FPGA-Based Acceleration of MCMC Algorithm through Self-Shrinking for Big Data

Shuanglong Liu, Shiyu Peng, Wan Shen
Key Laboratory of Low-Dimensional Quantum Structures and Quantum Control
Key Laboratory of Physics and Devices in Post-Moore Era, College of Hunan Province
Hunan Normal University, Changsha, China, *liu.shuanglong@hunnu.edu.cn*

***Abstract*—Markov chain Monte Carlo (MCMC) algorithms are widely used in Bayesian inference to compute the posterior distribution of complex models, facilitating sampling from probability distributions. However, the computational burden of evaluating the likelihood function in MCMC poses significant challenges in big data applications. To address this, sub-sampling methods have been introduced to approximate the target distribution by using subsets of the data rather than the entire dataset. Unfortunately, these methods often lead to biased samples, making them impractical for real-world applications. This paper proposes a novel scaling MCMC method that achieves exact sampling by utilizing a subset (mini-batch) of the data with locally bounded approximations of the target distribution. Our method adaptively adjusts the mini-batch size by automatically tuning a hyperparameter based on the sample acceptance ratio, ensuring optimal balance between sample efficiency and computational cost. Moreover, we introduce a highly optimized hardware architecture to efficiently implement the proposed MCMC method onto FPGA. Our accelerator is evaluated on an AMD Zynq UltraScale+ FPGA device using a Bayesian logistic regression model on the MNIST dataset. The results demonstrate that our design achieves unbiased sampling with a 47.6× speedup over the standard MCMC design, while also significantly reducing estimation errors compared to state-of-the-art MCMC methods.**

I. Introduction

Bayesian inference is a powerful tool for analyzing complex, structured data, with Markov chain Monte Carlo (MCMC) being the standard for sampling from challenging probabilistic models [1]. However, applying MCMC sampling to large datasets is often prohibitively computationally intensive, rendering it impractical for real-world use [2]. This challenge arises from the repeated need to compute the costly likelihood function across the entire dataset. As data volumes increase, traditional MCMC methods face significant performance bottlenecks, with the computational complexity and runtime of MCMC simulations escalating exponentially. This greatly restricts the applicability of MCMC methods in big data applications [3].

To enhance the scalability of MCMC for large datasets, researchers have developed subsampling techniques [4]–[8] that estimate the likelihood using data subsets rather than the full dataset. While these approaches reduce computational demands, they often introduce asymptotic bias, compromising accuracy for speed [9], making them unsuitable for exact inference [10]. On the other hand, existing exact subsampling methods [6], [7] often require restrictive conditions on the target distribution, making them impractical for real-world use.

In this paper, we introduce the Self-Shrinking MCMC (SS-MCMC) method, inspired from the foundational idea in [9]. Our approach employs a self-shrinking mechanism to dynamically adjust batch size based on local distribution bounds, achieving exact sampling while balancing batch size and convergence rate. We also propose a tailored hardware architecture for SS-MCMC, featuring a pipelined design that minimizes data transfer overhead through prefetching and efficient on-chip memory use.

The main contributions of this work are:

- A novel exact subsampling MCMC algorithm with self-shrinking of the batch size, which ensures unbiased estimates and an optimal balance between speed and sample efficiency (Section III);
- A highly optimized FPGA hardware architecture designed to efficiently implement the proposed MCMC algorithm, leveraging on-chip memory blocks for optimal performance (Section IV).

II. Background and Related Work

A. Markov chain Monte Carlo

MCMC methods generate dependent samples by evolving a Markov chain designed to converge to the target distribution. Among these methods, the Metropolis-Hastings (MH) algorithm [11], as shown in Code 1, is one of the most widely used. Consider the parameters of interest, denoted by θ, a vector of dimension D, with N observed data points $\{x_n\}_{n=1}^N$. MCMC samplers facilitate the transition from the current state θ to a new proposed state θ', ensuring the preservation of the posterior distribution $\pi(\theta|\{x_n\}_{n=1}^N)$ throughout the process. In each iteration, the MH algorithm generates a proposal using a Gaussian random walk (line 2) and stochastically accepts or rejects it based on the probability ratio a (line 3). The main computational cost arises from evaluating the full posterior likelihood with the respective parameter in line 3.

Using Bayes' theorem and assuming that the data points $\{x_n\}_{n=1}^N$ are independently and identically distributed (*i.i.d.*), a common assumption, along with a prior distribution $p(\theta)$, the posterior distribution can be expressed as the product of the likelihoods of the individual data points conditioned on θ:

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

978-3-9826741-0-0/DATE25/© 2025 EDAA

Code 1 Metropolis-Hastings (MH) MCMC

Input: initial setting θ_0, number of samples N_s;
Output: samples θ_i for $i = 1, \ldots, N_s$;

for ($i = 1$ to N_s) **do**
  Propose $\theta' \sim \theta_{i-1} + \mathcal{N}(0, \sigma^2 I)$;
  Compute $a = \frac{\pi(\theta'|\{x_n\}_{n=1}^N)}{\pi(\theta_{i-1}|\{x_n\}_{n=1}^N)}$;
  $u \sim \text{Uniform}(0, 1)$;
  if $u \leq a$ **then**
    Accept proposal: $\theta_i \leftarrow \theta'$;
  else
    Reject proposal: $\theta_i \leftarrow \theta_{i-1}$;
  end if
end for

For simplicity, we define the component of the log-likelihood functions $U_i(\theta)$ as follows:

$$U_i(\theta) = -\log p(x_i|\theta) \tag{2}$$

Then, the target posterior distribution can be rewritten as:

$$\pi(\theta|\{x_n\}_{n=1}^N) \propto \exp\left(-\sum_{i=1}^{N} U_i(\theta)\right) \tag{3}$$

In big data settings, calculating all N likelihoods becomes the main computational bottleneck. Therefore, this work focuses on strategies to minimize the computational burden of these evaluations while maintaining the accuracy of the MCMC estimate.

B. MCMC Output Analysis

MCMC samples are correlated due to the nature of the Markov chain, which results in an increased asymptotic variance of the MCMC estimate compared to using independent samples. This inefficiency is measured by the Effective Sample Size (ESS), defined as:

$$ESS = N_s/(1 + 2\sum_{j=1}^{k} \rho(j)) \tag{4}$$

where N_s is the number of post burn-in MCMC samples, and $\sum_{j=1}^{k} \rho(j)$ is the sum of the first k autocorrelations. ESS measures the "effective" number of independent samples, and the performance metric typically used for MCMC samplers is ESS per second, which takes into account both raw sampling speed (wall-clock time) and ESS.

While MCMC methods are theoretically asymptotically unbiased, subsampling can introduce bias when only a portion of the dataset is used to speed up likelihood estimation. These approximate methods sacrifice some accuracy for faster computation, resulting in trade-offs between bias and variance. To compare MCMC algorithms, the error in the estimate is often measured by the risk [6], defined as the mean squared error (MSE). The practical goal of MCMC is to minimize risk within a given execution time.

C. Related Work

MCMC Algorithm Optimization: Scaling MCMC methods for large datasets has led to various acceleration techniques, primarily divided into Consensus Monte Carlo (CMC) and subsampling-based algorithms. CMC methods [12] partition the dataset into batches, run MCMC on each batch separately, and then combine the results to approximate the target posterior distribution. However, merging these results effectively is challenging and lacks theoretical convergence guarantees [3].

Subsampling methods are popular for their efficiency, as they use only a portion of the dataset to estimate the likelihood, speeding up computations. Korattikara *et al.* [4] and Bardenet *et al.* [5] introduced approximate MH rules that control bias by accepting or rejecting samples based on partial data. However, they rely on bounding the difference in log-likelihood contributions, which can be restrictive. Maclaurin *et al.* [6] developed Firefly MC, an efficient subset-sampling method, but it demands a strict lower bound on the likelihood function, which is challenging to establish and lacks theoretical guarantees. Quiroz *et al.* [8] proposed a block-Poisson estimator for exact subsampling MCMC, suitable for large datasets but requiring importance sampling correction due to its tendency to produce negative values. Zhang *et al.* [9] introduced TunaMH, an exact minibatch method that balances batch size and convergence rate, but requires careful hyperparameter tuning, particularly in high-dimensional settings.

FPGA-Based MCMC Acceleration: Leveraging the parallel processing capabilities of FPGAs, several advancements have been made in FPGA-based MCMC accelerators. Mingas *et al.* [13] developed a population-based MCMC framework for multi-core CPUs, GPUs, and FPGAs, though precision loss during probability density evaluation posed challenges, necessitating complex optimizations to prevent biased sampling. Liu *et al.* [10] introduced an unbiased FPGA-based MCMC accelerator with custom precision arithmetic, and further improved on this with a CA-MCMC method [2] that reduced memory access issues for big data. However, the dynamic reallocation of data across memory hierarchies adds design complexity. Shukla *et al.* [14] presented the MC^{2}RAM framework, which integrates MCMC sampling within SRAM for high-speed Bayesian inference through in-memory computation and parallelization. Ni *et al.* [15] developed the PMBA, an FPGA-optimized MCMC accelerator that achieves high throughput but faces challenges with complex synchronization in the SIMD pipeline. Ye *et al.* [16] introduced the PMP-MCMC algorithm, which boosts MCMC efficiency on FPGAs through enhanced parallelization.

III. Self-Shrinking MCMC Method

In this section, we present the Self-Shrinking MCMC (SS-MCMC) method, a cutting-edge approach that integrates a self-shrinking mechanism for dynamic mini-batch adaptation, motivated from the idea in [9]. SS-MCMC ensures convergence to the exact target distribution while optimizing the trade-off between speed and accuracy through adaptive mini-batch updates.

Similar to the method described in [9], our algorithm uses an energy function $U_i(\theta)$ to quantify the "energy" or "cost" associated with observing a data point given the parameters θ. This function measures how well the parameters fit the data; a lower $U_i(\theta)$ indicates a better fit between θ and the observed data x_i. Consequently, the MH acceptance probability a in Code 1 can be expressed as:

$$a = \exp\left(\sum_{i=1}^{N} U_i(\theta) - U_i(\theta')\right) \tag{5}$$

The speed of the subsampling MCMC method is mainly determined by how often these functions are called during iterations. To ensure accurate approximation of the posterior distribution with a data subset, we constrain the energy differences between data points. Specifically, we use positive constants $c_1, \ldots, c_N$ that sum to a constant C and a symmetric function $M : \Theta \times \Theta \to \mathbb{R}_+$ to measure similarity between parameter values θ and θ' within the parameter space Θ. The energy difference between two parameter values is constrained by $|U_i(\theta) - U_i(\theta')| \leq c_i \cdot M(\theta, \theta')$, which bounds the variation in energy within a manageable range. When using a data subset for acceptance probability computation, the key difference from using the full dataset is the energy difference functions for the data points excluded from the subset. Thus, the logarithmic ratio of the acceptance probability with the full dataset to that with the subset can be formulated as:

$$\left|\log\left(\frac{a}{a_{\text{subsample}}}\right)\right| = \left|\sum_{i \notin S}(U_i(\theta) - U_i(\theta'))\right| \tag{6}$$

By applying these constraints, we can bound the energy difference between θ and θ' as follows:

$$\left|\sum_{i \notin S}(U_i(\theta) - U_i(\theta'))\right| \leq \sum_{i \notin S} c_i \cdot M(\theta, \theta') \tag{7}$$

It is important to note that since $\sum_{i=1}^{N} c_i = C$, for indices i not in S, we have $\sum_{i \notin S} c_i \leq C$. This allows us to further bound the expression:

$$\left|\sum_{i \notin S}(U_i(\theta) - U_i(\theta'))\right| \leq C \cdot M(\theta, \theta') \tag{8}$$

Based on Eq. (6), the difference in MH acceptance probability between using the full dataset and the subsample is also bounded as follows:

$$\left|\log\left(\frac{a}{a_{\text{subsample}}}\right)\right| \leq C \cdot M(\theta, \theta') \tag{9}$$

The local bound on the likelihood difference provides the theoretical foundation for our SS-MCMC algorithm, ensuring both convergence and accuracy. This approach allows the algorithm to concentrate computational resources on a selected data subset, significantly reducing computational overhead compared to traditional MH methods. The SS-MCMC algorithm is outlined in Code 2. As in [9], we assume the likelihood function U_i is L_i-Lipschitz continuous, setting $c_i = L_i$, and

Code 2 Self-Shrinking MCMC (SS-MCMC)

Input: initial parameter θ_0, number of samples N_s, initial hyperparameter χ_0, parameters $c_i = L_i$ and C, $target\_accept$, $adjust\_interval$, adjust ratio δ;
Output: samples θ_i, $i = 1, \ldots, N_s$;

1: $current\_accept \leftarrow 0$; $\chi \leftarrow \chi_0$;
2: **for** $i = 1$ **to** N_s **do**
3: Propose sample: $\theta' \sim \theta_{i-1} + \mathcal{N}(0, \sigma^2 I_D)$;
// Compute the upper bound using L2 norm.
4: $M(\theta_{i-1}, \theta') = \|\theta' - \theta_{i-1}\|$;
// Sample the minibatch size.
5: $B \sim \text{Poisson}(\chi C^2 M^2(\theta_{i-1}, \theta') + CM(\theta_{i-1}, \theta'))$;
6: $LH \leftarrow 0$;
7: **for** $b = 1$ **to** B **do**
// Sample the data point index j_b in the mini-batch.
8: $P(j_b = j) = c_j / C$, for $j = 1 \ldots N$;
// Compute $U_i(\theta)$ using Eq. (2.)
9: $\phi_1 = \frac{1}{2}\left(U_{j_b}(\theta') - U_{j_b}(\theta_{i-1}) + c_{j_b} M(\theta_{i-1}, \theta')\right)$,
$\phi_2 = \frac{1}{2}\left(U_{j_b}(\theta_{i-1}) - U_{j_b}(\theta') + c_{j_b} M(\theta_{i-1}, \theta')\right)$;
// add or remove the data point with a probability.
10: $p_{j_b} = \frac{\chi c_{j_b} CM^2(\theta_{i-1}, \theta') + \phi_1}{\chi c_{j_b} CM^2(\theta_{i-1}, \theta') + c_{j_b} M(\theta_{i-1}, \theta')}$;
11: $u_1 \sim \text{Uniform}(0, 1)$;
12: **if** $u_1 \leq p_{j_b}$ **then** // add the data.
13: $LH \mathrel{+}= \log\left(\frac{\chi c_{j_b} CM^2(\theta_{i-1}, \theta') + \phi_2}{\chi c_{j_b} CM^2(\theta_{i-1}, \theta') + \phi_1}\right)$;
14: **end if**
15: **end for**
16: $a = exp(LH)$;
17: $u_2 \sim \text{Uniform}(0, 1)$;
18: **if** $u_2 \leq a$ **then**
19: $\theta_i = \theta'$; $current\_accept + +$;
20: **else**
21: $\theta_i = \theta_{i-1}$;
22: **end if**
// self-shrinking the batch size using a ratio adjusted through hyperparameter updates.
23: **if** $i \% adjust\_interval = 0$ **then**
24: **if** $current\_accept < target\_accept$ **then**
25: $\chi \leftarrow (1 + \delta)\chi$, $current\_accept \leftarrow 0$;
26: **else**
27: $\chi \leftarrow (1 - \delta)\chi$, $current\_accept \leftarrow 0$;
28: **end if**
29: **end if**
30: **end for**

use the L2 Norm as the upper bound function (line 4). The batch size B is sampled from a Poisson distribution (line 5). Data points are then sampled with replacement based on c_i values, and each sampled point has a probability of being removed from the mini-batch.

Our SS-MCMC algorithm introduces several innovations compared to the TunaMH method from [9]. Notably, our algorithm updates the likelihood in line 9 with a single computation when determining the probability of selecting a data point. Unlike [9], where likelihood re-computation

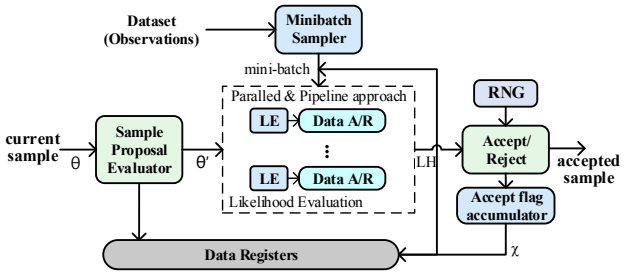

Fig. 1. The overall architecture of the proposed SS-MCMC accelerator.

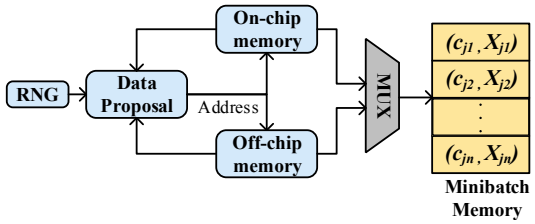

Fig. 2. The Minibatch Sampler module, which continuously samples from the observations' dataset via data prefetching from off-chip memory, where X_i represents the ith data point in the dataset.

is required, our method automatically updates the likelihood for the MH acceptance probability after selecting a data point. In contrast to standard MCMC, which only computes the likelihood for the proposal and keeps the previous one unchanged, our SS-MCMC method requires re-computation of the likelihood for both the proposal and previous parameters in each iteration due to subset changes. While this introduces additional computations, the use of a subset reduces the overall computational costs. The optimization of likelihood updates in our algorithm significantly reduces computational overhead compared to the method in [9], saving the repeated likelihood computations within the loop.

The key innovation of our method is the self-shrinking mechanism for batch size, illustrated in lines 23 to 29 of Code 2. This mechanism dynamically adjusts the required subset size by tuning the hyperparameter based on the sample acceptance ratio, ensuring an optimal balance between sample efficiency and computational cost. Experimental results, as discussed in Section V-E, show that our approach significantly outperforms [9] in effective samples per second, demonstrating a more efficient use of computational resources.

IV. Proposed Hardware Architecture

A. Design Overview

In this section, we present the hardware architecture designed to implement the SS-MCMC algorithm on an FPGA. In big data scenarios, the observed dataset is initially stored in off-chip DDR memory and then processed by the MCMC accelerator. Fig. 1 shows the overall architecture of the SS-MCMC accelerator, which consists of the Sample Proposal Evaluator, Minibatch Sampler, Likelihood Evaluation, and Acceptance flag accumulator modules.

The Sample Proposal Evaluator calculates the upper bound function of the proposal and current samples, storing the results in data registers for subsequent operations. The Minibatch Sampler efficiently forms mini-batches by leveraging data prefetching and continuous sampling from both on-chip and off-chip memory, as detailed in Section IV-B. The Likelihood Evaluation module accelerates the computation of likelihood probabilities through parallel and pipelined processing. Based on the probability ratio, the sample is then either accepted or rejected. Concurrently, the Acceptance flag accumulator adaptively updates the hyperparameter χ to dynamically adjust the mini-batch size during iterations.

B. Data Prefetching with Memory Subsystem Design

To reduce communication overhead when sampling from off-chip memory for minibatch generation, we introduce a data prefetching technique that optimally utilizes on-chip memory. As shown in Code 2, data points are selected based on their corresponding c_i values. To optimize this process, high-inclusion probability data points are stored in faster on-chip BRAMs, while those with lower probabilities are kept in slower off-chip memory. This allocation, determined during pre-processing, effectively eliminates the need for real-time data transfer. The tailored memory sub-system and Minibatch Sampler module are illustrated in Fig. 2. During pre-processing, we calculate the weights of the dataset points and store the highest-weighted data points in on-chip BRAMs, with their corresponding weights stored in the same memory address.

C. Likelihood Evaluation Design

The architecture of the Likelihood Evaluation module is illustrated in Fig. 3. Designed with a pipelined approach, this module efficiently computes the likelihood for each data point. Given that this stage represents the main computational load of the SS-MCMC algorithm, our design leverages parallel computing to handle multiple likelihood calculations concurrently. Each LE block features two likelihood computation sub-modules because SS-MCMC needs to recalculate the likelihood for both the proposal and the previous parameter in each iteration, as discussed in the above section.

V. Evaluation and Experiments

A. Case Study

We evaluate the performance of our SS-MCMC accelerator using logistic regression on the MNIST dataset of handwritten digits. Similar to previous works in literature [2], [6], [9], our focus is on classifying digits 7 and 9, utilizing the first 12 principal components as features. The likelihood function for logistic regression is given by:

$$p(x_n|\theta) = \frac{1}{1+\exp(-y_n\theta^\top x_n)} \tag{10}$$

where $x_n \in \mathbb{R}^D$ represents the feature vector for the n-th data point, and y_n is the corresponding class label, taking values of either -1 or 1.

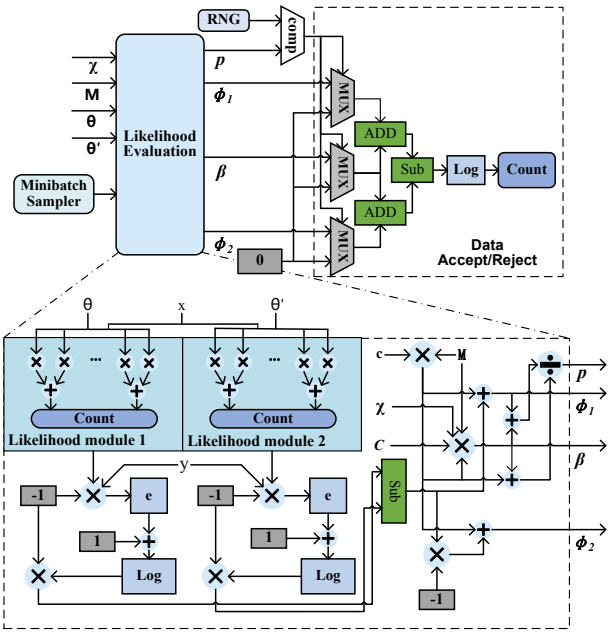

Fig. 3. The architecture of the Likelihood Evaluation module with parallel processing, where $\beta = \chi c_j CM^2(\theta_{i-1}, \theta')$.

B. Implementation Detail

Our SS-MCMC accelerator is implemented on an AMD Zynq UltraScale+ FPGA using Xilinx Vivado HLS, with all data paths utilizing single-precision floating-point arithmetic and operating at a frequency of 200 MHz. For performance comparison, we also implemented three other exact MCMC algorithms on the same FPGA as standalone implementations: 1) Standard MCMC (MH), which computes the likelihood on the entire dataset; 2) flyMCMC [6], which requires a global lower bound on the likelihood function; and 3) TunaMH [9], which uses a tunable hyperparameter to balance convergence rate and batch size. The latter two represent state-of-the-art subsampling MCMC methods.

C. Resource Utilization

Table I compares the resource utilization of our SS-MCMC accelerator with other MCMC methods. Although our SS-MCMC and the other two subsampling methods require more computational resources than the conventional MH architecture (MH), this additional resource use is necessary to support modules like the Minibatch Sampler and decision-making processes that go beyond the basic likelihood estimator used in MH. FlyMCMC, while using fewer logic resources, requires more BRAMs than our method. This is due to its reliance on a global bound; when this bound is not tightly approximated, flyMCMC needs larger batch sizes, consuming more BRAMs. In contrast, our SS-MCMC algorithm uses fewer logic resources and less on-chip BRAM compared to TunaMH. This efficiency is due to: 1) our algorithm updating the likelihood during minibatch sampling, which avoids redundant calculations when determining the probability ratio; and 2) the self-shrinking mechanism in SS-MCMC, which results in smaller batch sizes and reduced BRAM usage, as shown in Table II.

TABLE I
RESOURCE UTILIZATION OF OUR PROPOSED MCMC ACCELERATOR.

Methods	LUTs	FFs	DSPs	BRAMs
MH	34256 (48%)	32523 (23%)	294 (81%)	24 (5%)
flyMCMC	54722 (77%)	43232 (30%)	303 (83%)	83 (18%)
TunaMH	64689 (91%)	61417 (43%)	352 (98%)	59 (13%)
SS-MCMC	48612 (69%)	38918 (27%)	324 (90%)	39 (9%)
Total of board	70560	141120	360	432

TABLE II
SPEED AND SAMPLING EFFICIENCY COMPARISON OF OUR ACCELERATOR OVER OTHER METHODS.

Methods	MH	flyMCMC	TunaMH	SS-MCMC
Batch Size	12,214	4,360	216	178
Execution time (s)	133.4	53.3	14.1	2.8
Raw Speedup	1×	2.5×	9.5×	47.6×
ESS/second	4.5	10.8	16	80
Throughput Speedup	1×	2.4×	3.6×	17.8×

D. Raw Speed Comparison

We first compare the raw speed (wall-clock time) of our SS-MCMC accelerator with other methods. For each method, 100,000 sample points are generated, and each MCMC simulation is repeated 100 times to obtain average results. Table II presents the comparison of batch size, runtime, and speedup for these methods on the MNIST dataset. The MH method evaluates likelihood using the entire dataset of 12,214 data points, and all speedups are normalized against the standard MH method.

The results show that our SS-MCMC is the fastest method in terms of wall-clock time. The average batch size of SS-MCMC is 24.5× smaller than flyMCMC's and 1.2× smaller than TunaMH's. This efficiency is primarily due to the self-shrinking mechanism for batch size update in SS-MCMC. Besides, SS-MCMC outperforms both standard MH and TunaMH, achieving impressive speedups of 47.6× and 5×, respectively. This substantial increase in computational efficiency is attributed to the highly optimized hardware architecture, data prefetching techniques discussed in Section IV-B, and optimal batch size selection.

E. Effective Sampling Throughput and Risk Comparison

1) Effective Sampling Throughput: We then compare the Effective Sampling Throughput of the samples generated from our SS-MCMC with that of other methods, as described in Section II-B. Our experimental results confirm that SS-MCMC algorithm achieves unbiased sampling, yielding the same mean value for generated samples as the MH and other exact MCMC methods. For this reason, in order to compare the performance of these algorithms, we evaluate the sampling efficiency using effective sampling throughput, measured by ESS per second, as described in Section II-B. We follow the same experimental setup as mentioned above, generating 100,000 samples and

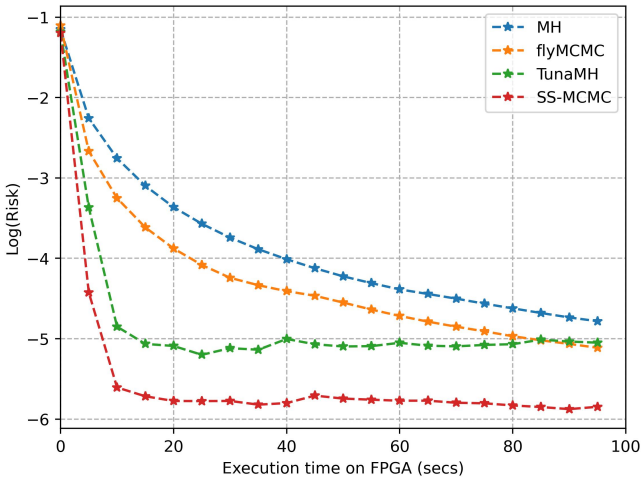

Fig. 4. The risks in the estimate of the mean value of the first parameter of the Logistic Regression on MNIST.

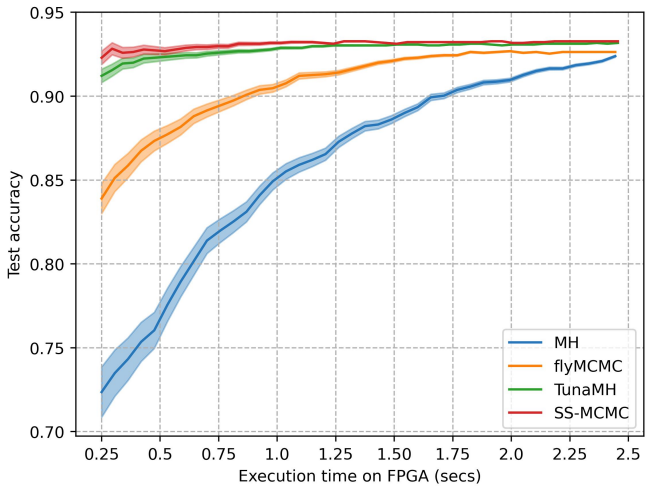

Fig. 5. MNIST logistic regression: test accuracy comparison.

repeating the process 100 times. This allows us to calculate the average number of effective samples and the average elapsed time across trials, providing a fair and comprehensive comparison of sampling efficiency for each algorithm.

The results, as shown in Table II, indicates that our SS-MCMC algorithm significantly outperforms other methods in sampling efficiency, achieving a performance improvement of 5× over TunaMH and 7.4× over flyMCMC. This substantial gain is primarily attributed to the shelf-shrinking mechanism and the co-optimized hardware-software design. Notably, the performance of TunaMH is sensitive to its initial hyperparameter. For the results presented in this paper, we used the optimal hyperparameter setting for TunaMH. In contrast, our SS-MCMC algorithm does not depend on an initial hyperparameter, as it automatically converges to the optimal setting after a certain number of iterations. Furthermore, our method maintains consistent performance across varying parameters such as the adjustment interval and ratio, demonstrating its robustness under different configurations.

2) Risk: We further evaluate the risk in the MCMC estimation errors over execution time on the FPGA. The true predictive mean is first determined using an extended run of the standard MH algorithm. The risk is then calculated as the mean squared error (MSE) of estimates generated by the four evaluated algorithms.

Fig. 4 shows the risk in estimating the mean of the first parameter on the MNIST dataset. The results indicate that SS-MCMC achieves the fastest risk reduction, decreasing variance more quickly within the same execution time. This rapid convergence is primarily due to the computational speedups from its dynamic batch sizing and optimized hardware design, offering a distinct advantage over the MH and flyMCMC methods in reducing risk. Moreover, SS-MCMC outperforms TunaMH in risk reduction within the same timeframe thanks to its adaptive hyperparameter tuning, which effectively optimizes the batch size.

F. Test Accuracy Comparison

Finally, we evaluate the test accuracy by using MCMC samples as parameters in the logistic regression model for inference on the MNIST test dataset, comparing classification accuracy across different methods. To ensure the reliability of the average results, the accuracy is measured over 100 repeated runs. The results are presented in Fig. 5, where the line widths in the figure represent the standard deviations observed across these runs. The results confirm that our SS-MCMC method consistently outperforms both the MH and flyMCMC methods in test accuracy. While SS-MCMC and TunaMH achieve similarly high accuracy, SS-MCMC reaches this level more quickly, highlighting its efficiency in producing accurate results.

VI. Conclusion

This paper presents a novel framework for an exact MCMC method based on data subsampling, aimed at reducing the computational burden of likelihood calculations while maintaining sampling efficiency and estimation accuracy. The key contribution of this work lies in the introduction of a self-shrinking mechanism for optimizing batch size, coupled with a tailored hardware architecture that delivers high computational efficiency and minimizes data communication overhead. Experimental results demonstrate that our MCMC accelerator achieves higher effective sampling throughput and lower risk compared to other state-of-the-art MCMC designs. Future work will focus on improving scalability and extending the algorithm's performance across broader applications.

Acknowledgment

The support of the National Natural Science Foundation of China (No. 62001165) and Scientific Research Foundation of Hunan Provincial Education Department (Key project 23A0087) is gratefully acknowledged.

REFERENCES

[1] R. Bardenet, A. Doucet, and C. Holmes, "On markov chain monte carlo methods for tall data," *Journal of Machine Learning Research*, vol. 18, no. 47, pp. 1–43, 2017.

[2] S. Liu and C.-S. Bouganis, "Communication-aware mcmc method for big data applications on fpgas," in *Proc. FCCM*, 2017, pp. 9–16.

[3] C. P. Robert *et al.*, "Accelerating mcmc algorithms," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 10, no. 5, p. e1435, 2018.

[4] A. Korattikara *et al.*, "Austerity in mcmc land: Cutting the metropolis-hastings budget," in *Proc. ICML*, 2014, pp. 181–189.

[5] R. Bardenet *et al.*, "Towards scaling up markov chain monte carlo: an adaptive subsampling approach," in *Proc. ICML*, 2014, pp. 405–413.

[6] D. Maclaurin and R. P. Adams, "Firefly monte carlo: Exact mcmc with subsets of data," *arXiv preprint arXiv:1403.5693*, 2014.

[7] R. Zhang and C. M. De Sa, "Poisson-minibatching for gibbs sampling with convergence rate guarantees," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[8] M. Quiroz *et al.*, "The block-poisson estimator for optimally tuned exact subsampling mcmc," *Journal of Computational and Graphical Statistics*, vol. 30, no. 4, pp. 877–888, 2021.

[9] R. Zhang *et al.*, "Asymptotically optimal exact minibatch metropolis-hastings," *Advances in Neural Information Processing Systems*, vol. 33, pp. 19 500–19 510, 2020.

[10] S. Liu, G. Mingas, and C.-S. Bouganis, "An unbiased mcmc fpga-based accelerator in the land of custom precision arithmetic," *IEEE Transactions on Computers*, vol. 66, no. 5, pp. 745–758, 2016.

[11] J. S. Liu, *Monte Carlo Strategies in Scientific Computing*. Springer, 2008.

[12] S. L. Scott *et al.*, "Bayes and big data: The consensus monte carlo algorithm," *Big Data and Information Theory*, pp. 8–18, 2022.

[13] G. Mingas and C.-S. Bouganis, "Population-based mcmc on multi-core cpus, gpus and fpgas," *IEEE Transactions on Computers*, vol. 65, no. 4, pp. 1283–1296, 2015.

[14] P. Shukla *et al.*, "MC^{2}RAM: Markov chain monte carlo sampling in sram for fast bayesian inference," in *Proc. ISCAS*, 2020, pp. 1–5.

[15] Y. Ni *et al.*, "PMBA: A parallel MCMC Bayesian computing accelerator," *IEEE Access*, vol. 9, pp. 65 536–65 546, 2021.

[16] G. Ye and S. Lu, "A prefetching multiple proposals markov chain monte carlo algorithm," *IEEE Transactions on Artificial Intelligence*, 2024.