state parameters
   Choose random number source
   Generate new states from random numbers
   Store new states
2. Computing kernel (N+1, Stationary Distribution)
   Computing probability
   Store stationary distribution of I
3. Sample kernel (N, Generate I)
   Load stationary distribution of I
   Generate value of I
   Store value of I
4. Merge kernel (N, Sampling Results)
   Load new states
   Load value of I
   Generate sampling results
   Store sampling results

FIGURE 3. Hardware instructions for single iteration sampling.

At step 4, the merge kernel is called to generate final sampling results. Instructions of different kernel will be sent to different hardware resource.

B. COMPILING FRONTEND OF PMBA

PMBA takes probabilistic programs as input. We design a frontend to handle probabilistic programs, BLOG in this work, and compile them into binary code instructions for hardware execution. Current probabilistic programming languages can be classified into two categories, extension of frequently used language such as PyMC [29] and Stan [30], and independent languages such as BLOG [31]. It is much more complex to develop a compiler for the first category of probabilistic languages because the baseline language is usually general-purposed and contains structures unrelated to probabilistic programing. The second category, on contrary, is more probabilistic-programming oriented and allows flexible development of our own compiler toolkit. In addition, algorithm-level transform is required in the frontend of PMBA. A given MCMC sampling procedure is first rewritten as parallel codelets and then mapped to PMBA as hardware instructions. We adopted the BLOG language as the programming language for PMBA due to its concise presentation of probabilistic model and relative ease of development of transformations. The PMBA frontend is developed as an extension of BLOG compiler [32] to meet the hardware demand.

As illustrated in Figure 4, the PMBA frontend performs three procedures on an input probabilistic program after parsing an input program.

1. Program splitting. The input BLOG program usually contains several segments sampling different probabilistic distribution (e.g. Poisson distribution and Gaussian distribution with varying parameters). PMBA frontend first divides the input program into codelets. A codelet targets a single distribution with fixed parameters. Each codelet will be executed individually on PMBA hardware. As some codelets can have data dependency on other codelets, the PMBA frontend tags such dependency for hardware controller to avoid wrong order of execution.
2. MCMC parallelism extraction. In this procedure, the PMBA frontend turning codelets derived from the first procedure into parallel binary instructions. The execution flow of each sampling block is supported by 5four hardware blocks as illustrated in Figure 3.
3. Data Mapping. In this procedure the PMBA frontend orchestrates the input and output data. As the input of some codelets comes from previously executed codelets, PMBA frontend allocated all data addresses in this procedure to ensure every codelet to get the right input.

With the above three procedures, the PMBA frontend splits input BLOG program as several parallel MCMC sampling blocks, and compiles them into hardware functions for PMBA hardware execution.

C. OVERALL MICROARCHITECTURE

The proposed PMBA microarchitecture largely follow a single instruction multiple thread (SIMT) paradigm. We extend a SIMT instruction set [33] with various operations in sampling, including choosing and loading random number source, loading MCMC states, generating sampling points and storing sampling results, to fully support MCMC process.

As illustrated in Figure 5, the PMBA microarchitecture is organized as a pipeline consisting of eight SIMD sampling units, one SIMD computing unit, one merge engine, and a random number generator together with a specialized buffer. The processing time of sampling units is around 6-8 times as much as computing units depending on specific applications. In this work eight sampling units are employed to avoid waste of hardware resource. Multiple MCMC sampling tasks can be processed simultaneously in pipeline on PMBA.

A sampling unit contains 32 sampling cores with each processing a single sampling thread. A sampling function mentioned in Section III can be mapped to one or more sampling units. All sampling cores are invoked in a SIMD manner.

There is only one computing unit used for the computing of a stationary distribution. It contains 32 SIMD computing cores. A merge engine is introduced to generate the final sampling results and thus offload the only computing unit. The merge engine works in sequential because the underlying operations are relatively simple.

We deploy a commercial IP core of random number generator [34] in this design to generate uniformly distributed bit streams. A random number buffer is added to store random numbers from the random number generator and supply random numbers to sampling units when needed. Each sampling unit has a random number file that can load uniform random

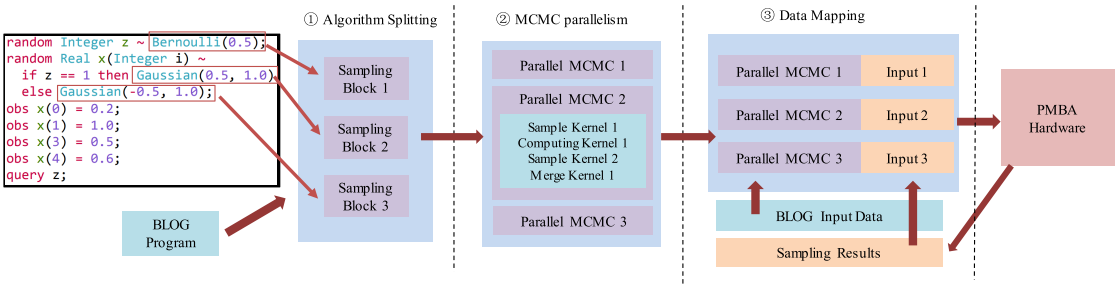


FIGURE 4. Frontend of PMBA.

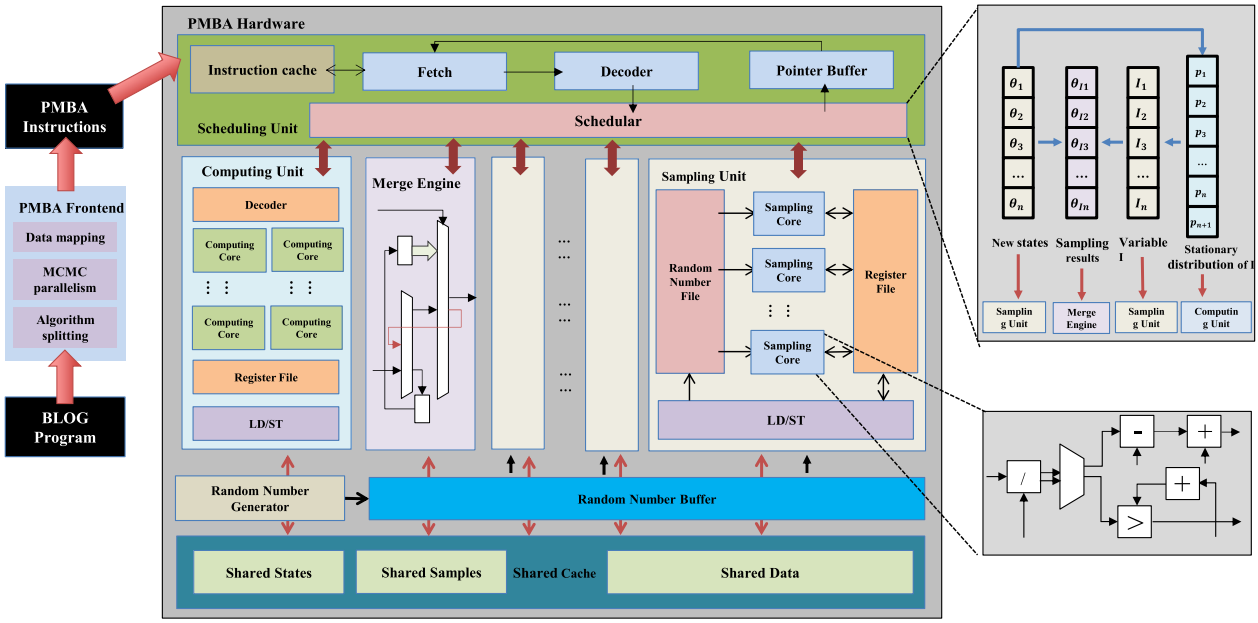


FIGURE 5. Overall microarchitecture of PMBA.

numbers from the random number buffer or specific random numbers from the shared cache.

All the units have access to a shared cache. The cache is divided into two regions, one reserved for shared states including new states and samples of I, and the other for shared samples of specific distributions.

A scheduling unit coordinates the overall computing resource by sending instructions of different kernels to corresponding sampling unit, computing unit or merge engine. Multiple tasks can be scheduled to PMBA to avoid idle of hardware resource. This makes it possible to run multiple MCMC chains on PMBA. While the actual MCMC processing may require more parallelism than PMBA hardware resource can provide, the scheduling unit realizes that by reusing the sampling units and computing unit.

IV. DESIGN DETAILS OF PMBA MICROARCHITECTURE

In this section, we elaborate the design details for implementing the microarchitecture of PMBA.

A. SCHEDULING UNIT

The detailed design of the scheduling unit is shown in Figure 6. It is in charge of coordinating instructions processing. The major components of the scheduling unit are as follows.

- Instruction cache: The instructions are kept in the instruction cache. Each task has its independent storage area and up to eight tasks can be supported.

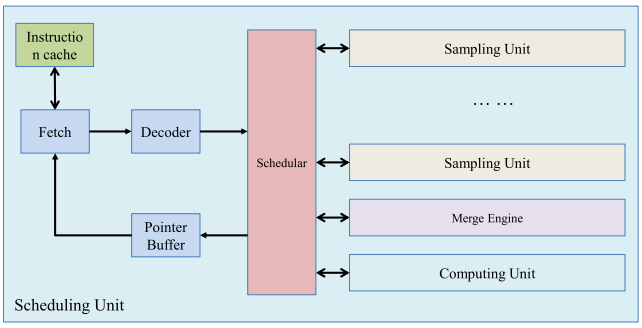


FIGURE 6. Scheduling unit.

- Fetch module: The fetch module keeps an instruction pointer for every task. The fetch module asks for instruction according to the pointer one task by one task, unless the instruction stack module feeds back a wait signal for current task.
- Instruction decoder: Fetched instructions are decoded by the decode module to classified for sampling, general computing or result merging.
- Scheduler: According to the results of decoder, the scheduler module performs the following actions. When a sampling instruction comes, scheduler module gives the instruction to available sampling units. If the available sampling units cannot meet the parallelism required by current tasks, the instruction is kept in FIFO buffer and dispatched once there is more available sampling unit. When a computing instruction comes, the scheduling model instead give the instruction pointer to the computing unit and set the pointer to end of current computing task. The computing unit uses its own fetch and decode pipeline for the rest of the computing task. The advantage of this mechanism is to avoid occupying the scheduler module for too long. In fact, the computing unit decodes faster because it only fetches from one task. Merging instructions are given to merge engine. Note that only one merge instruction is need to start the merge procedure. Every time an instruction is dispatched by scheduler module, the current pointer is given to pointer buffer. It means that this task must wait until the former instruction is processed. When the sampling unit, the computing unit or the merge engine returns a termination signal, the scheduler module will ask the pointer buffer to pop the pointer of current task.
- Pointer Buffer: It sends a wait signal for tasks which have an instruction pointer stored in the pointer buffer.

The scheduling pipeline is able to map multiple tasks to PMBA so as to efficiently utilize the computing resources.

B. PROCESSING RESOURCE

PMBA has three different types of processing resource, sampling unit, computing unit, and merge engine.

The sampling unit is the basic building block to implement sampling functions. The decoded operation comes directly from the scheduling unit. The sampling processing of proposed parallel MCMC algorithm offers sufficient data-level parallelism, so PMBA chooses a SIMD pattern to implement parallel execution. 32 sampling cores are installed in one sampling unit and they always follow the same processing flow but process different data.

A new state x_i is generated from the last state x_{i-1} in a Markov Chain according to the transition kernel as follows.

$$K(x_i, x_{i-1}) = p(x_i|x_{i-1}) \tag{4}$$

A random number from $p(\mathrm{x_i}|\mathrm{x_{i-1}})$ is needed to decide the possibility of drawing x_i. To improve computing efficiency, a dedicated register file is used to store the required random numbers and supply them to sampling cores. The capacity of the register file is set to hold 64 random numbers for 32 sampling cores in order to reduce latency. The random number can follow either uniform distribution or a specific one. They are loaded from random number buffer or shared samples cache through the load/store unit.

Figure 7 illustrates the detailed design of the sampling core. A sampling core has two function, generating new states of Markov Chain, and sample value of index I. Given the parameters needed for calculation (for example, $\mathrm{r_{max}}$ in equation 2), a decoder decides which function to calculate results. To generate a new state, we use a random walk transformation:

$$x_i = r/\frac{r_{max}}{p} - p + x_{i-1} \tag{5}$$

where r represents the random number, r_{max} represents the max value of random number and parameter p represents the maximum random walk step. A symmetric random walk is added to $\mathrm{x_{i-1}}$ to generate new state $\mathrm{x_i}$.

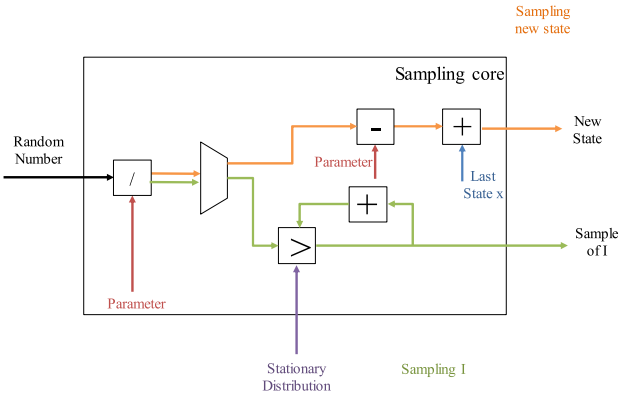


FIGURE 7. Design of sampling core.

In order to derive a sample of I from the stationary distribution, the sampling core first calculates $\mathrm{r/r_{max}}$ as a normalized random number. Then the result is compared with accumulated stationary distribution probabilities of I to decide the value of sampled I.

The computing unit is responsible for calculating the stationary distribution of I. Here PMBA adopt a 32-core general-purpose SIMD pipeline based on our previous work [33] running hardware instructions. The computing unit has its own decoder so the instructions from computing kernels can be directly dispatch to computing unit once for all.

When all the new states and samples of I are ready, the scheduling unit invokes the merge engine. Sampling results are chosen from new states according to samples of I. As shown in Figure 8, a load unit is able to load all new states to registers in merge engine with load instruction occur. As a result, the merge instruction actually initializes a vector operation.

C. RANDOM NUMBER BUFFER

In the process of parallel MCMC sampling on PMBA, each sampling core is allocated to one parallel thread. For every

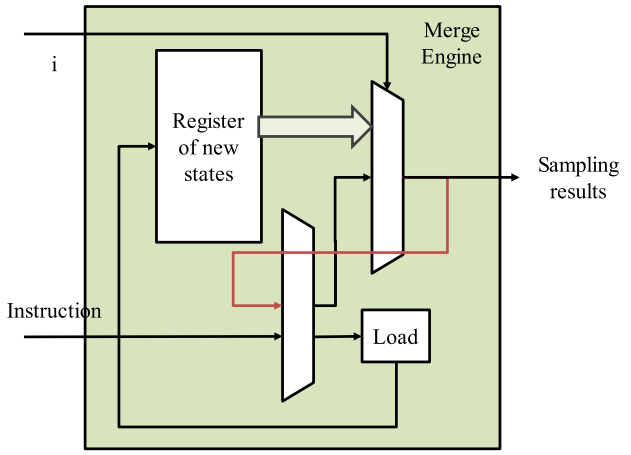


FIGURE 8. Merge engine.

iteration of sampling, two random numbers are used by each sampling core running a single thread. The massive usage of random numbers requires a robust source of randomness in terms of quality and efficiency of generated random numbers. For this reason, PMBA is equipped with dedicated IP cores for random number generation together with customized data path and memory blocks so that random distributions can be sampled by sampling cores and then stored on-chip for succeeding usage.

The sequential generating rate of random number generator is not sufficient when all the sampling units are requiring random numbers. So we implement a random number buffer continually drawing random numbers from the generator when there is storage available. Each sampling unit has its own area in the random number buffer.

D. SHARED CACHE

As illustrated in Figure 5, the shared cache offers storage space that can be used by all processing resource. There are three banks in shared cache for shared states, shared samples and shared data respectively. New states and sampled values of I are kept in shared states bank, they are in fact the synchronization data of the original parallel MH MCMC algorithm. Shared sample bank stores the sample results that will be used as random number input for sampling units in future process. Other shared data are sent to the shared data bank.

V. EVALUATION OF PMBA

In this section, we evaluate the implementation and applications of the PMBA microarchitecture. An FPGA implementation of PMBA is presented in subsection *A*. Then we evaluate the performance of the implementation for Metropolis-Hastings MCMC on PMBA compared with multi-core CPU and GPU in subsection *B*. We further implement a visual learning task on PMBA to which will be discussed in subsection *C*.

TABLE 1. PMBA configuration on the FPGA implementation.

FPGA Model	Xilinx VC709
Clock rate	400MHz
Random number generator	800Mbps
Random number buffer	1KB
Sampling Unit	
Register file	8KB
Random number register file	128B
Shared cache	48KB
Shared state bank	16KB
Shared samples bank	16KB
Shared data bank	16KB

A. FPGA PROTOTYPING

A prototyping PMBA microarchitecture is implemented on a Xilinx VC709 development board with a clock rate of 400MHz. The rate of random number generator is 512Mbps. It takes 2000 cycles to fill a 1KB random number buffer with on-chip generated random numbers. The capacity of register file for sampling unit is 256B. i.e., eight 32-bit registers for each sampling core. The random number register file inside a sampling unit has a size of 128B and holds sixty-four 32-bit random numbers. The volume of the shared cache is 48KB. The sampling unit is also equipped with a 16KB shared states bank, a 16KB shared sample bank, and a 16KB shared data bank. 16K random numbers can be stored in the shared samples bank for MCMC sampling on PMBA.

Table 2 shows the resource usage of FPGA hardware for implementing the PMBA microarchitecture. This summary indicates that PMBA uses a notably high ratio of block RAMs resource. Considering that the utilization of block RAMs can be replaced by using CLBs (including LUTs and Flipflops) in FPGA, it is feasible to implement more sampling or computing units.

TABLE 2. Usage of FPGA resource.

Type	LUTs	Flipflops	Block RAMs
Sampling Unit	4712	9561	32
Merge Engine	79	31	0
Computing Unit	8421	11749	16
RNG	5436	8239	10
Rest of Design	2816	3331	480
Overall	21464	32911	538
Usage	39.4%	32.9%	70.6%

B. PERFORMANCE EVALUATION

We first implement a single parallel Metropolis-Hastings algorithm on both PMBA and a Titan X acceleration card (Pascal GPU) using CUDA language. Then we evaluate

several BLOG benchmarks on PMBA to compare with previous works.

The number of parallel threads is configurable in the parallel MCMC algorithm. So, we first evaluate the performance of MCMC sampling at different configuration of parallel threads against GPU. The threads here indicate how many samples are generated in parallel for a single iteration of MCMC. Each configuration is evaluated with 300 runs of execution. Two version of algorithm are processed, we use random walk to sample points of Beta random numbers and Gaussian random numbers. The results in Figure 9 show that PMBA outperforms the GPU by a factor of 17x-21x. Noticed that it is capable to implement more threads, but the computing complexity during synchronization of parallel MCMC algorithm grows when proposing more threads. As a result, the optimal choice of number of proposals N varies with different tasks [15]. So we make the paralleled threads configurable and choose 1024 as the maximum of paralleled threads as evaluated in the algorithm research [15].

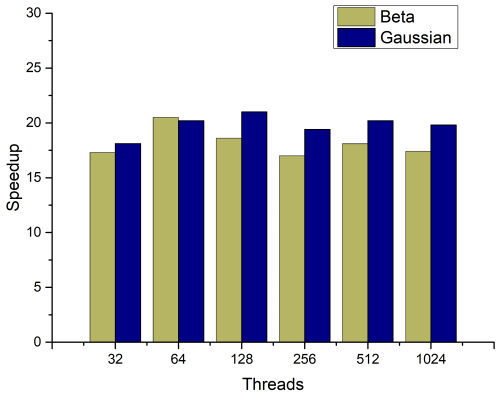


FIGURE 9. Speedup of PMBA against GPU sampling with Beta sampling and Gaussian sampling.

We then evaluate the performance of BLOG benchmarks [32] on PMBA hardware and Figure 10 shows the results. In order to compare the performance intuitively, we follow AcMC2 [28] and normalize the performance of the Swift microarchitecture [32] as 1. It can be inferred that the PMBA hardware improves the performance from 3.6 to 10.3 over AcMC2. There is difference that AcMC2 uses Power8 CPU and in this work we evaluate the performance against Xeon E7 CPU. These benchmarks contain sampling tasks of both continuous and discrete probability distributions as well as random variables with conditional dependencies, which verifies the ability of PMBA to handle general purpose probabilistic computing tasks.

C. EXEMPLAR BASED VISUAL CATEGORY LEARNING

The benchmarks in Section V testify the performance advantage of PMBA. In this sub-section, we evaluate the efficiency and effectiveness of PMBA for a complex statistical learning problem, visual category learning. The problem can be formulated as extracting the relationship among categories

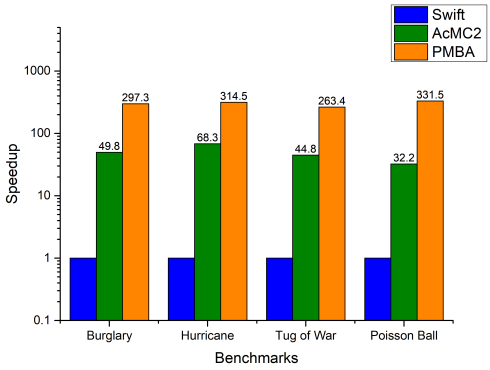


FIGURE 10. BLOG benchmarks on PMBA.

from natural images of multiple classes of objects. Basically, a category of visual stimuli corresponds to certain concepts, while the features for different categories reflect relationship among concepts. For instance, the images in the ImageNet dataset [35] are conceptually organized on the basis of WordNet [36] with a total of 21841 concepts following a complex hierarchical structure.

In this work, we use the classical exemplar based visual category learning framework [37]. In this framework, concepts are represented and memorized with the exemplar model. The basic idea of exemplar model is to model a concept with a few representative instances that cover the feature space of the concept as much as possible. First, images are represented by parameters extracted from several local features [38]. Then we pick instances $(I_{i1}, I_{i2}, \ldots I_{im})$ for exemplar E_i of a concept C_i through a Gaussian mixture model described in (3).

$$E_i \; of \; Concept \; C_i \sim \{\eta_{i1} * Gaussian\,(I_{i1}) + \ldots + \eta_{im} * Gaussian\,(I_{im})\,(\eta > 0.1) \quad (6)$$

We find E_i that best fits the data of concept C_i by running MCMC on PMBA. After getting the exemplars for all the concepts, we extract the structural relationship as follow.

Given MCMC sampling results S_i and S_j of exemplar E_i and E_j, θ_{ij} represents the ratio of sampling points in S_i that are also results of S_j

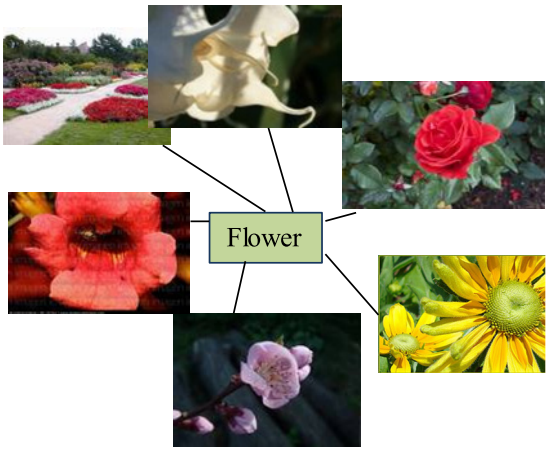
$$\theta_{ij} = (S_i \cap S_j)/S_i \quad (7)$$

If we can get j satisfying

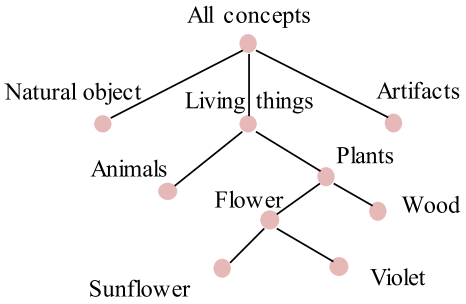
$$j = argmax\,(\theta_{ji})\,(\theta_{ij} > \delta) \quad (8)$$

then C_i is a sub-concept of C_j (δ is the parameter of the threshold of relationship, when $\theta_{ij} > \delta$, C_i can be seen as the child-concept of C_j).

We use the images from ImageNet [35] as the training data. In this work, we are only interested at the 971 concepts of natural object, living things, and artifacts. As shown in Figure 11, we first learn the exemplar of each concept by sampling a Gaussian mixture model [39] due to its performance in feature-based image learning. Given the relation of 300 concepts defined at higher levels in WordNet, we then



(a) Learning Exemplar



(b) Learning Category Relation

FIGURE 11. Exemplar based visual category learning.

learn the relationship of the remaining 671 concepts. The accuracy rate of sampling new concept to correct direct parent concept and to correct parent concept, is respectively 37.8% and 71.2%. The PMBA outperforms a Xeon E7 CPU by a factor of 60 in terms of learning time.

VI. CONCLUSION AND FUTURE WORK

In this work, we propose a Bayesian computing engine with native support for high-efficiency MCMC sampling as well as the corresponding computing framework for automatic parallelization and mapping of probabilistic programs. Leveraging the concurrency extracted by a parallel Metropolis-Hastings algorithm, the proposed PMBA architecture supports fully parallelized single-chain sampling of arbitrary probabilistic distributions. With the help of a compiling frontend for automatic parallelism extraction, PMBA can execute probabilistic programs for a wide range of statistical learning applications. The microarchitecture supports efficient execution of MCMC sampling based probabilistic computations. Experimental results demonstrate the significant performance advantages of PMBA over state-of-the-art GPUs. Comparison against previous dedicated Bayesian computing solutions proves that PMBA enables a performance improvement of up to over 10 times.

There still remains a large space to explore high performance Bayesian computing microarchitectures. First, PMBA adopts random walks to generate new states in the MCMC sampling process instead of directly transforming to new state, because directly transforming requires more complex hardware. It is worth exploring efficient design of such hardware. Second, PMBA only supports Metropolis-Hastings MCMC algorithm. It will be more flexible for PMBA if parallel Gibbs and other more advanced MCMC algorithm can be implemented. Third, as we are still at the dawn of probabilistic computing, the design space of basic building blocks like sampling unit and computing unit need extensive refining.

REFERENCES

[1] R. A. Jacobs and J. K. Kruschke, "Bayesian learning theory applied to human cognition," *Wiley Interdiscipl. Rev., Cognit. Sci.*, vol. 2, no. 1, pp. 8–21, Jan. 2011.
[2] J. Li, J. M. Bioucas-Dias, and A. Plaza, "Hyperspectral image segmentation using a new Bayesian approach with active learning," *IEEE Trans. Geosci. Remote Sens.*, vol. 49, no. 10, pp. 3947–3960, Oct. 2011.
[3] D. Ramage, D. Hall, R. Nallapati, and C. D. Manning, "Labeled LDA: A supervised topic model for credit attribution in multi-labeled corpora," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, vol. 1, Aug. 2009, pp. 248–256.
[4] D. Kersten, P. Mamassian, and A. Yuille, "Object perception as Bayesian inference," *Annu. Rev. Psychol.*, vol. 55, no. 1, pp. 271–304, Feb. 2004.
[5] F. Xu and J. B. Tenenbaum, "Word learning as Bayesian inference," *Psychol. Rev.*, vol. 114, no. 2, p. 245, 2007.
[6] W. J. Ma, M. Husain, and P. M. Bays, "Changing concepts of working memory," *Nature Neurosci.*, vol. 17, no. 3, p. 347, 2014.
[7] K. P. Körding and D. M. Wolpert, "Bayesian integration in sensorimotor learning," *Nature*, vol. 427, no. 6971, p. 244, 2004.
[8] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum, "Human-level concept learning through probabilistic program induction," *Science*, vol. 350, no. 6266, pp. 1332–1338, Dec. 2015.
[9] D. C. Knill and A. Pouget, "The Bayesian brain: The role of uncertainty in neural coding and computation," *Trends Neurosci.*, vol. 27, no. 12, pp. 712–719, Dec. 2004.
[10] M. Jones and B. C. Love, "Bayesian fundamentalism or enlightenment? On the explanatory status and theoretical contributions of Bayesian models of cognition," *Behav. Brain Sci.*, vol. 34, no. 4, p. 169, 2011.
[11] J. M. Hernández-Lobato and R. Adams, "Probabilistic backpropagation for scalable learning of Bayesian neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 1861–1869.
[12] A. S. Mahani and M. T. A. Sharabiani, "SIMD parallel MCMC sampling with applications for big-data Bayesian analytics," *Comput. Statist. Data Anal.*, vol. 88, pp. 75–99, Aug. 2015.
[13] X. Wang, F. Guo, K. A. Heller, and D. B. Dunson, "Parallelizing MCMC with random partition trees," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 451–459.
[14] A. E. Brockwell, "Parallel Markov chain Monte Carlo simulation by pre-fetching," *J. Comput. Graph. Statist.*, vol. 15, no. 1, pp. 246–261, Mar. 2006.
[15] B. Calderhead, "A general construction for parallelizing Metropolis–Hastings algorithms," *Proc. Nat. Acad. Sci. USA*, vol. 111, no. 49, pp. 17408–17413, 2014.

[16] L. Marni, M. Hosseini, J. Hopp, P. Mohseni, and T. Mohsenin, "A real-time wearable FPGA-based seizure detection processor using MCMC," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–4.
[17] J. A. Hartigan, "Bayes theory," *Bull. Amer. Math. Soc.*, vol. 12, pp. 294–297, 1985.
[18] W. K. Hastings, "Monte Carlo samplingmethods using Markov chains and their applications," *Biometrica*, vol. 57, no. 1, p. 97, 1970.
[19] J. S. Rosenthal, "Parallel computing and Monte Carlo algorithms," *Far East J. Theor. Statist.*, vol. 4, no. 2, pp. 207–236, 2000.
[20] R. V. Craiu and C. Lemieux, "Acceleration of the multiple-try Metropolis algorithm using antithetic and stratified sampling," *Statist. Comput.*, vol. 17, no. 2, p. 109, 2007.
[21] V. K. Mansinghka, E. M. Jonas, and J. B. Tenenbaum, "Stochastic digital circuits for probabilistic inference," Massachusetts Inst. Technol., Cambridge, MA, USA, Tech. Rep. MITCSAIL-TR 2069, 2008.
[22] L. Marni, M. Hosseini, and T. Mohsenin, "MC3A: Markov chain monte carlo manycore accelerator," in *Proc. Great Lakes Symp. VLSI*, May 2018, pp. 165–170.
[23] M. Hosseini, R. Islam, A. Kulkarni, and T. Mohsenin, "A scalable FPGA-based accelerator for high-throughput MCMC algorithms," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2017, p. 201.
[24] B. Darvish Rouhani, M. Ghasemzadeh, and F. Koushanfar, "Causalearn: Automated framework for scalable streaming-based causal Bayesian learning using FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 1–10.
[25] S. Liu, G. Mingas, and C.-S. Bouganis, "An unbiased MCMC FPGA-based accelerator in the land of custom precision arithmetic," *IEEE Trans. Comput.*, vol. 66, no. 5, pp. 745–758, May 2017.
[26] S. Liu, G. Mingas, and C.-S. Bouganis, "An exact MCMC accelerator under custom precision regimes," in *Proc. Int. Conf. Field Program. Technol. (FPT)*, Dec. 2015, pp. 120–127.
[27] G. Mingas and C.-S. Bouganis, "Population-based MCMC on multi-core CPUs, GPUs and FPGAs," *IEEE Trans. Comput.*, vol. 65, no. 4, pp. 1283–1296, Apr. 2016.
[28] S. S. Banerjee, Z. T. Kalbarczyk, and R. K. Iyer, "AcMC^2: Accelerating Markov chain Monte Carlo algorithms for probabilistic models," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 515–528.
[29] A. Patil, D. Huard, and C. J. Fonnesbeck, "PyMC: Bayesian stochastic modelling in Python," *J. Stat. Softw.*, vol. 35, no. 4, p. 1, 2010.
[30] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell, "Stan: A probabilistic programming language," *J. Stat. Softw.*, vol. 76, no. 1, pp. 1–32, 2017.
[31] B. Milch, B. Marthi, and S. Russell, "BLOG: Relational modeling with unknown objects," in *Proc. ICML Workshop Stat. Relational Learn. Connections Other Fields*, 2004, pp. 67–73.
[32] Y. Wu, L. Li, S. Russell, and R. Bodik, "Swift: Compiled inference for probabilistic programming languages," 2016, *arXiv:1606.09242*. [Online]. Available: http://arxiv.org/abs/1606.09242
[33] K. Fang, Y. Ni, J. He, Z. Li, S. Mu, and Y. Deng, "FastLanes: An FPGA accelerated GPU microarchitecture simulator," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, Oct. 2013, pp. 241–248.
[34] B. Sunar, W. Martin, and D. Stinson, "A provably secure true random number generator with built-in tolerance to active attacks," *IEEE Trans. Comput.*, vol. 56, no. 1, pp. 109–119, Jan. 2007.
[35] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015.
[36] G. A. Miller, "WordNet: A lexical database for English," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, 1995.
[37] D. L. Medin and M. M. Schaffer, "Context theory of classification learning," *Psychol. Rev.*, vol. 85, no. 3, p. 207, 1978.
[38] G. Chechik, V. Sharma, U. Shalit, and S. Bengio, "Large scale online learning of image similarity through ranking," *J. Mach. Learn. Res.*, vol. 11, pp. 1109–1135, Jan. 2010.
[39] H. Permuter, J. Francos, and I. Jermyn, "A study of Gaussian mixture models of color and texture features for image classification and segmentation," *Pattern Recognit.*, vol. 39, no. 4, pp. 695–706, Apr. 2006.

YUFEI NI received the B.S. degree in electronics engineering from Tsinghua University, China, in 2013, where he is currently pursuing the Ph.D. degree in electronics engineering with the Institute of Microelectronics. His research interests include parallel computing architecture and artificial intelligence accelerating methods.

YANGDONG DENG (Senior Member, IEEE) received the B.E. and M.S. degrees in electronic engineering from Tsinghua University, Beijing, China, in 1995 and 1998, respectively, and the Ph.D. degree in electrical and computer engineering with Carnegie Mellon University, Pittsburgh, PA, USA, in 2006. From 2005 to 2008, he was a Consulting Technical Staff with Magma Design Automation. From 2008 to 2013, he was an Associate Professor with the Institute of Microelectronics, Tsinghua University, where he has been an Associate Professor with the School of Software, since 2013. His research interests include computer architecture and predictive maintenance. His awards and honors include the ECE Fellowship (Department of Electrical and Computer Engineering, Carnegie Mellon University), the Best Paper Award of 2013 International Conference on Computer Design, and the NVIDIA Professor Partnership Award.

SONGLIN LI received the B.S. degree with the Department of Microelectronics and Nanoelectronics, Tsinghua University, where he is currently pursuing the M.S. degree with the Department of Microelectronics and Nanoelectronics. His research interests include computer architecture, brain-inspired computing, and neuromorphic computing.