ViX: Analysis-driven Compiler for Efficient Low-Precision Variational Inference

Ashitabh Misra, Jacob Laurel, Sasa Misailovic
University of Illinois Urbana-Champaign
Department of Computer Science
{misra8, jlaurel2, misailo}@illinois.edu

***Abstract*—As large quantities of stochastic data are processed onboard tiny edge devices, these systems must constantly make decisions under uncertainty. This challenge necessitates principled embedded compiler support for time- and energy-efficient probabilistic inference. However, compiling probabilistic inference to run on the edge is significantly understudied, and the existing research is limited to computationally expensive MCMC algorithms. Hence, these works cannot leverage faster variational inference algorithms which can better scale to larger data sizes that are representative of realistic workloads in the edge setting. However, naively writing code for differentiable inference on resource-constrained edge devices is challenging due to the need for expensive floating point computations. Even when using reduced precision, a developer still faces the challenge of choosing the right quantization scheme, as gradients can be notoriously unstable in the face of low-precision.**

To address these challenges, we propose ViX which is the first compiler for low-precision probabilistic programming with variational inference. ViX generates optimized variational inference code in reduced precision by automatically exploiting Bayesian domain knowledge and analytical mathematical properties to ensure that low-precision gradients can still be effectively used. ViX can scale inference to much larger data-sets than previous compilers for resource-constrained probabilistic programming while attaining both high accuracy and significant speedup. Our evaluation of ViX across 7 benchmarks shows that ViX-generated code is up to 8.15$\times$ faster than performing the same variational inference in 32-bit floating point and also up to 22.67$\times$ faster than performing the variational inference in 64-bit double precision, all with minimal accuracy loss. Further, on a subset of our benchmarks, ViX can scale inference to data sizes between $16-80\times$ larger than the existing state-of-the-art tool Statheros.

I. Introduction

As edge and IoT computing becomes ever more pervasive, developers are increasingly having to extract high performance over large datasets on low-power devices with limited compute resources. Indeed, a key challenge in TinyML has been the need for programming systems which can optimize data-intensive computations for heavily resource-constrained edge-devices, in which even a floating point unit is a luxury. Generating efficient code for those devices is complicated by the fact that (a) data intensive computations already require specialized, domain-specific programming frameworks, which are often not tailored to the resource-constrained edge setting and (b) the few existing domain specific languages optimized for edge applications are often severely limited in their scope.

These difficulties are especially true of probabilistic programming [8], which has emerged as a powerful domain specific programming paradigm for expressing stochastic uncertainty in models performing inference over data. Despite the multitude of probabilistic programming languages, only one, Statheros [12], has been developed for compiling low-precision probabilistic inference procedures to tiny, resource-constrained edge processors. Further, Statheros only supports a restricted version of MCMC inference algorithm and has difficulty scaling to larger datasets. Thus to make inference on a tiny edge device accurately scale to larger data sizes that are often encountered in applications, a different inference strategy is needed.

Variational Inference (VI) [5] has emerged as a highly scalable approximate Bayesian inference technique that reformulates inference as an optimization problem. Since VI is by design an *approximate* inference method, it represents an attractive use case for fixed-point quantization since the inferred posteriors need not be exact to begin with. However, performing any type of gradient-based optimization in reduced precision is quite challenging (as shown in Deep Learning [9]), especially in the face of significant resource constraints on the device. Furthermore, our goal of efficient, low precision VI at the edge is complicated by another challenge. While existing works [6] utilize fixed-point gradient optimization, these works leverage properties of the computation's structure that do not always hold for probabilistic programs. The reason is that, unlike highly structured models (e.g. neural networks), the expressivity of probabilistic programming languages means different programs can have drastically different model structure. Further, unlike related works targeting FPGAs [2], [6], our setting, on a tiny, resource-constrained CPU precludes us from leveraging expensive customizable hardware.

ViX To address these challenges, we propose ViX which is a general probabilistic programming framework for automating differentiable, variational inference with reduced precision. To make differentiable inference tractable, ViX must ensure that gradients computed in low precision do not unnecessarily explode or underflow while still maintaining enough precision for ensuring the gradient descent steps successfully converge. Furthermore, ViX is able to use a custom range analysis and leverage properties of fixed-point arithmetic in a way that is tailored to the structure of variational inference in order to scale to much larger datasets than any state-of-the-art probabilistic programming language for TinyML platforms.

Evaluation We evaluate ViX on 7 benchmark probabilistic programs. We perform this evaluation on an Arduino Due computing platform. ViX code using fixed-point precision is

978-3-9819263-7-8/DATE23/© 2023 EDAA

Model	:==	*Latent*$^+$; *Observed*$^+$; *DistStmt*$^+$; `Infer(`M`,`η`)`
Latent	:==	`Variable<`*Type*`>(`*Var*`)`
Observed	:==	`Constant<`*Type*`>(`*Var , c*`)`
DistStmt	:==	*Var* = *DistExpr* | *Expr*
	|	`for (int i=`c_1`; i<`c_2`; i++){` *Var* = *DistExpr* `}`
DistExpr	:==	`beta<`*Type*`>(`*Expr*, *Expr*`)`
	|	`uniform<`*Type*`>(`*Expr*, *Expr*`)`
	|	`normal<`*Type*`>(`*Expr*, *Expr*`)`
Expr	:==	*Expr* $\otimes$ *Expr* | `Constant<`*Type*`>(c)`
	|	`Variable<`*Type*`>(`*Var*`)` | `sqrt<`*Type*`>(`*Expr*`)`
	|	`log<`*Type*`>(`*Expr*`)` | `pow<`*Type*`>(`*Expr*`)`
Type	:==	`float` | `double` | `Fixed<int,int>`

Fig. 1. The Syntax of the ViX Base Language

up to $22.67\times$ faster (geomean $11.38\times$) than a standard 64-bit double precision version and up to $8.15\times$ faster (geomean $5.08\times$) compared to a standard 32-bit floating-point version. Further, we compare ViX to Statheros, a state-of-the-art MCMC based reduced-precision probabilistic programming language [12] to highlight how ViX is significantly better at scaling to larger data sets. Specifically, we show how on Bayesian regression tasks, ViX scales to $16,000$ data observations, which is between 16-$80\times$ more observations than Statheros.

To summarize, our main contributions are (1) **Fixed-point Variational Inference of probabilistic programs**: ViX is the first system for compiling probabilistic programs to fixed-point precision variational inference routines (2) **Static Analysis**: ViX supports a new range analysis for abstractly interpreting the ELBO and its gradient in order to ensure the selected precision does not suffer over-flows or under-flows. (3) **Evaluation**: Our evaluation on multiple probabilistic programming benchmarks showcases how ViX code is faster than both 32-bit floating-point and 64-bit double precision *and* can scale to much larger datasets than the existing state-of-the-art.

II. ViX Preliminaries

Language Syntax and Semantics Our core language is syntactically embedded inside of C++. The base syntax is given in Fig. 1. The syntax allows the developer to build the Bayesian network corresponding to the probabilistic program. Additionally, the syntax allows the developer to specify arithmetic expressions, which will ultimately be compiled into computational graphs that the variational inference engine can automatically differentiate through.

The semantic meaning of a probabilistic program is a joint posterior distribution over all the latent variables in the program. As shown in Fig. 1, the latent variables are syntactically specified with the `Variable<`*Type*`>(`*Var*`)` command, whereas observed data is specified with `Constant<`*Type*`>(`*Var, c*`)`.

Variational Inference VI is an approximate inference method that poses Bayesian inference as an optimization problem. The basic technique is shown in Algorithm 1 which is what the `Infer` statement in Fig. 1 ultimately does. In VI, one chooses to approximate each latent variable's posterior distribution, $p(\tau|d)$ (where τ is the latent and d are data observations) with an independent variational family approximation, $q_\theta(\tau)$, often called the *guide* distribution. The guide distribution will

Algorithm 1 Generic VI Algorithm

Inputs: unnormalized posterior $p(\tau)p(d|\tau)$, data observations d, variational guide $q_\theta(\tau)$ for each latent, learning rate η, number of iterations M
$\theta \leftarrow$ *initialize*(θ)
for $m = 1$ to M **do**
  $g \leftarrow \frac{1}{n}\sum_{i=1}^{n} \log(\frac{q_\theta(\tau_i)}{p(\tau_i)p(d|\tau_i)}) \cdot \nabla_\theta(\log(q_\theta(\tau_i))$ ▷ ELBO gradient
  $\theta \leftarrow \theta + \eta \cdot g$
end for

be chosen to have a tractable analytical density, typically a Gaussian. Thus performing inference involves finding the best parameters θ for the variational guide so that this posterior approximation is as consistent with the observed data as possible. As an example, if one knows the posterior over some latent variable is unimodal, one can choose the variational family to be a Gaussian and optimize for the best mean. The quality of how well the variational guide approximates the true posterior is determined by evaluating the Evidence lower bound (ELBO), shown in Equation 1. Maximizing the ELBO corresponds to minimizing the KL Divergence between the true posterior and the variational approximation.

$$ELBO(q_\theta) = \mathbb{E}\left[\log(\frac{q_\theta(\tau)}{p(\tau)p(d|\tau)})\right] \tag{1}$$

In order to optimize the ELBO, one has to perform gradient descent with respect to the parameters, which we denote as ∇_θ. The formula is given in Equation 2 as:

$$\nabla_\theta ELBO(q_\theta) = \mathbb{E}\left[\log\left(\frac{q_\theta(\tau)}{p(\tau)p(d|\tau)}\right) \cdot \nabla_\theta \log(q_\theta(\tau))\right] \tag{2}$$

ViX will automate the reduced precision evaluation of the ELBO (and its gradient) as well as the choosing, initializing and optimizing of the variational guide distributions. A key challenge that arises when performing this computation in reduced precision is ensuring the empirical evaluation of both the ELBO and its gradients does not overflow. Thus, one of our core contributions is a static analysis to prevent such overflows.

Fixed-Point Arithmetic Fixed-point arithmetic precision [17] is well-known alternative to floating-point that is particularly useful for hardware without a native support for floating point operations. For fixed-point precision one represents a real number with a fixed number of integer bits I, fractional bits F, and typically 1 bit for the sign. Hence, the numeric range for signed fixed-point precision numbers is $[-2^I, 2^I - 2^{-F}]$.

III. Compilation

We now present the core of ViX's compilation algorithm.

Algorithm 2 ViX Compilation

Inputs: Probabilistic Program P, wordsize w
$G \leftarrow$ `ConvertToBN(`P`)` ▷ Builds Bayes Net Computational Graph
$[e_{smallest}, e_{biggest}]$, `Bounds` $\leftarrow$ `GetELBOBounds(`G`)` ▷ Abstract ELBO
$I, K, F \leftarrow$ `ComputeScale(`$[e_{smallest}, e_{biggest}], w$`)` ▷ Infer bit sizes
$P \leftarrow$ `ApplyScale(`P, I, K, F`)` ▷ Applies the Scaling
$P \leftarrow$ `InitializeGuide(`P, `Bounds)` ▷ Initialize all guide functions
return P

A. Abstracting the ELBO

In order to statically determine the fixed-point precision configuration, we must perform a range analysis on the ELBO and the ELBO's gradient. Thus we need to abstractly compute these two expressions over the entire possible parameter space. This computation becomes difficult as it requires we also abstract the gradient range. However we build upon prior work [13], [14] that allows us to abstractly interpret the numeric range of values computable with automatic differentiation.

For any function $f : \mathbb{R}^m \to \mathbb{R}^p$ (e.g. $\log(x)$ or a probability density, $p(x)$) we can define its interval arithmetic lifting as $f^\sharp : \mathbb{IR}^m \to \mathbb{IR}^p$. Likewise all primitive arithmetic operations have corresponding interval arithmetic liftings. These liftings are how we will abstractly perform the range analysis. To perform the range analysis on the ELBO gradient, g of Algorithm 1 (line 4), we compute the following inside the `GetELBOBounds` function of Algorithm 2 (line 3),:

$$[g_{low}, g_{up}] = \log^\sharp \left(\frac{q^\sharp_\theta([\tau_{low}, \tau_{up}])}{p^\sharp([\tau_{low}, \tau_{up}]) p^\sharp([d_{low}, d_{up}]|[\tau_{low}, \tau_{up}])} \right) \cdot^\sharp \nabla^\sharp_\theta(\log^\sharp(q^\sharp_\theta([\tau_{low}, \tau_{up}]))) \quad (3)$$

where $[\tau_{low}, \tau_{up}]$ represent the range of values that samples from the variational distribution can take on and $[d_{low}, d_{up}]$ represents the range of the observed data samples. All arithmetic operations including the multiplications and division are likewise performed using the interval abstract domain.

The interval abstraction of the variational guide's likelihood gradient, $\nabla^\sharp_\theta$, is computed using interval automatic differentiation as in [13]. We must also compute the range of the log-likelihoods themselves not just their gradients.

Distribution Range Analysis. To abstract the range of the likelihoods, $p^\sharp(\cdot)$, we not only need to know the range of the data or variational samples whose likelihood we are scoring ($[d_{low}, d_{up}]$ and $[\tau_{low}, \tau_{up}]$, respectively), but also the parameters of each distribution in the original program which govern $p(\cdot)$. Determining these parameter ranges requires knowing the support of the distributions. To abstractly compute the range of the *support* of these distributions, we follow the technique of [12] to propagate intervals through the model for each latent and observed variable. This technique also makes the following assumption that for distributions like the Gaussian, while the true support is $(-\infty, \infty)$, in practice this range can be truncated to finite bounds. Thus while not sound, this heuristic technique gives practically useful bounds. The interval abstraction of the support of the distributions (adapted from [12]) is given below:

1) $\texttt{normal}^\sharp([\mu_l, \mu_u], [\sigma_l, \sigma_u]) = [\mu_l - 3\sigma_u, \mu_l + 3\sigma_u]$
2) $\texttt{uniform}^\sharp([a_l, a_u], [b_l, b_u]) = [\min(a_l, b_l), \max(a_u, b_u)]$
3) $\texttt{beta}^\sharp([a_l, a_u], [b_l, b_u]) = [0, 1]$

Extending the abstraction to other distributions is analogous, one merely returns the interval of the distribution's support or a truncation of that support to finite bounds if the support is unbounded. Thus for each abstracted likelihood $p^\sharp(\cdot)$, we obtain an interval range on the parameters governing it as well as an interval range over the data values whose likelihood we score.

B. Determining Integer and Fractional Bits

The ViX compiler will save the lower and upper interval bounds for *all* intermediate sub-expressions, likelihoods, and distributions. Thus we can heuristically over-approximate the largest range any value can take on over the course of the probabilistic program execution. As a heuristic, we only abstractly evaluate the ELBO and its gradient for a single step of VI.

We let the largest interval bound of any possible expression or sub-expression be given as $[e_{smallest}, e_{biggest}]$. In theory, one would need $\lceil \log_2(\max(|e_{smallest}|, |e_{biggest}|)) \rceil$ integer bits to ensure no overflow. However, even if one is not able to allocate sufficient integer bits to support this range, our analysis can simulate this number of integer bits by using scaling.

ELBO scaling. The key way in which we can avoid using too many integer bits is by scaling down the ELBO optimization to a more manageable numeric range. As the goal of VI is to optimize the ELBO, we can scale the ELBO objective by any positive constant and the parameter θ that maximizes the ELBO will be the same. Thus we have the following equality for any positive real $C \in \mathbb{R}_{>0}$.

$$\arg\max_\theta ELBO(q_\theta) = \arg\max_\theta \; C \cdot ELBO(q_\theta) \quad (4)$$

Furthermore, because we can choose any positive constant to divide the objective function by, we choose a power of two ($C = 2^K$), which allows the division to be implemented by our compiler with fast bitshift ($>>$) operations.

Thus the optimization problem reduces to performing gradient descent on $\frac{1}{2^K} ELBO(q_\theta)$ instead of directly on $ELBO(q_\theta)$. We can leverage this insight further, since by the linearity of both derivatives and expectations we know that:

$$\nabla_\theta \frac{1}{2^K} ELBO(q_\theta) = \mathbb{E}\left[\frac{1}{2^K} \cdot \log(\frac{q_\theta(\tau)}{p(\tau)p(d|\tau)}) \cdot \nabla_\theta \log(q_\theta(\tau)) \right]$$

Thus we can scale down the intermediate sub-expressions ($\log(\frac{q_\theta(\tau)}{p(\tau)p(d|\tau)})$ and $\nabla_\theta \log(g_\theta(\tau))$) with a bitshift of K *before* they become too large and potentially overflow. This allows us to use less integer bits than a fully sound, conservative interval analysis would permit. We will see in Section V that this scaling does not significantly affect the end to end accuracy of the inference. Programmatically, this step will be implemented by replacing line 4 of Algorithm 1 with:

$$g \leftarrow \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2^K} \log(\frac{q_\theta(\tau_i)}{p(\tau_i)p(d|\tau_i)}) \cdot \nabla_\theta(\log(q_\theta(\tau_i))$$

Choosing K. As bitshifting the fixed-point numbers in the evaluation of the ELBO gradient by K shifts allows us to use less integer bits than a naive interval analysis would suggest, the problem now becomes how to choose the right K? The idea will be to use less integer bits than $\lceil \log_2(max(|e_{smallest}|, |e_{biggest}|)) \rceil$, and whatever the difference is will be K. We first define a hyper-parameter $\alpha \in [0, 1]$. We compute the number of integer bits I as:

$$I = \lceil \alpha \cdot \log_2(max(|e_{smallest}|, |e_{biggest}|)) \rceil \quad (5)$$

$$K = \lceil \log_2(max(|e_{smallest}|, |e_{biggest}|)) \rceil - I \quad (6)$$

Intuitively if $\lceil \log_2(max(|e_{smallest}|, |e_{biggest}|)) \rceil$ is the true number of integer bits we need, we can effectively simulate that many bits using only I integer bits, but with everything first scaled down by K bitshifts. We lastly compute the fractional bits as:

$$F = w - 1 - I \tag{7}$$

where w is the wordlength (e.g. 32-bit, 64-bit, etc.) and the 1 accounts for a sign bit. This entire process is performed inside the `ComputeScale` function (line 4, Algorithm 2). During this procedure, ViX first checks if the number of fractional bits F computed in Eq. 7 is at least 8. If it is not (e.g. too many of the bits are being allocated for integer bits), then ViX will by default, force F to 8. This means the number of integer bits used will be *less* than the value of I originally computed in Eq. 5, i.e., the new number of integer bits will be $I = w-1-8$. To account for using less than the necessary number of integer bits, ViX will recompute the shift amount K in Eq. 6 using the new value of I.

This shifting is critical for making reduced-precision VI scale to larger data sets. Without these shifts, overflows occur leading to erroneous results once the number of observed data samples goes beyond a certain threshold as shown by the experiments in Section V-C. The value of α is currently selected manually by the developer, based on the properties of the program and the data. In the future, we plan to automate this process.

C. Guide Initialization

The interval analysis is useful not only for determining the number of integer bits to use, but also how to initialize the variational guide functions' parameters, as in the `InitializeGuide` function (Algorithm 2 line 6). A substantial part of the success of VI stems from how well initialized the guide functions are [15]. Since for each latent variable x in the original program, we have an interval $[x_{lower}, x_{upper}]$ stored in `Bounds`, computed as in Section III.A that abstracts the support of that variable, we can use this information to initialize the parameters of the guide function. We initialize each Gaussian variational guide (which approximates the posterior over x) to have its mean parameter set to $\frac{x_{upper}-x_{lower}}{2}$. This technique helps ensure that the guide's support overlaps significantly with the support of the respective latent variable's prior which ultimately gives better convergence. For the Gaussian variational guide distributions we also put a lower bound on the standard deviation parameter (in our experiments 0.1). ViX issues a warning if the inference suggests a standard deviation lower than this threshold. Intuitively, applications supported at this resource-constrained level are expected to operate with some amount of noise, and if very small variances are necessary, a developer would need to resort to float or double.

IV. Methodology

Benchmarks. As differentiable inference is tailored to continuous distributions, our benchmarks are continuous in nature. Our benchmarks include standard probabilistic programming benchmarks as well as Bayesian linear and polynomial regressions. Additionally to study inference-in-the-loop, as defined in ProbZelus [3], we evaluate ViX on dynamic Bayesian networks, each for a fixed number of time-steps. Inference-in-the-loop is implemented as a straight line ViX program inside of an outer `for` loop which corresponds to the number of timesteps. This number is fixed and specified at compile-time, following the setup of ProbZelus.

TABLE I
ViX Inferred Configs: $K+\langle I, F\rangle$, I integer and F fractional bits with bit shift K.

Benchmarks	Bitshifting + Precision	# Distributions
BetaBinomial [11]	$1 + \langle 10, 21\rangle$	2
Plankton [11]	$2 + \langle 19, 12\rangle$	2
LinReg [12]	$2 + \langle 21, 10\rangle$	52
PolyReg1	$5 + \langle 23, 8\rangle$	51
PolyReg2	$2 + \langle 21, 10\rangle$	52
Kalman1D [3]	$3 + \langle 21, 10\rangle$	502
HMM [3]	$3 + \langle 22, 9\rangle$	102

The BetaBinomial benchmark taken from [11] represents a continualized model where the Binomial has been continualized to a Gaussian (a requirement to run variational inference). Plankton (also from [11]) is another continualized model where one infers a continuous approximation of the number of plankton in an ecosystem. The linear regression benchmark requires inferring the slope and intercept of a line whose points have been perturbed by Gaussian noise. PolyReg1 is a polynomial regression model that infers a single coefficient of a second-degree polynomial given noisy evaluations of that polynomial. PolyReg2 is another polynomial regression that infers two coefficients of a second-degree polynomial given noisy evaluations. The last two benchmarks, HMM and Kalman1D, correspond to the inference-in-the-loop setting, in which we repeatedly perform inference over multiple timesteps. The HMM benchmark (from [3]) corresponds to a Gaussian prior over a Gaussian observed variable for monotonically increasing set of data where we infer the posterior at each timestep. Kalman1D (also from [3]) is another inference-in-the-loop benchmark that infers a Gaussian posterior at each timestep over noisy GPS data in one dimension.

Table I presents a comprehensive list of the benchmarks. It also presents the ViX compiler-inferred fixed-point sizes, including bit shifts, and the total number of distributions in the Bayesian network encoded by the probabilistic program. The number of distributions includes the number of latent and observed variables. For the Bayesian regression benchmarks, we later varied the amount of observed data the model conditions on to study how low precision variational inference with ViX scales, hence the configurations from Table I only apply to the experiments of Sections V.A and V.B.

Implementation. We implemented ViX in C++ using expression templating and built upon the FPM library [10] for fixed-point arithmetic. As our DSL is embedded in C++, we use the gcc-arm toolchain to assist in generating the executables. The underlying fixed-point implementation [10] requires a minimum of 8 Fractional bits (F), and the same is enforced by ViX. The maximum scaling amount (K) is set at 8, as any higher can lead to an unacceptable loss in precision.

Experimental Setup. To evaluate the benefits of ViX, we compare the accuracy and runtimes of variational inference in

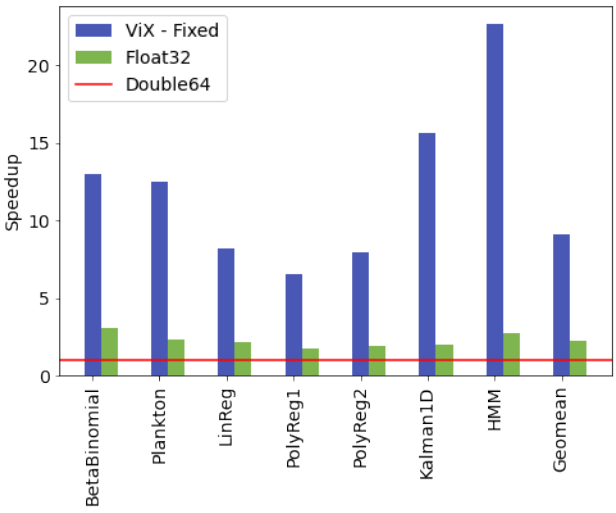

Fig. 2. Speedup of ViX compared with 32-bit float and 64-bit double

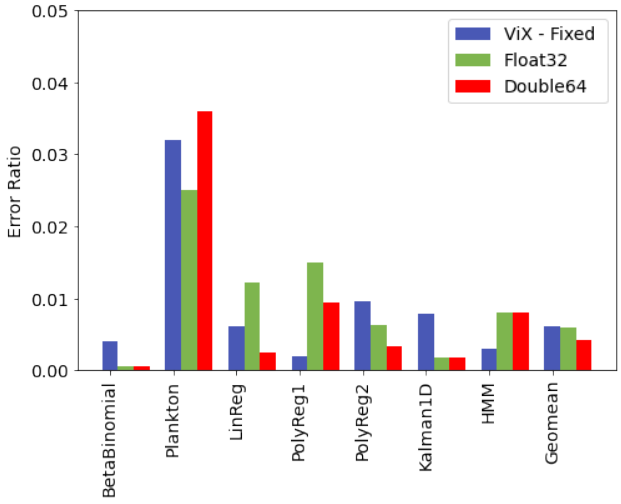

Fig. 3. Accuracy of ViX compared with 32-bit float and 64-bit double

reduced precision (ViX) against variational inference in both 32 and 64 bit floating point (baseline). To study how ViX's reduced precision variational inference scales beyond reduced precision MCMC inference, we also compare against Statheros [12]. We perform these comparisons on an Arduino Due (32-bit Atmel SAM3X8E M3, no FPU). The comparison with Statheros is performed on a CPU, as the Arduino itself lacks the memory to store the number of data observations we want to study.

We experimentally measure accuracy by measuring the error between the inferred posterior and the ground truth as follows: we take the mean of the variational approximation of the posterior as the point estimate of the latent, denoted as *est*, and compute the error ratio relative to the ground truth value. We obtain ground truth *gt* in two ways: for LinReg, PolyReg1, PolyReg2, Kalman1D and HMM we obtain fixed data; for BetaBinomial and Plankton we compute the mean of the posterior obtained by running MCMC offline for 100,000 iterations. For a single latent variable, the error ratio is computed as $Err = |\frac{gt-est}{gt}|$; for multiple latent variables, we take the geomean of the error ratios for each latent variable.

Variational Inference. For both the variational inference comparison between fixed-point (ViX), 32-bit floating-point and 64-bit double precision, and the comparison against Statheros we run the gradient descent in VI for 1500 iterations. For HMM and Kalman1D we run VI at each time-step for 200 iterations, because those streaming applications typically have low-latency requirements and strict convergence is often not necessary for initial time steps. We used the same learning rate of 0.01 for all the benchmarks. We selected the bit-shifting hyperparameter α to be 0.9 in our experiments for all benchmarks. We initialize the parameters of the variational guide by setting the mean of each Gaussian to be the center point of the interval that the static analysis assigns to that latent variable.

V. Evaluation

We study the following 3 research questions:

- **RQ1:** Does performing Variational Inference for probabilistic programs in fixed-point precision run faster than performing the same inference in floating point or double precision on a resource constrained edge device?
- **RQ2:** Is Variational Inference in fixed-point precision comparably accurate to performing the same inference in floating point or double precision?
- **RQ3:** Can ViX and its incorporation of a bit shift scaling factor allow Variational Inference in fixed-point precision to scale to datasets larger than what the fixed-point precision MCMC of existing work [12] supports?

A. RQ1: ViX Speed

Figure 2 presents the speedup ratios compared to 64-bit double precision baseline (red horizontal line). The x-axis presents each benchmark while the y-axis corresponds to the speedup ratio (relative to double precision). The geomean of the speedups of ViX (shown in blue) relative to 64-bit double precision is $11.38\times$ while the speedup of ViX compared to 32-bit floating point precision (shown in green) is $5.08\times$. The raw runtimes of ViX ranged from 247ms (BetaBinomial) to 7.9s (Kalman1D) while for 32-bit floating point the times range from 1s (BetaBinomial) to 63s (Kalman1D). Likewise, the raw runtimes of 64-bit double precision on the Arduino ranged from 3.2s (BetaBinomial) to 125s (Kalman1D).

The speedup stems from the fact that the Arduino Due like other resource-constrained edge-devices lacks hardware support for floating-point operations, and instead has to rely on software emulation which is significantly slower than fixed-point arithmetic.

B. RQ2: ViX Accuracy

Figure 3 presents the error ratio of the inferred posterior parameters returned by variational inference. The respective geomean accuracies of ViX (shown in blue), 32-bit floating point (shown in green) and 64-bit double precision (shown in red) are 0.0064, 0.0059, 0.0042 respectively. In all cases the

TABLE II
VIX VS. STATHEROS FOR DIFFERENT DATA SIZES (LOWER IS BETTER, × REPRESENTS FAILURE)

		Error Ratio for Observed Data Size N					
Method	Benchmark	N=50	200	500	1000	8000	16000
ViX	PolyReg1	0.003	0.016	0.007	0.014	0.006	0.022
	PolyReg2	0.003	0.012	0.015	0.003	0.003	0.004
	LinReg	0.013	0.020	0.002	0.005	0.021	0.068
Statheros	PolyReg1	0.0010	×	×	×	×	×
	PolyReg2	0.0002	0.0003	0.0023	×	×	×
	LinReg	0.0091	0.0050	×	×	×	×

error ratios were less than 0.05 meaning all methods were able to obtain comparably high precision. The results show that the error ratios obtained by using reduced-precision are comparable to those obtained with higher precision.

C. RQ3: ViX Scalability

Table II shows the results of how ViX scales to large datasizes compared to the previous state-of-the-art Statheros [12]. We define an inference failure to be a case where the inference returns an unacceptably high error ratio which for our experiments was taken to be 0.1. Thus all results in Table II are either less than that threshold or just a failure (denoted via $\times$). For the LinReg benchmark, while the MCMC inference performed by Statheros can successfully handle $\leq$ 200 observed data points, for ≥ 500 datapoints, Statheros' MCMC encounters an overflow that causes the entire computation to incur erroneous results. In contrast, ViX can successfully scale VI to $16,000$ observed datapoints.

On the PolyReg1 benchmark, Statheros handles 50 observations but encounters runtime errors before 200 observed datapoints. In contrast, ViX attains accurate inference for even $16,000$ observed datapoints thus ViX scales to at least $80\times$ more observed data points. Similarly, for PolyReg2, Statheros can support observations over 500 datapoints but fails before 1000, while ViX successfully scales inference to $16,000$ observed datapoints. Hence for PolyReg2, ViX supports at least $16\times$ more observations than Statheros.

We attribute the unacceptable error of Statheros for larger numbers of data observations to the fact that overflows eventually occur since there are not enough integer bits as Statheros limits the number of integer bits to at most 19. In contrast, because of how ViX uses the scaling factor K, we are able to "simulate" having more integer bits than we actually use. This step is precisely why ViX is able to scale to such large datasets. If ViX did not support scaling, then LinReg would become imprecise after 710, PolyReg1 after 250, and PolyReg2 after 650 observed data points.

VI. RELATED WORK

While there have been a few works on compiling probabilistic programming languages for embedded systems [2], [4], [12], the closest in spirit to ours is Statheros [12] as they are the only other work that performs reduced precision probabilistic programming in the resource constrained setting. However their work only supports the MCMC inference algorithm instead of variational inference, and their work is also severely limited by scalability issues as we have shown. Also closely related is ProbLP [16] which performs reduced precision inference, however they only support *exact* inference and thus cannot scale in the way ViX can.

Researchers also proposed performing variational inference on FPGAs [1], [6], but these techniques do not target the resource-constrained, CPU-only TinyML setting as we do. Further, those works support Bayesian neural networks instead of general probabilistic programs.

Lastly, while researchers have proposed static analyses for range analysis to automatically infer fixed-point sizes [7], [12], to the best of our knowledge none of these analyses have to infer ranges over the derivative terms in an optimization problem for differentiable inference the way ViX does.

VII. CONCLUSION

We proposed ViX, a probabilistic programming system that automates the task of performing variational inference in reduced precision. We showed that reducing the precision allows substantial speedups for inference time on resource constrained hardware when compared to floating point and that this speedup does not lead to sacrificing any accuracy. Furthermore we showed that low-precision, resource-constrained variational inference is still able to scale far beyond existing edge-computing probabilistic inference platforms.

ACKNOWLEDGEMENTS

This research was supported in part by NSF Grants No. CCF-1846354, CCF-1956374, CCF-200888, and a Sloan UCEM Graduate Scholarship.

REFERENCES

[1] Hiromitsu Awano and Masanori Hashimoto. Bynqnet: Bayesian neural network with quadratic activations for sampling-free uncertainty estimation on fpga. In *DATE*, 2020.

[2] Subho S Banerjee et al. Acmc 2: Accelerating markov chain monte carlo algorithms for probabilistic models. In *ASPLOS*, 2019.

[3] Guillaume Baudart et al. Reactive probabilistic programming. In *PLDI*, 2020.

[4] Guillaume Baudart et al. Jax based parallel inference for reactive probabilistic programming. In *LCTES*, 2022.

[5] David M Blei et al. Variational inference: A review for statisticians. *Journal of the American statistical Association*, 2017.

[6] Ruizhe Cai et al. Vibnn: Hardware acceleration of bayesian neural networks. *ACM SIGPLAN Notices*, 2018.

[7] Claire Fang Fang et al. Fast, accurate static analysis for fixed-point finite-precision effects in dsp designs. In *ICCAD*, 2003.

[8] Andrew Gordon et al. Probabilistic programming. In *FoSE*. 2014.

[9] Urs Köster et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. *Neurips*, 2017.

[10] Michael Lankamp et al. fpm library. 2020.

[11] Jacob Laurel and Sasa Misailovic. Continualization of probabilistic programs with correction. In *ESOP*, 2020.

[12] Jacob Laurel, Rem Yang, Atharva Sehgal, Shubham Ugare, and Sasa Misailovic. Statheros: Compiler for efficient low-precision probabilistic programming. In *Design Automation Conference (DAC)*, 2021.

[13] Jacob Laurel, Rem Yang, Gagandeep Singh, and Sasa Misailovic. A dual number abstraction for static analysis of clarke jacobians. *Proceedings of the ACM on Programming Languages*, (POPL), 2022.

[14] Jacob Laurel, Rem Yang, Shubham Ugare, Robert Nagel, Gagandeep Singh, and Sasa Misailovic. A general construction for abstract interpretation of higher-order automatic differentiation. (OOPSLA), 2022.

[15] Rajesh Ranganath et al. Black box variational inference. In *Artificial intelligence and statistics*. PMLR, 2014.

[16] Nimish Shah et al. Problp: A framework for low-precision probabilistic inference. In *DAC*, 2019.

[17] Randy Yates. Fixed-point arithmetic: An introduction. *Digital Signal Labs*, 81(83):198, 2009.