CoopMC: Algorithm-Architecture Co-Optimization for Markov Chain Monte Carlo Accelerators

Yuji Chai[1,2,†], Glenn G. Ko[1,2], Wei-Te Mark Ting[1,2], Luke Bailey[1,2], David Brooks[1], Gu-Yeon Wei[1]
[1]*Harvard University*, [2]*Stochastic Inc.*
[†]*yuc927@g.harvard.edu*

***Abstract*—Bayesian machine learning is useful for applications that may make high-risk decisions with limited, noisy, or unlabeled data, as it provides great data efficiency and uncertainty estimation. Building on previous efforts, this work presents CoopMC, an algorithm-architecture co-optimization for developing more efficient MCMC-based Bayesian inference accelerators. CoopMC utilizes *dynamic normalization* (DyNorm), *LUT-based exponential kernels* (TableExp), and *log-domain kernel fusion* (LogFusion) to reduce computational precision and shrink ALU area by 7.5× without noticeable reduction in model performance. Also, a *Tree-based Gibbs sampler* (TreeSampler) improves hardware runtime from $\mathcal{O}(N)$ to $\mathcal{O}(log(N))$, an 8.7× speedup, and yields 1.9× better area efficiency than the existing state-of-the-art Gibbs sampling architecture. These methods have been tested on 10 diverse workloads using 3 different types of Bayesian models, demonstrating applicability to many Bayesian algorithms. In an end-to-end case study, these optimizations achieve a 33% logic area reduction, 62% power reduction, and 1.53× speedup over previous state-of-the-art end-to-end MCMC accelerators.**

***Keywords*-Algorithm-Architecture Co-Design, Hardware Accelerator, Bayesian Machine Learning, Markov Chain Monte Carlo**

I. Introduction

Bayesian machine learning (ML), often called probabilistic computing, is a type of statistical machine learning that leverages Bayes' theorem to model event probabilities using observed evidence and prior knowledge. It has become an important class of machine learning algorithms for processing data in various scenarios, including integrating domain knowledge in models, handling sparse or noisy data, and handling hierarchical or time-series data. Compared to many deep learning (DL) algorithms, it provides much better data efficiency and accurately estimate uncertainty. These advantages make Bayesian ML a superior choice in certain fields.

Unlike DL models, Bayesian models do not have high requirements for dataset quality and quantity: Bayesian models can easily learn from a limited number of (and even unlabeled) data points, while still providing useful insights. With their high data efficiency, Bayesian models' outperform their DL counterparts in fields such as insurance [1], finance [2], and pharmaceuticals [3], where large amounts of labeled data are hard to acquire and often noisy.

At the same time, Bayesian models provide explicit uncertainty estimates with inference results. For tasks such as biomedical analysis and clinical diagnostics, a prediction result is far from sufficient; an uncertainty estimation about the model's inference prediction is crucial when making critical decisions with real-world consequences. A mission-critical model must communicate how certain it is about a query and to "know" when it is uncertain. From this perspective, DL falls short. Although a final output layer can provide a quality score for possible classes, this has been proven to be insufficient when estimating predictive uncertainty [4]. On the other hand, Bayesian models excel in problems where uncertainty is critical. A recent example is COVID-19 predictions, where Bayesian modeling helped predict daily COVID-19 cases by leveraging prior statistics on similar diseases, such as Severe Acute Respiratory Syndrome, and by modeling human behaviors, such as social distancing [5]–[7]. Incorporating Bayesian methods into DL preserves the learning capabilities of DL while providing superior uncertainty estimation for its outputs. This technique has shown its effectiveness in regression tasks [8], [9], image classification [4], and computer vision for autonomous vehicles [10].

A typical Bayesian model use its parameters to estimate the probability of certain events. Predicting the probability of an event taking an outcome from a Bayesian model is called Bayesian inference. Bayesian inference relies on one of two classes of inference algorithms: Markov chain Monte Carlo (MCMC) or variational inference (VI). For VI, the posterior inference can be cast as an optimization problem solvable using gradient-based algorithms, and thus enjoys the benefits of various acceleration tools built for DL [11], [12]. However, VI may not always converge and can introduce unwanted bias during inference, making it ill-suited for critical problems. On the other hand, MCMC is guaranteed to converge and with less bias, but does not scale well on existing computing platforms, namely CPUs and GPUs. It requires many complex kernels (e.g., generating many random numbers and sampling from discrete distributions) which may stall hardware and yield poor utilization. Table II shows runtime percentage breakdown for various workloads on CPU. The Probability Generation (PG) and Sampling from Distribution (SD) are computational steps consisting of complex computational kernels. They dominates the end-to-end runtime for various MCMC-based workloads. Specialized hardware acceleration to accommodate these novel kernels could lead to direct speedups. For this reason, specialized hardware and accelerators have shown great potential [13]–[17]. They already demonstrate dramatic improvements over CPUs/GPUs.

Despite Bayesian models' unparalleled advantages for specific classes of problems and acceleration potential, they have drawn less attention from the architecture and systems communities, in part due to the high costs associated with building specialized hardware. Thus, developers and researchers tend to focus on established fields and well-known algorithms. Unfortunately, this lack of attention may hurt machine learning development in the long run [18]. To highlight the potential of other promising ML algorithms, such as Bayesian ML, this paper seeks to further improve acceleration of Bayesian inference. By identifying core operations within Gibbs sampling, a widely used MCMC algorithm, we present a collection of optimizations that improve computational efficiency while maintaining the robustness of the algorithm against noise or errors introduced. We first generalize the computational flow of Bayesian inference into three main steps. Next, we utilize algorithm-architecture co-design to exploit numerical and structural properties of the computational flow in reducing hardware costs while accelerating inference. Finally, we evaluate the resulting design across a broad variety of workloads. The contributions of our paper are as follows:

- Generalization of the Bayesian inference computational flow to three stages: Probability Generation (PG), Sampling from Distribution (SD) and Parameter Update (PU).
- A collection of optimization methods, Dynamic Normalization (DyNorm), lookup-table-based Exponential Kernel (TableExp), and Log-domain Kernel Fusion (LogFusion), to collectively accelerate Bayesian inference, providing 7.5× ALU area reduction with negligible reduction in model performance.
- A tree-based sampler micro-architecture (TreeSampler) reducing Gibbs sampling cycle runtime from $\mathcal{O}(N)$ to $\mathcal{O}(log(N))$, with better hardware area efficiency.
- Evaluation across ten diverse Bayesian workloads to demonstrate robustness and broad applicability to common Bayesian models.
- A case study highlighting CoopMC's effectiveness in end-to-end designs, combining previously published designs.

Admittedly, there are some concepts or implementation that are similar to DyNorm, TableExp, and LogFusion in other domain's previous works. This paper's novelty is in how we combine the techniques for Bayesian inference acceleration. Combining the four optimization methods is also a deliberate decision: they are codependent. Without DyNorm, TableExp and LogFusion would not converge for most Bayesian learning algorithms due to precision loss. LogFusion is specifically tailored to Bayesian learning, which requires sequences of multiplications and divisions. Also, unlike previous works that only focus on a particular model type or inference algorithm, or on a single application, this work explores three different types of Bayesian models, each running disparate applications with widely varying numbers of variables and dimensions—from fewer than 10 to more than 6 million variables and up to 2 million dimensions each. Furthermore, CoopMC investigates optimizations for computational kernel efficiency, which has been larger ignored by previous works. Thus, methods proposed in CoopMC could be directly added into past or future accelerator computational pipeline designs as a plug-in optimization. We will demonstrate its end-to-end optimization capability in Section IV-D.

II. Background

Bayesian models consist of a collection of random variables. Each random variable has several components: the variable's current label, which can be discrete or continuous, and its probability distribution for taking different labels. The variable's correlation to other variables within a model affects its own probability distribution. The relationships between different variables inside a model may be expressed as a graph that incorporates prior knowledge for a given task. Different causal relationships result in different graph structures, enabling the application of Bayesian ML for a wide variety of inference tasks. To show the generality of our method, we cover three types of Bayesian models with ten different workloads, as seen in Table I: Markov Random Field (MRF), Bayesian Network (BN) and Latent Dirichlet Allocation (LDA).

Although we only discuss these ten workloads in this paper, methods are designed to provide general acceleration for MCMC-based sampling algorithm. The MCMC algorithm is analogous to stochastic gradient descent in deep learning: it is a general tool used in a wide range of Bayesian machine learning models, including the three (MRF, BN, LDA) presented in the paper. So CoopMC methods are applicable to any MCMC algorithm with a discrete sampling process. For the simplicity of this paper, we will focus on those ten workloads in Table I.

A. Model Evaluation Metrics

One thing to note is that most of the models discussed in this work are designed for unsupervised learning. Due to the nature of unsupervised learning, it is not possible to calculate model accuracy for an inference task because data is unlabeled. Therefore, we evaluate the performance of our optimization methods by comparing the hardware-optimized performance to a baseline vanilla algorithm performance. The comparison metric will usually be the pos-

Workload	#Variables	#Labels
MRF-Image Restoration	6,656	64
MRF-Stereo Matching	110,592	16
MRF-Image Segmentation	150,000	2
MRF-Sound Source Separation	64,125	2
BN-ASIA	8	2
BN-EARTHQUAKE	5	2
BN-SURVEY	6	3
LDA-NIPS	1,932,365	128
LDA-Enron	6,412,172	128
LDA-RNA	540,393	128

Table I: Summary of various benchmark workloads.

Workload	PG%	SD%	PU%
MRF-Image Restoration	88.00%	9.20%	2.81%
MRF-Stereo Matching	76.49%	14.78%	8.73%
MRF-Image Segmentation	45.71%	31.69%	22.60%
MRF-Sound Source Separation	46.14%	31.63%	22.23%
BN-ASIA	46.00%	52.37%	1.63%
BN-EARTHQUAKE	44.97%	53.36%	1.68%
BN-SURVEY	45.96%	52.45%	1.59%
LDA-NIPS	40.26%	53.23%	6.50%
LDA-Enron	42.84%	56.34%	0.83%
LDA-RNA	39.14%	53.20%	7.66%

Table II: Runtime percentage breakdown of various benchmark workloads.

terior probability of the final result or the difference to a best convergence result that the vanilla algorithm is able to achieve. This is a common practice for evaluating unsupervised Bayesian models and inference. Details about model performance evaluation are provided in the introductions to each algorithm.

B. Markov Random Field (MRF)

A Markov random field [19], [20] is a Bayesian model with a correlation graph similar to a grid-like structure V, containing all nodes. The event abstraction for MRF is defined as a graph node taking a certain label, which means each variable will represent a node in the structure. Every node is correlated to four neighbors surrounding it. For any node $i \in V$, its probability distribution is determined by the observed data y_i and other correlated nodes' labels, x_j for $j \in N_i$, which represent the set containing all correlated nodes of i. The label of x_i may take on l discrete labels in the range $[0, l)$. The posterior probability of i taking the label x_i, given its observed label y_i and the correlation set N_i, is:

$$P_i(x_i, y_i) = \frac{1}{Z}\phi_i(x_i, y_i) \prod_{j \in N_i} (\phi_{i,j}(x_i, x_j)) \tag{1}$$

$$Z = \sum_{x_i \in [0,l)} (P_i(x_i, y_i)) \tag{2}$$

ϕ_i is the probability of i taking the label x_i, given y_i, while $\phi_{i,j}$ is its probability, given the correlation set N_i. The MRF formulation is often rewritten as a sum of energy functions. For every variable i, its energy functions consist of two components. The first is data cost, $DC(x_i, y_i)$, determined by the difference between y_i and x_i. The second is smooth cost, $SC(x_i, x_j)$, determined by x_i and y_i. The sum of $DC(x_i, y_i)$ and $SC(x_i, x_j)$ is the total cost, $TC(x_i, y_i)$. TC is fed into a negative exponential function to generate the probability distribution. The probability distribution takes the following form:

$$TC_i(x_i, y_i) = DC_i(x_i, y_i) + \sum_{x_i \in [0,l)} (SC_{i,j}(x_i, x_j)) \tag{3}$$

$$P_i(x_i, y_i) = e^{-\beta * TC_i(x_i, y_i)} \tag{4}$$

The model will continue training until the energy function converges. The inference result will be the label of each variable from the last iteration. Four different applications are used as benchmarks for MRF:

- Image Restoration: An application to restore an image from one with added random Gaussian noise and black boxes. An ideal output completely removes all noise and restores the original image.
- Stereo Matching: An application to understand the 3D depth view of an input image. Objects in the same level of the depth should share the same label.
- Image Segmentation: An application to separate the foreground from the background of an input image.
- Sound Source Separation: an application to separate distinct sound sources from an input audio containing a mix of audio sources.

As previously mentioned, it is unreasonable to provide a correct answer to an unsupervised workload. Therefore, performance for each MRF application is evaluated by the mean-square-error (MSE) of the inference output with a reference result, or golden result. The golden result is generated by running a vanilla floating-point inference algorithm for an excessively large number of iterations, and represents the best quality MRF is able to achieve on a given workload. Due to the randomness of MRF, any two inference results will almost never be exactly the same, so it is natural to observe a non-zero MSE, even for the vanilla algorithm. To fairly compare different applications, we normalize each MSE result with the MSE of the inference result from an untrained model. A smaller MSE means better inference quality.

C. Bayesian Network (BN)

A Bayesian network [21], [22] is a Bayesian model whose variables and their conditional dependence are modeled as a directed acyclic graph. They are commonly used for understanding probabilistic relationships between incidents sharing a predefined casual relationship. Each node in the graph represents an event for this type of Bayesian model. A typical workload queries the probability of a variable being a certain label, given some other variables' label as evidence e.

Each node assumes a label in $[0, n)$. n represents the number of labels that each node could take. Each edge represents a conditional probability between two variables in the graph. The MCMC inference algorithm will update every variable's label x_i by sampling based on its probability, as given by:

$$P(X_i = x_i|e) = P(X_i = x_i|Pa(X_i), e) \times \prod_{Y_j \in Ch(X_i)} \Big(P(Y_j|Pa(Y_j), e)\Big) \times \frac{1}{P(e)} \quad (5)$$

$Pa(X_i)$ represents all parent nodes of X_i, while $Ch(X_i)$ represents all children nodes of X_i. After each iteration, the label of X_i will be recorded, and the probability of X_i taking different labels is generated from counting the number of times each label is selected during the inference process. The sampling process will stop when the probability of the variables reach steady-state. Three different datasets are used as benchmarks for BN:

- ASIA [23]: A network that describes the relationship between different patients' symptoms and their potential causes.
- EARTHQUAKE [24]: A network that describes the relationship between earthquakes and neighbors' calls due to an alarm system.
- SURVEY [25]: A network that describes how different peoples' demographic factors will influence their transportation methods.

Similar to before, the inference quality of the BNs is evaluated by comparing their MSEs to golden results. Here, a golden result consists of the average results of several inferences using vanilla floating-point Gibbs sampling inference. Again, the smaller the MSE, the better the result.

D. Latent Dirichlet Allocation (LDA)

Latent Dirichlet Allocation [26], [27] is a generative model that views documents in a corpus as "bags of words" generated from a vocabulary. Each word may be treated as an event associated with a source document, vocabulary, and latent topic (to be inferred). Different word labels are represented as different event outcomes. The distribution of topics across documents and vocabulary are captured by two parameter tables: a Document-Topic table (DT) and Vocabulary-Topic table (VT). DT and VT have sizes $D \times T$ and $V \times T$, respectively, where D represents the number of documents, V represents the size of vocabulary and T is the number of topics. Training an LDA model requires updating DT and VT using an input corpus until convergence. Once trained, these tables may be used to infer the distribution of topics for new documents. In this paper, we adopt a common inference method, called collapsed Gibbs sampling [28], [29]. Collapsed Gibbs sampling uses the following probability distribution expression:

$$P(k) = \frac{(DT_{d,t} + \alpha)(VT_{t,v} + \beta)}{\sum_{v \in [0,V)} (VT_{t,v}) + \beta V} \quad (6)$$

Three different datasets are used as benchmarks for LDA:

- NIPS: A collection of academic papers from NeurIPS. The workload is to identify key discussion topics for each paper in the dataset.
- Enron: A collection of email transcripts. The LDA model aims to determine the essential topics for each e-mail.
- RNA: Multiple generic RNA sequences. A typical task is to find similar sequences for genomic expression.

The inference quality of a dataset is evaluated by the log-likelihood of the final converged model. This log-likelihood value is determined by the likelihood of the model variables' configurations, given its observed data. This is the common comparison standard used in previous works [30]–[32] for LDA. Higher log-likelihood indicates better quality.

III. Algorithm and Architecture Co-Optimization

We propose co-optimization methods generally applicable to inference across a broad variety of Bayesian models. To provide such generality, we first construct a common computational model that works for most Bayesian inference. Second, we discover common patterns such as solving exponential equations, multiplication-division sequences, low-precision requirements, and sampling. Third, we exploit these characteristics to reduce precision and hardware requirements, fuse different computational kernels, and reduce sampling run time.

Based on the discussion in Section II, this work focuses on Bayesian inference using Gibbs sampling. The inference process for any Bayesian model is essentially a continuous sampling of new labels from a probability distribution for each random variable in the model until convergence. More concretely, for each random variable x in a model B, its sampling process consists of the following three steps, also visualized in Figure 1:

- Step 1 - Probability Generation (PG): Based on the current labels of all other variables correlated with x, a probability distribution P_x (a vector of size N) is generated. The n^{th} element in P_x represents the probability of x taking the label of n, given the current state of B. In an accelerator, these computations require hardware components that can perform several types of kernels, including division, multiplication, addition, and exponentiation. P_x is stored as a probability vector, usually in a register file (ProbReg), for use in the next step. Multiple processing elements can generate several elements of P_x in parallel.
- Step 2 - Sampling from Distribution (SD): Based on the probability distribution P_x from PG, a new label n_{new} is randomly sampled for random variable x. The probability of n_{new} being n is proportional to the n^{th}

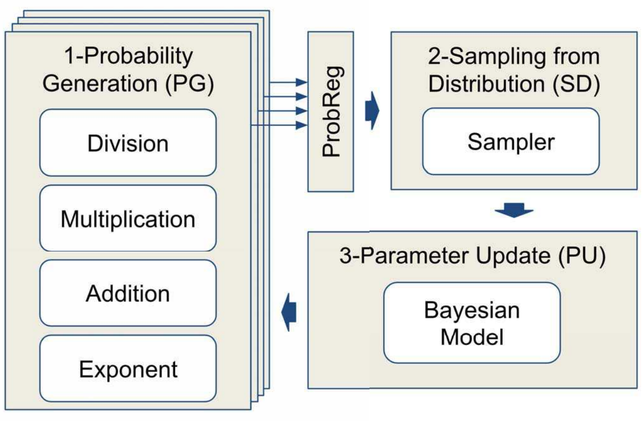

Figure 1: A diagram shows three essential steps for Bayesian inference, and its computation flow.

element in P_x. This step usually requires special hardware to sample from a distribution generated during PG.

- Step 3 - Parameter Update (PU): Update the label of x in the model B with the sampling result n_{new} from SD. Other variables correlated with x require the update to be completed before moving to the next variable.

For each iteration, every variable in the model goes through these three steps to update its label. The inference result will be the final converged model. Any hardware aiming to accelerate Bayesian inference must implement these steps in their processing element. Previous works on accelerating Bayesian inference have focused on optimizing the PU step. Various methods such as chromatic sampling and asynchronous updates help to relax the sequential requirements for PU [16]. These methods enable much better parallelism and almost linear speedup with the number of PE cores. However, their optimization and co-design never touch on PG or SD. As reported in their work, the probability distribution is generated using a standard 32-bit fixed-point computational pipeline. Every computational kernel, such as multiplication, exponentiation, and sampling, follows the vanilla algorithm exactly. At the same time, their optimizations for PU have only demonstrated effectiveness for MRFs. Building on these previous works, we investigate how to further optimize hardware accelerators for Bayesian inference by exploiting patterns or characteristics commonly found in Bayesian inference models.

A. DyNorm: Dynamic Normalization

Exponential kernels are commonly used in Bayesian models. Take the stereo matching application using MRF as a concrete example. The probability function for MRF is expressed as an exponential family in canonical form. In previous MRF accelerators, implementations used a 32-bit fixed point exponential ALU for the exponential kernel.

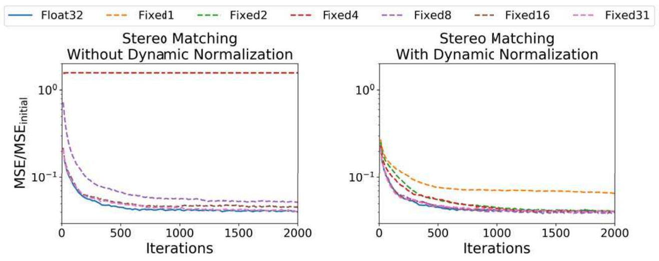

Figure 2: Dynamic normalization greatly improves algorithms' tolerance to low-precision computation.

However, whether 32-bit precision is required for MRF has not been investigated. To understand the precision requirements for the exponential kernel, we investigate how model performance will change as the precision of the exponential kernel varies.

As shown in Figure 2, different bitwidths, or precisions, have a significant impact on the convergence performance of the model. If fewer than 8 bits are used, the model does not successfully converge. The model is trapped at a steady state due to the very low precision of the exponential kernel, and most probabilities in this workload require a much smaller domain, usually smaller than 2^{-5}. With just 4 bits assigned to the exponentiation kernel, only values larger than 2^{-4} can be expressed, yielding a vector of zeroes for the P_x generated from Stage 1. Consequently, instead of sampling based on P_x, Stage 2 will only be able to select a label uniformly at random and assign that label to x. The sampler no longer correctly updates the label of x, resulting in a horizontal line in the right plot of Figure 2. Even when the number of bits is increased to 16, the final result is still unable to attain the same level of performance as with 31 bits.

However, if we confine the activation range of the exponential kernel output to a predetermined range, low precision is still viable. To achieve this, we designed a method called Dynamic Normalization and apply this to the original MRF algorithm. In the original algorithm, each element in P_x is equal to the following expression:

$$p_x(m) = \frac{exp(TC_m)}{\sum_{m=1}^{N} exp(TC_m)} \tag{7}$$

TC_m is the input for each of the exponential kernels, when variable x takes label m. Constraining TC_m will prevent $exp(TC_m)$ from becoming too small. Dividing the numerator and denominator by a constant $exp(C)$ does not affect the value of $p_x(m)$. Thus, the constant C may be used to limit the size of each of the inputs to the exponential kernel. However, since every $p_x(m)$ for each x has a wide activation range, it is impractical to predetermine the value for the normalizing constant C. Instead, this constant is determined at runtime and takes the maximum value of

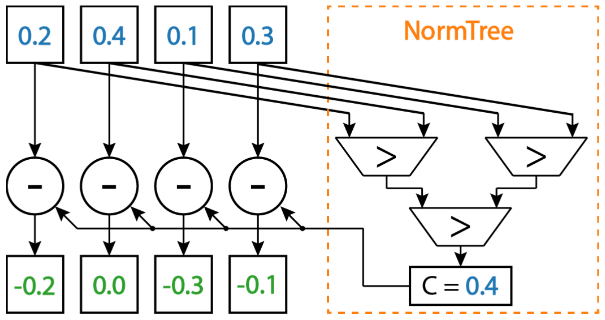

Figure 3: Hardware for applying DyNorm onto an array of inputs. NormTree structure is marked by the dashed box, which is used for finding the maximum value in the input array.

$p_x(m)$ for each x.

$$p_x(m) = \frac{exp(input_m - C)}{\sum_{m=1}^{N} exp(input_m - C)} \quad (8)$$

$$= \frac{exp(input'_m)}{\sum_{m=1}^{N} exp(input'_m)}, MAX(input'_m) = 0 \quad (9)$$

With the help of DyNorm, the maximum value of TC'_m for each x is always zero. This limits the maximum output value of the exponential kernel to one. A visualization of this process is shown in Figure 3. Since the sampling process cares most about the m with the highest $p_x(m)$, shifting all exponential kernel output values back towards zero greatly helps the sampler in generating more meaningful results, even with lower precision. This effect is clearly shown by the right plot in Figure 2. After applying Dynamic Normalization, even exponential kernels with only 1-bit precision retain partial inference capabilities. Using more precision, such as 8 bits, shows identical results to the full 31-bit exponential kernel. Both the convergence rates and final convergence results show no noticeable differences.

The DyNorm operation introduces an additional kernel into the design. This operation needs to find the maximum value from a probability distribution P_x. We propose the NormTree design, which is a tree-based maximum kernel for finding the maximum value from an array of inputs. This tree-based design may be extended for more complex usage, as discussed in the following section. Figure 3 shows the hardware design for supporting DyNorm operation with a NormTree structure. Each Tree node in the NormTree is a comparator, which compares between two incoming inputs and outputs the larger value to the next layer. The output of the last layer is the maximum value from the input array. The runtime will be $\mathcal{O}(log(N_{pipe})+1)$, where N_{pipe} is the number of pipelines for the PG step, offering very good scalability for large numbers of parallel pipelines. The hardware cost for this component is amortized by N as well, resulting in a minuscule hardware cost for DyNorm.

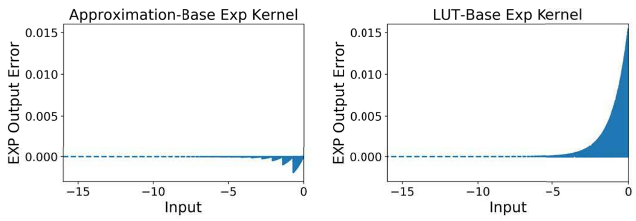

Figure 4: Comparison of kernel output error for approximation-based exp kernel and LUT-based exp kernel (TableExp)

B. *TableExp: LUT-Based Exp Kernel*

Dynamic Normalization shows great potential to lower the precision requirements for the exponential kernel. However, the actual hardware is still relatively complex even when using approximations to compute the exponential value. Since the precision can be reduced dramatically (e.g., to only 8-bit), it is not necessary to provide a very precise approximation. Instead, a fully lookup-table-based (LUT-based) exponential kernel can be used. Since DyNorm will also be applied to TableExp, all input values to this kernel will always be non-positive. For the lookup table, $size_{lut}$ is the number of elements in the lookup table, and $step_{lut}$ is the step between two adjacent quantized inputs. TableExp will quantize any negative input x_{in} into k, the largest integer smaller than $-x_{in}/step_{lut}$. If we set $step_{lut}$ to a power of 2, then the quantization operation is easily implemented using simple shifts. The quantized input k is also used to index the lookup table. If the index is smaller than the size of the lookup table, the output value will simply be the k^{th} element of the table. If the index is greater than the size of the lookup table, the output value will be zero. Each element in the table is defined using the following expression:

$$TableExp(x_{in}) = \begin{cases} exp(-k \cdot step_{lut}) & (k < size_{lut}) \\ 0 & (k \geq size_{lut}) \end{cases} \quad (10)$$

The lookup table design is also defined by another parameter, the number of bits ($\#bit_{lut}$) used to express each entry in the table. After numerical analysis of different workloads, we rarely found x_{in} to be smaller than -16 after DyNorm. Thus, we fixed $step_{lut}$ to $16/size_{lut}$. Using TableExp can greatly simplify the exp kernel design, but can also introduce more error into the kernel output. Figure 4 shows an error comparison between the approximation-based exp kernel and a TableExp with $size_{lut} = 1024$ and $\#bit_{lut} = 32$.

To understand how this introduced error affects inference quality, we perform a case study using the stereo matching benchmark, sweeping $size_{lut}$ and $\#bit_{lut}$, two key design parameters for TableExp. The parameter sweep result is shown in Figure 7, and demonstrates that Bayesian is very resilient to lower-precision hardware kernels. Using only

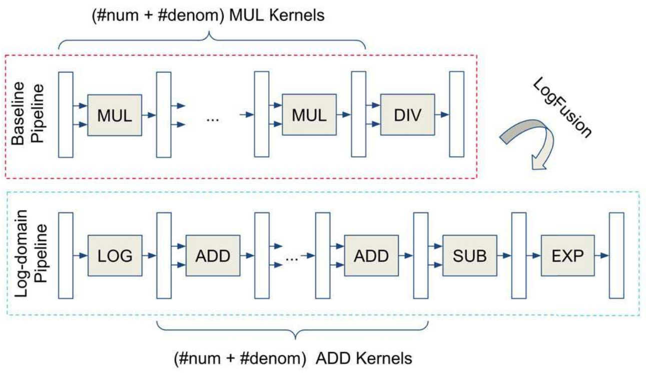

Figure 5: Visualization of a computational pipeline with LogFusion.

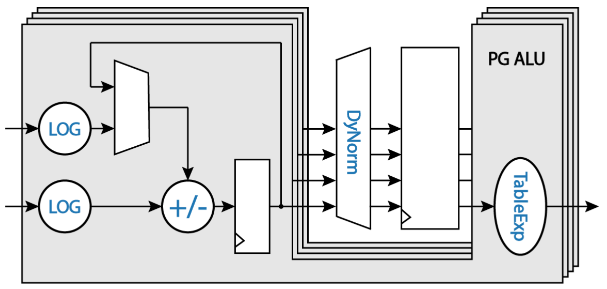

Figure 6: Micro-architecture of a probability generation (PG) computational core with LogFusion, DyNorm and TableExp.

8-bit precision results in almost the same performance as using a full 32 bits. There is also no significant overhead associated with the number of elements used in the lookup table. Storing 64 quantized values in a lookup table does not significantly impact learning rate or the final convergence result, compared to the lookup table with 1024 elements. By applying TableExp, any accelerator design can circumvent the complex approximation-based exponential kernel and instead replace it with an efficient low-precision read-only memory (ROM) while preserving inference quality.

The computational flow of Bayesian inference explains the tolerance to lower precision. In sampling from distributions, Bayesian inference is not a process that requires exact computation. Random error and noise are common even during the inference process using floating-point probabilities. Furthermore, during the sampling process, the most probable labels will have a much larger probability than other labels for each variable, so adding some additional error into the system should not significantly influence the sampling result and the most probable label will still be most frequently selected. Low-precision computation is already very common in DL accelerators [33], [34], so it is reasonable to also observe this kind of resilience to lower-precision in Bayesian inference.

C. LogFusion: Log-Domain Kernel Fusion

Apart from exp kernels, there are other types of kernels commonly used in PG. Algorithms like LDA and BN contain a sequence of multiplications and divisions after PG. This pattern requires the system to incorporate a high-cost divider component into the accelerator design. Additionally, the number of multiplications and divisions could be proportional to the number of correlated variables. If an accelerator is to generally accommodate any kind of Bayesian inference, ALUs supporting exponentiation, multiplication, and division need to be incorporated into the system.

However, from the previous section, we concluded that high precision is not a strict requirement for Bayesian inference, so long as all intermediate results stay within a reasonable activation range. With this in mind, we propose a kernel fusion method called Log-Domain Kernel Fusion (LogFusion) for Bayesian inference. Instead of directly computing the result from a long sequence of expensive multiplications and divisions, LogFusion reduces computational cost by performing all computation in the log-domain, where multiplications and divisions are transformed into cheaper additions and subtractions. This is easily shown by the following equation:

$$\frac{\prod_{i=1}^{\#num} a_i}{\prod_{j=1}^{\#denom} b_j} = exp(\sum_{i=1}^{\#num} log(a_i) - \sum_{j=1}^{\#denom} log(b_j)) \tag{11}$$

In the equation, the numerator needs to take the product of all a_i together, while the denominator needs to take the product of all b_j together. Through this conversion, division and multiplication kernels are no longer needed and are replaced by $(\#num+\#denom)$ additions and one subtraction. After a sequence of additions and subtractions has been computed, the output result is converted back to real results using the exp kernel. Additionally, with the help of DyNorm, input values to the exp kernel will always be brought back into an acceptable range for the kernel. Although there is an additional overhead for the exp kernel and the log kernel, the high cost of incorporating sequences of multiplications and divisions still makes this trade-off favorable. Figure 6, shows the Mirco-architecture, combining all optimizations for PG. The PG ALU can calculate any sequence of multiplication and division without the need for a multiplier or divider. Multiple PG ALUs can be used to exploit parallelism. Since DyNorm works on a vector of inputs, it will be shared by multiple PG pipelines. As a result, its additional hardware cost will be amortized by the number of parallel pipelines in the compute core.

An accelerator using LogFusion still has its distinctive advantages, even if it is compared with an alternative accelerator on FPGA using Digital Signal Processor (DSP). For example, the Xillinx DSP48E1 DSP Module supports 25-bit x 18-bit multiplication and 48-bit addition. Using a single

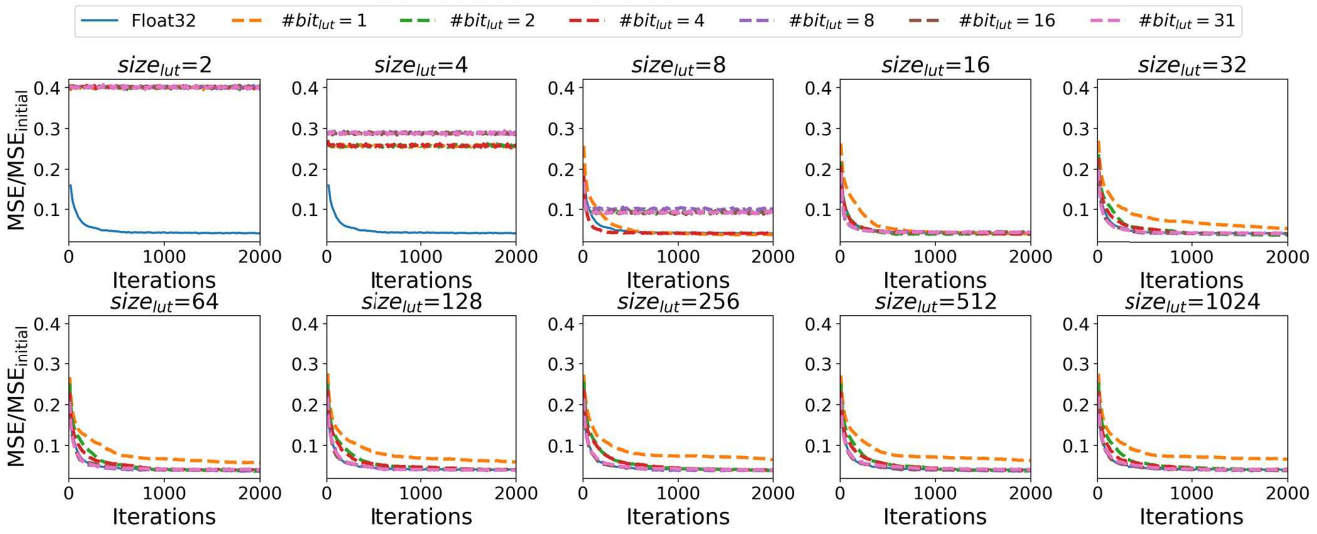

Figure 7: Using smaller $size_{lut}$ and $\#bit_{lut}$ can still provide similar inference quality as the Float32 baseline.

DSP unit, a 32-bit multiplication needs four cycles, but only 1 cycle for 32-bit addition. Even accounting for log and exp conversions (2 cycles), log-domain computation is still faster. Probability computations for Bayesian models usually require many consecutive multiplications and divisions, so benefits easily accumulate. Treating division as inverse multiplication would cause underflow issues for fixed-point formats. Resolving division-by-zero problems requires larger bit widths for mantissa alignment and multiplication. In contrast, LogFusion incurs no additional cost.

D. TreeSampler: Tree-Based Sampler

In addition to the optimizations for PG, operations in SD can also be improved. Sampling to find a new label based on a vector is essentially a vector search operation. Prior implementations apply a for-loop pointer to this kernel. However, this iterative search method requires $\mathcal{O}(N)$ runtime and is not ideal for larger numbers of labels. Utilizing a tree-based structure for search algorithms is common practice to reduce runtime. Taking inspiration from the binary search tree data structure, we propose the TreeGibbs architecture, generalizing the sampling process into a tree structure to achieve $\mathcal{O}(log(N))$ runtime.

A sampler samples a topic from a distribution provided as a vector of probability values P_x for a variable x, which takes a label $n \in [0, N)$, where N is the number of possible labels for x. The vector P_x contains a probability $P_x(n)$ for every possible label of n. The larger $P_x(n)$ is, the more likely the sampler assigns label n to variable x. If computation is done sequentially, P_x may be converted into a vector A_x containing cumulative probabilities $A_x(m) = \sum_{m=0}^{n} P_x(m)$. A threshold value T is generated by multiplying the total sum of P_x (i.e., $A_x(N-1)$) by a random value from the standard uniform distribution. The smallest n that makes $A_x(n)$ larger than T will be assigned to x as its new label. The sampling process requires cumulative summing so, if N is not large, using an iterative compute architecture to compute each probability uses the least amount of logic. This structure requires at least $2N+1$ cycles for the hardware to sample a token. However, if N is very large, the required time for generating a sample increases significantly.

To solve this runtime issue, we propose TreeSampler. Fig. 8 shows a simplified diagram of TreeSampler divided into three modules: *TreeSum, ThresholdGen* and *TraverseTree*. TreeSum is the tree structure that sums all elements of P_x. Each node sums the outputs of its children nodes and passes the summed value to its counterpart in TraverseTree. ThresholdGen generates T by multiplying the total sum by a uniform random number from a hardware Pseudo-random Number Generator (PRNG). TraverseTree enables the tree structure to select the correct leaf node. Each node will compare its parent node's value N_p with its left child node's value N_l. If the left child has a higher value, the node will update its own value to N_p and activate the left child node. Otherwise, the node will update its own value to $N_p - N_l$ and activate the right child node. Using this structure, the runtime for sampling each token is reduced from $\mathcal{O}(2N+1)$ to $\mathcal{O}(log(N))$.

TreeSampler shows great runtime reduction compared to a sequential sampler and scales much better, as shown in Figure 9. As the number of labels in the workload increases, the runtime speedup also increases. This is crucial for workloads that require a higher number of labels for each random variable. While this tree-based design is much more complex, TreeSampler provides far better hardware

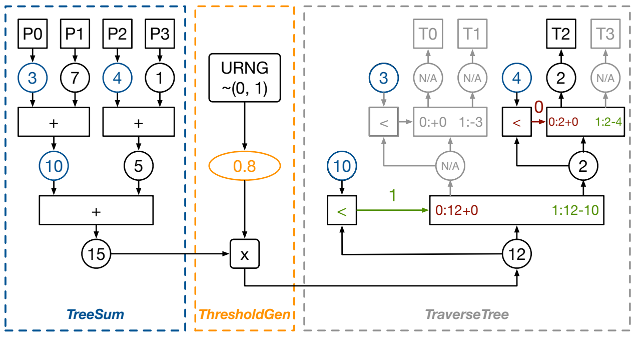

Figure 8: Micro-architecture diagram of TreeSampler

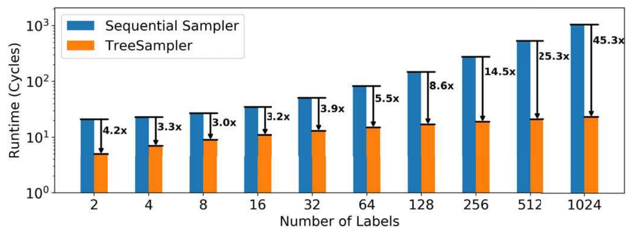

Figure 9: Runtime speedup of TreeSampler scales well with number of labels.

utilization and is more efficient than a sequential sampler.

By adding additional shift registers between the same layers in Tree Sum and TraverseTree, we ensure information can be passed from TreeSum to TraverseTree. This means the entire TreeSampler hardware can operate on several different inputs without interference. The additional shift register from TreeSum to TraverseTree creates a pipelined version of TreeSampler (PipeTreeSampler). Compared to TreeSampler, the runtime is exactly the same, but the maximum throughput is further improved to one sample per cycle. Although much more complex and expensive to build, its hardware efficiency is the best among the sequential sampler, TreeSampler, and itself. More discussion on this topic is in Section IV-C.

IV. Experimental Results

We applied our methods to ten different benchmarks using three types of Bayesian models introduced in Section II. These three algorithms and the various datasets for each provide a good representation of common discrete Bayesian inference workloads.

In this section, we first provide an algorithmic evaluation of DyNorm, TableExp, and LogFusion on ten different workloads. Their combined effect enables low-precision computation without noticeable loss in model performance. Secondly, we will evaluate the effectiveness of hardware cost reduction by jointly applying the presented methods. Thirdly, we will show the throughput speedup from TreeSampler and discuss its hardware resource efficiency. There is no need for a discussion on algorithm inference quality for TreeSampler, because it implements the same sampling algorithm as a vanilla sequential Gibbs sampler.

A. Algorithmic Evaluation of DyNorm, TableExp and LogFusion

1) MRF Benchmarks: In Section III-A, we presented the effects of applying DyNorm and TableExp for Stereo Matching. DyNorm allows MRF inference with a precision as low as 8-bit. Without this, inference results show a significant degradation compared to the Float32 results. In addition, TableExp can further reduce the hardware requirement for the compute pipeline. The algorithm shows strong resilience to lower-precision compute. These patterns do not pertain to Stereo Matching alone. We applied the two techniques to all four benchmarks of MRF.

DyNorm is able to reduce hardware precision without loss of inference quality. In Figure 10, four different applications are provided. For each application, 4-bit and 8-bit fixed-point results are shown on the graph, with the floating point MRF result as reference. For all applications, directly using fixed-point compute is not ideal for inference quality. However, once DyNorm is applied, model inference results immediately show improvement, attaining almost identical quality to the floating point result. For Stereo Matching, there still exists a slight degradation in convergence speed for the 4-bit fixed-point result. Despite this, 8-bit precision with DyNorm allows all applications to reach the same inference quality as the floating point baseline.

Combining all three proposed methods shows their effectiveness across all MRF applications. To investigate how different TableExp design parameters affect model performance, we sweep two key design parameters and record the final converged result. The first parameter is $size_{lut}$, which represents the number of elements stored in the lookup table. The second parameter is $\#bit_{lut}$, which is the number of fractional bits used for each element in the lookup table. Theoretically, larger $size_{lut}$ and $\#bit_{lut}$ should result in better precision for the exp kernel. However, a better precision does not always result in better model convergence. Figure 11 shows convergence results on four different applications. There are very small differences when varying $\#bit_{lut}$, while $size_{lut}$ appears to be more influential on the final converged result. Achieved inference quality is already similar to Float32 once the $size_{lut}$ reaches 32. Intuitively, this behavior is expected as the resolution of the EXP kernel output is determined by $size_{lut}$ and has a larger influence on the determination of discrete labels, moreso than the accuracy of the quantized output values. Considering the convergence speed difference shown in Figure 11, 8 bits are needed to reach the same convergence speed. As a result, using 8-bit precision and $size_{lut}$ of 32 is

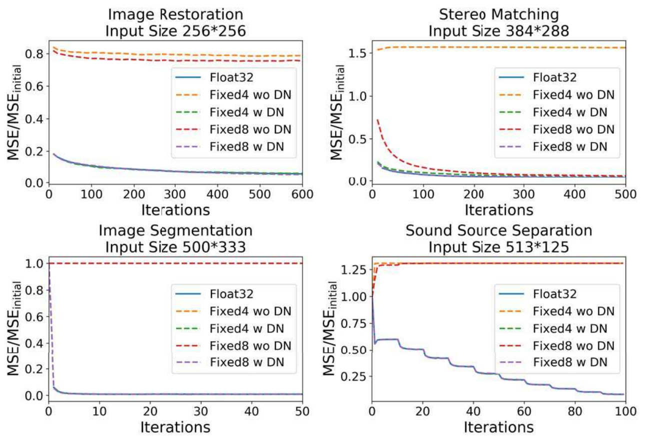

Figure 10: Dynamic normalization makes low-precision MRF reach the same inference quality as the floating-point MRF for four different applications.

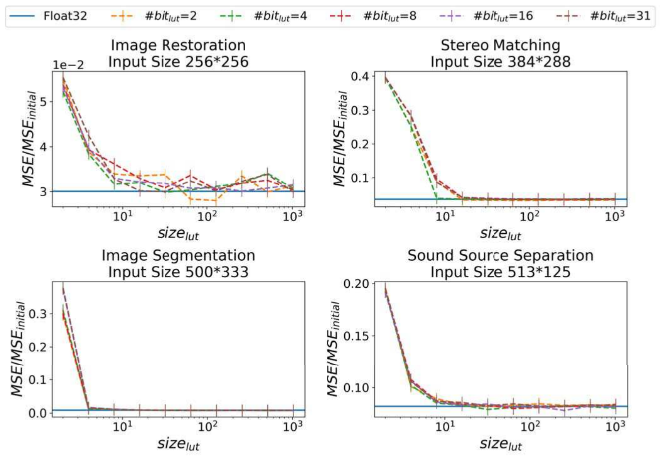

Figure 11: Design parameter sweep of TableExp on MRF. Float32 result is given as a reference baseline. Lower normalized MSE means better inference quality.

sufficient for the hardware to reach the same quality as the floating point algorithm.

2) BN Benchmarks: CoopMC can also be applied to other discrete Bayesian workloads, such as Bayesian networks or latent Dirichlet allocation. Figure 12 shows how $\#bit_{lut}$ and $size_{lut}$ influence the final converged result. Unlike results for MRF, both $\#bit_{lut}$ and $size_{lut}$ significantly influence the converged inference result. Using lower precision values in TableExp produces a dramatic change in the final accuracy. These three Bayesian network workloads' small size contributes to a larger sensitivity to precision. Although the sensitivity to precision affects the converged performance metrics for lower precision LUTs, the converged results are well above the threshold precision of 8 bits. Meanwhile, $size_{lut}$ demonstrates very similar behavior to the MRF

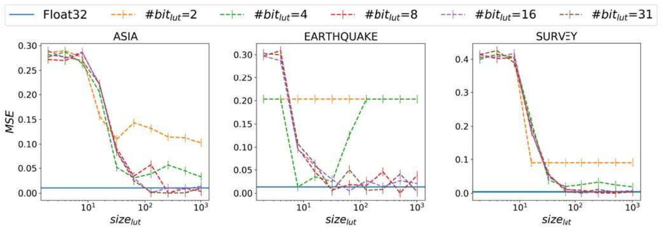

Figure 12: Design parameter sweep of TableExp on BN. Float32 result is given as a reference baseline. Lower MSE means better inference quality.

applications. A larger $size_{lut}$ will lead to better convergence results for Bayesian networks, until a threshold has been reached, after which convergence results remain similar. For the three tested Bayesian network workloads, the threshold is 128.

3) LDA Benchmarks: Figure 13 shows inference results of LDA after applying DyNorm, TableExp, and LogFusion. The x-axis shows different sizes for the lookup table $size_{lut}$, and the y-axis shows the log-likelihood, a common evaluation metric representing the probability of the current state of an LDA model, given an observed dataset. A higher probability implies better model inference quality. The points in the plots show convergence results for their corresponding dataset. Each line represents the number of bits $\#bits_{lut}$ used for lookup table entries in TableExp.

Experimental results from LDA shows a very similar trend to the BN results. Both $size_{lut}$ and $\#bit_{lut}$ have significant impact on inference quality. For $size_{lut}$, the inference result shows a clear saturation trend, once a certain threshold has been reached. For all three cases, $size_{lut}$ of 128 is sufficient for the convergence result to reach parity with floating point. For $\#bit$, the requirements are tighter than what was observed with BN. To match the full-precision version, 16-bit precision is required. Also, the separation between lines with different $\#bit_{lut}$ is much clearer than with the MRF or BN results.

The higher precision requirement results from the high connectivity between different variables in the model, due to the nature of the LDA algorithm. Unlike MRF or BN, each variable is connected to every other variable within the same document, as well as every other instance of the same vocabulary in the corpus. This implies the probability distributions for the labels of connected variables are similar. The only way to distinguish these variables from each other is by the slight differences in their probabilities. However, lower precision fails to represent those small differences, which eventually results in poor performance. Nevertheless, DyNorm, TableExp, and LogFusion still give the same level of performance as the baseline, given $\#bit_{lut} \geq 16$ and $\#size_{lut} \geq 128$.

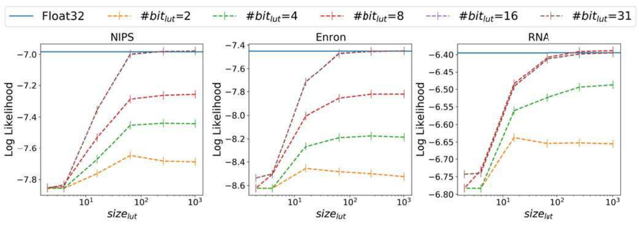

Figure 13: Design parameter sweep of TableExp on LDA. Float32 result is given as a reference baseline. Higher log-likelihood implies better inference quality.

Type	Area for Divider Baseline (um^2)					
	Divider				Total	Reduction
Baseline	3831				3831	1×
Type	**Area for LogFusion (um^2)**					
	LOG	ADD	DN	EXP	Total	Reduction
DN+LF	267	76	84	830	1257	3.05×
DN+LF+TE	267	76	84	80	507	7.56×

Table III: Hardware Area Comparison among DyNorm(DN) + LogFusion(LF), DyNorm(DN) + LogFusion(LF) + TableExp(TE) and Divider Baseline.

B. DyNorm, TableExp, and LogFusion Hardware Evaluation

Combining Dynamic Normalization, LUT-Based EXP kernel and log-domain Kernel Fusion shows great potential in reducing hardware precision requirements and unnecessary compute kernels. To evaluate CoopMC from a hardware resource perspective, we implement our proposed methods and synthesize our design in RTL. The accelerator RTL is synthesized using a commercial EDA tool, Cadence Genus, based on the commercial GlobalFoundries 12nm technology. Furthermore, the memories in the accelerator are generated using a commercial 12nm SRAM compiler to obtain realistic area and energy numbers. The frequency of the accelerator is signed off at 500 MHz at 0.8V based on a typical process corner. The hardware area estimation is shown in Table III.

The first row of the table shows the baseline logic area requirements of a pipelined 32-bit divider. As a reference point, it does not use any hardware optimization methods. LogFusion can remove the need for a divider by introducing additional components to the hardware. The baseline sampler is replaced by a log kernel, an add kernel, a DyNorm kernel, and an exp kernel. For both design choices using LogFusion, ADD is the addition/subtraction kernel, which is used by the hardware to compute division or multiplication in log-domain. DN is the hardware for DyNorm, and its design is shown in Figure 3. Since DyNorm is shared by multiple pipelines for generating probabilities, its hardware cost is also amortized by the number of parallel inputs it handles. The reported area cost has been averaged by the number of parallel input pipelines.

In the case of DyNorm+LogFusion(DN+LF), both log and exp kernels are 32-bit approximation-function-based kernels (16 bits each, for the integer and fractional parts). Due to the higher precision and complexity of the approximation function, the exp ALU has a relatively high area cost, contributing to most of the hardware used for DN+LF. This design choice still has a lower cost than that of the divider, providing more than 3.05× reduction in chip area. In the case of DyNorm+LogFusion+TableExp(DN+LF+TE), an additional layer of optimization is applied to the design. Rather than using the approximation-based exp kernel, TableExp helps to further reduce area overhead. The only major component in the lookup table is a ROM containing pre-computed values, so TableExp is only 10% of its counterpart's size.

This reduction improves the Kernel Fusion area reduction to more than 7.56×. One thing to note is that the hardware size we are considering here is the largest LUT dimension (32-bit with $size_{lut}$ of 1024), as discussed in previous sections. If further low-precision optimization is applied for the LUT, the hardware area reduction continues to improve, making 7× a lower bound. With these area savings, the benefits of using Kernel Fusion significantly outweighs the overhead of adding more components into the design. Any future accelerator design for discrete Bayesian inference could benefit from this methodology.

C. Hardware Evaluation of TreeSampler

In previous works on accelerating Gibbs sampling for discrete Bayesian inference [16], [35], discrete sampling is implemented as a sequential process with a runtime of $\mathcal{O}(N)$. With our tree-based sampler, the runtime is significantly reduced to $\mathcal{O}(log(N))$. The reduction in runtime is already shown in Figure 8. However, the TreeSampler structure is considerably more complex than its sequential counterpart. To further investigate the hardware area usage of TreeSampler, we implemented a sequential sampler, TreeSampler, and pipelined TreeSampler (PipeTreeSampler), and synthesized them using GlobalFoundries' 12nm technology node. Their hardware area comparison is shown in Figure 14.

As shown in the plot, TreeSampler and PipeTreeSampler are significantly more costly than the sequential sampler. This would seem to indicate a large sacrifice in hardware resource efficiency for the $\mathcal{O}(log(N))$ runtime. However, a more useful metric is throughput per unit area, shown in Figure 15. One thing to note is that the TreeSampler design is determined by $\#labels$, not $\#variables$. The $\#labels$ represent possible discrete labels each random variable could take, while $\#variables$ is the number of random variables for a workload. Although larger workloads, like LDA-Enron might have millions of variables, they just need 128 for their $\#labels$, shown by Table I. TreeSampler samples a new label from a vector with the size of $\#labels$, not

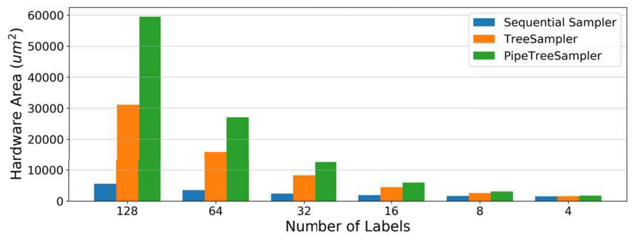

Figure 14: Hardware area of various sampler design for different number of labels.

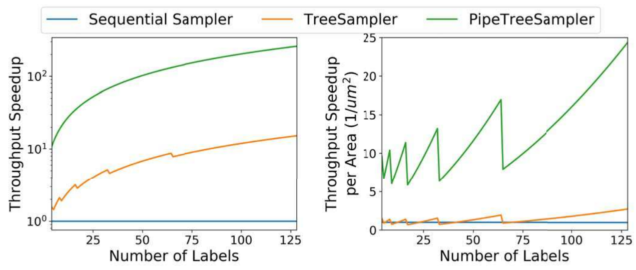

Figure 15: Throughput and area efficiency of various sampler design for different number of labels.

$\#variables$, so we sweep the design parameters from 2 to 128 in those two plots.

The left plot shows the relative throughput speedup from using the three types of discrete sampler, with the sequential sampler throughput used as the reference point. The speedup for TreeSampler shows a step function pattern: due to the nature of TreeSampler, its runtime will be the smallest integer that is larger than $log(N)$. That is, in between two exponents of 2, its throughput will remain constant.

The right plot shows throughput speedup normalized by area. PipeTreeSampler always leads in performance and efficiency, albeit at higher hardware cost. The amount of throughput generated per μm^2 is still superior to the sequential sampler for most cases. Due to the step function behavior, it can be slightly less efficient than the sequential sampler in some cases, but still offers better throughput per core, which could condense a multi-core design into a single-core one while maintaining the same level of hardware throughput. Also, as the number of labels increases, maximum speedup also increases, and minimum throughput per area improves. Compared to previous designs with 64 labels, TreeSampler provides 8.7× speedup while being 1.9× more area-efficient. Generally, when there is sufficient accelerator area to use a TreeSampler, it will provide better throughput, latency, and area efficiency compared to a sequential sampler.

Version	Logic Area (um^2)		Power (mW)		Speedup
$V_{Baseline}$	14491	100%	7.96	100%	1×
V_{PG}	9719	67%	3.08	38%	0.98×
V_{TS}	25657	177%	14.9	187%	1.59×
V_{PG+TS}	19874	137%	9.53	120%	1.53×

Table IV: Logic area and estimated power comparison between different versions of end-to-end implementation

D. Hardware End-to-End Case Study

To ensure past and future accelerator designs can easily benefit from our methods, we designed CoopMC such that significant overhauls to the overall architecture or memory system are unnecessary. CoopMC can serve as a well-optimized plug-and-play design to improve the performance and area usage of computational kernels.

More concretely, we reconstructed previous work by implementing a similar MCMC computational core, which is based on the design of the Gibbs Sampler in [16], and the SPU in [36]. The computational core is designed for both the PG and SD steps, and is implemented using GlobalFoundries' 12nm libraries, the same technology node used for previous area comparisons. We benchmark our design with a 64-label MRF workload. As a baseline, $V_{Baseline}$ of this computational core follows the naive design, using a single PG compute pipeline and a discrete sampler, with full 32-bit precision. V_{PG} adopts the PG step optimizations, which are DyNorm, TableExp and LogFusion. V_{TS} adopts TreeSampler for the SD step without using the optimization methods in V_{PG}. V_{PG+TS} combines all optimizations for the best performance and efficiency. Compute logic area and power estimates are shown in Table IV.

As shown by the table, V_{PG} shows 33% area and 62% power improvement compared to the baseline. The combination of DyNorm, TableExp and LogFusion shows significant improvements in the MCMC computational core and maintains its advantages even for end-to-end implementations. V_{TS} requires more area and power, but it provides 59% end-to-end cycle speedup. Using PG step optimizations, we can further reduce the area and energy cost in V_{PG+TS} to achieve a 1.53× speedup, requiring only 37% more area and 20% more power. With more parallel pipelines for the PG step, end-to-end speedup could be further improved. The combined advantages of our proposed optimization methods significantly out-weigh their potential drawbacks.

Our RTL implementation follows in the spirit of [16, 36], accelerating an MRF workload with 64 labels and streams in data cost. With those assumptions, computing each variable inside the accelerator engine requires 2072 bits for reading and 6 bits for output. Based on the roofline model, we are compute-limited if 2078 bits are transferable within the computation duration. The baseline ($V_{Baseline}$) has a threshold bandwidth of 15 bits/cycle while the fully optimized version (V_{PG+TS}) requires 22 bits/cycle. This is easily achievable using 32-bit SRAM, consuming 8.8mW. After applying our optimizations, memory bandwidth is not the limiting factor, so optimizations for PG and SD directly contribute to end-to-end performance.

V. Related Works

MCMC-based Bayesian inference is very computationally intensive, and its inherent sequential requirements make parallelism difficult to exploit. Much of the existing literature focuses on algorithm improvements such as methods to parallelize the sequential algorithm while minimizing added bias. Some of these methods relax some of the mathematical properties to create a nearly identical parallelized version. MultiBUGS [37] is an example of work that does not alter statistical guarantees while parallelizing sampling. It extends a well-known Bayesian modeling software called BUGS [38] with multi-core functionality, by taking tasks within MCMC that can be calculated in parallel, including operations such as likelihood computation, and runs them on multiple cores while also sampling conditionally independent variables in parallel. [39] presents a suite of Bayesian inference workloads, BayesSuite, with characterization and profiling results on various processors with different microarchitectures. It also presents schedule and optimization techniques for faster execution on the processors.

Various works also present hardware implementations for Bayesian inference architectures that outperform MultiBUGS, including FPGA-based implementations. [40] is an implementation of Hamiltonian Monte Carlo sampling, an MCMC variant known to be efficient for sampling continuous distributions. [14] presents a compiler to generate efficient FPGA implementations using predefined hardware templates for parallel execution of MCMC sampling methods. [15], [41] present FPGA-based accelerators that show significant gains over ARM CPUs for audio processing and computer vision tasks in a mobile setting. They demonstrate an architecture that concurrently samples conditionally independent variables for parallel Gibbs sampling.

Additionally, several works also propose ASIC solutions. [16], [35] have presented the first programmable Bayesian inference accelerator for computer vision and audio processing on mobile settings. These works support asynchronous (or Hogwild!) Gibbs sampling as well as parallelization of conditional independent variables, enabling an additional level of parallelism on top of the other existing work mentioned above. Another implementation of these parallel Gibbs sampling architectures optimized specifically for sound source separation to be applied for automatic speech recognition SoC [42]. [17] is another chip implementation that uses an exact deterministic inference called Sum-Product Networks.

Most previous designs focus on how to parallelize MCMC-based inference. In most of their implementations, their computational core design is naive and without sufficient optimizations. Taking another approach, [36] investigated the statistical robustness of MCMC accelerators when using reduced precision for probabilities to improve efficiency. This work defined sampling quality, convergence diagnostics, and goodness of fit as metrics for qualitative evaluation of correctness for probabilistic accelerators. CoopMC looks into optimizations for the entire MCMC computational pipeline. Our work includes, but is not limited to, techniques such as reduced precision to increase efficiency, and takes advantage of statistical robustness for optimizing MCMC. Our optimizations do not rely on changes to the parallel architecture of computational cores or their memory hierarchies. Consequently, our design can be used in conjunction with the previous hardware approaches and provide benefits for past and future designs.

VI. Conclusion

We present CoopMC, an algorithm-architecture co-optimization for Markov Chain Monte Carlo accelerators. We generalize MCMC-based Bayesian inference into three computational steps: probability generation (PG), sampling from distributions (SD), and parameter updates (PU). Based on the numerical characteristics of the PG step, we propose DyNorm, TableExp and LogFusion to jointly exploit the low-precision robustness of Bayesian inference and avoid unnecessary division and multiplication kernels. Fusion of these techniques help reduce the required precision and shrink the computational kernel area cost by up to 7.5×. For the SD step, we propose TreeSampler to reduce hardware runtime dramatically, from $\mathcal{O}(N)$ to $\mathcal{O}(log(N))$. This results in an 8.7× speedup, compared to the published state-of-the-art Gibbs sampler architecture, while simultaneously increasing area efficiency by 1.9×. In an end-to-end case study, CoopMC shows a 33% logic area reduction and 62% power reduction, and that a 1.53× speedup can be achieved with better area and power efficiency. All of our proposed methods have been tested on ten diverse workloads, using three different types of Bayesian models, without noticeable reduction in model performance. The general applicability of these methods suggests its extensibility to other Bayesian models.

Acknowledgment

The authors would like to thank Dr. Tianyu Jia for providing hardware synthesis technical support and ARM Inc. for providing SRAM synthesis library. This work was funded in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation (SRC) program co-sponsored by DARPA; NSF Awards 1718160 (Virtualized Accelerators for Scalable, Composable Architectures); Harvard University Physical Sciences and Engineering Accelerator Award.

References

[1] L. Hong and R. Martin, "A flexible bayesian nonparametric model for predicting future insurance claims," *North American Actuarial Journal*, vol. 21, no. 2, pp. 228–241, 2017. [Online]. Available: https://doi.org/10.1080/10920277.2016.1247720

[2] *The Bayesian Paradigm: Likelihood Function and Bayes' Theorem*. John Wiley & Sons, Ltd, 2012, ch. 2, pp. 6–21. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119202141.ch2

[3] E. Lesaffre, G. Baio, and B. Boulanger, *Bayesian Methods in Pharmaceutical Research*, 05 2020.

[4] K. Osawa, S. Swaroop, A. Jain, R. Eschenhagen, R. E. Turner, R. Yokota, and M. E. Khan, "Practical deep learning with bayesian principles," in *NeurIPS*, 2019.

[5] D. Manevski, N. Ružić Gorenjec, N. Kejžar, and R. Blagus, "Modeling covid-19 pandemic using bayesian analysis with application to slovene data," *Mathematical Biosciences*, vol. 329, p. 108466, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0025556420301279

[6] A. C. S. de Oliveira, L. H. M. Morita, E. B. da Silva, L. A. R. Zardo, C. J. F. Fontes, and D. C. T. Granzotto, "Bayesian modeling of covid-19 cases with a correction to account for under-reported cases," *Infectious Disease Modelling*, vol. 5, pp. 699–713, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S246804272030049X

[7] A. Berihuete, M. Sanchez-Sanchez, and A. Suarez-Llorens, "A bayesian model of covid-19 cases based on the gompertz curve," *Mathematics*, vol. 9, no. 3, 2021. [Online]. Available: https://www.mdpi.com/2227-7390/9/3/228

[8] S. Farquhar, M. A. Osborne, and Y. Gal, "Radial bayesian neural networks: Beyond discrete support in large-scale bayesian deep learning," in *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, S. Chiappa and R. Calandra, Eds., vol. 108. PMLR, 26–28 Aug 2020, pp. 1352–1362. [Online]. Available: http://proceedings.mlr.press/v108/farquhar20a.html

[9] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1050–1059. [Online]. Available: http://proceedings.mlr.press/v48/gal16.html

[10] M. E. Khan, D. Nielsen, V. Tangkaratt, W. Lin, Y. Gal, and A. Srivastava, "Fast and scalable bayesian deep learning by weight-perturbation in adam," in *ICML*, 2018.

[11] A. Kucukelbir, D. Tran, R. Ranganath, A. Gelman, and D. M. Blei, "Automatic differentiation variational inference," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 430–474, 2017.

[12] M. D. Hoffman, D. M. Blei, C. Wang, and J. Paisley, "Stochastic variational inference." *Journal of Machine Learning Research*, vol. 14, no. 5, 2013.

[13] S. Wang, X. Zhang, Y. Li, R. Bashizade, S. Yang, C. Dwyer, and A. R. Lebeck, "Accelerating markov random field inference using molecular optical gibbs sampling units," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 558–569.

[14] S. S. Banerjee, Z. T. Kalbarczyk, and R. K. Iyer, "Acmc 2: Accelerating markov chain monte carlo algorithms for probabilistic models," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 515–528. [Online]. Available: https://doi.org/10.1145/3297858.3304019

[15] G. G. Ko, Y. Chai, R. A. Rutenbar, D. Brooks, and G. Wei, "Flexgibbs: Reconfigurable parallel gibbs sampling accelerator for structured graphs," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 334–334.

[16] G. G. Ko, Y. Chai, M. Donato, P. N. Whatmough, T. Tambe, R. A. Rutenbar, D. Brooks, and G. Y. Wei, "A 3mm2 programmable bayesian inference accelerator for unsupervised machine perception using parallel gibbs sampling in 16nm," in *2020 IEEE Symposium on VLSI Circuits*, 2020, pp. 1–2.

[17] N. Shah, L. I. G. Olascoaga, S. Zhao, W. Meert, and M. Verhelst, "9.4 piu: A 248gops/w stream-based processor for irregular probabilistic inference networks using precision-scalable posit arithmetic in 28nm," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 150–152.

[18] S. Hooker, "The hardware lottery," *ArXiv*, vol. abs/2009.06489, 2020.

[19] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, and C. Rother, "A comparative study of energy minimization methods for markov random fields with smoothness-based priors," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 6, pp. 1068–1080, 2008.

[20] J. H. Kappes, B. Andres, F. A. Hamprecht, C. Schnörr, S. Nowozin, D. Batra, S. Kim, B. X. Kausler, J. Lellmann, N. Komodakis, and C. Rother, "A comparative study of modern inference techniques for discrete energy minimization problems," in *2013 IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 1328–1335.

[21] K. Korb and A. Nicholson, *Bayesian Artificial Intelligence*, ser. Chapman & Hall/CRC Computer Science & Data Analysis. CRC Press, 2010. [Online]. Available: https://books.google.com/books?id=LxXOBQAAQBAJ

[22] T. Nielsen and F. JENSEN, *Bayesian Networks and Decision Graphs*, ser. Information Science and Statistics. Springer New York, 2009. [Online]. Available: https://books.google.com/books?id=37CAgCykQaAC

[23] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 50, no. 2, pp. 157–224, 1988. [Online]. Available: http://www.jstor.org/stable/2345762

[24] K. Korb and A. Nicholson, *Bayesian artificial intelligence, second edition*. Australia: CRC Press, Jan. 2010.

[25] M. Scutari and J.-B. Denis, *Bayesian Networks with Examples in R*. Boca Raton: Chapman and Hall, 2014, iSBN 978-1-4822-2558-7, 978-1-4822-2560-0.

[26] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, no. null, p. 993–1022, Mar. 2003.

[27] T. Griffiths and M. Steyvers, "Finding scientific topics," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 101 Suppl 1, pp. 5228–35, 04 2004.

[28] J. Foulds, L. Boyles, C. DuBois, P. Smyth, and M. Welling, "Stochastic collapsed variational bayesian inference for latent dirichlet allocation," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 446–454. [Online]. Available: https://doi-org.ezp-prod1.hul.harvard.edu/10.1145/2487575.2487697

[29] L. Yao, D. Mimno, and A. McCallum, "Efficient methods for topic model inference on streaming document collections," in *KDD*, 2009.

[30] X. Xie, Y. Liang, X. Li, and W. Tan, "Culda: Solving large-scale lda problems on gpus," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 195–205. [Online]. Available: https://doi.org/10.1145/3307681.3325407

[31] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T.-Y. Liu, and W.-Y. Ma, "Lightlda: Big topic models on modest computer clusters," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2015, p. 1351–1361. [Online]. Available: https://doi.org/10.1145/2736277.2741115

[32] ——, "Lightlda: Big topic models on modest computer clusters," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2015, p. 1351–1361. [Online]. Available: https://doi.org/10.1145/2736277.2741115

[33] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, "Binary neural networks: A survey," *Pattern Recognition*, vol. 105, p. 107281, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0031320320300856

[34] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *ECCV*, 2016.

[35] G. Ko, Y. Chai, M. Donato, P. N. Whatmough, T. Tambe, R. A. Rutenbar, G. Wei, and D. Brooks, "A scalable bayesian inference accelerator for unsupervised learning," in *2020 IEEE Hot Chips 32 Symposium (HCS)*. Los Alamitos, CA, USA: IEEE Computer Society, aug 2020, pp. 1–27. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HCS49909.2020.9220686

[36] X. Zhang, R. Bashizade, Y. Wang, S. Mukherjee, and A. R. Lebeck, "Statistical robustness of markov chain monte carlo accelerators," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 959–974. [Online]. Available: https://doi.org/10.1145/3445814.3446697

[37] R. J. Goudie, R. M. Turner, D. De Angelis, and A. Thomas, "Multibugs: A parallel implementation of the bugs modelling framework for faster bayesian inference," *Journal of statistical software*, vol. 95, 2017.

[38] D. J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter, "Winbugs-a bayesian modelling framework: concepts, structure, and extensibility," *Statistics and computing*, vol. 10, no. 4, pp. 325–337, 2000.

[39] Y. Emma Wang, Y. Zhu, G. G. Ko, B. Reagen, G. Wei, and D. Brooks, "Demystifying bayesian inference workloads," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 177–189.

[40] B. Darvish Rouhani, M. Ghasemzadeh, and F. Koushanfar, "Causalearn: Automated framework for scalable streaming-based causal bayesian learning using fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–10. [Online]. Available: https://doi.org/10.1145/3174243.3174259

[41] G. G. Ko, Y. Chai, R. A. Rutenbar, D. Brooks, and G. Wei, "Accelerating bayesian inference on structured graphs using parallel gibbs sampling," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 159–165.

[42] T. Tambe, E. Y. Yang, G. G. Ko, Y. Chai, C. Hooper, M. Donato, P. N. Whatmough, A. M. Rush, D. Brooks, and G. Y. Wei, "9.8 a 25mm2 soc for iot devices with 18ms noise-robust speech-to-text latency via bayesian speech denoising and attention-based sequence-to-sequence dnn speech recognition in 16nm finfet," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 158–160.