# Compiling Markov Chain Monte Carlo Algorithms for Probabilistic Modeling

Daniel Huang
Harvard University
Cambridge, MA, USA
dehuang@fas.harvard.edu

Jean-Baptiste Tristan
Oracle Labs
Burlington, MA, USA
jean.baptiste.tristan@oracle.com

Greg Morrisett
Cornell University
Ithaca, NY, USA
jgm19@cornell.edu

## Abstract

The problem of probabilistic modeling and inference, at a high-level, can be viewed as constructing a (*model*, *query*, *inference*) tuple, where an *inference* algorithm implements a *query* on a *model*. Notably, the derivation of inference algorithms can be a difficult and error-prone task. Hence, researchers have explored how ideas from *probabilistic programming* can be applied. In the context of constructing these tuples, probabilistic programming can be seen as taking a language-based approach to probabilistic modeling and inference. For instance, by using (1) appropriate languages for expressing models and queries and (2) devising inference techniques that operate on encodings of models (and queries) as program expressions, the task of inference can be automated.

In this paper, we describe a compiler that transforms a probabilistic model written in a restricted modeling language and a query for posterior samples given observed data into a Markov Chain Monte Carlo (MCMC) inference algorithm that implements the query. The compiler uses a sequence of intermediate languages (ILs) that guide it in gradually and successively refining a declarative specification of a probabilistic model and the query into an executable MCMC inference algorithm. The compilation strategy produces composable MCMC algorithms for execution on a CPU or GPU.

***CCS Concepts*** **• Mathematics of computing → Bayesian networks**; **Bayesian computation**; **Markov-chain Monte Carlo methods**; **• Software and its engineering → Compilers**

***Keywords*** probabilistic programming, intermediate languages, Markov-chain Monte Carlo kernels

## 1. Introduction

Consider the problem of clustering a set of data points in $D$-dimensional Euclidean space $\mathbb{R}^D$ into $K$ clusters. One approach is to construct a *probabilistic (generative) model* to explain how we believe the observations are generated. For instance, one explanation might be that there are (1) $K$ cluster centers chosen randomly according to a normal distribution and (2) that each data point (independently of each other data point) is normally distributed around a randomly chosen cluster center. This describes what is known as a Gaussian Mixture Model (GMM).

More generally, a probabilistic model induces a probability distribution, which we can *query* for quantities of interest. For example, the query "what is the most likely cluster assignment of each observation under the GMM model?" formulates our original problem of clustering a collection of points in probabilistic terms. To answer such a query, practitioners implement an *inference algorithm*. Inference is analytically intractable in general. Consequently, practitioners devote a significant amount of time to developing and implementing *approximate* inference algorithms to (approximately) answer a query for a given model.

In summary, we can think of the problem of probabilistic modeling and inference as constructing a (*model*, *query*, *inference*) tuple, where the *inference* algorithm implements a *query* on a *model*. To simplify this task, researchers have explored how ideas from *probabilistic programming* can be applied. In the context of constructing these tuples, probabilistic programming can be seen as taking a language-based approach to probabilistic modeling and inference. For instance, by using (1) appropriate languages for expressing models and queries and (2) devising inference techniques that operate on encodings of models (and queries) as program expressions, the task of inference can be automated. Due to the promise of this approach, many probabilistic programming languages (PPLs) have been proposed in the literature to explore the various points of the design space [5, 10, 11, 15, 16, 23, 26, 28, 29].

In this paper, we a present a tool called AugurV2 for constructing (*model*, *query*, *inference*) tuples of a restricted form, where (1) the *model* is expressed in a domain-specific modeling language similar in expressive power to Bugs [23] or Stan [8] (as compared to modeling languages embedded in general-purpose languages [11, 28]), (2) the *query* is for posterior samples given observed data, and (3) the *inference* is automatically derived and is restricted to a family of algorithms based on MCMC sampling (as compared to languages that provide a set of fixed inference strategies [8, 23, 29] or those that provide domain-specific languages for the specification of inference [5, 14, 26]). Even in this restricted setting, automating posterior inference based on MCMC poses at least three challenges.

- As inference is intractable in general, it is unlikely that a black-box approach to MCMC will work well for all probabilistic models of interest. Consequently, we would like a way to derive model-specific facts useful for constructing MCMC algorithms. For instance, systems such as Bugs [23] leverage conjugacy relations and Stan [8] leverages gradients.
- There are many kinds of MCMC algorithms. Ideally, our tool should support as many as possible, given that these algorithms may exhibit different computational and statistical behaviors depending on the model to which it is applied. More generally, end users may want more fine-grained control over how to perform inference. For example, Blaise [5] provides a domain-specific graphical language for expressing models and inference algorithms. Other systems such as Edward [26] and Venture [14] explore other designs for providing fine-grained control of inference.
- MCMC sampling is computationally expensive [7]. To improve the computational efficiency, we may want to leverage parallelism such as in our previous work on Augur [27] and other systems such as Anglican [28].

Our solution is to come up with a sequence of intermediate languages (ILs) that enable our tool to gradually and successively refine a *declarative* specification of a model and a query for posterior samples into an *executable* Markov Chain Monte Carlo (MCMC) inference algorithm. For the purposes of this paper, we will refer to the process above as *compilation*, and hence, will also refer to our tool as a compiler. We address the challenges listed above with the following aspects of our compiler design.

- The compiler uses a *Density IL* to express the density factorization of a model (Section 3.1). This IL can be analyzed to symbolically compute the conditionals of a model so that the compiler can generate *composable* MCMC algorithms.
- The compiler uses a *Kernel IL* to express the high-level structure of a MCMC algorithm as a composition of MCMC algorithms (Section 4.1). In particular, the AugurV2 compiler supports multiple kinds of basic MCMC algorithms, including ones that leverage conjugacy relations (*e.g.*, Gibbs) and gradient information (*e.g.*, HMC). The Kernel IL is similar to the subset of the Blaise language [5] that corresponds to inference. We use the IL for the purposes of compilation, whereas Blaise focuses on the expressivity of the language.
- The compiler uses *Low++* and *Low-- ILs* to express the details of MCMC inference (Section 4.3). The first exposes parallelism and the second exposes memory management. The AugurV2 compiler leverages these ILs to support both CPU and GPU (Section 5) compilation of composable MCMC algorithms.

Our preliminary experiments show that such a compilation strategy both enables us to leverage the flexibility of composable MCMC inference algorithms on the CPU and GPU as well as improve upon the scalability of automated inference (Section 7). We have also found it relatively easy to add new base MCMC updates to the compiler without restructuring its design. Hence, the compiler is relatively extensible. It would be an interesting direction to see what aspects of AugurV2's ILs and compilation strategy could be reused in the context of a larger infrastructure for compiling PPLs, but that is not in scope for this paper. AugurV2 is available at `https://github.com/danehuang/augurv2/`.

## 2. AugurV2 Overview

At a high-level, the AugurV2 modeling language expresses *Bayesian networks* whose graph structure is fixed. This class contains many practical models of interest, including regression models, mixture models, topic models, and deep generative models such as sigmoid belief networks. Models that cannot be expressed include ones that use non-parametric distributions (which can be encoded in a general-purpose language) and models with undirected dependency structure (*e.g.*, Markov random fields whose dependency structure is hard to express in a functional setting). Note that even in this setting, inference can still be difficult due to the presence of high-dimensional distributions. For example, the dimensionality of a mixture model such as the GMM scales with the number of observations—each observed point introduces a corresponding latent (*i.e.*, unobserved) cluster assignment. Consequently, the design of AugurV2 focuses on generating efficient MCMC algorithms for posterior inference on this restricted class of models.

### 2.1 Probabilistic Modeling Setup

AugurV2 provides a simple, first-order modeling language for expressing probabilistic generative models with *density factorization*

$$p(\theta, y) = p(\theta)\, p(y \mid \theta)\,,$$

where $p(\theta)$ is a distribution over parameters $\theta$ called the *prior* and $p(y \mid \theta)$ is the *conditional distribution* of the data $y$ given the parameters $\theta$. In this paper, we will only

```
(K, N, mu_0, Sigma_0, pis, Sigma) => {
  param mu[k] ~ MvNormal(mu_0, Sigma_0)
      for k <- 0 until K ;
  param z[n] ~ Categorical(pis)
      for n <- 0 until N ;
  data x[n] ~ MvNormal(mu[z[n]], Sigma)
      for n <- 0 until N ;
}
```

Figure 1: An AugurV2 program encoding a GMM. At a high-level, the modeling language mirrors random variable notation.

consider probability distributions with densities[1], and hence, will interchangeably use the two terms. Given observed data, written $y^*$ to distinguish it from the formal parameter $y$ in the distribution $p(y \mid \theta)$, the AugurV2 system implements a MCMC algorithm to draw samples from the *posterior distribution*

$$p(\theta \mid y^*) = \frac{1}{Z} p(\theta)\, p(y^* \mid \theta)\,,$$

where $Z = \int p(\theta)\, p(y^* \mid \theta)\, d\theta$. The notation $p(y^* \mid \theta)$ is perhaps better written as $p(y = y^* \mid \theta)$ to indicate that it is a function of $\theta$ and is known as the *likelihood function*. In the rest of the section, we will show how to encode the GMM we gave in the introduction in AugurV2's modeling language, how to use the system to perform posterior inference, and overview the compilation process.

### 2.2 Modeling Language

Figure 1 contains an AugurV2 program expressing the GMM we introduced earlier. At a high-level, the modeling language is designed to mirror random variable notation. The *model body* is a sequence of declarations, each consisting of a random variable and its distribution. Each declaration is annotated as either a model parameter (`param`) or observed data (`data`). Model parameters are inferred (*i.e.*, output) whereas model data is supplied by the user (*i.e.*, input). In the case of a GMM, the means `mu` and cluster assignments `z` are model parameters[2], while the `x` values are model data. It is also possible to define a random variable as a deterministic transformation of existing variables, although this feature is not needed to express the GMM in this example.

```
import AugurV2Lib
import numpy as np

# Part 1: Load data
x = load_gmm_data('/path/to/data')
N, D = x.shape; K = 3
mu0 = np.zeros(D); S0 = np.eye(D);
S = np.eye(D); pis = np.full(K, 1.0/K)
,
# Part 2: Invoke AugurV2
with AugurV2Lib.Infer('path/to/model') as aug:
  opt = AugurV2Lib.Opt(target='cpu')
  aug.setCompileOpt(opt)
  sched = 'ESlice mu (*) Gibbs z'
  aug.setUserSched(sched)
  aug.compile(K, N, mu0, S0, pis, S)(x)
  samples = aug.sample(numSamples=1000)
```

Figure 2: Fitting a GMM with AugurV2 using a Python interface.

At the top-level, the model closes over any free variables mentioned in the model body. These include the model hyper-parameters (`mu_0, Sigma_0, pis, Sigma`) and any other variables over which the model is parameterized by (`K, N`). The modeling language can be considered as alternative notation for expressing the density factorization, which we give for the GMM below.

$$\prod_{k=1}^{K} p(\mu_k \mid \mu_0, \Sigma_0) \prod_{n=1}^{N} p(z_n \mid \pi)\, p(y_n^* \mid \mu_{z_n})$$

Random vectors (*e.g.*, `mu[k]`) are specified using comprehensions with the `for` construct. The semantics of AugurV2 comprehensions are *parallel*, meaning that they do not depend on the order of evaluation. The idea is to provide a syntactic construct that corresponds to the mathematical phrase "let $\mu_k \sim \mathcal{N}(\vec{\mu_0}, \Sigma)$ for $0 \leq k < K$." Because such a mathematical statement is implicitly parallel and occurs frequently in the definition of probabilistic models, we opt for syntactic constructs that capture standard statistical practice, instead of more standard programming language looping constructs (*e.g.*, an imperative for loop).

As AugurV2 provides only parallel comprehensions, we discourage users from expressing models with sequential dependencies. For example, we would need to write a Hidden Markov Model, where each hidden state depends on the previous state, by unfolding the entire model. This is *doable*, but does not take advantage of the design of AugurV2.

AugurV2 imposes two further restrictions. First, comprehension bounds cannot mention model parameters. This forces the comprehension bounds to be constant (although they can still be ragged). For this reason, we say that AugurV2 expresses *fixed-structure* models. Second, AugurV2 provides only primitive distributions whose probability density function (PDF) or probability mass function (PMF) has known functional form. Hence, the models AugurV2 expresses are *parametric*. Currently, AugurV2 does not support non-parametric distributions (*i.e.*, distributions with an infinite number of parameters so that the number of parameters used scales with the number of observations).

[1] That is, with respect to Lebesgue measure, counting measure, or their products.

[2] To be pedantic, we should call these latent variables.

### 2.3 Using AugurV2

Figure 2 contains an example of how to invoke AugurV2's inference capabilities contained in the Python module `AugurV2`. The first part of the code loads the data and hyper-parameters and the second part invokes AugurV2.

Instances of the `AugurV2Infer` class provide methods for obtaining posterior samples (*e.g.*, `sample`). The class takes a path to a file containing the model. Once we have created an instance of an AugurV2 inference object (`aug`), we need to indicate to the compiler what kind of MCMC sampler it should generate (via `setCompileOpt` and `setUserSched`). For example, we can set the target for compilation as either the CPU or GPU (`target`). We can also customize the MCMC algorithm by choosing our own MCMC schedule (via `setUserSched`).

In this example, we decide to apply Elliptical Slice sampling [17] to the cluster means (`ESlice mu`) and Gibbs sampling to the cluster assignments (`Gibbs z`). Hence, this schedule indicates a *compositional* MCMC algorithm, where we apply different MCMC updates to different portions of the model. This feature is inspired by the *programmable inference* proposed by other probabilistic programming systems (*e.g.*, Venture [14]). If a user schedule is not specified, the compiler uses a heuristic to select which combination of MCMC methods to use.

To compile the model, we supply the model arguments, hyper-parameters, and data (as Python variables) in the order that they are specified in the model. Thus, the AugurV2 compiler is invoked at *runtime*. Consequently, given different data sizes and hyper-parameter settings, the AugurV2 compiler may choose to generate a different MCMC algorithm. The compiler generates Cuda/C code depending on whether the target is the GPU or the CPU. The native inference code is then further compiled using Nvcc (the Cuda compiler) or Clang into a shared library which contains inference code for a specific instantiation of a model. After compilation, the object `aug` contains a collection of inference methods that wrap the native inference code. The Python interface handles the conversion of Python values to and from Cuda/C via the Python CTypes interface. Hence, the user can work exclusively in Python. In this example, we ask for 1000 samples from the posterior distribution.

### 2.4 Compilation Overview

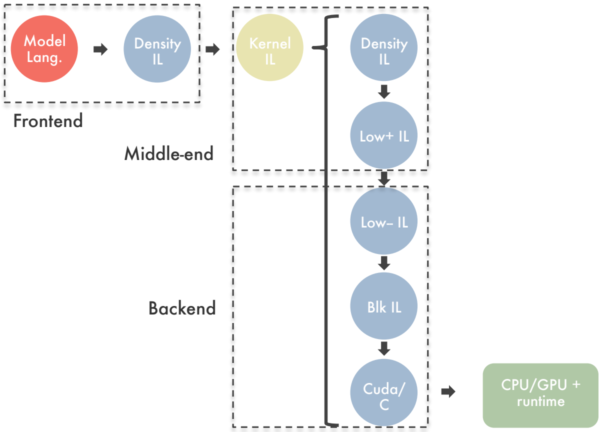

Figure 3: Overview of the AugurV2 compilation process. It is comprised of a Frontend (model to density factorization), Middle-end (density factorization to executable inference), and Backend (executable inference to native inference).

The compiler uses ILs to structure the compilation (summarized in Figure 3) from model and query into inference.[3] Notably, the ILs enable the AugurV2 compiler to successively and gradually refine a *declarative* specification of a model and query into an *executable* inference algorithm that contains details such as memory consumption and parallelism. The successive removal of layers of abstraction is the typical manner in which ILs aid the compilation of traditional language.

Of course, we will need a semantics to justify the compilation process. The semantics of probabilistic programs is an active area of research due to the need to model continuous distributions and its interaction with standard programming language features [6, 12, 25]. As AugurV2 provides a simple language that intuitively expresses Bayesian networks, its semantics is not an issue, in contrast to more expressive languages that provide higher-order functions and recursion. Nevertheless, we point out that AugurV2 can be given semantics in terms of (Type-2) computable distributions [12]. This enables us to think of a compiler as a (computable) function that witnesses the result that (Type-2) computable distributions are realizable by (Type-2) computable sampling algorithms. In our case, the compiler realizes a MCMC sampling algorithm.

## 3. Frontend

In the first step of compilation, the compiler transforms a model expressed in the modeling language into its corresponding density factorization in the *Density IL*. This follows standard statistical practice, where models expressed using random variables (AugurV2's modeling language) are converted into its description in terms of densities (Density IL). The compiler can analyze the density factorization and symbolically compute the *conditionals* (up to a normalizing constant) of the model for each model parameter (see Section 3.3) to support the generation of composable MCMC algorithms. The analysis is based off the one implemented in our previous work [27], although the results of the analysis then were only used to generate Gibbs samplers.

[3] The first author's dissertation contains more details about AugurV2 and its semantics.

$$
\begin{aligned}
&\mathit{obj} ::= \lambda(\overrightarrow{x}).\, \mathit{fn} \qquad\qquad \mathit{gen} ::= e\ \texttt{until}\ e \\
&\mathit{fn} ::= p_{\mathit{dist}(\overrightarrow{e})}(e) \ \mid\ \mathit{fn}\,\mathit{fn} \ \mid\ \prod_{x \leftarrow \mathit{gen}} \mathit{fn} \\
&\quad \mid\ \texttt{let}\ x = e\ \texttt{in}\ \mathit{fn} \ \mid\ [\mathit{fn}]_{x=e} \\
&e ::= x \ \mid\ i \ \mid\ r \ \mid\ \mathit{dist}(\overrightarrow{e}) \ \mid\ \mathrm{op}^n(\overrightarrow{e}) \ \mid\ e[e] \\
&\sigma ::= \texttt{Int} \ \mid\ \texttt{Real} \qquad \tau ::= \sigma \ \mid\ \texttt{Vec}\ \tau \ \mid\ \texttt{Mat}\ \sigma
\end{aligned}
$$

Figure 4: The Density IL encodes the density factorization of a probabilistic model.

### 3.1 Representing Models: The Density IL

The syntax for the Density IL is summarized in Figure 4. It encodes the density factorization of a model. As an example, the GMM from before (Figure 1) is encoded in the Density IL as

$$
\lambda(K, N, \mu_0, \Sigma_0, \pi, \Sigma, \mu, z, x).\ \prod_{k \leftarrow \mathit{gen}_1} p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu[k])
$$
$$
\prod_{n \leftarrow \mathit{gen}_2} p_{\mathcal{D}(\pi)}(z[n]) \prod_{n \leftarrow \mathit{gen}_2} p_{\mathcal{N}(\mu[z[n]], \Sigma)}(x[n] \mid \mu[z[n]])
$$

where $\mathit{gen}_1 = 0\ \texttt{until}\ K$ and $\mathit{gen}_2 = 0\ \texttt{until}\ N$.

At the top-level, a model is represented as a single function $\lambda(\overrightarrow{x}).\, \mathit{fn}$ with bindings $\overrightarrow{x}$ for model hyper-parameters, meta-parameters, and data, and a density function body *fn*. A density function is either (1) the density of a primitive, parameterized distribution $p_{\mathit{dist}}(\overrightarrow{e})$, (2) the composition of two density functions $\mathit{fn}_1\ \mathit{fn}_2$ (*i.e.*, multiplication) of two density functions, (3) a structured product $\prod_{x \leftarrow \mathit{gen}} \mathit{fn}$ that specifies a product of density functions according to the comprehension $x \leftarrow \mathit{gen}$, (4) a standard let-binding $\texttt{let}\ x = e\ \texttt{in}\ \mathit{fn}$, or (5) an indicator function $[\mathit{fn}]_{x=e}$ that takes on the value inside the brackets if the condition $x = e$ is satisfied and 1 otherwise.

The Density IL is simply-typed. Base types include integers `Int` and reals `Real`. Compound types include vectors `Vec` $\tau$ and matrices `Mat` $\sigma$. Thus, compound types such as vectors of matrices are allowed, whereas matrices of vectors are rejected. The type system is used to check simple properties. For example, the type system checks that densities are defined on the appropriate spaces and that the comprehension bounds *gen* in a product of density function $\prod_{x \leftarrow \mathit{gen}} \mathit{fn}$ is indeed a vector of integers.

### 3.2 MCMC and Conditionals

Before we describe the symbolic computation on the Density IL, we briefly introduce background on MCMC. Suppose we would like to construct a sampler to draw from the probability density $p(x, y)$. The Metropolis Hastings (MH) algorithm is a MCMC algorithm that gives a sampler for $p(x, y)$ given a transition function $q(x, y \rightarrow x', y')$, *i.e.*, a conditional density $q(x', y' \mid x, y)$[4]. The transition function is also called a *proposal*. In particular, every MCMC algorithm can be seen as a special case of the MH algorithm with a proposal of a specific form. For instance, the HMC algorithm uses a gradient-based proposal [7].

Given a multivariate distribution such as $p(x, y)$, it may be undesirable to construct the entire proposal $q(x, y \rightarrow x', y')$ in a single step. For instance, the GMM has both discrete and continuous variables, so one cannot directly apply the HMC algorithm (which uses gradients) to the entire model without handling the discrete variables in a special manner. Here, it is useful to construct the MCMC sampler as the *composition* of two MCMC samplers, one that targets the conditional $p(x \mid y)$ and another that targets the conditional $p(y \mid x)$. In particular, this involves constructing the two simpler proposals $q_1(x \rightarrow x'; y)$ (holding $y$ fixed) and $q_2(y \rightarrow y'; x)$ (holding $x$ fixed). For situations like this, the AugurV2 compiler supports the symbolic computation of conditionals.

### 3.3 Approximating Conditionals

In a more traditional setting, one might represent the density factorization of a model using a Bayesian network. As AugurV2 programs denote Bayesian networks, we could compile the Density IL into a Bayesian network and use standard, graph-based algorithms for determining how the conditionals factorize. However, the resulting graph could be quite large and would be difficult to use in other phases of compilation (*e.g.*, for generating GPU code). Instead, the compiler symbolically computes the conditionals of the model.

Computing the conditional of the density factorization $p(x)\, p(y \mid x)\, p(y \mid z)$ with respect to (w.r.t.) $x$ up to a normalizing constant is equivalent to simplifying the expression below.

$$
p(x \mid y, z) = \frac{p(x)\, p(y \mid x)\, p(y \mid z)}{\int p(x)\, p(y \mid x)\, p(y \mid z) dx}
$$

The expression simplifies to

$$
p(x \mid y, z) \propto p(x)\, p(y \mid x)
$$

where we cancel the terms that have no functional dependence on $x$. This computation is isomorphic to the computation of conditional independence relationships in Bayesian networks [4].

The challenge with symbolically computing conditionals of the density factorization comes from handling structured products in the Density IL. Conceptually, these products can be unfolded because we compile at runtime when the size of the structured product is known. However, the resulting density factorization loses regularity and can be quite large. Hence, the compiler treats structured products symbolically at the cost of precision of the conditional computed.

[4] More generally, $q$ would be a *probability kernel.*

For example, suppose we would like to compute the conditional of

$$\prod_{k \leftarrow gen} p(x_k) \prod_{k' \leftarrow gen} p(y_{k'} \mid x_{k'})$$

w.r.t. $x_k$ for some $k$. In particular, the compiler cannot tell that this expression simplifies to $p(x_k)\, p(y_k \mid x_k)$ because the terms are syntactically different. To handle this, the compiler uses a factoring rewrite rule, given below.

$$\prod_{i \leftarrow gen_1} \mathit{fn}_1 \prod_{j \leftarrow gen_2} \mathit{fn}_2 \rightarrow \prod_{i \leftarrow gen_1} \mathit{fn}_1\, \mathit{fn}_2 \text{ when } gen_1 = gen_2$$

As a reminder, comprehension bound expressions in AugurV2 cannot refer to random variables and so are constant. If the compiler cannot determine the equality of comprehension bounds, it will not be factored, and precision in the approximation of the conditional can be lost.

The compiler also implements the normalization rule

$$\prod_{i \leftarrow gen_i} \mathit{fn} \rightarrow \prod_{k \leftarrow gen_k} \prod_{i \leftarrow gen_i} [\mathit{fn}]_{k=z} \,,$$

where $z$ is a Categorical variable with range $gen_k$ mentioned in *fn*. This rule deals with mixture models, which is a common pattern in probabilistic models. For example, the conditional of $\mu$ for the GMM (omitting the bounds)

$$\prod_k p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu[k]) \prod_n p_{\mathcal{N}(\mu[z[n]], \Sigma)}(y[n])$$

can be transformed into

$$\prod_k p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu[k]) \prod_k \prod_n [p_{\mathcal{N}(\mu[z[n]], \Sigma)}(y[n])]_{k=z[n]} \,,$$

after which we can apply the factoring rule. This says that each cluster location $\mu_k$ depends only on the data points $y_n$ whose cluster assignment $z_n$ is associated with the current cluster $k$. Currently, the compiler attempts to apply the categorical indexing rule first and then attempts to factor. For the restricted modeling language that we consider, we have found the two normalization rules to be precise enough for practical use.

## 4. Middle-End

The AugurV2 middle-end converts a model encoded as its density factorization into a high-level, executable inference algorithm. This phase uses two additional ILs. First, the *Kernel IL* represents the high-level structure of a MCMC algorithm as a composition of basic MCMC updates applied to the model's conditionals. Second, the *Low++ IL* serves as the first pass target for executable MCMC code. It is an imperative language that makes sources of parallelism explicit, but abstracts away memory management. Importantly, we can leverage domain-specific knowledge of each base MCMC update to directly annotate parallelism in the code. Thus, we do not need to (re)discover parallelism at a lower-level of abstraction.

$$\begin{aligned}
\mathit{sched}\ \alpha &::= \lambda(\overrightarrow{x}).\ k\ \alpha \\
k\ \alpha &::= (\kappa\ \alpha)\ \mathit{ku}\ \alpha \ \mid\ k\ \alpha \otimes k\ \alpha \\
\mathit{ku} &::= \texttt{Single}(x) \ \mid\ \texttt{Block}(\overrightarrow{x}) \\
\kappa\ \alpha &::= \texttt{Prop}\ (\texttt{Maybe}\ \alpha) \ \mid\ \texttt{FC} \\
&\quad\ \mid\ \texttt{Grad}\ (\texttt{Maybe}\ \alpha) \ \mid\ \texttt{Slice}
\end{aligned}$$

Figure 5: The Kernel IL encodes the structure of an MCMC algorithm. It is parametric in $\alpha$, which is instantiated with successively lower level ILs that encode how the MCMC algorithm is implemented.

### 4.1 Representing MCMC I: The Kernel IL

The AugurV2 compiler represents a MCMC algorithm as a composition of base updates in the *Kernel IL*, whose syntax is summarized in Figure 5. As an example, the user schedule presented in Figure 2 is encoded in the Kernel IL as

$$\begin{aligned}
&\lambda(K, N, \mu_0, \Sigma_0, \pi, \Sigma, \mu, z, x). \\
&\quad \texttt{Slice}\ \texttt{Single}(\mu)\, \mathit{fn}_\mu \otimes \texttt{FC}\ \texttt{Single}(z)\, \mathit{fn}_z \,,
\end{aligned}$$

where $\mathit{fn}_\mu$ and $\mathit{fn}_z$ correspond to the conditionals of $\mu$ and $z$ up to a normalizing constant respectively.

The syntax

$$\texttt{Slice}\ \texttt{Single}(\mu)\, \mathit{fn}_\mu$$

encodes a base update indicating that we apply Slice sampling to the variable $\mu$ with proportional conditional $\mathit{fn}_\mu$. The kernel unit *ku* specifies whether we should sample a variable $x$ by itself (*i.e.*, Single(x)), or whether to sample a list of variables $\overrightarrow{x}$ jointly (*i.e.*, Block(x)). Sampling variables jointly, also known as blocking, is useful when those variables are heavily correlated. In general, a base update $\kappa\ \mathit{ku}\ \alpha$ is parametric in the representation of the proportional conditional $\alpha$. This enables the compiler to successively instantiate $\alpha$ with lower-level ILs that expose more computational details of inference (*e.g.*, parallelism or memory usage). Given two MCMC updates $k_1\ \alpha$ and $k_2\ \alpha$, we can sequence (*i.e.*, compose) the two updates with the syntax $k_1\ \alpha \otimes k_2\ \alpha$. Sequencing is not commutative, *i.e.*, the MCMC update $k_2\ \alpha \otimes k_1\ \alpha$ is different from $k_1\ \alpha \otimes k_2\ \alpha$.

In addition to Slice (Slice) and closed-form conditional updates (FC), the Kernel IL also supports proposal-based updates (Prop) and gradient-based updates (Grad). These updates each contain an optional piece of code of type $\alpha$ that specifies the proposal and the gradient respectively for the corresponding conditional that the base update is applied to. In contrast, Slice and closed-form conditional updates can be derived purely from the model conditionals.

The ergodic convergence of base MCMC kernels in the Kernel IL can be established via standard proofs in the MCMC literature [7, 24]. Composite kernels created with

$$
\begin{aligned}
\mathit{decl} &::= \mathit{name}(\overrightarrow{x})\{\texttt{global} : \overrightarrow{g},\ \texttt{body} : e,\ \texttt{ret} : e\} \\
s &::= e \mid x\ \mathit{sk}\ e \mid e[\overrightarrow{e}]\ \mathit{sk}\ e \mid s\ s \\
&\quad \mid \texttt{if}(e)\{s\}\{s\} \mid \texttt{loop}\ \mathit{lk}\ (i \leftarrow \mathit{gen})\{s\} \\
\mathit{sk} &::= \ \texttt{=} \mid \texttt{+=} \\
\mathit{lk} &::= \texttt{Seq} \mid \texttt{Par} \mid \texttt{AtmPar} \\
e &::= x \mid i \mid r \mid \mathit{dist}(\overrightarrow{e}).\mathit{dop} \mid op^{n}(\overrightarrow{e}) \mid e[e] \\
\mathit{dop} &::= \texttt{ll} \mid \texttt{samp} \mid \texttt{grad}i
\end{aligned}
$$

Figure 6: The Low++ IL is an imperative language that exposes parallelism in the computation of a MCMC update, but abstracts away from details of memory management.

$\otimes$ can be shown to preserve the weaker property of invariance of the target distribution [24]. Ergodic convergence of a compound MCMC kernel, which requires considerations of irreducibility and aperiodicity, is not checked by the compiler. Static checking of such properties would be an interesting direction for future work.

### 4.2 Specifying a High-Level MCMC Algorithm

Once the compiler has decomposed the model according to its unnormalized conditionals, it will determine which base MCMC updates can be applied to each conditional. In this step, the compiler simply chooses *which* updates to apply and not *how* to implement them. Thus, the output is a program in the Kernel IL with conditionals specified in the Density IL. As a reminder, the user can supply the MCMC schedule, in which case the compiler will check that it can indeed generate the desired schedule and fail otherwise. When a user does not supply a MCMC schedule, the compiler uses a heuristic to accomplish this task. First, it determines which variables it can perform Gibbs sampling with conjugacy relations. For the remaining discrete variables, it will also apply Gibbs sampling by approximating the closed-form conditional. For the remaining continuous variables, it will apply HMC sampling to take advantage of gradient information.

### 4.3 Representing MCMC II: The Low++ IL

The Low++ IL expresses parallelism available in a MCMC algorithm. After the compiler generates code in this IL, the compiler only needs to reason at the level of an inference algorithm—all aspects of the model are eliminated. Figure 6 summarizes the syntax for the Low++ IL. The language is largely standard, so we highlight the aspects useful for encoding MCMC.

First, the IL provides additional distributions operations *dop*, including the log-likelihood `ll`, sampling `samp`, and gradients `grad`$i$, where the integer $i$ refers to the position of the argument to take the gradient with respect to. At the level of the Density IL, the distribution operation was implicitly its density. For expressing inference algorithms, we may need other operations on distributions such as sampling from them.

| MCMC update | likelihood | full-conditional | gradient |
|---|---|---|---|
| MH | ✓ | ✗ | ✗ |
| Gibbs | ✗ | ✓ | ✗ |
| HMC | ✓ | ✗ | ✓ |
| Reflective slice | ✓ | ✗ | ✓ |
| Elliptical slice | ✓ | ✗ | ✗ |

Figure 7: A summary of base MCMC updates and the primitives required to implement them.

Second, the IL contains a dedicated increment and assign statement $x$ `+=` $e$. (We can also increment and store to locations $e[\overrightarrow{e}]$ `+=` $e$.) We added this additional syntactic category because many MCMC updates require incrementing some quantity. For example, we may need to count the number of occurrences satisfying some predicate to compute a conjugacy relation or accumulate derivative computations after an application of the chain-rule. Because we hope to generate parallel inference code, this separate syntactic category indicates to the compiler that the increment and assign must be done atomically.

Third, The ILs annotate loops with whether they can be executed sequentially (`Seq`), in parallel (`Par`), or in parallel given that all increment and assign operations are done atomically (`AtmPar`). For instance, we can annotate a loop that samples a collection of conditionally independent variables in parallel (*e.g.*, when implementing a conjugacy relation) with `Par`. As we will see later when we describe AugurV2's implementation of gradients (Section 4.4), these computations will use the loop annotation `AtmPar`.

### 4.4 Primitive Support for Base MCMC Updates

The AugurV2 compiler currently supports user-supplied MH proposals, Gibbs updates, HMC updates[5], and (reflective and Elliptical) Slice updates. Fortunately, each base update can be decomposed into yet further primitives, summarized in Figure 7. The rest of the functionality can be supported as library code—the primitives encapsulate the parts of the MCMC algorithm that are specific to the full-conditional. This helps us manage the complexity of the compiler. These primitives include (1) likelihood evaluation, (2) closed-form full-conditional derivation, and (3) gradient evaluation. The compiler will implement these primitives in the Low++ IL.

***Likelihood evaluation*** It is straightforward to generate Low++ code that reifies a likelihood computation from a

[5] There is also a prototype of No-U-Turn sampling.

density factorization. It is also straightforward to parallelize these computations as a map-reduce.

***Closed-form conditional derivation*** The compiler only computes the conditionals of the model's density factorization up to a normalizing constant. To obtain a closed-form solution for the conditional, the compiler needs to solve for the normalizing constant, which requires solving an integral. As this is not analytically possible in general, the AugurV2 compiler supports closed-form conditionals in two cases.

First, like Bugs and Augur, AugurV2 exploits conjugacy relations. Recall that a *conjugacy relation* exists when the form of the conditional distribution $p(\theta \mid x)$ takes on the same functional form as $p(\theta)$. There is a well-known list of conjugacy relations. Consequently, the AugurV2 compiler supports conjugacy relations via table lookup. Computing a conjugacy relation typically involves traversing the involved variables and computing some simple statistic of the variables.

The compiler may fail to detect a conjugacy relation if (1) the approximation of the conditional is imprecise or (2) the compiler needs to perform mathematical rearrangements beyond structural pattern matching. It would be interesting to see if we can improve upon the latter situation by combining AugurV2 with a computer algebra system (CAS) as other systems have done (*e.g.*, [18]) and leveraging the CAS to solve the integral, but we leave this for future work.

Second, AugurV2 can also approximate the closed-form conditional for a discrete variable as a finite sum, even if a conjugacy relation does not exist. That is, it can generate code that directly sums over the support of the discrete variable up to some predetermined bound.

$$p(z = k \mid x) = \frac{p(z = k)\, p(x \mid z = k)}{\sum_{k'} p(z = k')\, p(x \mid z = k')}$$

***Gradient evaluation*** The compiler implements source-to-source, reverse-mode automatic differentiation (AD) to support gradient evaluation of the model likelihood. For more background, we refer the reader to the literature on AD (*e.g.*, [2]). Figure 8 summarizes the translation implemented by the AugurV2 compiler, where the input program is assumed to contain only simple expressions. The output is an *adjoint program*, *i.e.*, a program that computes the derivative. We highlight two interesting aspects of AD in the context of AugurV2.

First, the AugurV2 compiler implements source-to-source AD, in contrast with other systems (*e.g.*, Stan [8]) that implement AD by instrumenting the program. This choice was largely motivated by the lack of complex control flow in the AugurV2 modeling language, which is the primary difficulty of implementing source-to-source AD. Here, the AugurV2 compiler leverages the semantics of parallel comprehensions to optimize the gradient code. Observe that the translation reverses the order of evaluation (see the sequence translation). In particular, this means that a sequential looping construct needs a stack to save the order of evaluation. However, as parallel comprehensions have order-independence semantics, the stack can be optimized away. We also found that it was easier to support the composition of different MCMC algorithms by directly generating code that implements gradients when needed, instead of designing the system runtime to support the instrumentation required for AD. Lastly, this choice makes it easier to support compilation to the GPU as we do not need to write two runtimes, one for the CPU and one for the GPU.

Second, for the purposes of parallelization, the compiler generates atomic increments in the AD translation when accumulating gradient computations (*e.g.*, see Figure 8a). The presence of these atomic increments means that parallelization is not straightforward. To illustrate the problem, consider applying the AD transformation to the GMM model conditional $p(\mu \mid z, y^*)$, which results in the (excerpted) code:

```
grad_mu_k(K, N, ..., Sigma, mu, z, y) {
  ...
  loop AtmPar (n <- 0 until N) {
    t0 = z[n];
    t1 = MvNormal(mu[t0], Sigma[t0], y[n]).grad2;
    adj_mu[t0] += adj_ll * t1;
  }
  ret adj_mu[k];
}
```

When the number of data points $N$ greatly exceeds the number of clusters $K$, launching $N$ threads in parallel while executing the increments atomically can lead to high contention, and consequently, poor performance. Hence, the compiler needs to parallelize loops marked `AtmPar` carefully to reduce contention in an atomic increment and assign (Section 5.4).

## 5. Backend

The AugurV2 backend performs the last step of compilation and generates *native* inference code. Except for the final step where the compiler synthesizes a complete MCMC algorithm and eliminates the Kernel IL, this step is largely similar to a traditional compiler backend in terms of the transformations it performs. For AugurV2, this includes (1) size-inference to statically bound the amount of memory usage an AugurV2 MCMC algorithm consumes and (2) reifying parallelism. Towards this end, the compiler uses an IL called the *Low-- IL*.

### 5.1 Representing MCMC III: The Low-- IL

The Low-- IL is structurally the same as the Low++ IL (Figure 5), except that programs must manage memory explicitly. However, details of how to reify parallelism are still left abstract.

$$
\begin{aligned}
\overline{(x, y)} &= \bar{y} \texttt{ += } \bar{x} \\
\overline{(x, \mathit{dist}(y_1, \ldots, y_n))} &= \bar{y_1} \texttt{ += } \bar{x} * \mathit{dist}(y_1, \ldots, y_n).\texttt{grad1} \\
&\cdots \\
&\bar{y_n} \texttt{ += } \bar{x} * \mathit{dist}(y_1, \ldots, y_n).\texttt{grad}n \\
\overline{(x, y_1[y_2])} &= \bar{y_1}[y_2] \texttt{ += } \bar{x}
\end{aligned}
$$

(a) Selected adjoint expression translation. A variable notated $\bar{x}$ is the adjoint of variable $x$. It contains the result of the partial derivative with respect to $x$.

$$
\begin{aligned}
\overline{p_{\mathit{dist}(\overrightarrow{e})}(x)} &= \overline{(x, \mathit{dist}(\overrightarrow{e}))} \\
\overline{\mathit{fn}_1 \mathit{fn}_2} &= \overline{\mathit{fn}_2}; \overline{\mathit{fn}_1} \\
\overline{\texttt{let } x = e \texttt{ in } \mathit{fn}} &= x \texttt{=} e; \overline{\mathit{fn}}; \overline{(x, e)} \\
\overline{\prod_{x \leftarrow \mathit{gen}} \mathit{fn}} &= \texttt{loop AtmPar}\,(x \leftarrow \mathit{gen})\{\overline{\mathit{fn}}\} \\
\overline{[\mathit{fn}]_{x=e}} &= \texttt{if}(x = e)\{\overline{\mathit{fn}}\}\{0;\}
\end{aligned}
$$

(b) The adjoint density function translation. The absence of complex control-flow in the Density IL simplifies the implementation.

Figure 8: The construction of an adjoint program for source-to-source, reverse-mode AD in AugurV2 from the Density IL to the Low++ IL.

$$
\begin{aligned}
b ::= {} & \texttt{seqBlk}\ \{s\} \mid \texttt{parBlk}\ \mathit{lk}\ x \leftarrow \mathit{gen}\ \{s\} \\
\mid {} & \texttt{loopBlk}\ x \leftarrow \mathit{gen}\ \{b\} \\
\mid {} & e_{\mathit{acc}} = \texttt{sumBlk}\ e_0\ x \leftarrow \mathit{gen}\ \{s\ ;\ \texttt{ret}\ e\}
\end{aligned}
$$

Figure 9: The Blk IL exposes the different kinds of parallelism, including data-parallel (parBlk), reduction (sumBlk), and the absence of parallelism (seqBlk).

### 5.2 Size Inference

As a reminder, AugurV2 programs express fixed-structure models. Consequently, we can bound the amount of memory an inference algorithm uses and allocate it up front. Moreover, for GPU inference, it is necessary to determine how much memory a MCMC algorithm will consume up front because we cannot dynamically allocate memory while executing GPU code. To accomplish this, the compiler first makes all sources of memory usage explicit. For instance, primitives such as vector addition that produce a result that requires allocation will be converted into a side-effecting primitive that updates an explicitly allocated location. These functional primitives made the initial lowering step from model and query into algorithm tractable and can be removed at this step. Next, the compiler performs size inference to determine how much memory to allocate. Currently, the compiler does not support standard optimizations such as destructive updates.

### 5.3 Representing Parallelism: The Blk IL

When AugurV2 is set to target the GPU, the compiler produces Cuda/C code from Low-- IL code. Here, the compiler can finally leverage the loop annotations present in the inference code. As a reminder, the loops were annotated when the compiler first generated a base MCMC update. At this point, the compiler chooses *how* to use utilize the GPU. As with size-inference and memory management, this step is also similar to a more traditional code-generation step. Towards this end, the compiler introduces an additional IL called the *Blk IL*. The syntax is summarized in Figure 9.

The design of the IL is informed by the SIMD parallelism provided by a GPU. For example, the construct $\texttt{parBlk}\ \mathit{lk}\ x \leftarrow \mathit{gen}\ \{s\}$ indicates a parallel block of code, where *gen* copies of the statement $s$ (in the Low-- IL) can be run in parallel according to the loop annotation *lk*. This corresponds to launching *gen* threads on the GPU. The syntax $e_{\mathit{res}} = \texttt{sumBlk}\ e_0\ x \leftarrow \mathit{gen}\ \{s\ ;\ \texttt{ret}\ e\}$ indicates a summation block, where the body $s$ maps some computation across *gen* and returns the expression $e$. The result is summed with $e_0$ as the initial value and is assigned to the expression in $e_{\mathit{res}}$. Hence, this corresponds to a GPU map-reduce. The construct $\texttt{loopBlk}\ x \leftarrow \mathit{gen}\ \{b\}$ loops a block $b$. Thus, this corresponds to launching *gen* parallelized computations encoded in $b$ in sequence. The construct $\texttt{seqBlk}\ \{s\}$ encodes a sequential block of code consisting of a statement $s$. This corresponds to the absence of parallelism.

### 5.4 Parallelizing Code

To parallelize the body of a declaration in the Low-- IL, the compiler first translates it into the Blk IL. Every top-level loop we encounter in the body is converted to a parallel block with the same loop annotation. The remaining top-level statements that are not nested within a loop are generated as a sequential block. Loop blocks and summation blocks are not generated during the initial translation step, but are generated as the result of additional transformations. The current parallelization strategy is a proof-of-concept that illustrates how to reify the parallelism specific to the base MCMC updates the compiler generates and we expect that there are many opportunities for improvement here. We summarize some optimizations that we have found useful in the context of the MCMC algorithms generated by AugurV2.

***Commuting loops*** As a reminder, AugurV2 compiles at runtime so it has access to the sizes of the data and parameters. It can use this information to commute IL blocks of the form

```
parBlk Par (k <- 0 until K) {
  loop Par (n <- 0 until N) {
    ...
  }
}
```

when $K \ll N$ so that the code utilizes more GPU threads.

***Inlining*** The compiler inlines primitive functions that are implemented with loops. For example, the compiler can inline the sampling of a Dirichlet distribution, which samples a vector of Gamma distributed variables and then normalizes the vector (with `normalize`).

```
sample_dirichlet(alpha) {
  loop Par (v <- 0 until V) {
    x[v] = Gamma(alpha).samp;
  }
  normalize(x);
  ret x;
}
```

Inlining expose additional sources of parallelism. For example, if we inline the sampling of the Dirichlet distribution `sample_dirichlet` above in a `ParBlk`, then the loops may be commuted.

***Conversion to summation blocks*** Lastly, the compiler analyzes parallel blocks marked atomic parallel to see which should be converted to summation blocks. To illustrate this more concretely, suppose we have the following code produced by AD:[6]

```
parBlk AtmPar (n <- 0 until N) {
  adj_var += adj_ll * Normal(y[n], 0, var).grad3;
}
```

Parallelizing this code by launching $N$ threads leads to high contention in updating the variable `adj_var`. Instead, the compiler estimates the contention rate as the ratio of the number of threads we are parallelizing with (*i.e.*, $N$ in this example) compared to the number of locations the atomic additions are accessing (*i.e.*, 1 in this example). If the ratio is high as it is in this example ($N/1$), then the compiler converts it to a summation block.

```
adj_var = sumBlk adj_var (n <- 0 until N) {
  t = adj_ll * Normal(y[n], 0, var).grad3;
  ret t;
}
```

Importantly, the compiler is invoked at runtime so the symbolic values can be resolved.

The compiler uses the following basic heuristic to optimize a block of code. First, it inlines primitive function as described above. If inlining code results in a loop being commuted or a conversion to a summation block, then it stops and returns that block of code. Otherwise, it does not inline and returns the block as is. We have not investigated other optimizations, different orderings of the optimizations, or more generally, a cost-model to guide optimization. As an example of an area for improvement, consider the following piece of code, which the AugurV2 compiler will currently fail to parallelize well. This code could result from inlining the log-likelihood computation for a Dirichlet distribution (ignoring the normalizing constant) evaluated at the vector `x[k]` for each `k` with concentration parameter `alpha`.

```
...
parBlk AtmPar (k <- 0 until K) {
  tmp_ll = 0;
  loop AtmPar (v <- 0 until V) {
    tmp_ll += (alpha[v] - 1) * log(x[k][v]);
  }
  ll += tmp_ll;
}
...
```

In this example, the compiler would map-reduce over the outer-loop, which would be inefficient if $K \ll V$ (*e.g.*, consider a model such as LDA that has $K$ topics and $V$ words in the vocabulary). By inspection, a better strategy would be to map-reduce over both loops ($K \times V$ elements) as addition is *associative*.

Once the compiler has translated the body into the Blk IL and performed the optimizations above, it will generate Cuda/C code. The Blk IL maps in a straightforward manner onto Cuda/C code. In general, such a compilation strategy will generate multiple GPU kernels for a single Low-- declaration.

### 5.5 Synthesizing a Complete MCMC Algorithm

In this step, the compiler eliminates the Kernel IL and synthesize a complete MCMC algorithm. More concretely, the compiler generates code to compute the *acceptance ratio* (AR) associated with each base MCMC update.

Recall that a base MCMC update targeting the distribution $p(x)$ can be thought of as a MH algorithm with a proposal $q(x \to x')$ of a specific form. To ensure that the distribution $p(x)$ is sampled from appropriately, the proposals are taken (or *accepted*) with probability

$$\alpha(x, x') = \min\left(1, \frac{p(x')q(x \to x')}{p(x)q(x' \to x)}\right) ,$$

where $\alpha(x, x')$ is known as the AR. If the proposal is not taken, it is said to be *rejected*. Some base MCMC updates such as Gibbs updates are always accepted, *i.e.*, have AR

[6] This code could arise from a model with density factorization $p_{\text{Exp}}(\sigma^2)\, p_{\mathcal{N}}(x_n^* \mid 0, \sigma^2)$ where $p_{\text{Exp}}(\sigma^2)$ is an Exponential distribution and $p_{\mathcal{N}}(x_n^* \mid 0, \sigma^2)$ is a Normal distribution with variance $\sigma^2$. Importantly, any model where there is a higher-level parameter (*e.g.*, the variance parameter) that controls the shape of lower-level distributions (*e.g.*, the likelihood), will result in gradient code of this form.

$\alpha(x, x') = 1$ for any $x$ and $x'$. Thus, the acceptance ratio does not need to be computed for such updates. Other MCMC updates such as HMC updates require the computation of the AR. The compiler generates code that computes the AR after every base update that requires it. Because such base updates can be rejected, the compiler maintains two copies of the MCMC state space, one for the current state and one for the proposal state, and enforces the invariant that the two are equivalent after the execution of a base MCMC update. The invariant ensures that the execution of a base MCMC update always uses the most current state.

## 6. System Implementation

In this section, we summarize AugurV2's system implementation, which includes the compiler, the user interface, and the runtime library.

### 6.1 Compiler Implementation

The AugurV2 compiler is written in Haskell. The frontend is roughly $950$ lines of Haskell code, the middle-end is roughly $1860$ lines, and the backend is roughly $3420$ lines. The entire compiler is roughly $9000$ lines, which includes syntax, pretty printing, and type checking. We found the backend to be the most tedious to write, particularly the details of memory transfer between the Python interface and Cuda/C.

### 6.2 Runtime Library

The AugurV2 runtime library is written in Cuda/C. It provides functionality for primitive functions, primitive distributions, additional MCMC library code, and vector operations. The libraries are written for both CPU and GPU inference. We believe there is room for improvement, particularly in the GPU inference libraries. For example, in the GMM example, we need to perform the same matrix operation on many small matrices in parallel. In contrast, the typical GPU use case is to perform one matrix operation on one large matrix.

The runtime representation of AugurV2 vectors are flattened. That is, AugurV2 supports vectors of vectors (*i.e.*, ragged arrays) in its surface syntax, but the eventual representation of the data will be contained in a flattened, contiguous region of memory. This enables us to use a GPU efficiently when we want to map an operation across all the data in a vector of vectors, without following a pointer-directed structure. For this reason, the runtime representation of vectors of vectors pairs a separate pointer-directed structure with a flattened contiguous array holding the actual data. The former provides random access capabilities, while the latter enables an efficient mapping operation across the data structure. The flattened representation is also beneficial for CPU inference algorithms because of the increased locality.

## 7. Evaluation

In this section, we evaluate AugurV2's design and compare against Jags (a variant of Bugs) and Stan, systems with similar modeling languages.

### 7.1 Extensibility

As MCMC methods are improved, it is important to design systems that can be extended to incorporate the latest advances. In this part of the evaluation, we comment on the extensibility of AugurV2's design in supporting base MCMC updates.

To support a new base update, we need to (1) add a node to the Kernel IL AST and modify the parser, (2) extend common Kernel IL operations to support the new node, and (3) implement the Cuda/C code for the base update against the AugurV2 runtime. From experience, we have found the first two items to be simple, provided that the base update uses the MCMC primitives already implemented. For instance, once we have an implementation of AD, we can easily support both reflective Slice sampling and HMC using the same AD transformation. In contrast, supporting Gibbs updates were difficult because we need to implement a separate code-generator for each conjugacy relation. Fortunately, there is a well-known list of conjugacy relations so this can be done once. The third item is between $0$ lines of C code (for a Gibbs update) to $30$ lines of C code ( *e.g.*, an implementation of Leapfrog integration for the HMC update) depending on the complexity of the base update. We have found that it often takes on the order of days to add a new base update, where most of the time is spent understanding the statistics and implementing the C code.

### 7.2 Performance

In this part of the evaluation, we compare AugurV2 against Jags and Stan, concentrating on how design choices made in the AugurV2 system—(1) compositional MCMC inference, (2) compilation, and (3) parallelism—compare against those made in Jags and Stan. We consider three probabilistic models commonly used to assess the performance of PPLs (*e.g.*, see the Stan user manual [22]): (1) Hierarchical Logistic Regression (HLR), (2) Hierarchical Gaussian Mixture Model (HGMM), and (3) Latent Dirichlet Allocation (LDA). We ran all the experiments on an Ubuntu 14.04 desktop with a Core-i7 CPU and Nvidia Titan Black GPU.

***Models*** The generative model for a HLR is summarized below. It is a model that can be used to construct classifiers.

$$\begin{aligned}
\sigma^2 &\sim \text{Exponential}(\lambda)\\
b &\sim \text{Normal}(0, \sigma^2)\\
\theta_k &\sim \text{Normal}(0, \sigma^2)\\
y_n &\sim \text{Bernoulli}(\text{sigmoid}(x_n \cdot \theta_k + b))
\end{aligned}$$

The generative model for a HGMM is summarized below. It is a model that clusters points on $D$-dimensional Euclidean space.

$$
\begin{aligned}
\pi &\sim \text{Dirichlet}(\alpha) \\
\mu_k &\sim \text{Normal}(\mu_0, \Sigma_0) \\
\Sigma_k &\sim \text{InvWishart}(\nu, \Psi) \\
z_n &\sim \text{Categorical}(\pi) \\
y_n &\sim \text{Normal}(\mu_{z_n}, \Sigma_{z_n})
\end{aligned}
$$

The generative model for LDA is summarized below. It is a model that can be used to infer topics from a corpus of documents.

$$
\begin{aligned}
\theta_d &\sim \text{Dirichlet}(\alpha) \\
\phi_k &\sim \text{Dirichlet}(\beta) \\
z_{dj} &\sim \text{Categorical}(\theta_d) \\
w_{dj} &\sim \text{Categorical}(\phi_{z_{dj}})
\end{aligned}
$$

***Compositional MCMC*** To assess the impact of generating compositional MCMC algorithms, we use the HLR model and the HGMM model. The HLR model contains only continuous parameters. Hence, a system such as Stan, which is specifically designed for gradient-based MCMC algorithms, should perform well. The HGMM model is fully-conjugate. Hence, a system such as Jags, which is specifically designed for Gibbs sampling, should perform well.

For the HLR model, we visually verified the trace plots (*i.e.*, a plot of the values of the parameter from one sample to the next) of each system. On the German Credit dataset [13], we found that AugurV2 configured to generate a CPU HMC sampler with manually picked parameters to be roughly 25 percent slower than Stan set to use the same HMC sampling algorithm in generating 1000 samples (with no thinning). It would be interesting to investigate more the differences in AugurV2's implementation of AD and Stan's. Jags had the poorest performance as it defaults to adaptive rejection sampling. It takes roughly 35 seconds for Stan to compile the model (due to the extensive use of C++ templates in its implementation of AD). AugurV2 compiles almost instantaneously when generating CPU code, while it takes roughly 8 seconds to generate GPU code. The difference between CPU and GPU compile times is due to the difference in speed between Clang and Nvcc (the GPU compiler we use). Although fast compilation times are not an issue in our setting where we just need to compile the model (and query) once, this may become more of an issue in structure learning [19].

Figure 10 contains plots of the log-predictive probability versus training time for a $2D$-HGMM model with 1000 synthetically-generated data points and 3 clusters. A log-predictive probability plot can be seen as a proxy for learning–as training time increases, the algorithm should be able to make better predictions. We configured AugurV2 to generate 3 different MCMC samplers corresponding to sampling the cluster locations with Elliptical Slice updates, Gibbs updates, and HMC updates. The plot shows that every system converges to roughly the same log-predictive probability, the difference being the amount of time it takes. For instance, Jags and AugurV2's Gibbs sampler have better computational performance because they can leverage the conjugacy relation, whereas Stan uses gradient-based MCMC.

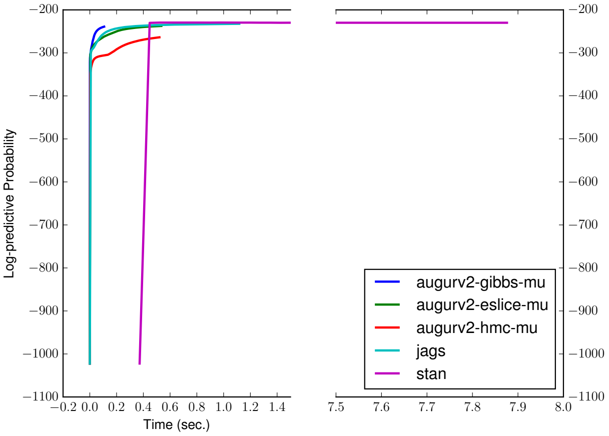

Figure 10: The log-predictive probability of a HGMM when using the samplers generated by AugurV2, Jags, and Stan. We use AugurV2 to generate 3 different MCMC inference algorithms. We set AugurV2 and Jags to draw 150 samples with no burn-in and no thinning. We set Stan to draw 100 samples with an initial tuning period of 50 samples and no thinning.

| $(k, d, n)$ | AugurV2 | Jags | Speedup |
|---|---|---|---|
| $(3, 2, 1000)$ | 0.2 | 1.1 | $\sim 5.5x$ |
| $(3, 2, 10000)$ | 1.4 | 17.4 | $\sim 12.4x$ |
| $(10, 2, 10000)$ | 3.7 | 51.5 | $\sim 13.9x$ |
| $(3, 10, 10000)$ | 15.6 | 93.0 | $\sim 5.9x$ |
| $(10, 10, 10000)$ | 17.8 | 301.9 | $\sim 16.9x$ |

Figure 11: Approximate timing results comparing the performance of AugurV2's compiled Gibbs sampler versus Jag's Gibbs sampler on a HGMM with varying clusters ($k$), dimensions ($d$), and data points ($n$).

***Compilation*** Figure 11 summarizes the timing results to generate 150 samples for a HGMM on a synthetically generated dataset for a varying number of clusters, dimensions, and data points. We set AugurV2 to generate a Gibbs update for all the variables and compare against Jags' Gibbs sampler so that both are running the same high-level inference algorithm. The difference is that Jags reifies the Bayesian network structure and performs Gibbs sampling on the graph structure, whereas AugurV2 directly generates code that performs Gibbs sampling using symbolically computed condi-

| Dataset-Topics | CPU (sec.) | GPU (sec.) | Speedup |
|---|---|---|---|
| Kos-50 | 159 | 60 | $\sim 2.7x$ |
| Kos-100 | 265 | 73 | $\sim 3.6x$ |
| Kos-150 | 373 | 82 | $\sim 4.6x$ |
| Nips-50 | 504 | 161 | $\sim 3.1x$ |
| Nips-100 | 880 | 168 | $\sim 5.2x$ |
| Nips-150 | 1354 | 235 | $\sim 5.8x$ |

Figure 12: Approximate timing results comparing the performance of AugurV2's CPU Gibbs inference against GPU Gibbs inference for LDA. The Kos dataset [13] has a vocabulary size of 6906 and contains roughly $460k$ words. The Nips dataset [13] has a vocabulary size of 12419 and roughly 1.9 million words.

tionals. The experiments show that AugurV2's approach outperforms Jags' approach. We also compared to the performance of Stan as well as checked the log-predictive probabilities for all three systems, but didn't include timing results for Stan because they aren't particularly meaningful. For instance, we applied Stan's No-U-Turn sampler to the largest model with 10 clusters in 10 dimensions, for which it takes approximately 19 hours to draw 150 samples. Notably, Stan does not natively support discrete distributions so the user must write the model to marginalize out all discrete variables, which increases the complexity of computing gradients. This highlights the importance of being able to support compositional MCMC algorithms.

***Parallelism*** Jags and Stan support parallel MCMC by running multiple copies of a chain in parallel. In contrast, AugurV2 supports parallel MCMC by parallelizing the computations *within* a single chain. As these methods are not comparable, we will focus on AugurV2's GPU inference capabilities. For these experiments, we will use all three models. We found that GPU inference can improve scalability of inference, but it is highly model-dependent.

For example, when we fit the HLR to the German Credit dataset using AugurV2's GPU HMC sampler, the computational performance was roughly an order of magnitude worse compared to AugurV2's CPU HMC sampler. This can be attributed to the small dataset size (roughly 1000 points) and the low dimensionality of the parameter space (26 parameters). When we apply the GPU HMC sampler to the Adult Income dataset [13], which has roughly 50000 observations and 14 parameters, the gradients were parallelized differently due to the summation block optimization—it is more efficient to run 14 map-reduces over 50000 elements as opposed to launching 50000 threads all contending to increment 14 locations.

In contrast to a model such as a HLR, models such as HGMM and LDA have much higher dimensional spaces. Indeed, the number of latent variables scales with the number of data points. In these cases, GPU inference was much more effective. Figure 12 summarizes the (approximate) timing results for AugurV2's performance on LDA across a variety of datasets, comparing its CPU Gibbs performance against its GPU Gibbs performance. We also checked the log-predictive probabilities to make sure that they were roughly the same for the CPU and GPU samplers. The general trend is that the GPU provides more benefit on larger datasets, with larger vocabulary sizes, and with more topics. We have also tried this with the HGMM and found the same general trend. We could not get Jags or Stan to scale to LDA, even for the smallest dataset.

## 8. Related work

The design of AugurV2 builds upon a rich body of prior work on PPLs. In this section, we compare AugurV2's design with related PPLs, using the (*model*, *query*, *inference*) tuple (*i.e.*, the probabilistic modeling and inference tuple) as a guide.

To begin, we summarize the different kinds of modeling languages proposed for expressing models. At a high-level, modeling languages either focus on (1) expressing well-known probabilistic modeling abstractions such as Bayesian networks [8, 23] as in AugurV2 or (2) are embedded into general-purpose programming languages [11, 20, 21, 28, 29]. Within both categories, there are variations with how models are expressed. For example, in the first category, the Stan [8] language enables users to specify a model by both imperatively updating its (log) density as well as writing the generative process. Bugs [23] provides an imperative language for specifying the generative process. For comparison, AugurV2 provides a first-order functional language for specifying the generative process, which simplifies the analysis a compiler performs. The Factorie [15] language expresses Factor graphs, and hence, does not necessarily express generative processes. The second category of languages provide richer modeling languages.

There are also a multitude of approaches to what queries and inference algorithms are supported. AugurV2 takes a sampling approach to posterior inference as with many other systems [20, 28, 29]. However, some systems such as Stan [8] and Infer.net [16] support multiple kinds of inference strategies, including Automatic Differentiable Variational Inference (AVDI) and Expectation Propagation respectively. Hence, these systems provide a fixed number of possible inference strategies. Hakaru [18] and Psi [10] provide capabilities for exact inference via a symbolic approach to integration.

The Blaise [5] system provides a domain-specific graphical language for expressing both models and MCMC algorithms. Hence, such a system provides a more expressive language for encoding inference algorithms beyond choosing from a preselected and fixed number of strategies. As we mentioned in the introduction, the Kernel IL used by AugurV2 is closely related to the Blaise language. In particular, it should be possible to translate the Kernel IL instantiated with the Density IL to a Blaise graph. We use the Kernel IL for the purposes of compilation, whereas in Blaise the language is interpreted. It would be an interesting direction of future work to see if the subset of the Blaise language related to inference can also be used for the purposes of compilation.

Edward [26] and Venture [14] also explore increasing the expressivity of different query and inference languages. Edward is built on top of TensorFlow [1], and hence, provides gradient-based inference strategies such as HMC and AVDI. Notably, the TensorFlow system efficiently supports the computation of gradients for an input computational graph (*e.g.*, the computational graph of the log-likelihood calculation for a probabilistic model). Consequently, Edward leverages the benefits of TensorFlow. It would be interesting direction for future work to see how much of TensorFlow's infrastructure for AD can be used in AugurV2 as done in Edward. However, note that AugurV2 provides higher-level functionality than TensorFlow. For example, AugurV2 can generate non-gradient-based MCMC algorithms (*e.g.*, Gibbs samplers) as well as compose gradient-based MCMC with non-gradient-based MCMC. Venture [14] provides an inference language that is closer to a general-purpose programming language, and hence, explores the more expressive regime of the design space for inference algorithms.

In addition to exploring various points of the (*model*, *query*, *inference*) tuple space, many PPLs have also proposed interesting implementation decisions that are related to AugurV2's. For example, the Hierarchical Bayes Compiler (HBC) [9] explores the compilation of Gibbs samplers to C code. In this work and our previous work on Augur, we support compilation of Gibbs samplers to the GPU. Swift [29] uses a compiler pass to optimize how conditional independence relationships are tracked at runtime for a modeling language that can express dynamic relationships. In contrast, AugurV2 provides a simpler modeling language that expresses static relationships, and hence, statically approximates these relationships. Bhat *et al.* [3] give a proof of correctness for a compiler that transforms a term in a first-order modeling language into a density. This language subsumes AugurV2's (*e.g.*, AugurV2 does not provide branching constructs) and can be seen as justifying the implementation of the AugurV2 frontend, which translates a term in the modeling language into its density factorization.

## 9. Conclusion

In summary, we present a sequence of ILs and the steps of compilation that can be used to convert a description of a fixed-structure, parametric model and a query for posterior samples into a composable MCMC inference algorithm. These ILs were designed to support (1) the static analysis of models to derive facts useful for inference such as the model decomposition, (2) generating multiple kinds of inference algorithms and their composition, and (3) CPU and GPU compilation. We show that this compilation scheme can scale automatic inference to probabilistic models with many latent variables (*e.g.*, LDA). We follow the traditional strategy of using ILs to cleanly separate out the requirements of each phase of compilation. It would be an interesting to see what aspects of AugurV2's ILs and compilation strategy could be reused in the context of a larger infrastructure for compiling PPLs.

## Acknowledgments

We would like to thank Vikash Mansinghka for providing valuable feedback during the shepherding process. We would also like to thank our anonymous reviewers, our reviewers from the artifact evaluation committee, and Owen Arden. Finally, we would like to thank all those who contributed to version 1 of Augur: Adam C. Pocock, Joseph Tassarotti, Guy L. Steele Jr., and Stephen X. Green. The first author is supported by Oracle Labs.

## References

[1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., KUDLUR, M., LEVENBERG, J., MONGA, R., MOORE, S., MURRAY, D. G., STEINER, B., TUCKER, P., VASUDEVAN, V., WARDEN, P., WICKE, M., YU, Y., AND ZHENG, X. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[2] BARTHOLOMEW-BIGGS, M., BROWN, S., CHRISTIANSON, B., AND DIXON, L. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics 124*, 1 (2000), 171–190.

[3] BHAT, S., BORGSTRÖM, J., GORDON, A. D., AND RUSSO, C. Deriving Probability Density Functions from Probabilistic Functional Programs. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2013), pp. 508–522.

[4] BISHOP, C. M., Ed. *Pattern Recognition and Machine Learning*. Springer, 2006.

[5] BONAWITZ, K. A. *Composable Probabilistic Inference with Blaise*. PhD thesis, Massachusetts Institute of Technology, 2008.

[6] BORGSTRÖM, J., GORDON, A. D., GREENBERG, M., MARGETSON, J., AND VAN GAEL, J. Measure Transformer Semantics for Bayesian Machine Learning. In *Programming Languages and Systems* (2011), pp. 77–96.

[7] BROOKS, S., GELMAN, A., JONES, G. L., AND MENG, X.-L., Eds. *Handbook of Markov Chain Monte Carlo*, 1 ed. Chapman and Hall/CRC, 2011.

[8] CARPENTER, B., LEE, D., BRUBAKER, M. A., RIDDELL, A., GELMAN, A., GOODRICH, B., GUO, J., HOFFMAN, M., BETANCOURT, M., AND LI, P. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* (2016). in press.

[9] Daume, III, H. Hierarchical Bayesian Compiler. `http://www.umiacs.umd.edu/~hal/HBC/`, 2008.

[10] Gehr, T., Misailovic, S., and Vechev, M. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification* (2016), Springer International Publishing, pp. 62–83.

[11] Goodman, N., Mansinghka, V., Roy, D. M., Bonawitz, K., and Tenenbaum, J. B. Church: A language for generative models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence* (2008), AUAI, pp. 220–229.

[12] Huang, D., and Morrisett, G. An Application of Computable Distributions to the Semantics of Probabilistic Programming Languages. In *Programming Languages and Systems* (2016), pp. 337–363.

[13] Lichman, M. UCI Machine Learning Repository, 2013.

[14] Mansinghka, V. K., Selsam, D., and Perov, Y. N. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR abs/1404.0099* (2014).

[15] McCallum, A., Schultz, K., and Singh, S. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems 23* (2009), Neural Information Processing Systems Foundation, pp. 1249–1257.

[16] Minka, T., Winn, J., Guiver, J., Webster, S., Zaykov, Y., Yangel, B., Spengler, A., and Bronskill, J. Infer.NET 2.6, 2014. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[17] Murray, I., Adams, R. P., and MacKay, D. J. Elliptical Slice Sampling. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (2010), SAIS, pp. 541–548.

[18] Narayanan, P., Carette, J., Romano, W., Shan, C.-c., and Zinkov, R. Probabilistic inference by program transformation in Hakaru (System Description). In *International Symposium on Functional and Logic Programming* (2016), Springer, pp. 62–79.

[19] Neapolitan, R. E. *Learning Bayesian Networks.* Prentice

[19] Neapolitan, R. E. *Learning Bayesian Networks.* Prentice Hall, 2004.

[20] Nori, A. V., Hur, C.-K., Rajamani, S. K., and Samuel, S. R2: An efficient mcmc sampler for probabilistic programs. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence* (2010), AAAI, pp. 2476–2482.

[21] Paige, B., and Wood, F. A Compilation Target for Probabilistic Programming Languages. In *Proceedings of the 31st International Conference on Machine Learning* (2014), JMLR.

[22] Stan Development Team. Stan Modeling Language: User's Guide and Reference Manual. http://mc-stan.org/documentation/, 2015.

[23] Thomas, A., Spiegelhalter, D. J., and Gilks, W. R. BUGS: A program to perform Bayesian inference using Gibbs sampling. *Bayesian Statistics 4*, 9 (1992), 837–842.

[24] Tierney, L. Markov Chains for Exploring Posterior Distributions. *Ann. Statist. 22*, 4 (12 1994), 1701–1728.

[25] Toronto, N., McCarthy, J., and Van Horn, D. Running probabilistic programs backwards. In *Programming Languages and Systems* (2015), Springer, pp. 53–79.

[26] Tran, D., Hoffman, M. D., Saurous, R. A., Brevdo, E., Murphy, K., and Blei, D. M. Deep Probabilistic Programming. *arXiv preprint arXiv:1701.03757* (2017).

[27] Tristan, J.-B., Huang, D., Tassarotti, J., Pocock, A. C., Green, S., and Steele, G. L. Augur: Data-Parallel Probabilistic Modeling. In *Advances in Neural Information Processing Systems 28* (2014), Neural Information Processing Systems Foundation, pp. 2600–2608.

[28] Wood, F., van de Meent, J. W., and Mansinghka, V. A new approach to probabilistic programming inference. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics* (2014), SAIS, pp. 2–46.

[29] Wu, Y., Li, L., Russell, S., and Bodík, R. Swift: Compiled Inference for Probabilistic Programming Languages. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence* (2016), pp. 3637–3645.