CausaLearn: Automated Framework for Scalable Streaming-based Causal Bayesian Learning using FPGAs

Bita Darvish Rouhani
UC San Diego
bita@ucsd.edu

Mohammad Ghasemzadeh
UC San Diego
mghasemzadeh@ucsd.edu

Farinaz Koushanfar
UC San Diego
farinaz@ucsd.edu

ABSTRACT

This paper proposes CausaLearn, the first automated framework that enables real-time and scalable approximation of Probability Density Function (PDF) in the context of causal Bayesian graphical models. CausaLearn targets complex streaming scenarios in which the input data evolves over time and independence cannot be assumed between data samples (e.g., continuous time-varying data analysis). Our framework is devised using a HW/SW co-design approach. We provide the first implementation of Hamiltonian Markov Chain Monte Carlo on FPGA that can efficiently sample from the steady state probability distribution at scales while considering the correlation between the observed data. CausaLearn is customizable to the limits of the underlying resource provisioning in order to maximize the effective system throughput. It uses physical profiling to abstract high-level hardware characteristics. These characteristics are integrated into our automated customization unit in order to tile, schedule, and batch the PDF approximation workload corresponding to the pertinent platform resources and constraints. We benchmark the design performance for analyzing various massive time-series data on three FPGA platforms with different computational budgets. Our extensive evaluations demonstrate up to *two orders-of-magnitude* runtime and energy improvements compared to the best-known prior solution. We provide an accompanying API that can be leveraged by data scientists and practitioners to automate and abstract hardware design optimization.

ACM Reference Format:
Bita Darvish Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar. 2018. CausaLearn: Automated Framework for Scalable Streaming-based Causal Bayesian Learning using FPGAs. In *Proceedings of 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. ACM, Monterey, CA, USA, 10 pages. https://doi.org/10.1145/3174243.3174259

1 INTRODUCTION

Probabilistic learning and graphical modeling of time-series data with causal structure is a challenging task in various scientific fields, ranging from machine learning [1] and stochastic optimization [2] to economics [3] and medical imaging [4]. Bayesian networks are an important class of directed graph analytics used to model dynamic systems. Unlike undirected graphical networks such as Markov Random Field, Bayesian networks are capable of learning causal structure in time-series data. In a Bayesian network, the posterior Probability Density Function (PDF) over the model parameters should be continuously updated to accommodate for the newly added structural trends as data evolves over time. Dynamic (a.k.a., *streaming*) learning of random variables is particularly important in *time-series data analysis* to enable effective decision making before the system encounters natural changes, rendering much of the collected data irrelevant to the current decision space.

Energy and runtime efficiency play a key role in building viable computing systems for analyzing massive and densely correlated data. Several recent theoretical works have shown the importance of data and model parallelism in analyzing Bayesian graphical networks [5–8]. These set of works, however, are designed at the algorithmic and data abstraction level and are oblivious to the hardware characteristics. Given the diminishing benefits of technology scaling, it is important to devise specialized hardware accelerators for efficient realization of different learning models [9, 10]. A number of prior research works have provided FPGA accelerators for Bayesian networks, e.g., [11, 12]. Although these works demonstrate significant improvement for deployment of specific Bayesian models, their predominant assumption is that data samples are independently and identically drawn from a certain distribution. As such, they cannot effectively capture dynamic data correlation in causal streaming applications (e.g., correlated time-series data).

We propose CausaLearn, the first *scalable FPGA* framework to compute on and update *continuous* random variables and their associated PDFs for *streaming-based* causal Bayesian analysis. Our key observation is that without simultaneous optimization of hardware resource allocation and algorithmic solution, the best performance efficiency cannot be achieved. To fulfill this objective, CausaLearn incorporates hardware characteristics into the higher-level hierarchy of the algorithmic solution and enables automated customization per application data and/or physical constraints. In particular, CausaLearn performs a one-time hardware physical profiling to find the pertinent resource constraints (e.g., memory bandwidth, computing power, and available energy). This information is automatically integrated into CausaLearn's resource-aware customization unit to tile, schedule, and batch the pertinent computational workload such that it best fits the platform. CausaLearn's automated compilation disengages users from hardware resource optimization task while providing synthesizable solutions that are co-optimized for the underlying hardware architecture and execution schedule.

CausaLearn leverages Gaussian processes (GP) to capture data dynamics in streaming settings. GP form a core methodology in probabilistic machine learning [13, 13, 14] to model the causality structure of time-series data. Markov Chain Monte Carlo (MCMC) is the mainstream method that is used in practice to explore the state space of probabilistic models such as GP. Given the wide range of MCMC applications, it is thus not surprising that a number of implementations on CPUs [15], GPUs [16–20], and FPGAs [11, 12, 21, 22] have been reported in the literature. MCMC incurs a complex data flow consisting of various sequential computing kernels to construct the pertinent Markov chain. As such, FPGAs provide a more flexible programmable substrate for MCMC acceleration compared to GPU accelerators that are particularly designed for Single Instruction Multiple Data (SIMD) operations. The existing works on FPGA, however, have mainly focused on the acceleration of MCMC for analyzing independent and identically distributed (i.i.d.) samples that are drawn from a simple multivariate Gaussian distribution, e.g., [11, 12]. Such assumption, however, does not hold for dynamic Bayesian analytics with causal structure as we illustrate in our practical design experiments. Perhaps, the only prior works on FPGA that have considered causal data dependency

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FPGA '18, February 25–27, 2018, Monterey, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5614-5/18/02…$15.00
https://doi.org/10.1145/3174243.3174259

Table 1: Common MCMC methodologies for analyzing Bayesian networks.

MCMC Methods	Description
Population-based	Population-based MCMC is a method designed to address the issue of multi-modality using a population of Markov chains. This method is particularly inefficient for analyzing high-dimensional data, due to the high cost of unnecessary space exploration.
State Space Model	State Space Model (SSM) MCMC targets Bayesian applications in which evaluating the closed-form PDF is not feasible. SSMs assumes the availability of unbiased estimators to compute the acceptance ratio in each MCMC step. This assumption does not often hold in practice.
Gibbs Sampling	Gibbs sampling decomposes the proposal distribution into its individual components by computing the full conditional distribution of the variable θ_i conditional on all the remaining ones. Gibbs sampling encounters serious computational inefficiency in solving high-dimensional tasks with highly correlated variables.
Slice Sampling	Slice sampling method uniformly samples from the area under the $p(\theta)$ graph as an equivalent to sampling from the probability distribution. This technique improves mixing performance in learning tasks with highly correlated variables. The complexity of Slice sampling scales exponentially with the data dimensionality.
Hamiltonian	Hamiltonian MCMC method uses the gradient of the target probability distribution to select better movements in each iteration. This method is particularly of interest as it can handle both *strong correlations* and *high-dimensionality* of the probability distribution.
Adaptive	Adaptive MCMC method adjusts the proposal distribution in the execution time to achieve a better sampling efficiency. The adaptive kernel might converge to a non-stationary distribution if not designed carefully.

in the context of Bayesian networks are [21, 22]. Authors in [21, 22] have used Dirichlet processes in *discrete space* to facilitate human T-cell analysis. We emphasize that due to the discrete nature of Dirichlet processes these works are inapplicable to the analysis of dynamic *continuous* random variables.

CausaLearn adopts Hamiltonian Markov Chain Monte Carlo (H_MCMC) to effectively explore the state space of GP parameters by moving toward the gradient of the associated PDF given the observed data samples. The prior MCMC acceleration works on FPGA, e.g., [11, 12, 21, 22] leverage random walks to sample from the target density function. Exploration of the parameters' space using random walks is particularly inefficient in analyzing *high-dimensional streaming* data due to the high cost of mitigating the impact of an unnecessary movement in constructing the Markov chain. CausaLearn overcomes this inefficiency by moving toward the gradient of the model using Hamiltonian dynamics. Computing the gradient of the target density function involves a variety of operations with complex data flows. CausaLearn provides a set of novel algorithmic and hardware optimization techniques to enable real-time execution of H_MCMC algorithm using FPGAs. In particular, our optimization includes: (i) Revising the conventional H_MCMC routine to iteratively update the corresponding gradients of the probability function using incremental data decomposition. Our algorithmic modification effectively reduces the hardware implementation complexity of computing the inverse of large matrices with no drops in the output's accuracy. (ii) Devising an automated tree-based memory management system that facilitates multiple concurrent loads/stores in order to effectively increase the system throughput by enabling data parallelism to the limits of the hardware resources. (iii) Designing an automated compilation tool to tile and schedules matrix-based computations to best fits the data dimensionality and the available resource provisioning.

We provide an accompanying API to make CausaLearn available to a broader community who rely on probabilistic data analysis and often have a limited hardware design expertise. Our API libraries can be leveraged for deployment of widely used classes of data analytics such as various regression and classification methods, belief propagation, expectation maximization, and neural networks. In summary, our explicit contributions are as follows:

- Introducing CausaLearn, the first *scalable* framework that enables *automated real-time* multi-dimensional PDF approximation for *causal* Bayesian analysis. CausaLearn provides support for streaming settings where the latent variables should be adaptively updated as data evolves over time.
- Developing a resource-aware customization tool to optimize system performance. Our automated optimization attains a balance between parallel operations and data reuse by slicing the computation and configuring the design to best fit the intrinsic physical resources and constraints.
- Devising the first scalable floating-point realization of causal Gaussian processes on FPGA by adopting stochastic Hamiltonian Markov Chain Monte Carlo (H_MCMC).
- Designing an accompanying API to facilitate automation and adaptation of CausaLearn for rapid prototyping of an arbitrary causal Bayesian data analysis. Our API minimizes the required user interaction while providing high performance and efficiency gains for FPGA acceleration.
- Providing proof-of-concept evaluations by analyzing large time-series data on three FPGA platforms with different computational budgets. Our evaluations demonstrate up to 320-fold runtime and 770-fold energy improvement compared to a highly-optimized software deployment.

2 PRELIMINARIES

Decomposition of time-series data into estimated latent variables provides an important alternative view from the time domain perspective [1, 2]. Let us denote the input data samples $\mathbf{D}$ as the pair of $(\mathbf{x}, \mathbf{y})$ values, where $\mathbf{x} = \{x_i = [x_{i1}, ..., x_{id}]\}_{i=1}^{n}$ includes the input data features and $\mathbf{y} = [y_1, ..., y_n]$ are the observation values. Here, d is the feature space size and n specifies the number of data measurements that may grow over time. Each output observation y_i can be either continuous as in most regression tasks, or discrete as in classification applications. The key to performing Bayesian graph analytics is to find a *probabilistic likelihood function* that maps each input feature x_i to its corresponding observation y_i such that:

$$y_i = f(x_i) + \epsilon_i. \tag{1}$$

The variable ϵ_i is an additive observation noise that determines how different the observation vector y_i can be from the latent function value $f(x_i)$. The observation noise is usually modeled as a Gaussian distribution variable with zero mean and a variance of σ_n^2.

2.1 Gaussian Processes

In probabilistic graphical models, all parameters should be represented as random variables. *Gaussian processes* are commonly used as the prior density over the set of latent functions $\{f(x_i)\}_{i=1}^{n}$ for analyzing time-series data. In Gaussian processes, each data point x_i is associated with a Normally distributed random variable f_i. Every finite collection of those random variables has a multivariate Gaussian distribution. GP is represented as:

$$\mathbf{f}(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), K(\mathbf{x}, \mathbf{x}')), \tag{2}$$

where $m(\mathbf{x})$ and $K(\mathbf{x}, \mathbf{x}')$ are the mean and covariance kernels that capture the correlation between data samples. With a GP prior, the observations $\mathbf{y} = [y_1, ..., y_n]$ can be assumed to be conditionally

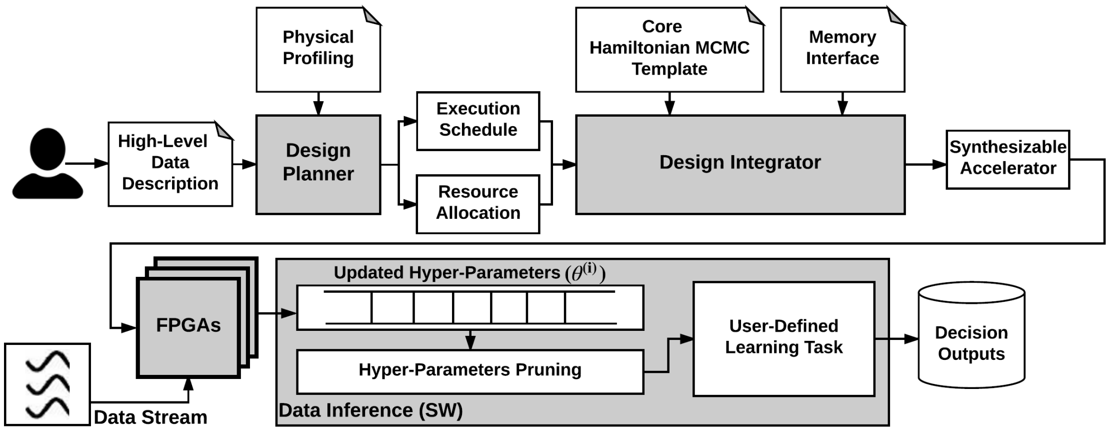


Figure 1: CausaLearn Global Flow: CausaLearn takes the stream of data samples as its input and learns the hyper-parameters of the corresponding posterior probability density function $P(\theta|\mathbf{D})$ using Hamiltonian MCMC. Our proposed Hamiltonian MCMC template is adaptively customized to the limits of the underlying platform and data structure. The updated hyper-parameters are used to perform a particular user-defined Bayesian learning task (e.g., regression or classification).

independent given the latent function **f(.)**. Therefore, the likelihood $p(\mathbf{y}|\mathbf{f})$ can be factorized over data samples as $\prod_{i=1}^{N} p(y_i|f_i)$, where $\mathbf{f} = [f(x_1), ..., f(x_n)]$. Note that the observations themselves are not independent (e.g., $p(\mathbf{y}) \neq \prod_{i=1}^{N} p(y_i)$). The mean and covariance kernel of a GP are also random variables with certain hyper-parameters (γ) that should be tuned with respect to the input data. The choice of the mean and covariance kernels determines the smoothness and variability of the latent function **f(.)** to be estimated.

2.2 Bayesian Graphical Analysis

To make our notation explicit, we write the likelihood as $p(\mathbf{y}|\mathbf{f}, \sigma_n^2)$ where σ_n^2 is the parameter of the observation noise, and $p(\mathbf{f}|\gamma)$ is the GP prior. The quantities $\theta = [\gamma, \sigma_n^2]$ are the hyper-parameters of the underlying probabilistic model. The posterior distribution $p(\theta|\mathbf{D})$ must be computed to make predictions for the incoming data samples in different learning tasks including various regression and classification techniques, stochastic optimizations, and neural networks. Let us denote the function of interest to be evaluated with $g(\theta)$. Thereby, the underlying learning task can be expressed as the evaluation of the following integral:

$$E_{p(\theta|\mathbf{D})}[g(\theta)] = \int g(\theta)p(\theta|\mathbf{D})d\theta. \tag{3}$$

For instance, by setting $g(\theta) = p(y^*|\theta)$, one can predict the probability of a future observation y^* based on the previously observed data per $p(y^*|\mathbf{D}) = \int p(\mathbf{y}^*|\theta)p(\theta|\mathbf{D})d\theta$.

Given the large cardinality of the hyper-parameter set $|\theta|$, and the high dimensionality of input data in real-world applications, it is computationally impractical to analytically evaluate the integral in Eq. (3). Thus, estimation algorithms such as MCMC are often the methods of choice [23]. Table 1 summarizes different MCMC algorithms. MCMC methods work sequentially by constructing a Markov chain with each state of the chain corresponding to a new random sample from the posterior distribution $p(\theta|\mathbf{D})$. The samples are then used to approximate Eq. (3) as follows:

$$\tilde{E}_{p(\theta|\mathbf{D})}[g(\theta)] = \frac{1}{N}\Sigma_{i=1}^{N} g(\theta^{(i)}). \tag{4}$$

3 CAUSALEARN GLOBAL FLOW

Figure 1 illustrates the high-level block diagram of CausaLearn framework. CausaLearn leverages Hamiltonian MCMC to devise a generic scalable framework that can be directly applied to different Bayesian applications. Hamiltonian technique is particularly of interest due to two main reasons: (i) It can handle both strong correlation and high-dimensionality in real-world applications by stochastically computing the gradient of the posterior distribution. (ii) It evades the requirement to compute the costly Metropolis-Hastings ratio commonly used in the alternative MCMC methods. This is because the acceptance rate tends to be high in the Hamiltonian method by moving toward the gradient of the target density function at each MCMC iteration as opposed to the use of random walks. CausaLearn involves two automated steps to schedule and customize the underlying data flow (Section 6). An API is also devised (Section 6.3) to ensure ease of use by users who do not necessarily possess a certain level of hardware-design knowledge.

(i) Design Planner. The design planner takes the high-level description of data from the user as its input. This description includes the rate of data arrival and feature space size in the target application. CausaLearn adopts platform profiling to abstract the physical characteristic of the target FPGA. The platform characteristics include the Block-RAM (BRAM) budget, available Digital Signal Processing (DSP) units, and memory bandwidth. The acquired physical characteristics along with the data description are fed into the design planner unit to find the optimal execution schedule and resource allocation (Section 6.1).

(ii) Design Integrator. The design integrator employs our core Hamiltonian MCMC (Section 5) as a template and customizes it according to the data schedule and resource allocation provided by the design planner. The integrator converters the acquired execution schedule into state machines and microcodes embedded in the target hardware design. CausaLearn tiles, batches, and pipelines the subsequent computational workload such that it best fits the target platform and application data (Section 6.2). The final synthesizable code is created after adding the memory interface to the design.

CausaLearn leverages a HW/SW co-design methodology. Bayesian analysis of streaming data involves: (i) Fine-tuning the pertinent hyper-parameters priors, and (ii) Performing a particular inference task (e.g., regression or classification) using the updated hyper-parameters. CausaLearn leverages FPGA as the primary hardware accelerator to enable real-time updating of the corresponding random variables and their associated PDFs inline with the data arrival. The FPGA is programmed with the Verilog code automatically generated as the output of the design integrator unit. The inference phase is performed on the general purpose processor that hosts the FPGA board. This is because data inference is a one-time process per input data sample and incurs a much lower computational overhead compared to that of updating the posterior distribution [6]. We use Peripheral Component Interconnect Express (PCIe) port to load the data to the FPGA and write back the updated parameters to the host. All computations are performed using IEEE 754 single precision floating-point format. Floating-point representation enables CausaLearn to be readily adopted in different learning tasks without requiring the user to modify the core implementation. It is worth noting that the fixed-point solutions are of limited applicability due to the variant nature of ultimate learning tasks and the unpredictability of data range in different applications.

4 CAUSALEARN FRAMEWORK

CausaLearn leverages a three-level model hierarchy to capture the causality structure of time-series data. It solves the following *objective function* to model the complex correlation of data samples:

$$\begin{aligned}
&\textit{Observation model}: \ \mathbf{y}|\mathbf{f}, \sigma_n^2 \sim \prod_{i=1}^{N} p(y_i|f_i, \sigma_n^2),\\
&\textit{GP prior}: \ \mathbf{f}(\mathbf{x})|\gamma \sim \mathcal{GP}(m(\mathbf{x}), K(\mathbf{x}, \mathbf{x}'|\gamma)),\\
&\textit{Hyper parameters prior}: \ \theta = [\gamma, \sigma_n^2] \sim p(\gamma)p(\sigma_n^2),
\end{aligned} \tag{5}$$

where σ_n^2 is the variance of the observation noise per Eq. (1) and γ is the hyper-parameter set of the predictive function $\mathbf{f(.)}$ defined as GP. All hyper-parameters $\theta = [\sigma_n^2, \gamma]$ are iteratively updated in CausaLearn framework as data evolves over time to dynamically approximate the posterior distribution $p(\theta|\mathbf{D})$.

A GP model is fully defined by its second order statistics (i.e., mean and covariance). A common prior density choice for the GP covariance kernel is the squared-exponential function [14]:

$$K_{ij}(\mathbf{x}) = \sigma_k^2 e^{(-\frac{1}{2}(x_i - x_j)^T \Sigma^{-1}(x_i - x_j))}. \tag{6}$$

Here, σ_k^2 is the variance of the kernel function and Σ is a diagonal positive definite matrix, $\Sigma = diag[\mathcal{L}_1^2, ..., \mathcal{L}_d^2]$, in which each diagonal element is the length-scale parameter indicating the importance of a particular input dimension in deriving the ultimate output.

Algorithm 1 outlines the pseudocode of CausaLearn framework. The hyper-parameter set includes the variances of the observation noise and covariance kernel along with the length-scales variables ($\theta = [\sigma_n^2, \sigma_k^2, \mathcal{L}_1, ..., \mathcal{L}_d]$). We further assume a log-uniform prior for the variance parameter σ_k^2 and a multivariate Gaussian prior for the length-scale parameters. Algorithm 1 involves four main steps:

❶ **Platform Profiling:** CausaLearn provides a set of automated subroutine that characterize the available resource provisioning. Our subroutines measure the performance of the following four basic operations involved in the H_MCMC algorithm: matrix-matrix multiplication, dot-product, back-substitution, and random number generation. Our subroutines run the operations with varying sizes to find the target platform constraints. Note that the realization of each operation can be highly diverse depending on the target platform. For instance, based on the sizes of the matrices being multiplied, a matrix multiplication can be compute-bound, bandwidth-bound, or occupancy-bound on a specific platform.

❷ **Automated Customization:** CausaLearn design customization uses the output of physical profiling along with a set of user-defined constraints to schedule and balance the computational workload. The user-defined physical constraints can be expressed in terms of runtime (T_u), memory (M_u), and power consumption (P_u). The building blocks of the customization unit are design planner and design integrator. The details of these blocks are discussed in Section 6.

Algorithm 1 CausaLearn Pseudocode

Inputs: Stream of input data ($D = [X, Y]$), Initial parameters $\theta^{(1)}$, Desired Markov Chain length (C_{len}), discretization factor dt, number of discretization steps n_{step}, Updating frequency n_u, Mass matrix (M), Constant friction term (F), Portion of newly arrived data in each data batch η, Physical constraints $C_u = [T_u, M_u, P_u]$.

Outputs: Posterior Distribution Samples $\theta^{(i)}$, and output decision set $\mathcal{O}$.

```
1:  $HW_{spec} \leftarrow PlatformProfiling()$ ❶
2:  $[b_s, HW_{code}] \leftarrow Customization(HW_{spec}, C_u)$ ❷
3:  $ProgramingFPGA(HW_{code})$
4:  for $i = 1, 2, ..., C_{len}$ do
5:      if $(i \mod n_u) == 0$ then
6:          $[\tilde{X}, \tilde{Y}] \leftarrow DataPartitioning(X, Y, b_s, \eta)$
7:          $Transferring\ Data\ Batch\ \tilde{D}\ to\ FPGA$
8:      $r^{(i)} \sim \mathcal{N}(0, M)$ ❸
9:      $(\theta_1, r_1) \leftarrow (\theta^{(i)}, r^{(i)})$
10:     $B = \frac{1}{2}\sigma_n^2 dt$
11:     $E = \sqrt{2|F - B|dt}$
12:     for $t = 2, ..., n_{step}$ do
13:         $\theta_t \leftarrow \theta_{t-1} + M^{-1} r_{t-1} dt$
14:         $\nabla\tilde{U}(\theta_t) \leftarrow gradient(\tilde{D}, \theta_t)$
15:         $r_t \leftarrow r_{t-1} - \nabla\tilde{U}(\theta_t)dt - FM^{-1}r_{t-1}dt + \mathcal{N}(0, E)$
        end for
16:     $(\theta^{(i+1)}, r^{(i+1)}) \leftarrow (\theta_{n_{step}}, r_{n_{step}})$
17:     $Sending\ Back\ \theta^{(i+1)}\ to\ the\ Host$
18:     $\tilde{\theta} = HyperParameterPrunning(\theta)$ ❹
19:     $\mathcal{O} = UserDefinedDataInference(\tilde{\theta})$
    end for
```

❸ **Dynamic Parameter Updating on FPGA:** CausaLearn takes the stream of data as its input and adaptively updates the pertinent PDF model using H_MCMC. We discuss the template H_MCMC accelerator architecture and its detailed hardware implementation in Section 5. Note that our proposed accelerator architecture is the first realization of Hamiltonian MCMC on the FPGA platform.

❹ **Parameter Pruning and Data Inference:** CausaLearn leverages the hyper-parameter samples drawn form the posterior distribution $p(\theta|\mathbf{D})$ to perform a user-defined data inference task (e.g., Eq. (4)). We use autocorrelation metric $\rho(.)$ to evaluate the mixing property of the generated samples:

$$\rho_k = \frac{\Sigma_i^{N-k}(\theta_i - \bar{\theta})(\theta_{i+k} - \bar{\theta})}{\Sigma_i^N(\theta_i - \bar{\theta})}, \tag{7}$$

where $\bar{\theta}$ is the running average of the previous hyper-parameter samples and k is a user-defined constant that denotes the desired lag in computing the autocorrelation. CausaLearn prunes the correlated hyper-parameter samples to further reduce the computational overhead of the inference phase while providing an effective exploration of the parameters' space. We provide extensive evaluations for both regression and classification tasks in Section 8.

5 ACCELERATOR ARCHITECTURE

CausaLearn leverages batch data processing to update the hyper-parameters of the probability density function. The size of data batch to be evaluated at each MCMC iteration explicitly governs the computational workload of the underlying task. As we will discuss in Section 6, CausaLearn performs physical profiling and resource-aware customization to adjust the data batch size (b_s) and schedule the subsequent computations such that it best fits the target platform and application data requirements.

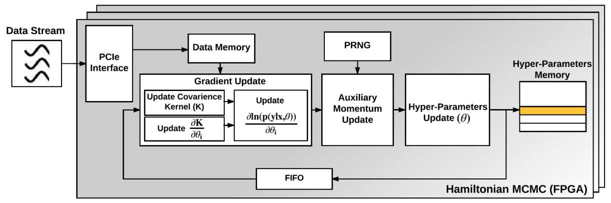

Data Stream
PCIe Interface
Data Memory
PRNG
Gradient Update
Update Covarience Kernel (K)
Update $\frac{\partial \mathbf{K}}{\partial \theta_i}$
Update $\frac{\partial \ln(p(y|x,\theta))}{\partial \theta_i}$
Auxiliary Momentum Update
Hyper-Parameters Update (θ)
Hyper-Parameters Memory
FIFO
Hamiltonian MCMC (FPGA)

Figure 2: High-level block diagram of Hamiltonian MCMC.

Figure 2 illustrates the high-level block diagram of the H_MCMC methodology. At each H_MCMC iteration, a data batch consisting of both newly arrived data samples and a random subset of previous samples are loaded into the FPGA through PCIe to be processed using Hamiltonian dynamics (*Lines 5-7 in Algorithm 1*). We use η to denote the portion of new data in each data batch ($0 < \eta \leq 1$). Performing Hamiltonian MCMC includes three main steps:
(i) Computing the gradient of posterior distribution given the prior density function of each hyper-parameter (Line 14 of Algorithm 1). In H_MCMC, the posterior distribution of θ given a set of independent observations $\mathbf{y} \in \mathbf{D}$ is represented as $p(\theta|\mathbf{D}) \propto e^{(-U(\theta))}$, where the energy function U is:

$$U = -\Sigma_{\mathbf{y} \in \mathbf{D}}\ ln\ p(\mathbf{y}|\mathbf{x}, \theta) - ln\ p(\theta). \tag{8}$$

(ii) Updating the auxiliary momentum variable r. CausaLearn adds a friction term to the momentum updating step as suggested in [6] to minimize the impact of injected noise as a result of bypassing the Metropolis-Hastings correction step in conventional MCMC. CausaLearn includes a scaled Pseudo Random Number Generator (PRNG) to sample from $\mathcal{N}(0, E)$ (Line 15 of Algorithm 1).

(iii) Drawing new hyper-parameter samples based on the currently computed gradients and momentum values. The Mass matrix, M, in Line 13 of Algorithm 1 is used to precondition the MCMC sampler when specific information about the target PDF is available. In many applications, the matrix M is set to the identity matrix I.

The main computational workload in Algorithm 1 is associated with computing the gradient of density function. Algorithm 2 outlines the process of computing the gradient vector $\nabla\tilde{U}(\theta_t)$ to perform H_MCMC with GP prior. Evaluating the $\frac{\partial ln(p(\mathbf{y}|\mathbf{x}, \theta))}{\partial \theta_i}$ term in Line 11 of Algorithm 2 requires computing the inverse of the covariance kernel ($K_{b_s \times b_s}$). Computing the inverse of a dense $b_s \times b_s$ matrix with $b_s \gg 2$ involves a variety of operations with complex data flow. As such, we suggest adopting QR decomposition in the MCMC routine to reduce the hardware implementation complexity and make the algorithm well-suited for FPGA acceleration.

Algorithm 3 details the incremental QR decomposition by modified Gram-Schmitt technique. QR decomposition returns an *orthogonal* matrix Q and an *upper-triangular* matrix R. Utilizing QR decomposition facilitates the gradient computing step by transforming the inversion of the dense kernel matrix into the inversion of an upper-triangular matrix ($K^{-1} = R^{-1}Q^T$), which is performed using simple back substitution (Section 5.1.3).

Algorithm 2 GP Gradient Computing

Inputs: Batch of input data ($\tilde{D} = [\tilde{X}, \tilde{Y}]$), Hyper-parameter set $\theta = [\sigma_n^2, \sigma_k^2, \mathcal{L}_1, ...\mathcal{L}_d]$
Outputs: Gradient of energy function $\bigtriangledown\tilde{U}(\theta)$.

```
1:  $Q^{(0)} \leftarrow [\ ]$
2:  $R^{(0)} \leftarrow [\ ]$
3:  $H \leftarrow [0, 0, ..., 0]^T_{1 \times b_s}$
4:  for $i = 1, 2, ..., b_s$ do
5:      for $j = 1, 2, ..., b_s$ do
6:          $v^2 \leftarrow \Sigma_{k=1}^{d} \frac{(\tilde{X}_{ik} - \tilde{X}_{jk})^2}{\mathcal{L}_k^2}$
7:          $H_j \leftarrow \sigma_k^2 exp(\frac{-v^2}{2})$
        end for
8:      $H_i \leftarrow H_i + \sigma_n^2$
9:      $[Q^{(i)}, R^{(i)}] \leftarrow QR\_Update(Q^{(i-1)}, R^{(i-1)}, H)$
    end for
10: $Z_i \leftarrow R^{-1} Q^T \frac{\partial K}{\partial \theta_i}$
11: $\frac{\partial ln(p(Y|X,\theta))}{\partial \theta_i} \leftarrow -\frac{1}{2}(Tr(Z_i) + Y^T Z_i R^{-1} Q^T Y)$
12: $\bigtriangledown\tilde{U}(\theta_i) \leftarrow \frac{|D|}{|\tilde{D}|}(\frac{\partial ln(p(Y|X,\theta))}{\partial \theta_i} - \bigtriangledown ln(p(\theta_i)))$
```

Algorithm 3 Incremental QR decomposition

Inputs: New column H, Last iteration Q^{s-1} and R^{s-1}.
Output: Q^s and R^s.

```
1:  $R^s \leftarrow \begin{pmatrix} R^{s-1} & 0 \\ 0 & 0 \end{pmatrix}$
2:  for $j$ = 1,...,s-1 do
3:      $R^s_{js} \leftarrow (Q^{s-1}_j)^T H$
4:      $H \leftarrow H - R^s_{js} Q^{s-1}_j$
    end for
5:  $R^s_{ss} \leftarrow \|H\|_2$
6:  $Q^s \leftarrow [Q^{s-1}, \frac{H}{R^s_{ss}}]$
```

5.1 Hardware Implementation

In this section, we explain the realization of H_MCMC module step by step. We leverage both algorithmic and hardware optimization techniques to provide an efficient implementation of H_MCMC.

5.1.1 Memory Management

To effectively pipeline the data flow in Algorithm 1 and optimize the system throughput, it is necessary to perform multiple concurrent loads and stores from a particular RAM. To cope with the concurrency, we suggest having multiple smaller-sized block memories to store particular data matrices instead of using a unified large BRAM. We devise and automate a memory management system to tile and schedule the matrix computations such that it best fits the data geometry and the physical hardware resources.

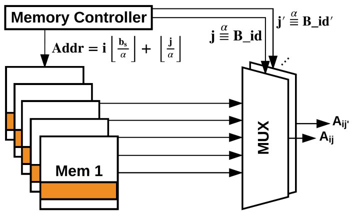

Memory Controller
$\mathbf{Addr = i\left\lfloor \frac{b_s}{\alpha} \right\rfloor + \left\lfloor \frac{j}{\alpha} \right\rfloor}$
$\mathbf{j \overset{\alpha}{\equiv} B\_id}$
$\mathbf{j' \overset{\alpha}{\equiv} B\_id'}$
Mem 1
MUX
$A_{ij'}$
A_{ij}

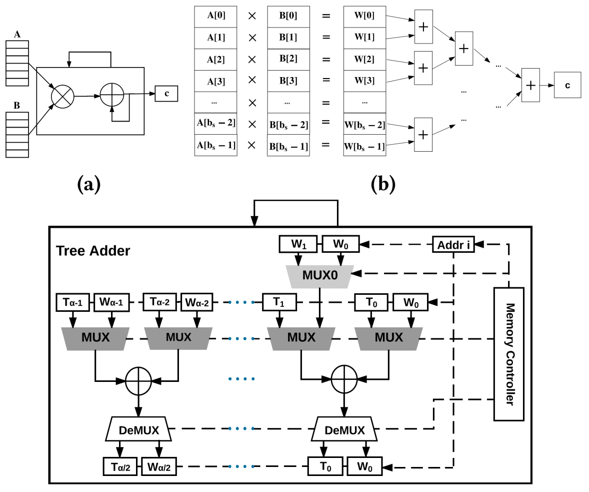
Figure 3: CausaLearn uses cyclic interleaving to facilitate concurrent load/store in performing matrix computations.

Figure 3 illustrates the schematic depiction of the memory management unit in CausaLearn framework. The block memories corresponding to a specific data matrix share the same address signal (*addr*) generated by the memory controller. The block identification index (B_id) is used in conjunction with the address signal to locate a certain element of the pertinent data matrix. To perform a matrix-based operation, one requires having access to sequential matrix indexes. CausaLearn's memory controller fills the corresponding memory blocks using *cyclic interleaving*. Employing cyclic interleaving enables accessing multiple successive elements of a matrix simultaneously which, in turn, facilitates parallelizing matrix-vector and matrix-matrix multiplications.

For a given data batch size (b_s), the number of concurrent floating-point adders/multipliers used to perform a matrix operation is directly controlled by the unrolling factor by which the data matrices are partitioned into smaller blocks. Let us denote the pertinent unroll factor with α. CausaLearn provides a set of subroutines that characterize the impact of unrolling factor α on the subsequent resource consumption. Our automated subroutines take the available resource provisioning into account and provide guidelines for an efficient hardware mapping. These guidelines are leveraged to customize the matrix-based computational workloads to the resource limits of the target platform while avoiding mapping of the data matrices into registers due to an excessive array partitioning. For instance, in Xilinx vendor libraries, every 10 floating-point numbers or less will be mapped to registers during the design synthesis. As such, α should take an integer value less than or equal to $\alpha \leq \frac{b_s}{10}$ to avoid excessive data partitioning. Note that mapping of large data matrices into the registers exhausts the LUT units on the target FPGA resulting in a complex control logic. This, in turn, translates to a larger critical path to accommodate for the underlying computations.

5.1.2 Tree-based Reduction

Performing matrix-vector and matrix-matrix multiplication results in frequent appearance of dot product operations similar to c += $A[i] \times B[i]$. Due to the sequential nature of dot products (Figure 4a), simple use of pipelining/unrolling does not significantly reduce the Initiation Interval (II) between two successive operations. As such, we suggest to transform such sequential operations (c += $A[i] \times B[i]$) into a series of operations that can be independently run in parallel (e.g., $W[i] = A[i] \times B[i]$). In particular, we implement a tree-based adder to find the final sum value c by adding up the values stored in a BRAM called W. We use cyclic interleaving for storing all the involving arrays including A, B, and W to facilitate pipelining the subsequent multiplications and additions (Figure 4b).



(c)

Figure 4: Facilitating matrix multiplication and dot product. (a) Conventional sequential approach. (b) Proposed tree-based model. Our approach reduces the II of dot product operations to 1. (c) The inner structure of tree-based adder.

Figure 4c illustrates the inner structure of CausaLearn tree-based adder. We utilize a temporary array T within the tree adder module to store the intermediate results. In our tree adder module, the number of additions performed at each stage is halved and the result is stored in the other array. E.g., in the even stages, the values in the array W are summed up and the results are stored in the array T. CausaLearn's memory controller generates the appropriate source/destination addresses to load/store the intermediate results at each stage of the tree. The number of floating-point adders/multipliers in the tree-based reduction module is equivalent to the unrolling factor α used to partition data matrices (assuming dual port memory blocks). Let us index the elements of arrays W and T with variable k. Each sub-block W_i and T_i in Figure 4c is filled such that $k \equiv i \mod \alpha$ where $k \in \{0, 1, ..., b_s\}$. In the tree adder structure, the multiplexer denoted by "MUX0" is necessary for performing the last b_s/α additions on the remaining values in W_{+1}. The final result (c) is stored in the address 0 of memory assigned to array W. As will be discussed in Section 5.1.4, CausaLearn attains a balance between parallel operations and data reuse by scheduling a slice of operations to be performed at each clock cycle.

5.1.3 Matrix Inverse Computation

Computing the inverse of the covariance kernel K is a key step in finding the gradient direction in the H_MCMC routine. Employing QR decomposition within the H_MCMC routine facilitates such operations given that K^{-1} can be computed as $R^{-1}Q^T$. For instance, to solve an equation similar to $V = K^{-1}B$, one needs to find the vector V such that $RV = Q^T B$. Given the upper-triangular structure of matrix R, the latter equation can be solved using back-substitution [24–26] in which (starting from the last row index) each element of the vector V can be uniquely recovered by solving a linear equation as illustrated in Figure 5. Let us denote the product of $Q^T B$ with vector C. The Processing Element (PE) in Figure 5 is a multiply-add accumulator that computes:

$$V_i = \frac{C_i - \Sigma_{j=i+1}^{b_s} R_{ij} V_j}{R_{ii}}. \tag{9}$$

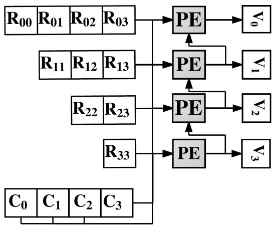


Figure 5: Schematic depiction of back-substitution.

CausaLearn performs back-substitution by parallelizing the computations as shown in Figure 6. Cyclic interleaving along the second dimension (matrix columns) is used to store the Q and R matrices. This enables us to pipeline the design and reduce the II between two successive operations into only 1 clock cycle. Indices of vectors and the second dimension of matrices in Figure 6 correspond to their actual values modulo α. We batch the operations in the back-substitution module to parallelize computations that share the same variables. E.g., in computing Line 10 of Algorithm 2, multiple columns of matrix $Z_{b_s \times b_s}$ may be batched together to facilitate computations given that the columns of matrix Z can be computed independently using the same set of Q and R values.

5.1.4 Data Parallelism

CausaLearn gains a balance between parallel operations and data reuse by partitioning matrix-based computations into smaller

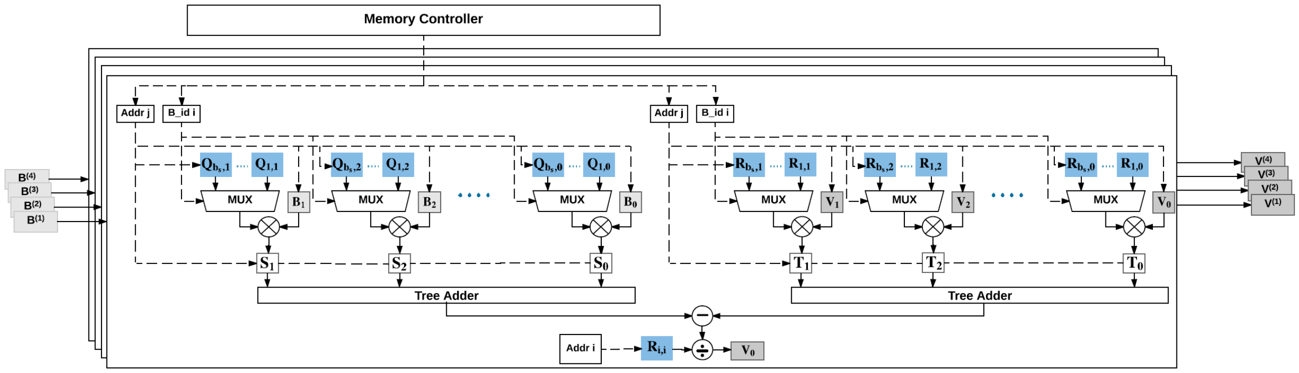


Figure 6: CausaLearn architecture for computing back-substitution. The operations in the right and left side of the equation, $R_{b_s \times b_s} V_{b_s \times b_s} = Q^T_{b_s \times b_s} B_{b_s \times b_s}$, are parallelized to optimized system throughput per iteration. We use cyclic interleaving along the second dimension to store the Q and R matrices. Each column of matrix $B^{(i)}$ is partitioned into smaller blocks to further accommodate parallelism. In this figure, we used dash lines to indicate the control signals.

slices of operations that best match the available computational resources such as DSP units. Figure 7a shows an example where multiple columns of matrix V are scheduled as a slice of operations to be evaluated in parallel in the matrix inversion unit ($R_{b_s \times b_s} V_{b_s \times b_s} = Q^T_{b_s \times b_s} B_{b_s \times b_s}$). As shown in Figure 7b, there is a trade-off between the number of samples per slice of computations and resource utilization. CausaLearn leverages this trade-off to optimize the template design such that the throughput per resource unit is maximized. The effective throughput per resource unit decreases for large values of slice factor p. This performance drop is due to the saturation of the pertinent resource provisioning which, in turn, makes it infeasible to perform more operations in parallel. We leverage batch data parallelism within different parts of the framework (e.g., tree-based reduction module, matrix inversion unit, etc.) to improve the efficiency of the system.

6 CAUSALEARN CUSTOMIZATION

The architecture discussed in Section 5 serves as a template for the accelerator's micro-architecture. Here, we outline our design customization methodology to adapt the H_MCMC routine to the resource boundaries of the target platform.

6.1 Design Planner

Table 2 details the memory footprint and runtime cost in CausaLearn framework. Memory constraint on computing platforms is one of the main limitations in big data regime. CausaLearn updates the posterior distribution samples of a dynamic data collection by breaking down the input data into data batches that best fit the memory budget. CausaLearn's memory footprint outlined in Table 2 specifies the storage requirement for the gradient matrices corresponding to each GP hyper-parameter, the covariance kernel $K = QR$, and the intermediate matrices Z_i (Line 10 in Algorithm 2).

The runtime requirement for data analysis in CausaLearn framework can be approximated as:

$$T_{CausaLearn} \propto T^{comm} + T^{Comp}. \tag{10}$$

The T^{comm} term denotes the communication overhead of sending a data batch of size $b_s \times d$ from host to the FPGA platform and reading back the updated posterior distribution parameters θ. The T^{Comp} term represents the runtime cost of updating the covariance matrix K and computing the gradients as outlined in Algorithms 1, 2, and 3. The computation and communication costs in CausaLearn framework are detailed in Table 2. As we demonstrate in Section 8, CausaLearn's overall runtime is mainly dominated by the computational workload while the communication cost contributes to a small fraction of the overall runtime (e.g., $\leq 0.03\%$).

Table 2: CausaLearn memory and runtime characterization.

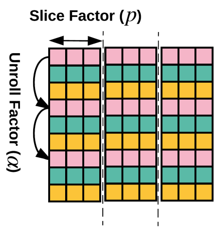
Physical Performance of CausaLearn Framework	
Memory Footprint	$M_{CausaLearn} \approx N_{bits} n_k (4+d) b_s^2$ N_{bits}: *Number of signal representation bits* n_k: *Number of H_MCMC units working in parallel* b_s: *Number of samples per data batch-size* d: *Feature space size of the incoming data samples*
Computation Runtime	$T^{comp} \approx \beta_{flop} C_{len} n_{step} (6b_s^2 d + b_s d)$ β_{flop}: *Computational cost per floating-point operation* C_{len}: *Desired Markov chain's length* n_{step}: *Number of discretization steps in H_MCMC*
Communication Runtime	$T^{comm} \approx \beta_{net} + \frac{N_{bit} C_{len} [b_s d + (d+2)]}{BW}$ β_{net}: *Constant network latency* BW: *Operational communication bandwidth*

There is a trade-off between the selected data batch size b_s and the required runtime to reach the Markov chain steady state distribution, a.k.a., *mixing time* [6, 27]. On the one hand, a high value of b_s reduces the number of iterations to reach the steady state distribution. However, it also reduces the throughput of the system as data can no longer fit in the fast BRAM of the target board. On the other hand, a low value of b_s may degrade the overall performance due to the significant increase in the number of required posterior samples to compute a steady approximation of Eq. (3). CausaLearn carefully leverages this trade-off to customize computations to the limits of the physical resources and constraints while minimally affecting the mixing time in the target application.

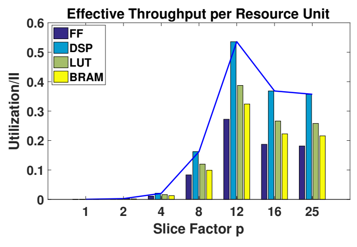
To deliver the most accurate approximation within the given resource provisioning, CausaLearn solves the optimization objective described in Eq. (11). CausaLearn's constraint-driven optimization can be expressed as:

$$\begin{aligned} &\underset{b_s,\, n_k}{\textit{minimize}}\ (\textit{MC mixing time}), \\ &\textit{subject to: } T^{comm} + T^{Comp} \leq T_u, \\ &\qquad \eta n_k b_s \leq f_{data} T_u, \\ &\qquad M_{CausaLearn} \leq M_u, \\ &\qquad P_{CausaLearn} \leq P_u, \\ &\qquad n_k \in \mathbb{N}, \end{aligned} \tag{11}$$

where T_u, P_u, and M_u are a set of user-defined parameters that imply the application constraints in terms of runtime, power, and



(a)



(b)

Figure 7: Example data parallelism in CausaLearn matrix inversion unit: (a) Partitioning of matrix computations into slices of operation. (b) Resource utilization divided by the pertinent initiation interval as a function of the number of samples per operation slice on Virtex-7-XC7VX485T FPGA.

memory respectively. The maximum number of newly arrived samples that should be processed in each time unit is either dictated by the arriving rate of data samples (f_{data}) or the buffer size for storing incoming samples (M_u). Here, η is the proportion of newly arrived samples versus the old ones in each data batch. For a fixed set of parameters, power consumption ($P_{CausaLearn}$) has a linear correlation with the number of MCMC modules that are run in parallel. CausaLearn tunes the number of concurrent MCMC modules accordingly to adapt to possible power limitations imposed by the target setting.

CausaLearn approximates the solution of Eq. (11) by fixing the number of parallel H_MCMC units (n_k) and solving for data batch size (b_s) using the Karush-Kuhn-Tucker (KKT) conditions. To facilitate automation, we provide a solver for our optimization approach. The solver gets the constraints from the user as inputs and uses our Mathematica-based computational software program to solve the optimization. Note that the constraint-driven optimization is a one-time process and incurs a constant, negligible overhead.

6.2 Design Integrator

The design integrator unit in CausaLearn framework takes the acquired execution schedule into consideration and generates the corresponding state machines and microcodes to manage the memory controller and data parallelisms discussed in Section 5.1. The customized synthesizable code is generated after embedding the microcodes within the template H_MCMC architecture. In our prototype designs, we leverage PCIe to transfer data back and forth between the FPGA and the general purpose processor hosting the FPGA. The PCIe interface can be replaced by any other data transfer link such as Ethernet depending on the application.

6.3 CausaLearn API

CausaLearn API consists of a set of high-level automated subroutines which perform the subsequent steps outlined in Figure 1. Programmers interact with our API only through providing the input data stream and pertinent physical constraints in terms of the available memory, runtime, and/or power inside a bash file. CausaLearn finds the optimal batch size (b_s) using our Mathematica-based optimizer as discussed in Section 6.1. The API then calls Vivado-HLS to search for optimal values of various design directives including unroll factor, slice factor, and pipeline depth that yield the maximum throughput while complying with the user-defined constraints. Eventually, the customized H_MCMC core along with the required I/O interface modules are generated to be implemented on FPGA.

In CausaLearn, API follows specific steps to find the optimal values for each HLS directive in an automated manner. For instance, the optimal value of slice factor is obtained by synthesizing the design using different values of slice factor and collecting utilization and initiation interval from the synthesis report. The optimal value is either the local optima of the effective throughput per resource unit as depicted in Figure 7b or the maximum value that allows the design to fit user-specific constraints (when using the local optima exceeds the user constraints). After setting the slice factor, unroll factor is determined to increase data parallelism while maintaining the design metrics below the specific physical constraints. It is noteworthy that the whole customization process is automated so that data practitioners with different scientific backgrounds that do not necessarily possess any particular hardware design knowledge can benefit from CausaLearn end-to-end design.

Depending on the synthesis speed on the host machine and data dimensionality, profiling can take 5 to 30 minutes on commodity personal computers. Note that profiling is performed once per application/platform and its cost is amortized over-time as the system is used for processing data streams.

7 HARDWARE SETTING AND RESULTS

We evaluate CausaLearn using three off-the-shelf FPGA evaluation boards namely Zynq ZC702 (XC7Z020), Virtex VC707 (XC7VX485T), and Virtex UltraScale VCU108 (XCVU095) as the primary hardware accelerator. We use an Intel core-i5 CPU with 8GB memory running on the Windows OS at 2.40GHz as the general purpose processor hosting the FPGA. The software realization of CausaLearn is employed for comparison purposes. We leverage PCIe library provided by [28] to interconnect the host and FPGA platforms. Vivado HLS 2016.4 is used to synthesize and simulate our MCMC units. All FPGA platforms work at 100MHz frequency.

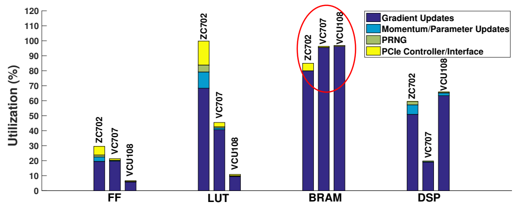


Figure 8: Resource utilization on different platforms assuming a hyper-parameter set of size $|\theta|$= 10. The output of our automated customization characterizes the hardware accelerator which, in turn, helps us to fully exploit the available on-chip memory. As shown, the resource utilization is mainly dominated by the Gradient update unit.

Figure 8 shows the breakdown resource utilization of CausaLearn deployed on three FPGAs. Each FPGA platform has a different computational budget. The total resource utilization accounts for both the H_MCMC unit (including the gradient update, momentum and parameter update, and PRNG modules) as well as the PCIe controller. Table 3 details CausaLearn performance per iteration of H_MCMC for processing different number of data samples (n). The earlier MCMC hardware accelerators are developed based on the assumption that input data samples are independent and identically distributed. These works cannot handle time-series data with causal structure as shown in Figure 9. As such, we opt to compare CausaLearn runtime (with n_k = 1) and energy consumption against a highly optimized C++ software solution. The software baseline is optimized using Eigen and OpenMP libraries. Eigen library exploits Intel Stream SIMD Extension (SSE) instructions to enhance the performance of intensive matrix computation. All the available cores on the Intel Core-i5 CPU (with 8GB memory running at 2.40GHz) were used to execute the H_MCMC routine.

FPGA power is simulated using Vivado power analyzer which accounts for both static and dynamic power. We use Intel Power Gadget 3.0.7. to measure CPU execution power. The power consumption for the H_MCMC unit is 0.95, 3.74, and 3.84 *Watts* for

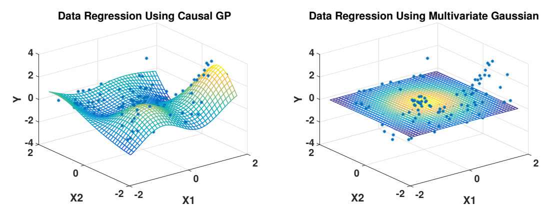
Table 3: Relative runtime/energy improvement per H_MCMC iteration achieved by CausaLearn on different platforms compared to the optimized software implementation for $|\theta| = 10$ and $n_{step} = 100$. The conventional H_MCMC algorithm incurs $O(n^2)$ runtime complexity, whereas, our batch optimization approach scales linearly with $\left\lceil \frac{n}{b_s} \right\rceil$.

n	Communication Overhead	Runtime per Iteration SW	CausaLearn Runtime per Iteration			Runtime Improvement			Energy Improvement		
			ZC702	VC707	VCU108	ZC702	VC707	VCU108	ZC702	VC707	VCU108
256	16.92 *msec*	113.01 *sec*	96.18 *sec*	47.10 *sec*	76.71 *sec*	1.2×	2.4×	1.5×	8.1×	4.1×	2.4×
512	33.65 *msec*	902.98 *sec*	192.37 *sec*	94.23 *sec*	153.42 *sec*	4.7×	9.6×	5.9×	31.8×	16.5×	9.8×
1024	67.18 *msec*	8601.37 *sec*	384.72 *sec*	188.61 *sec*	230.13 *sec*	22.4×	42.7×	37.4×	136.3×	65.9×	56.3×
2048	134.19 *msec*	33.52 *hr*	769.44 *sec*	376.81 *sec*	460.26 *sec*	156.8×	320.2×	262.2×	769.1×	398.9×	318.2×

ZC702, VC707, and VCU108, respectively. As illustrated, the computational time in CausaLearn grows linearly with respect to the number of data samples. In this experiment, the optimal batch size for each platform is used to maximize the on-chip memory usage as shown in Figure 8. The optimal data batch sizes (output of CausaLearn customization) are 88, 256, and 360 on ZC702, VC707, and VCU108, respectively. In cases where the number of data samples is not divisible by the data batch, $\left\lceil \frac{n}{b_s} \right\rceil$ iterations are performed to analyze all data samples.

8 PRACTICAL DESIGN EXPERIENCES

We use CausaLearn to analyze three large time-series data with strong causal structure. In particular, we analyze:
(i) Dow Jones Index stock's change over time. This data [29] includes daily stock data of 30 different companies collected over 6 months. Each data sample x_i contains 8 features including different statistics of the stock price during the previous week (e.g., the highest and lowest price). The task is to predict the percentage of return for each stock in the following week.
(ii) Sensor data to classify different daily human activities. The dataset [30] comprises body motion and vital signs recordings for ten volunteers while performing different activities. Each data sample x_i includes 23 features. In this experiment, we use the data collected for two subjects to distinguish jogging and running activities. Each activity is recorded for 1 minute with a sampling rate of $50Hz$ resulting in more than $6K$ samples per subject.
(iii) Time-variant data for regression purposes [31]. The data is generated using a time-variant (unknown) function where the task is to predict the function's output given the previously observed samples. Figure 9a shows the regression's output using the posterior distribution samples learned by CausaLearn (Figure 11c). In Figure 9, we compare the regression's output using MCMC samples learned by assuming a causal GP prior versus i.i.d. data measurements with multivariate Gaussian prior (e.g., [11, 32]). The data points denoted by star signs are the training observations $\mathbf{y}$.



(a) **(b)**

Figure 9: Time-variant data analysis using MCMC samples by assuming (a) causal GP prior (CausaLearn), vs. (b) i.i.d. assumption with multivariate Gaussian prior (e.g., [11, 32]).

Data batch size, b_s, is a key tunable parameter that characterizes CausaLearn's resource utilization and runtime performance as outlined in Table 2. Figure 10 demonstrates the impact of data batch size b_s on the subsequent resource utilization and system throughput per H_MCMC unit in each application. Multiple H_MCMC units can work in parallel within the confine of the resource provisioning to further boost the system throughput for smaller data batch sizes.

Figure 11 shows CausaLearn's posterior distribution samples obtained with a batch size of $b_s = 128$ in each application. The red cross sign on each graph demonstrates the maximum a posterior (MAP) estimate obtained by solving:

$$\underset{\theta}{argmax}\ ln(p(\mathbf{y}|\mathbf{x}, \theta)) + ln(p(\theta)). \qquad (12)$$

Due to the space limit and high dimensionality of the target datasets, Figure 11 selectively shows the MCMC samples obtained for the observation noise variance (σ_n^2). The same trend is observed for the other hyper-parameters (e.g., σ_k^2 and $\mathcal{L}_i$).

9 RELATED WORK

Bayesian network is a key method to model dynamic systems in various statistical and machine learning tasks. Significant theoretical strides have been made to design Bayesian graphical analytics that can be used at scales by exploiting task and data level parallelism [5–8, 33]. Available Bayesian inference tools on CPUs [15], GPUs [16–18, 34], and FPGAs [21, 22], however, are either application specific or include direct mappings of algorithms to hardware. As such, the idea of customizing the Bayesian networks to make them well-suited for the underlying platform is unexplored. Recently, authors in [19, 20] have introduced a generic GPU-accelerated framework for Bayesian inference. Even these works are built based the assumption that input data samples are i.i.d; thus lack the capability to capture the inherent causal structure of time series data. To the best of our knowledge, CausaLearn is the first automated framework that enables end-to-end prototyping of complex causal Bayesian analytics with continuous random variables. CausaLearn is capable of handling both strong correlation and high-dimensionality in streaming scenarios with severe resource constraints.

FPGAs have been used to accelerate computationally expensive MCMC methods. Recent works in [11, 32, 35] have proposed reconfigurable architectures with custom precision for efficient realization of population-based MCMC routine applied to Bayesian graphical models. Authors in [11, 32, 35] targets simple multivariate Gaussian densities where observations are assumed to be independent and identically distributed. Thus, these works cannot be readily employed in more sophisticated streaming scenarios where independence cannot be assumed between data samples. To the best of our knowledge, CausaLearn is the first to provide a scalable FPGA realization of generic H_MCMC routine applied to streaming applications with large and densely correlated data samples. We emphasize that the use of data precision optimization technique proposed in [32, 36] provide an orthogonal means to our resource-aware customization for performance improvement. Therefore, CausaLearn can achieve even greater improvement by leveraging such optimizations.

10 CONCLUSION

This paper presents CausaLearn, the first automated reconfigurable framework to compute on and continuously update time-varying probability density functions for causal Bayesian analysis.

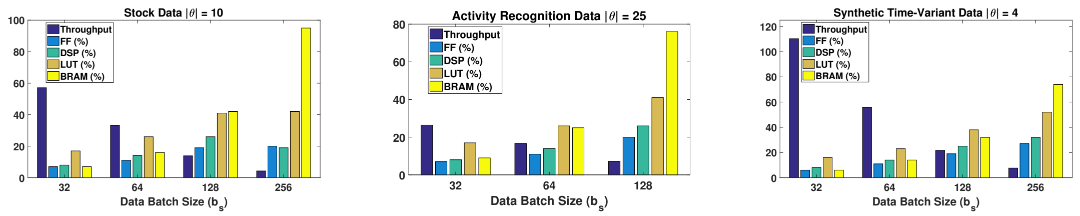


Figure 10: VC707 resource utilization and system throughput per H_MCMC unit ($n_k = 1$) as a function of data batch size b_s in different applications. The reported throughputs indicate batch per second processing rate corresponding to $n_{step} = 100$.

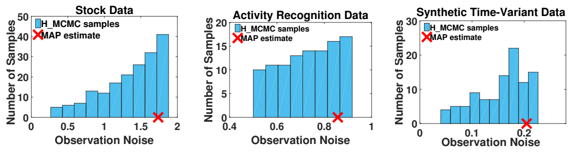


Figure 11: Example CausaLearn's posterior distribution samples. The red cross sign on each graph demonstrates the maximum a posterior estimate in each experiment.

CausaLearn targets probabilistic learning in streaming scenarios in which the number of data samples grows over time and computational resources are severely limited. To boost the computational efficiency, CausaLearn provides a scalable implementation of Hamiltonian MCMC on FPGA. We modify the conventional MCMC algorithm using QR decomposition to make it amenable for hardware-based acceleration performed by FPGA platforms. We further provide novel memory management, tree-based reduction, and data parallelism techniques to effectively pipeline and balance the underlying matrix computations on FPGA. CausaLearn is devised with an automated constraint-driven optimization unit to customize H_MCMC workload to the limits of the resource provisioning while minimally affecting the MC mixing time. An accompanying API ensures automated applicability of CausaLearn for an end-to-end realization of complex Bayesian graphical analysis on massive datasets with densely correlated samples.

REFERENCES

[1] R. Salakhutdinov, A. Mnih, and G. Hinton, "Restricted boltzmann machines for collaborative filtering," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 791–798.
[2] L. Bottolo and S. Richardson, "Evolutionary stochastic search for bayesian model exploration," *Bayesian Analysis*, vol. 5, no. 3, pp. 583–618, 2010.
[3] T. Flury and N. Shephard, "Bayesian inference based only on simulated likelihood: particle filter analysis of dynamic economic models," *Econometric Theory*, vol. 27, no. 05, pp. 933–956, 2011.
[4] S. Brooks, A. Gelman, G. Jones, and X.-L. Meng, *Handbook of Markov Chain Monte Carlo*. CRC press, 2011.
[5] M. Welling and Y. W. Teh, "Bayesian learning via stochastic gradient langevin dynamics," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 681–688.
[6] T. Chen, E. B. Fox, and C. Guestrin, "Stochastic gradient hamiltonian monte carlo." in *ICML*, 2014, pp. 1683–1691.
[7] W. Neiswanger, C. Wang, and E. Xing, "Asymptotically exact, embarrassingly parallel mcmc," *arXiv preprint arXiv:1311.4780*, 2013.
[8] U. Simsekli, A. Durmus, R. Badeau, G. Richard, E. Moulines, and T. Cemgil, "Parallelized stochastic gradient markov chain monte carlo algorithms for non-negative matrix factorization," in *42nd International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017.
[9] D. Auras, S. Birke, T. Piwczyk, R. Leupers, and G. Ascheid, "A flexible mcmc detector asic," in *SoC Design Conference (ISOCC), 2016 International*. IEEE, 2016, pp. 285–286.
[10] M. Lin, I. Lebedev, and J. Wawrzynek, "High-throughput bayesian computing machine with reconfigurable hardware," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2010, pp. 73–82.
[11] 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, 2016.
[12] S. Liu, G. Mingas, and C.-S. Bouganis, "An exact mcmc accelerator under custom precision regimes," in *Field Programmable Technology (FPT), 2015 International Conference on*. IEEE, 2015, pp. 120–127.
[13] C. E. Rasmussen, "Gaussian processes for machine learning," 2006.
[14] M. K. Titsias, N. Lawrence, and M. Rattray, "Markov chain monte carlo algorithms for gaussian processes," *Inference and Estimation in Probabilistic Time-Series Models*, vol. 9, 2008.
[15] D. Maclaurin and R. P. Adams, "Firefly monte carlo: Exact mcmc with subsets of data," *arXiv preprint arXiv:1403.5693*, 2014.
[16] M. M. Tibbits, M. Haran, and J. C. Liechty, "Parallel multivariate slice sampling," *Statistics and Computing*, vol. 21, no. 3, pp. 415–430, 2011.
[17] S. Henriksen, A. Wills, T. B. Schön, and B. Ninness, "Parallel implementation of particle mcmc methods on a gpu," *IFAC Proceedings Volumes*, vol. 45, no. 16, pp. 1143–1148, 2012.
[18] A. L. Beam, S. K. Ghosh, and J. Doyle, "Fast hamiltonian monte carlo using gpu computing," *Journal of Computational and Graphical Statistics*, vol. 25, no. 2, pp. 536–548, 2016.
[19] A. G. d. G. Matthews, M. van der Wilk, T. Nickson, K. Fujii, A. Boukouvalas, P. León-Villagrá, Z. Ghahramani, and J. Hensman, "Gpflow: A gaussian process library using tensorflow," *Journal of Machine Learning Research*, vol. 18, no. 40, pp. 1–6, 2017.
[20] D. Tran, A. Kucukelbir, A. B. Dieng, M. Rudolph, D. Liang, and D. M. Blei, "Edward: A library for probabilistic modeling, inference, and criticism," *arXiv preprint arXiv:1610.09787*, 2016.
[21] N. B. Asadi, T. H. Meng, and W. H. Wong, "Reconfigurable computing for learning bayesian networks," in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. ACM, 2008, pp. 203–211.
[22] N. Bani Asadi, C. W. Fletcher, G. Gibeling, E. N. Glass, K. Sachs, D. Burke, Z. Zhou, J. Wawrzynek, W. H. Wong, and G. P. Nolan, "Paralearn: a massively parallel, scalable system for learning interaction networks on fpgas," in *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 2010, pp. 83–94.
[23] C. Andrieu, N. De Freitas, A. Doucet, and M. I. Jordan, "An introduction to mcmc for machine learning," *Machine learning*, vol. 50, no. 1, pp. 5–43, 2003.
[24] B. D. Rouhani, A. Mirhoseini, E. M. Songhori, and F. Koushanfar, "Automated real-time analysis of streaming big and dense data on reconfigurable platforms," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 10, no. 1, p. 8, 2016.
[25] B. D. Rouhani, E. M. Songhori, A. Mirhoseini, and F. Koushanfar, "Ssketch: An automated framework for streaming sketch-based analysis of big data on fpga," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 2015, pp. 187–194.
[26] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar, "Rise: An automated framework for real-time intelligent video surveillance on fpga," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, p. 158, 2017.
[27] R. Bardenet, A. Doucet, and C. Holmes, "An adaptive subsampling approach for mcmc inference in large datasets," in *Proceedings of The 31st International Conference on Machine Learning*, 2014, pp. 405–413.
[28] XILLYBUS, "http://xillybus.com/," 2017.
[29] UCI Machine Learning Repository, "https://archive.ics.uci.edu/ml/datasets/Dow+Jones+Index," 2016.
[30] ——, "https://archive.ics.uci.edu/ml/datasets/MHEALTH+Dataset," 2016.
[31] J. Vanhatalo, J. Riihimäki, J. Hartikainen, P. Jylänki, V. Tolvanen, and A. Vehtari, "Gpstuff: Bayesian modeling with gaussian processes," *Journal of Machine Learning Research*, vol. 14, no. Apr, pp. 1175–1179, 2013.
[32] G. Mingas and C.-S. Bouganis, "A custom precision based architecture for accelerating parallel tempering mcmc on fpgas without introducing sampling error," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012, pp. 153–156.
[33] Y.-A. Ma, T. Chen, and E. Fox, "A complete recipe for stochastic gradient mcmc," in *Advances in Neural Information Processing Systems*, 2015, pp. 2917–2925.
[34] C. Hall, W. Ji, and E. Blaisten-Barojas, "The metropolis monte carlo method with cuda enabled graphic processing units," *Journal of Computational Physics*, vol. 258, pp. 871–879, 2014.
[35] S. Liu, G. Mingas, and C. Bouganis, "An unbiased mcmc fpga-based accelerator in the land of custom precision arithmetic," *IEEE Transactions on Computers*, 2016.
[36] G. Mingas, F. Rahman, and C.-S. Bouganis, "On optimizing the arithmetic precision of mcmc algorithms," in *Field-Programmable Custom Computing Machines (FCCM), 21st Annual International Symposium on*. IEEE, 2013, pp. 181–188.